Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Συστηματικός Έλεγχος Ορθότητας του Read-Copy-Update υπό Ακολουθιακά Συνεπή και Ασθενή Μοντέλα Μνήμης

Διπλωματική Εργασία

ΚΟΚΟΛΟΓΙΑΝΝΑΚΗΣ ΜΙΧΑΗΛ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και
Υπολογιστων

Συστηματικός Έλεγχος Ορθότητας του Read-Copy-Update υπό Ακολουθιακά Συνεπή και Ασθενή Μοντέλα Μνήμης

Διπλωματική Εργασία

ΚΟΚΟΛΟΓΙΑΝΝΑΚΗΣ ΜΙΧΑΗΛ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 13η Οκτωβρίου 2016.

.....	.....	.....
Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Νικόλαος Σ. Παπασπύρου Αν. Καθηγητής Ε.Μ.Π.	Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2016

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

Περίληψη

Ο συστηματικός έλεγχος ορθότητας και η επαλήθευση παράλληλων προγραμμάτων, παρουσιάζουν σημαντικές δυσκολίες. Ας θεωρήσουμε, για παράδειγμα, μια βιβλιοθήκη προγραμμάτων που χρησιμοποιείται για το συγχρονισμό μεταξύ διεργασιών, ή για τον έλεγχο πρόσβασης σε κάποιες κοινές μεταξύ διαφόρων νημάτων δομές δεδομένων. Κατά την κατασκευή μιας συστηματικής διαδικασίας ελέγχου για μία τέτοια βιβλιοθήκη, πρέπει να ληφθούν υπ' όψιν: οι πολλοί πιθανοί τρόποι με τους οποίους τα νήματα-πελάτες έχουν πρόσβαση στη βιβλιοθήκη, οι διαφορετικοί τρόποι με τους οποίους τα νήματα και η βιβλιοθήκη αλληλεπιδρούν, καθώς και οποιεσδήποτε επιπτώσεις οφείλονται στο ασθενές μοντέλο μνήμης (weak memory model) που χρησιμοποιείται από τους σύγχρονους μικροεπεξεργαστές.

Ο σκοπός της διπλωματικής αυτής εργασίας είναι ο συστηματικός έλεγχος ορθότητας της βιβλιοθήκης Read-Copy-Update, ενός μηχανισμού συγχρονισμού που χρησιμοποιείται ευρέως στον πυρήνα του λειτουργικού συστήματος Linux. Για το σκοπό αυτό, χρησιμοποιήθηκε το Nidhugg, ένα εργαλείο ελέγχου μοντέλου χωρίς αποθήκευση της κατάστασης (stateless model checking tool) για C/pthreads προγράμματα, το οποίο ενσωματώνει επεκτάσεις για τη μελέτη των επιπτώσεων διαφόρων ασθενών μοντέλων μνήμης, όπως TSO, PSO και POWER. Κατασκευάσαμε μία μη-φορμαλιστική, αλλά πλήρη λίστα προδιαγραφών για το Read-Copy-Update, καθώς και μία κατάλληλη συλλογή προγραμμάτων που στοχεύουν στον συστηματικό έλεγχό του. Στην εργασία αυτή, παρουσιάζουμε την πρώτη μηχανική επαλήθευση της ιδιότητας της "Περιόδου Χάριτος" (Grace-Period guarantee) του RCU, χρησιμοποιώντας το Tree RCU, το οποίο αποτελεί την κύρια υλοποίηση που χρησιμοποιείται στον πυρήνα του Linux. Επιπρόσθετα, αναπαρηγάγαμε ένα γνωστό σφάλμα (bug) στον πυρήνα χρησιμοποιώντας το Nidhugg, ενώ αντίθετα, δείξαμε ότι ένα προηγουμένως αναφερόμενο ως bug, στην πραγματικότητα δεν αποτελεί bug. Τέλος, ελέγχθηκαν επίσης κάποιες ακόμη ιδιότητες του Tree RCU και του Tiny RCU.

Λέξεις κλειδιά

Συστηματικός Έλεγχος Ορθότητας, Τεχνικές Συγχρονισμού, Παράλληλος Προγραμματισμός, Επαλήθευση Προγραμμάτων, Read-Copy-Update, RCU, Linux kernel.

Abstract

Thorough verification and testing of concurrent programs is an important, but also challenging problem. Consider, for instance, a library for synchronization between processes, or for providing access to some shared data structure by many threads. When setting up a thorough testing procedure for such a library, one must consider: the many possible ways in which client threads can access this library, all the different ways in which the threads in the library itself can interact, and any possible effects of the weak memory consistency models which are employed in modern microprocessors.

This thesis investigates the systematic concurrency testing of the Read-Copy-Update library, a synchronization mechanism used heavily in the Linux kernel. For this purpose we used Nidhugg, a stateless model checking tool for C/pthreads programs, which incorporates extensions for checking effects of weak memory models such as TSO, PSO and POWER. An informal, yet precise specification for Read-Copy-Update has been constructed, along with a suitable systematic test suite for it. In this thesis, we present the first mechanical validation of the Grace-Period guarantee of Linux kernel's Tree RCU implementation, for non-preemptible environments. Furthermore, we managed to reproduce a known bug in the kernel using Nidhugg, and to prove that a previously reported bug does not in reality qualify as a bug. Finally, some other properties of Tree RCU and Tiny RCU have been tested as well.

Key words

Formal Verification, Stateless Model Checking, Systematic Concurrency Testing, Synchronization Techniques, RCU, Read-Copy-Update, Deferred Destruction, Linux Kernel, Parallel Programming.

Ευχαριστίες

Καταρχάς, οφείλω να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Κωστή Σαγώνα, για όλη τη βοήθεια και την υποστήριξή του κατά τη διάρκεια αυτής της προσπάθειας. Μέσω του ενθουσιασμού του και του ενδιαφέροντός του προς την έρευνα, κατάφερε να με παρακινήσει και να με εμπνεύσει, ενώ οι ουσιαστικές συμβουλές του και η καθοδήγησή του καθ' όλη την διάρκεια της παρούσας εργασίας ήταν πολύτιμες. Η εργασία μαζί του τον τελευταίο χρόνο ήταν απολαυστική, κι ελπίζω να έχω την ευκαιρία να ξαναδουλέψω μαζί του μελλοντικά.

Επίσης, οφείλω να ευχαριστήσω τον Paul McKenney, που αφιέρωσε πολύ από τον -λιγοστό- διαθέσιμο χρόνο του για να απαντήσει στις ερωτήσεις μου. Η βαθιά γνώση και κατανόησή του σε θέματα σχετικά με το RCU με βοήθησαν σε πολλές περιπτώσεις, ενώ οι προτάσεις και οι συμβουλές του ήταν εξαιρετικά χρήσιμες.

Τέλος, χρωστάω ένα μεγάλο ευχαριστώ στους γονείς μου και τον αδερφό μου για την υπόμονή τους, την αγάπη τους και την υποστήριξή τους όλα αυτά τα χρόνια. Εξάλλου, αν δεν ήταν αυτοί, η προσπάθεια αυτή δεν θα ήταν δυνατή.

Κοκολογιαννάκης Μιχαήλ,

Αθήνα, 13η Οκτωβρίου 2016

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-1-16, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Οκτώβριος 2016.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Κατάλογος σχημάτων

Κώδικες

Κεφάλαιο 1

Εισαγωγή

Όλοι όσοι έχουν ασχοληθεί με την ανάπτυξη λογισμικού διαθέτουν εμπειρία με προγράμματα που δεν δούλεψαν, ή, δεν τηρούσαν ορισμένες προδιαγραφές. Η αξιοπιστία του λογισμικού έχει τεράστιες επιπτώσεις στους ανθρώπους που το χρησιμοποιούν, με τα αναξιόπιστα λογισμικά να είναι υπεύθυνα για μία πληθώρα προβλημάτων: από χαμένο χρόνο (που μάλλον έχει ξοδεφθεί για αποσφαλμάτωση του λογισμικού) έως και θάνατο (αν, λόγου χάρη, ένα σύστημα κρίσιμο για την υγεία ή την ασφάλεια ανθρώπων αποτύχει). Αναλογιζόμενοι τα παραπάνω, μαζί με το γεγονός ότι ένα μεγάλο ποσοστό των σημερινών προγραμμάτων έχει σχεδιασθεί με βάση τις δυνατότητες ταυτοχρονισμού που προσφέρουν οι σημερινοί επεξεργαστές, καταλήγουμε στο συμπέρασμα ότι η σωστή συμπεριφορά των παράλληλων προγραμμάτων είναι ζωτικής σημασίας.

Ωστόσο, η συγγραφή ενός παράλληλου προγράμματος το οποίο λειτουργεί όπως αναμένεται είναι πολύ πιο δύσκολη από τη συγγραφή ενός μονονηματικού προγράμματος. Υπάρχει ένα πολύ μεγαλύτερο σύνολο κινδύνων το οποίο περιλαμβάνει όλους τους κινδύνους που παρουσιάζονται σε μονονηματικά προγράμματα, συν κάποιους άλλους κινδύνους παρόντες μόνο σε παράλληλα προγράμματα. Οι τελευταίοι περιλαμβάνουν: καταστάσεις ανταγωνισμού (race conditions), αδιέξοδα (deadlocks) και ζωντανά αδιέξοδα (livelocks). Επιπρόσθετα, κατά τη συγγραφή τέτοιων προγραμμάτων πρέπει να λαμβάνονται υπ' όψιν και οι πιθανές επιπτώσεις που έχουν τα ασθενή (αδύναμα) μοντέλα μνήμης [Wikia, Rela] που εφαρμόζονται στους μοντέρνους επεξεργαστές. Στα ασθενή μοντέλα μνήμης είναι δυνατό να υπάρξουν αναδιατάξεις μνήμης (memory reorderings) οι οποίες πραγματοποιούνται για την πλήρη εκμετάλλευση του εύρου ζώνης διαύλου διαφορετικών τύπων μνήμης, όπως η κρυφή μνήμη. Συνεπώς, αν δεν χρησιμοποιηθούν αποτελεσματικά φράχτες μνήμης (memory barriers), η συμπεριφορά ενός παράλληλου προγράμματος μπορεί να μην είναι η αναμενόμενη.

Συστηματικός Έλεγχος Ορθότητας Παράλληλων Προγραμμάτων

Είναι προφανές ότι ο συστηματικός έλεγχος ορθότητας και η επαλήθευση παράλληλων προγραμμάτων παρουσιάζουν σημαντικές δυσκολίες. Ο *έλεγχος μοντέλου* [Wikib] είναι μία τεχνική που εξερευνεί με συστηματικό τρόπο τον χώρο καταστάσεων ενός προγράμματος, και επιβεβαιώνει ότι κάθε προσβάσιμη κατάσταση ικανοποιεί ορισμένες προδιαγραφές. Με άλλα λόγια, ο έλεγχος μοντέλου είναι μια τεχνική που στοχεύει στην φορμαλιστική επαλήθευση παράλληλων συστημάτων. Οι προδιαγραφές για ένα σύστημα εκφράζονται σαν λογικές φόρμουλες, και αποδοτικοί αλγόριθμοι χρησιμοποιούνται για τη διάσχιση του μοντέλου που ορίζεται από το σύστημα και τον έλεγχο των προδιαγραφών.

Όμως, τα εργαλεία ελέγχου μοντέλου είναι αντιμέτωπα με την συνδυαστική έκρηξη του χώρου καταστάσεων, αφού καλούνται να αποθηκεύσουν έναν τεράστιο αριθμό καθολικών καταστάσεων του συστήματος. Για την αντιμετώπιση αυτού του προβλήματος, έχουν αναπτυχθεί διάφορες έξυπνες τεχνικές.

Ο *φραγμένος έλεγχος μοντέλου* [Bier03] (bounded model checking), για παράδειγμα, εξερευνεί το πεπερασμένο αυτόματο που προκύπτει από τον χώρο καταστάσεων μόνο

για ένα συγκεκριμένο αριθμό βημάτων, έστω k, κι ελέγχει για παραβιάσεις των προδιαγραφών του συστήματος μόνο εντός των k αυτών βημάτων. Ο φραγμένος έλεγχος μοντέλου συνδυασμένος με τη χρήση μερικών διατάξεων για τη μοντελοποίηση εκτελέσεων του προγράμματος [Algl13] έχει υλοποιηθεί αποτελεσματικά σε εργαλεία όπως ο CBMC [Clar04].

Ο *έλεγχος μοντέλου χωρίς αποθήκευση της κατάστασης* [Gode97] (stateless model checking), γνωστός και σαν συστηματικός έλεγχος ταυτοχρονισμού, είναι μία τεχνική με μικρές απαιτήσεις μνήμης εφαρμόσιμη σε προγράμματα με εκτελέσεις πεπερασμένου μήκους. Ο έλεγχος μοντέλου χωρίς αποθήκευση της κατάστασης εξερευνεί τον χώρο καταστάσεων ενός προγράμματος χωρίς να αποθηκεύει ρητά καταστάσεις. Η τεχνική αυτή έχει υλοποιηθεί με επιτυχία σε εργαλεία όπως το VeriSoft [Gode05], το CHESS [Musu08], το Concuerror [Chri13], και πρόσφατα το Nidhugg [Abdu15]. Μερικά από τα εργαλεία αυτά επιχειρούν επίσης να αντιμετωπίσουν το πρόβλημα της συνδυαστικής έκρηξης στον αριθμό των εκτελέσεων που εξερευνούν διατηρώντας όμως πλήρη κάλυψη του χώρου καταστάσεων με χρήση τεχνικών *αναγωγής μερικών διατάξεων* [Valm91, Pele93, Gode96, Clar99] (partial order reduction). Η αναγωγή μερικών διατάξεων βασίζεται στην παρατήρηση ότι δύο εκτελέσεις μπορούν να θεωρηθούν ισοδύναμες αν η μία προκύπτει από την άλλη αλλάζοντας γειτονικά, ανεξάρτητα βήματα εκτέλεσης. Τεχνικές *Δυναμικής Αναγωγής Μερικών Διατάξεων* (Dynamic Partial Order Reduction – DPOR) βρίσκουν τις εξαρτήσεις μεταξύ των εντολών των διαφόρων νημάτων ενώ το πρόγραμμα εκτελείται [Flan05]. Η εξερεύνηση ξεκινά με μία τυχαία εκτέλεση, της οποίας τα βήματα χρησιμοποιούνται για τον εντοπισμού αλληλοεξαρτώμενων εντολών και σημείων όπου διαφορετικές εκτελέσεις πρέπει να εξερευνηθούν για να καλυφθούν όλες οι πιθανές συμπεριφορές του προγράμματος.

Ο έλεγχος μοντέλου χωρίς αποθήκευση της κατάστασης μπορεί να επεκταθεί για να χειρίζεται ασθενή μοντέλα μνήμης. Για παράδειγμα, το Nidhugg [Abdu15], είναι ένα εργαλείο ελέγχου μοντέλου χωρίς αποθήκευση της κατάστασης για προγράμματα σε C/C++ που χρησιμοποιούν pthreads, το οποίο ενσωματώνει επεκτάσεις για την εύρεση σφαλμάτων (bugs) που οφείλονται σε ασθενή μοντέλα μνήμης όπως TSO, PSO και POWER. Το Nidhugg χρησιμοποιεί έναν πολύ αποδοτικό αλγόριθμο που ονομάζεται source-DPOR [Abdu14], και στην εργασία αυτή χρησιμοποιήσαμε το Nidhugg σε όλους τους ελέγχους που πραγματοποιήθηκαν.

Σκοπός της Διπλωματικής

Η διπλωματική αυτή στοχεύει στον συστηματικό έλεγχο ορθότητας του Read-Copy-Update (RCU), ενός μηχανισμού συγχρονισμού που χρησιμοποιείται στον πυρήνα του λειτουργικού συστήματος Linux [Linu]. Το Read-Copy-Update είναι ένας μηχανισμός συγχρονισμού που χρησιμοποιείται ευρέως στον πυρήνα, και πολλά από τα υποσυστήματα του πυρήνα βασίζονται στο RCU. Φυσικά, το γεγονός ότι ο πυρήνας του Linux χρησιμοποιείται σε περισσότερες από ένα δισεκατομμύριο συσκευές σήμερα [Call15] καθιστά τον έλεγχο του RCU ακόμη πιο σημαντικό.

Προηγούμενες προσπάθειες που έχουν γίνει για τον συστηματικό έλεγχο ορθότητας του RCU περιλαμβάνουν την περιγραφή της σημασιολογίας του RCU σε όρους λογικής διαχωρισμού [Gots13] (separation logic), καθώς και τον συστηματικό έλεγχο ορθότητας του RCU χώρου χρήστη (userspace-RCU) σε μία λογική για ασθενή μοντέλα μνήμης [Tass15]. Μια ψηφιακή αρχιτεκτονική για τη μοντελοποίηση προσβάσεων μνήμης εκτός σειράς (out-of-order) και τη χρονοδρομολόγηση εντολών έχει επίσης προταθεί [Desn13]. Ένας συστηματικός έλεγχος ορθότητας του RCU χώρου χρήστη έχει γίνει χρησιμοποιώντας το εργαλείο ελέγχου μοντέλου SPIN [Desn12]. Ακόμη, ερευνητές στο πανεπιστήμιο Stony Brook παρήγαγαν έναν εντοπιστή καταστάσεων ανταγωνισμού δεδομένων (data races) που μπορεί να χρησιμοποιηθεί για το RCU [Dugg10, Seys12].

Ο Paul E. McKenney επαλήθευσε τη βασική εγγύηση που προσφέρει το RCU για το Tiny RCU (έκδοση του RCU για μονοεπεξεργαστικά συστήματα) χρησιμοποιώντας τον CBMC [McKe15d], και αυτή σηματοδότησε την πρώτη προσπάθεια συστηματικού ελέγχου του RCU χρησιμοποιώντας τον κώδικα του πυρήνα απευθείας. Τέλος, στρατηγικές ελέγχου με χρήση μεταλλάξεων (mutation testing) έχουν εφαρμοστεί στον κώδικα του RCU που υπάρχει στον πυρήνα [Ahme15].

Η δική μας συνεισφορά περιλαμβάνει τον συστηματικό έλεγχο ορθότητας των βασικών ιδιοτήτων τόσο του Tiny RCU, όσο και του Tree RCU, υλοποιήσεις του RCU χρησιμοποιούμενες στον πυρήνα του Linux. Για το σκοπό αυτό, παρουσιάζεται μια μη-φορμαλιστική αλλά πλήρης συλλογή των προδιαγραφών του RCU, μαζί με μία συλλογή προγραμμάτων για τον συστηματικό έλεγχο του RCU που κατασκευάστηκε με βάση τις προδιαγραφές αυτές. Επιπρόσθετα, αναπαραγήγαμε ένα γνωστό σφάλμα (bug) που υπήρχε στον πυρήνα, και δείξαμε ότι ένα προηγουμένως αναφερθέν ως σφάλμα, στην πραγματικότητα δεν αποτελεί σφάλμα. Όλοι οι έλεγχοι που πραγματοποιήσαμε χρησιμοποίησαν τον κώδικα του πυρήνα απευθείας με ελάχιστες τροποποιήσεις, ενώ οι περισσότερες από τις προσπάθειες για συστηματικό έλεγχο του RCU που έχουν γίνει χρησιμοποιούσαν μοντέλα που απαιτούσαν τον μετασχηματισμό του πηγαίου κώδικα του RCU. Επιπρόσθετα, τέσσερις διαφορετικές εκδόσεις του πυρήνα συμπεριλήφθηκαν, με σκοπό να καλυφθεί ένα μεγάλο εύρος συμπεριφορών στους ελέγχους μας.

Οργάνωση της Διπλωματικής

Στο Κεφάλαιο 2 παρουσιάζονται οι θεμελιώδεις μηχανισμοί του RCU, καθώς και ορισμένες προδιαγραφές που κάθε υλοποίηση του RCU οφείλει να τηρεί. Στα Κεφάλαια 3 και 4 παρουσιάζεται ο τρόπος που μοντελοποιήσαμε το περιβάλλον του πυρήνα, καθώς και η ανάπτυξη μιας συλλογής προγραμμάτων, σχεδιασμένων για τον συστηματικό έλεγχο του Tiny RCU και του Tree RCU. Στο Κεφάλαιο 5 πραγματοποιείται μια ανακεφαλαίωση των παραπάνω και παρουσιάζονται συμπεράσματα που αντλούνται από την εργασία αυτή, καθώς και πιθανές μελλοντικές επεκτάσεις της.

Κεφάλαιο 2

Η βιβλιοθήκη Read-Copy-Update (RCU)

Στο κεφάλαιο αυτό θα περιγράψουμε τη βασική ιδέα και την υλοποίηση του μηχανισμού Read-Copy-Update (RCU) του πυρήνα του Linux. Θα αναλύσουμε όλους τους μηχανισμούς που προσφέρει το RCU, καθώς και τον τρόπο που αυτοί οι μηχανισμοί δουλεύουν. Επιπρόσθετα, θα παρουσιάσουμε ορισμένες απαιτήσεις που πρέπει να ικανοποιεί κάθε υλοποίηση του RCU, συμπεράσματα που αντλούνται από τη χρήση του RCU, και, φυσικά, τον τρόπο που ένα καθοριστικό πρόγραμμα ελέγχου το οποίο στοχεύει στον συστηματικό έλεγχο ορθότητας του μηχανισμού αυτού, μπορεί να κατασκευαστεί.

2.1 Εισαγωγή στο RCU

Το Read-Copy-Update (RCU) είναι ένας μηχανισμός συγχρονισμού που εφευρέθηκε από τους Paul E. McKenney και John (Jack) D. Slingwine, και προστέθηκε στον πυρήνα (kernel) του λειτουργικού συστήματος Linux τον Οκτώβριο του 2002 [Wikic, McKe13, McKe04, Kernc, McKe]. Το κύριο χαρακτηριστικό του RCU είναι ότι επιτρέπει στις αναγνώσεις (reads) να συμβαίνουν ταυτόχρονα με τις εγγραφές (updates), κάτι που το καθιστά εξαιρετικά κλιμακώσιμο.

Αν και το παραπάνω μπορεί να φαίνεται παράλογο ή αδύνατο εκ πρώτης όψεως, το RCU το πετυχαίνει με έναν πολύ απλό, αλλά εξαιρετικά αποδοτικό τρόπο: διατηρώντας πολλαπλά αντίγραφα των δεδομένων. Το RCU είναι προσεκτικά υλοποιημένο με τρόπο που όχι μόνο διαφαλίζει τη συνάφεια (coherency) των αναγνώσεων και ότι δεν θα διαγραφούν δεδομένα μέχρις ότου όλα τα προϋπάρχοντα κρίσιμα τμήματα των αναγνωστών (read-side critical sections) ολοκληρωθούν, αλλά επίσης χρησιμοποιεί αποδοτικούς και κλιμακώσιμους μηχανισμούς οι οποίοι κάνουν τα μονοπάτια ανάγνωσης (read paths) εξαιρετικά γρήγορα. Στην πραγματικότητα, σε μη-προεκχωρητικούς (non-preemptible) πυρήνες το RCU δεν προσθέτει κανενός είδους καθυστέρηση στους αναγνώστες. Φυσικά, οι ενημερώσεις μπορεί να είναι σχετικά ακριβές μιας και οι εγγραφείς πρέπει να διατηρούν πολλαπλά αντίγραφα των δεδομένων, έτσι ώστε να εξυπηρετούνται οι προϋπάρχοντες αναγνώστες. Γίνεται προφανές ότι, αφού υπάρχουν πολλές εκδόσεις των δεδομένων, το RCU μάλλον δεν εφαρμόζεται σε περιπτώσεις που οι αναγνώστες δεν πρέπει να διαβάζουν μη-ενημερωμένα δεδομένα (βλ. Ενότητα 2.3.2). Παρ' όλο που το RCU χρησιμοποιείται σε πολλά μέρη στον πυρήνα του Linux, η ακριβής έκταση της εφαρμοσιμότητάς του είναι ακόμη ένα θέμα υπό έρευνα [Trip09].

Η βασική ιδέα πίσω από το RCU είναι ο χωρισμός των εγγραφών/ενημερώσεων (updates) σε δύο φάσεις, τη φάση της *αφαίρεσης* (removal phase) και τη φάση της *ανάκτησης* (reclamation phase). Κατά τη φάση της αφαίρεσης οι αναφορές σε κάποια δεδομένα *αφαιρούνται*, είτε καταστρέφοντάς τες (π.χ. θέτοντάς τες `NULL`), είτε αντικαθιστώντας τες με αναφορές σε νεότερες εκδόσεις των αντικειμένων αυτών. Η φάση αυτή μπορεί να τρέξει παράλληλα με τις αναγνώσεις λόγω του ότι οι μοντέρνοι επεξεργαστές εγγυώνται ότι οι αναγνώστες θα δουν είτε την καινούργια, είτε την παλιά αναφορά σε ένα αντικείμενο, και όχι μια περίεργη μίξη της παλιάς και της καινούργιας αναφοράς,

ή μια μερικώς ανανεωμένη αναφορά. Κατά τη φάση της ανάκτησης *ανακτώνται* (π.χ. ελευθερώνονται καλώντας τη `free()`) τα δεδομένα που αφαιρέθηκαν κατά τη φάση της αφαίρεσης. Φυσικά, επειδή το RCU επιτρέπει ταυτόχρονες εγγραφές και αναγνώσεις, η φάση της ανάκτησης πρέπει να ξεκινήσει *μετά* τη φάση της αφαίρεσης και, πιο συγκεκριμένα, όταν είναι σίγουρο ότι δεν υπάρχουν αναγνώστες που προσπελαύνουν, ή έχουν αναφορές στα δεδομένα που ανακτώνται.

Έτσι, η τυπική διαδικασία ενημερώσεων με χρήση του RCU μοιάζει με την παρακάτω [McKe98]:

0. Διασφάλιση ότι όλοι οι αναγνώστες προσπελαύνουν δομές δεδομένων που προστατεύονται από το RCU μέσα σε κάποιο κρίσιμο τμήμα ανάγνωσης.

1. Αφαίρεση των αναφορών σε μια δομή δεδομένων, έτσι ώστε επακόλουθοι αναγνώστες να μην μπορούν να αποκτήσουν αναφορές στη δομή αυτή (φάση αφαίρεσης).

2. Αναμονή μέχρις ότου όλοι οι προϋπάρχοντες αναγνώστες να έχουν ολοκληρώσει το κρίσιμο τμήμα ανάγνωσής τους (RCU read-side critical section), ούτως ώστε να μην υπάρχει κανείς που να έχει κάποια αναφορά στο αντικείμενο που αφαιρείται.

3. Στο σημείο αυτό δεν γίνεται να υπάρχουν αναγνώστες που να διατηρούν αναφορές στη δομή δεδομένων, οπότε η δομή αυτή μπορεί να ανακτηθεί με ασφάλεια, π.χ. να απελευθερωθεί με χρήση της `free()` (φάση ανάκτησης).

Τα βήματα 1 και 3 στην προαναφερθείσα διαδικασία δεν πραγματοποιούνται κατ' ανάγκην από το ίδιο νήμα.

Το βήμα 2 στη διαδικασία αυτή είναι η βασική ιδέα πίσω από το RCU και την αναβαλλόμενη καταστροφή των δεδομένων. Η ικανότητα να αναμένουμε μέχρις ότου όλοι οι προϋπάρχοντες αναγνώστες να έχουν ολοκληρώσει το κρίσιμο τμήμα ανάγνωσής τους επιτρέπει στους αναγνώστες να λειτουργούν με συγχρονισμό ελαφριάς μορφής, ή, σε μερικές περιπτώσεις, χωρίς κανενός είδους συγχρονισμό. Η αναμονή για τους προϋπάρχοντες αναγνώστες μπορεί να επιτευχθεί είτε με φραγμό του νήματος (blocking), είτε απλώς καταχωρίζοντας μία επανάκληση συνάρτησης (callback) η οποία θα πραγματοποιηθεί αφού όλοι οι προϋπάρχοντες αναγνώστες έχουν ολοκληρώσει το κρίσιμο τμήμα ανάγνωσής τους.

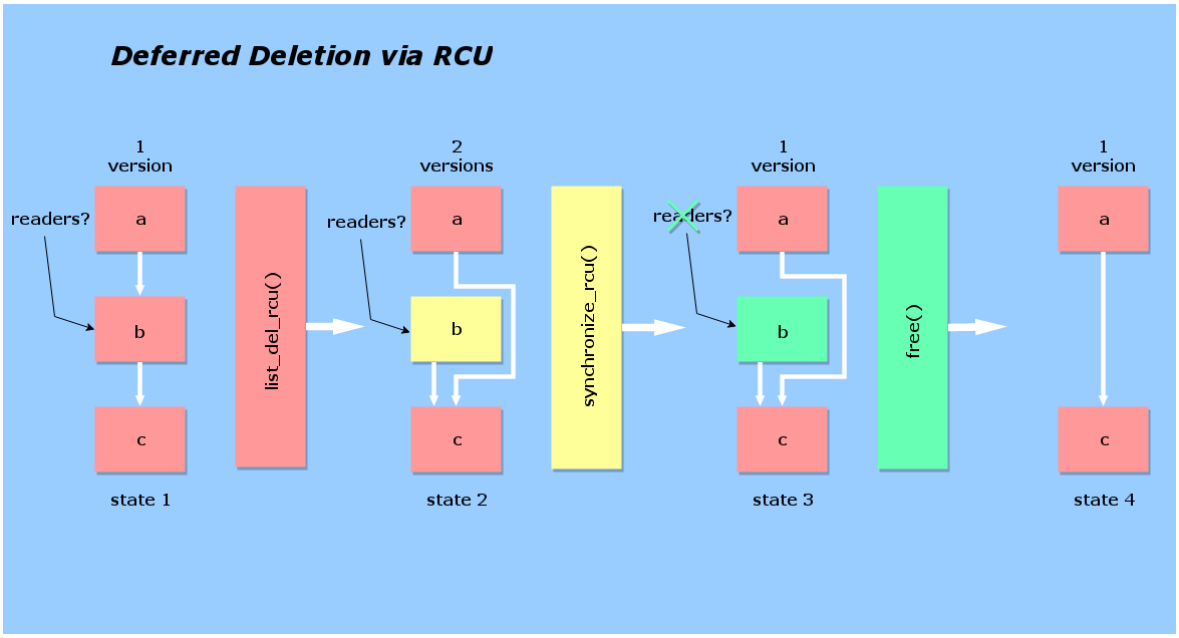
Η τυπική διαδικασία ενημερώσεων με χρήση του RCU παρουσιάζεται στο Σχήμα 2.1.

2.2 Αναλυτική Περιγραφή των Μηχανισμών του RCU

Το RCU αποτελείται από τρεις θεμελιώδεις μηχανισμούς [McKe07c]: έναν για την εισαγωγή δεδομένων, έναν για την αναμονή ολοκλήρωσης των κρίσιμων τμημάτων ανάγνωσης όλων των προϋπαρχόντων αναγνωστών, και έναν για τη διατήρηση πολλαπλών αντιγράφων των δεδομένων. Οι μηχανισμοί αυτοί περιγράφονται στις επόμενες υποενότητες μαζί με παραδείγματα τα οποία είναι χρήσιμα στην κατανόηση του τρόπου με τον οποίον δουλεύουν.

2.2.1 Μηχανισμός Έκδοσης–Συνδρομής

Όπως έχει ήδη αναφερθεί, το βασικό χαρακτηριστικό του RCU είναι ότι επιτρέπει ταυτόχρονες εγγραφές και αναγνώσεις, ή, με άλλα λόγια, το γεγονός ότι οι εγγραφείς μπορούν να τροποποιήσουν κάποια δεδομένα ενώ αναγνώστες ταυτόχρονα διαβάζουν αυτά τα δεδομένα. Το RCU επιτυγχάνει το παραπάνω παρέχοντας έναν μηχανισμό για την εισαγωγή δεδομένων, ο οποίος προσομοιάζει έναν μηχανισμό Έκδοσης-Συνδρομής



Σχήμα 2.1: Αναβαλλόμενη διαγραφή με χρήση του RCU (σχήμα προσαρμοσμένο από [McKe13]).

(Publish-Subscribe). Ο καλύτερος τρόπος για να κατανοήσουμε τον μηχανισμό αυτόν είναι μέσω ενός απλού παραδείγματος.

Αρχικά, ας θεωρήσουμε έναν καθολικό δείκτη `gp`, ο οποίος είναι αρχικοποιημένος σε `NULL`. Ας υποθέσουμε ακόμη ότι ένας εγγραφέας τροποποιεί τον δείκτη αυτόν ώστε να δείχνει σε μία νέα, αρχικοποιημένη δομή δεδομένων που δεσμεύθηκε δυναμικά. Ένα τμήμα κώδικα που πραγματοποιεί τα παραπάνω παρουσιάζεται στον Κώδικα 2.1.

```
1  struct foo {
2    int a;
3    int b;
4  };
5  struct foo *gp = NULL;
6
7  /* . . . */
8
9  p = kmalloc(sizeof(*p), GFP_KERNEL);
10 p->a = 42;
11 p->b = 42;
12 gp = p;
```

Listing 2.1: Αναδιάταξη των εντολών κατά την Έκδοση.

Δυστυχώς, δεν υπάρχει καμία απολύτως εγγύηση για το ότι οι τελευταίες τέσσερις εντολές θα εκτελεστούν με τη σειρά. Τόσο ο μεταγλωττιστής όσο κι ο επεξεργαστής μπορούν να αναδιατάξουν τις εντολές αυτές στο βωμό της αποδοτικότητας. Τα γεγονότα όμως που μπορούν να οδηγήσουν σε μία τέτοια αναδιάταξη δεν είναι τόσο προφανή, άρα αξίζει να δείξουμε τον τρόπο που τέτοιες αναδιατάξεις μπορούν να συμβούν.

Καταρχάς, ας υποθέσουμε ότι ο μεταγλωττιστής έχει την τιμή του `gp` σε έναν καταχωρητή. Αν ο μεταγλωττιστής θέλει να πετάξει (spill) την τιμή αυτή για να γεννήσει τον κώδικα για τις γραμμές 10-12, μπορεί να γεννήσει τον κώδικα για τη γραμμή 12 *πριν* γεννήσει τον κώδικα για τις γραμμές 10-11 (υποθέτοντας φυσικά ότι οι τοποθεσίες μνήμες για το `p` και το `gp` δεν συμπέφτουν). Ο μεταγλωττιστής είναι ελεύθερος να πραγματοποιήσει ό,τι βελτιστοποιήσεις επιθυμεί, υπό την προϋπόθεση ότι αυτές δεν αλλοιώνουν τη

συμπεριφορά ενός μονονηματικού προγράμματος. Αυτός είναι ο λόγος που μία εντολή προς τον μεταγλωττιστή (π.χ. ένα φράγμα μεταγλώττισης – compiler barrier, ή η χρήση του **volatile** αν ενδείκνυται) είναι απαραίτητη για τη διατήρηση της επιθυμητής διάταξης.

Επίσης, ακόμη κι αν ο κώδικας που γεννήσει ο μεταγλωττιστής προβλέπει την εκτέλεση των παραπάνω εντολών με τη σειρά, ο επεξεργαστής μπορεί να αλλάξει τη συμπεριφορά του κώδικα με διάφορους τρόπους. Ακόμη κι αν ο επεξεργαστής εκτελέσει τις εντολές με τη σειρά, η τιμή του `gp` μπορεί να φύγει (flush) από τον ενταμιευτή αποθήκευσης (store buffer) πριν φύγουν οι τιμές των αναθέσεων 10-11 (π.χ. αν η γραμμή της κρυφής μνήμης στην οποία είναι αποθηκευμένο το `p` δεν ανήκει στον επεξεργαστή ο οποίος εκτελεί τις παραπάνω εντολές, ενώ η γραμμή που είναι αποθηκευμένο το `gp` του ανήκει). Ωστόσο, υπάρχει πληθώρα περιπτώσεων όπου ο επεξεργαστής εκτελεί εκτός σειράς τις παραπάνω εντολές έτσι κι αλλιώς, π.χ. σε superscalar αρχιτεκτονικές. Σε τέτοιες περιπτώσεις, οι λόγοι για τους οποίους αναδιατάσσονται οι παραπάνω εντολές γίνονται αμέσως προφανείς.

Ας υποθέσουμε τώρα ότι ένας αναγνώστης διαβάζει χωρίς κάποιο κλείδωμα την τιμή του `gp` και, αν δεν είναι `NULL`, διαβάζει - επίσης χωρίς κάποιο κλείδωμα- τις τιμές των πεδίων `a` και `b`. Στην περίπτωση αυτή, υποθέτοντας ότι οι εντολές των γραμμών 10-12 έχουν αναδιαταχθεί, ένας αναγνώστης μπορεί να δει μη-αρχικοποιημένες τιμές και για τα δύο πεδία του `gp`.

Για να αποφύγουμε προβλήματα όπως τα παραπάνω, πρέπει να χρησιμοποιήσουμε φράχτες μνήμης (memory barriers). Το RCU προσφέρει το `rcu_assign_pointer()` το οποίο είναι παρόμοιο με μία εντολή ανάθεσης, αλλά επίσης προσφέρει επιπλόν εγγυήσεις όσον αφορά στη διάταξη των εντολών. Πιο συγκεκριμένα, το `rcu_assign_pointer()` έχει σημασιολογία παρόμοια με την εντολή `memory_order_release` της C11, αλλά επιπλέον εμποδίζει "περίεργες" βελτιστοποιήσεις που μπορεί να κάνει ο μεταγλωττιστής. Με το `rcu_assign_pointer()`, οι τελευταίες τρεις γραμμές του Σχήματος 2.1 θα γίνουν ως εξής:

```
1  p->a = 42;
2  p->b = 42;
3  rcu_assign_pointer(gp, p);
```

Listing 2.2: Μηχανισμός Έκδοσης.

Στον παραπάνω κώδικα, η μακροεντολή `rcu_assign_pointer()` *εκδίδει* τη νέα δομή δεδομένων, αναγκάζοντας τόσο τον μεταγλωττιστή όσο και τον επεξεργαστή να πραγματοποιήσουν την ανάθεση στο `gp` μόνο *αφού* έχουν αρχικοποιηθεί τα πεδία της δομής δεδομένων.

Φυσικά, παρόμοια προβλήματα επηρεάζουν τους αγαγνώστες. Ας θεωρήσουμε, για παράδειγμα, τον ακόλουθο κώδικα:

```
1  p = gp;
2  if (p != NULL) {
3    /* do something with p->a, p->b */;
4  }
```

Listing 2.3: Αναδιάταξη εντολών κατά τη Συνδρομή.

Αν και με μια πρώτη ματιά το παραπάνω τμήμα κώδικα δεν φαίνεται να μπορεί να προκαλέσει προβλήματα, μια πιο προσεκτική ματιά αποκαλύπτει ότι αυτό δεν ισχύει. Βελτιστοποιήσεις εικασίας τιμής (value speculation optimizations) που πραγματοποιούνται από τον μεταγλωττιστή, καθώς και αναδιάταξη εξαρτώμενων αναγνώσεων (dependent loads reordering) που πραγματοποιούνται από μερικούς επεξεργαστές (όπως ο διαβόητος DEC Alpha), μπορεί να οδηγήσουν στην ανάγνωση των τιμών των πεδίων `a`

και b πριν την ανάγνωση του gp, ή στην ανάγνωση των τιμών p->a και p->b από διαφορετικές δομές δεδομένων. Ένας άλλος τρόπος που ο μεταγλωττιστής μπορεί να αλλοιώσει τον κώδικα παρουσιάζεται παρακάτω:

```
1  if (gp) {
2    /* do something with gp->a, gp->b */
3  }
```

Listing 2.4: Αναδιάταξη εντολών κατά τη Συνδρομή – βελτιστοποιημένος κώδικας.

Στο παραπάνω τμήμα κώδικα ο μεταγλωττιστής ξαναδιαβάζει από το gp και δεν χρησιμοποιεί το p καθόλου! Αυτό μπορεί να συμβεί αν ο μεταγλωττιστής δεν έχει ελεύθερους καταχωρητές. Και ενώ η ανάγνωση μόνο από το gp μπορεί να φαίνεται λογική επιλογή, αν αυτός ο κώδικας έτρεχε παράλληλα με έναν εγγραφέας ο οποίος αντικαθιστούσε την τρέχουσα έκδοση μιας δομής δεδομένων με μία νέα, ο αναγνώστης θα μπορούσε να είχε διαβάσει διαφορετική τιμή για τα πεδία a και b, αν οι τιμές που διαβάστηκαν προήλθαν από διαφορετικές δομές δεδομένων. Έτσι, για να αποφευχθούν τέτοιου είδους προβλήματα (και πολλά άλλα, π.χ. load tearing [Howe]), το RCU παρέχει το rcu_dereference():

```
1  rcu_read_lock();
2  p = rcu_dereference(gp);
3  if (p != NULL) {
4    /* do something with p->a, p->b */
5  }
6  rcu_read_unlock();
```

Listing 2.5: Μηχανισμός Συνδρομής.

Το rcu_dereference() ενσωματώνει όλους τους φράχτες μεταγλώττισης και τους φράχτες μνήμης που απαιτούνται για τον σκοπό του. Μπορεί να θεωρηθεί σαν μια *συνδρομή* στην τιμή ενός συγκεκριμένου δείκτη, και εγγυάται ότι επακόλουθες εντολές αποδεικτοποίησης θα δουν οποιαδήποτε αρχικοποίηση συνέβη πριν το rcu_assign_pointer() (έκδοση). Το rcu_dereference() έχει σημασιολογία παρόμοια με της ανάγνωσης memory_order_consume της C11, και χρησιμοποιεί τόσο το volatile όσο και φράχτες μνήμης (για τον DEC Alpha), έτσι ώστε να προσφέρεται η προαναφερθείσα εγγύηση.

Η χρήση των rcu_assign_pointer() και rcu_dereference() είναι απαραίτητη για προγράμματα που προσπελαύνουν δείκτες προστατευόμενους από το RCU. Επιπρόσθετα, αν ένας δείκτης προστατεύται από το RCU (επισημαίνεται με χρήση του __rcu), όλες οι αποδεικτοποιήσεις αυτού του δείκτη πρέπει να πραγματοποιηθούν μέσα σε κρίσιμα τμήματα ανάγνωσης RCU χρησιμοποιώντας το rcu_dereference(), και όλες οι αναθέσεις σε δείκτες προστατευόμενους από το RCU πρέπει να χρησιμοποιούν το rcu_assign_pointer(). Ένα εργαλείο που ονομάζεται sparse χρησιμοποιείται στον πυρήνα citeSparse για τη διαβεβαίωση (μεταξύ άλλων) της σωστής μεταχείρισης δεικτών που προστατεύονται από το RCU, και για την αποφυγή σφαλμάτων που οφείλονται στη χρήση του RCU. Τα κρίσιμα τμήματα ανάγνωσης RCU ορίζονται από τα rcu_read_lock() και rcu_read_unlock(). Οι μακροεντολές αυτές δεν φράσσουν (block) το νήμα που τις εκτελεί, ούτε ελέγχουν επανειλημμένα την ικανοποίηση κάποιας συνθήκης (spinning). Μάλιστα, σε μη-προεκχωρητικούς πυρήνες, στην ουσία δεν γεννούν καθόλου κώδικα.

Τέλος, για διευκόλυνση των προγραμματιστών, το RCU προσφέρει ένα ολόκληρο API για την ευκολότερη μεταχείριση λιστών (π.χ. διάσχιση λίστας, εισαγωγή στοιχείου σε λίστα, διαγραφή στοιχείου, αντικατάσταση στοιχείου, κλπ), τόσο για διπλά συνδεδεμένες λίστες, όσο και για κυκλικές λίστες, δομές δεδομένων που χρησιμοποιούνται κατά κόρον στον πυρήνα. Η παρουσίασή του API δεν θα γίνει εδώ, χάριν συντομίας, αλλά ενδιαφερόμενοι αναγνώστες μπορούν να βρουν περισσότερες πληροφορίες στις ακόλουθες πηγές: [McKe08b, McKe10b, McKe14].

2.2.2 Αναμονή Ολοκλήρωσης Προϋπαρχόντων Αναγνωστών

Στην προηγούμενη ενότητα, αναφέρθηκε ότι, για να αποδεσμευθεί με ασφάλεια μία δομή δεδομένων, το RCU πρέπει να περιμένει όλους τους προϋπάρχοντες αναγνώστες να ολοκληρώσουν το κρίσιμο τμήμα ανάγνωσής τους. Αλλά ο τρόπος που το κάνει αυτό το RCU δεν είναι σε καμία περίπτωση προφανής σε κάποιον που δεν είναι εξοικειωμένος με το RCU. Στην ενότητα αυτή θα συζητήσουμε τον τρόπο που το RCU ξέρει ότι όλοι οι προϋπάρχοντες αναγνώστες έχουν ολοκληρώσει τα κρίσιμα τμήματα ανάγνωσής τους, και θα παρουσιάσουμε μια στοιχειώδη υλοποίηση για τον μηχανισμό αυτόν.

Ουσιαστικά, το RCU είναι *ένας τρόπος να περιμένουμε για πράγματα να ολοκληρωθούν*. Το RCU μπορεί να θεωρηθεί ένας μαζικός μηχανισμός μετρήματος αναφορών. Τα πράγματα για τα οποία το RCU αναμένει είναι τα *κρίσιμα τμήματα ανάγνωσης RCU*, και το RCU μπορεί να περιμένει την ολοκλήρωση χιλιάδων κρίσιμων τμημάτων ανάγνωσης, χρησιμοποιώντας έναν πολύ απλό αλλά εξαιρετικά αποδοτικό τρόπο.

Τα κρίσιμα τμήματα ανάγνωσης RCU ξεκινούν με το `rcu_read_lock()` και τελειώνουν με το `rcu_read_unlock()`. Οι μακροεντολές αυτές μπορούν να εμφωλευθούν, και το RCU αντιμετωπίζει ένα σύνολο εμφωλευμένων εντολών σαν ένα μεγάλο κρίσιμο τμήμα ανάγνωσης RCU. Ωστόσο, οι μακροεντολές αυτές δεν επιτρέπεται να οδηγούν σε φραγμό (block/sleep) του νήματος που τις χρησιμοποιεί[1]. Στην ουσία, το RCU μπορεί να περιμένει για την ολοκλήρωση *οποιουδήποτε* τμήματος κώδικα, αρκεί αυτός ο κώδικας να βρίσκεται εντός ενός κρίσιμου τμήματος ανάγνωσης RCU.

Όπως αναφέρθηκε στην Ενότητα 2.1, η τυπική ακολουθία ενημερώσεων με χρήση του RCU μοιάζει με την παρακάτω:

1. Ενημέρωση μιας δομής δεδομένων (π.χ. αντικατάσταση ενός στοιχείου σε μία διπλά συνδεδεμένη λίστα με χρήση της `list_replace_rcu()`).

2. Αναμονή για την ολοκλήρωση όλων των προϋπαρχόντων κρίσιμων τμημάτων αναγνωστών RCU. Το παραπάνω μπορεί να επιτευχθεί με διάφορους τρόπους. Ένας τρόπος είναι η κλήση του `synchronize_rcu()` (θα συζητηθεί στη συνέχεια), το οποίο φράσσει το νήμα που το καλεί μέχρις ότου όλοι οι προϋπάρχοντες αναγνώστες να έχουν ολοκληρώσει το κρίσιμο τμήμα τους. Ωστόσο, όταν όλα τα προϋπάρχοντα κρίσιμα τμήματα ανάγνωσης RCU έχουν ολοκληρωθεί, δεν υπάρχει κανένας τρόπος ένας αναγνώστης να αποκτήσει μια αναφορά στο στοιχείο που αφαιρέθηκε.

3. Ανάκτηση, π.χ. αποδέσμευση της μνήμης που καταλαμβάνει το στοιχείο που αντικαταστάθηκε.

Στο παραπάνω παράδειγμα, υπάρχουν δύο πράγματα που χρήζουν της προσοχής μας. Πρώτον, τα κρίσιμα τμήματα ανάγνωσης RCU που ξεκινούν *μετά* τη `synchronize_rcu()` μπορούν να ολοκληρωθούν αφού η συνάρτηση αυτή έχει επιστρέψει. Δεύτερον, κατά την αναμονή της ολοκλήρωσης του `synchronize_rcu()`, μπορεί να υπάρχουν αναγνώστες που βλέπουν τόσο το παλίο, όσο και το νέο στοιχείο στη συγκεκριμένη θέση της λίστας. Το γεγονός αυτό θα συζητηθεί περαιτέρω στην επόμενη ενότητα.

Η τυπική ακολουθία ενημερώσεων που περιγράφηκε παραπάνω γίνεται πιο εύκολα κατανοητή με το παράδειγμα του Κώδικα 2.6. Στο παράδειγμα αυτό, οι τελευταίες τρεις γραμές αντιπροσωπεύουν την τυπική ακολουθία ενημερώσεων. Ο εγγραφέας ψάχνει στη λίστα για ένα συγκεκριμένο στοιχείο και, αν το στοιχείο βρεθεί, *αντιγράφει* το στοιχείο (ενώ μπορεί να υπάρχουν *αναγνώστες* που τρέχουν ταυτόχρονα) κι έπειτα εκτελεί μια *ενημέρωση*.

[1] Μία ειδική έκδοση του RCU η οποία ονομάζεται Sleep RCU (SRCU) επιτρέπει σε ένα νήμα να "κοιμάται" κατά τη διάρκεια ενός κρίσιμου τμήματος ανάγνωσης RCU [McKe06].

```
1  struct foo {
2    struct list_head list;
3    int a;
4    int b;
5  };
6  LIST_HEAD(head);
7
8  /* ... */
9
10 p = list_search(head, key);
11 if (p == NULL) {
12   /* ... */
13 }
14 q = kmalloc(sizeof(*p), GFP_KERNEL);
15 *q = *p;
16 q->a = 42;
17 q->b = 42;
18 list_replace_rcu(&p->list, &q->list); /* Modifies an element in the list */
19 synchronize_rcu();                    /* Waits for pre-existing readers */
20 kfree(p);                             /* Frees the replaced element */
```

Listing 2.6: Παράδειγμα τυπικής ακολουθίας ενημερώσεων με χρήση του RCU.

Γενικά, το μόνο θέμα που δεν συζητήθηκε ως τώρα είναι ο τρόπος που το RCU γνωρίζει ότι όλα τα προϋπάρχοντα τμήματα ανάγνωσης έχουν ολοκληρωθεί. Και το γεγονός ότι τα `rcu_read_lock()` και `rcu_read_unlock()` δεν γεννούν καθόλου κώδικα σε μη-προεκχωρητικούς πυρήνες είναι σίγουρα περίεργο. Εξάλλου, πώς είναι δυνατόν να έχουμε συγχρονισμό αν δεν αλλάζει η κατάσταση του συστήματος;

Το κλειδί εδώ είναι ότι τα κρίσιμα τμήματα ανάγνωσης RCU δεν επιτρέπεται να περιέχουν κάποιου είδους φραγμό[2] (block/sleep). Έτσι, όταν ένας επεξεργαστής πραγματοποιήσει, λόγου χάρη, μία εναλλαγή περιβάλλοντος λειτουργίας (context switch), οποιοδήποτε κρίσιμο τμήμα ανάγνωσης RCU είχε προηγηθεί θα έχει ολοκληρωθεί. Συνεπώς, όταν κάθε επεξεργαστής έχει πραγματοποιήσει τουλάχιστον μία εναλλαγή περιβάλλοντος λειτουργίας, είναι εγγυημένο ότι όλοι οι προϋπάρχοντες αναγνώστες θα έχουν ολοκληρώσει το κρίσιμο τμήμα ανάγνωσης RCU τους, και η `synchronize_rcu()` μπορεί να επιστρέψει με ασφάλεια. Μία οπτική παρουσίαση των παραπάνω παρέχεται στο Σχήμα 2.2. Αν και υπάρχουν αρκετά περισσότερα πράγματα πίσω από τον μηχανισμό αυτόν, τα παραπάνω είναι αρκετά για μια βασική κατανόησή του.

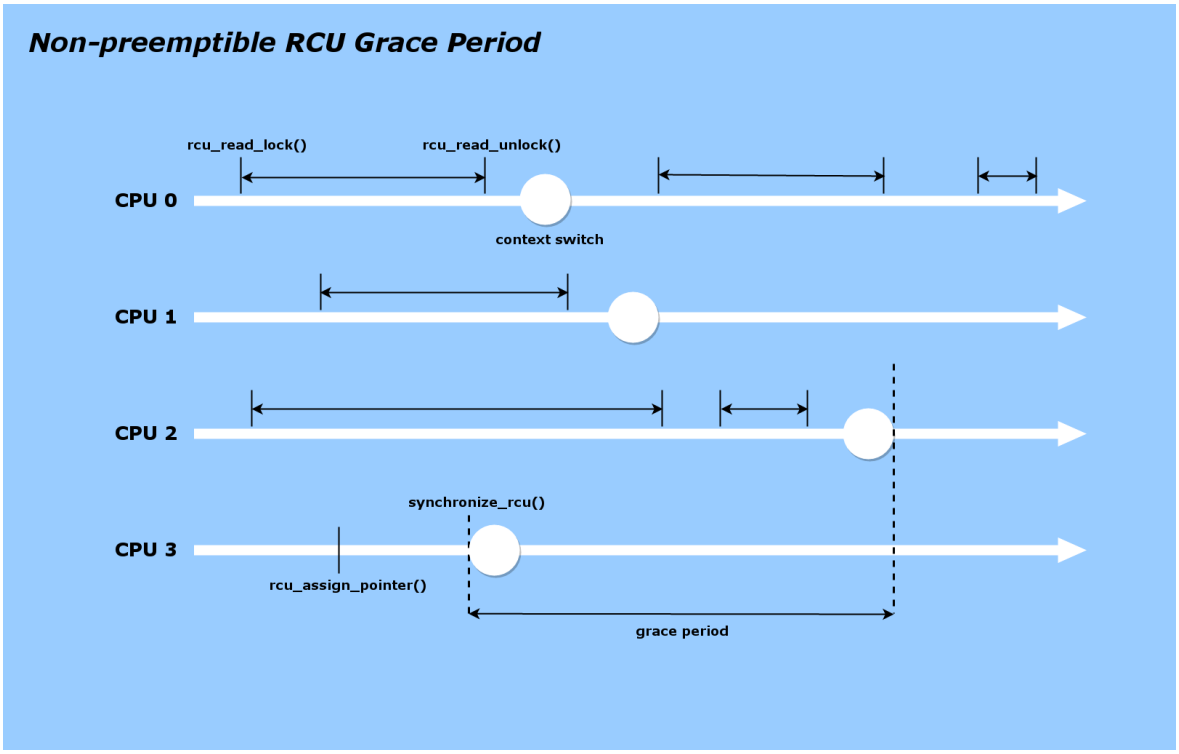
Στη συνέχεια παρέχονται κάποιοι ορισμοί, έτσι ώστε να παρουσιασθούν ακριβέστερα οι έννοιες που περιγράφηκαν προηγουμένως.

Definition 2.1. Οποιαδήποτε εντολή δεν βρίσκεται εντός ενός κρίσιμου τμήματος ανάγνωσης RCU λέγεται ότι είναι σε *κατάσταση ηρεμίας*.

Τέτοιες εντολές δεν επιτρέπεται να έχουν αναφορές σε δομές δεδομένων που προστατεύονται από το RCU (αυτό ελέγχεται στον πυρήνα με το `sparse`). Αξίζει να σημειωθεί ότι διαφορετικές εκδόσεις του RCU, έχουν διαφορετικά σύνολα καταστάσεων ηρεμίας. Για παράδειγμα, για το Tree RCU (την κύρια έκδοση του RCU που χρησιμοποιείται στον πυρήνα, βλ. Ενότητα 4.1), οι καταστάσεις ηρεμίας περιλαμβάνουν:

- Την εναλλαγή περιβάλλοντος λειτουργίας,
- Την λειτουργία αναμονής – idle (είτε είναι το idle loop, είτε η λειτουργία `dynticks-idle`),

[2] Αυτό ισχύει τόσο για το Tree RCU όσο και για το (παρωχημένο πλέον) Classic RCU. Υπάρχουν άλλες εκδόσεις που επιτρέπουν τον φραγμό εντός κρίσιμων τμημάτων ανάγνωσης ([McKe06, McKe07a]).



Σχήμα 2.2: Αναμονή για την ολοκλήρωση όλων των προϋπαρχόντων αναγνωστών RCU (σχήμα προσαρμοσμένο από [McKe13]).

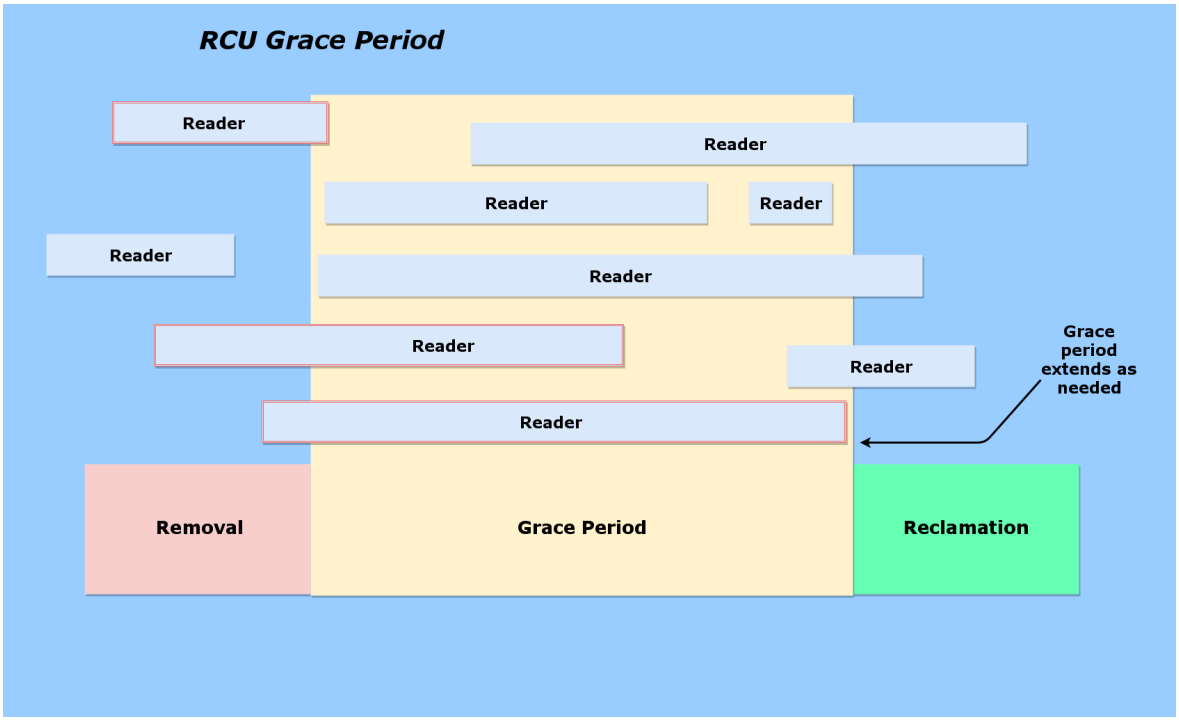
- Την εκτέλεση κώδικα χώρου χρήστη

ενώ για το Tree RCU-bh (μία έκδοση που έχει σχεδιασθεί για να αντέχει επιθέσεις DDoS) οι καταστάσεις ηρεμίας στην ουσία είναι οποιοσδήποτε κώδικας εκτός softirq με τις διακοπές ενεργοποιημένες. Φυσικά, ο μηχανισμός για την αναμονή των προϋπαρχόντων αναγνωστών δεν απαιτείται να περιμένει για επεξεργαστές που βρίσκονται σε κατάσταση ηρεμίας. Πρέπει να περιμένει μόνο για προϋπάρχοντες αναγνώστες.

Definition 2.2. *Οποιαδήποτε* χρονική περιόδος κατά την οποία κάθε επεξεργαστής περάσει τουλάχιστον μία φορά από κατάσταση ηρεμίας λέγεται *περίοδος χάριτος*.

Συνεπώς, αν ένα κρίσιμο τμήμα ανάγνωσης RCU είχε ξεκινήσει πριν την έναρξη μιας συγκεκριμένης περιόδου χάριτος, τότε το τμήμα αυτό πρέπει να ολοκληρωθεί πριν τη λήξη αυτής της περιόδου χάριτος. Αυτό σημαίνει ότι ο μηχανισμός για την αναμονή των προϋπαρχόντων αναγνωστών πρέπει να περιμένει να παρέλθει *τουλάχιστον* μία περίοδος χάριτος. Άπαξ και παρέλθει μία περίοδος χάριτος, δεν γίνεται να υπάρχουν αναγνώστες με αναφορές στην παλιά έκδοση μιας ενημερωμένης δομής δεδομένων (αφού κάθε επεξεργαστής πέρασε από κατάσταση ηρεμίας) και η φάση της ανάκτησης μπορεί να ξεκινήσει.

Λαμβάνοντας υπ' όψιν τα παραπάνω, μια στοιχειώδης υλοποίηση του `synchronize_rcu()` είναι η ακόλουθη: `for_each_online_cpu(cpu) { run_on(cpu);}`. Στον κώδικα αυτόν, το `run_on()` απλά εκτελεί το τρέχον νήμα στον επιθυμητό επεξεργαστή (φυσικά, αυτό οδηγεί σε μια εναλλαγή περιβάλλοντος λειτουργίας στον συγκεκριμένο επεξεργαστή). Όπως φαίνεται, η παραπάνω δομή επανάληψης εξαναγκάζει μια εναλλαγή περιβάλλοντος λειτουργίας σε όλους τους επεξεργαστές, κάτι που εγγυάται ότι όλα τα προϋπάρχοντα κρίσιμα τμήματα ανάγνωσης RCU θα έχουν ολοκληρωθεί. Φυσικά, υλοποίηση αυτή δεν θα δούλευε σε πυρήνες που χρησιμοποιούν προεκχωρητικότητα πραγματικού χρόνου (real-time preemption). Όπως αναμένεται, η υλοποίηση του `synchronize_rcu()` στο Tree RCU (την κύρια έκδοση



Σχήμα 2.3: Η φάση της ανάκτησης ξεκινά αφού παρέλθει μία περίοδος χάριτος (σχήμα προσαρμοσμένο από [McKe07c].

του πυρήνα) δεν μοιάζει καθόλου με αυτή τη στοιχειώδη υλοποίηση. Η πραγματική υλοποίηση του πυρήνα είναι πολύ πιο πολύπλοκη, αφού απαιτείται να χειρίζεται πολλά διαφορετικά γεγονότα (συμπεριλαμβανομένων των διακοπών, των Μη Απενεργοποιήσιμων Διακοπών – Non Maskable Interrupts, την αφαίρεση επεξεργαστών εν ώρα λειτουργίας – CPU hotplugs, την πλήρη λειτουργία `dynticks-idle`, κλπ), και διαφέρει ανάλογα με την έκδοση του RCU, την έκδοση του πυρήνα, κ.ο.κ. Σε κάθε περίπτωση όμως, η στοιχειώδης αυτή υλοποίηση του `synchronize_rcu()` είναι πολύ χρήσιμη, καθώς η βασική ιδέα πίσω από τον μηχανισμό αυτό παραμένει η ίδια, και είναι αυτή που παρουσιάστηκε στις προηγούμενες παραγράφους. Τέλος, αξίζει να σημειωθεί ότι ο μηχανισμός αναμονής των προϋπαρχόντων αναγνωστών προσφέρει επίσης μία συνάρτηση που λειτουργεί ασύγχρονα, η οποία ονομάζεται `call_rcu()`. Η συνάρτηση αυτή καταχωρίζει μία επανάκληση συνάρτησης η οποία θα εκτελεστεί αφού όλοι οι προϋπάρχοντες αναγνώστες έχουν ολοκληρώσει το κρίσιμο τμήμα ανάγνωσης RCU τους, δηλαδή, αφού τουλάχιστον μία περίοδος χάριτος έχει παρέλθει.

2.2.3 Διατήρηση Πολλαπλών Αντιγράφων Πρόσφατα Ενημερωμένων Αντικειμένων

Ο τρίτος και τελευταίος θεμελειώδης μηχανισμός του RCU που θα περιγράψουμε, απαντά σε ερωτήματα που αφορούν στο τί βλέπουν οι αναγνώστες όταν διασχίζουν μία λίστα που ενημερώνεται ταυτόχρονα. Παρ' όλο που οι βασικές ιδέες μπορεί να είναι ήδη προφανείς από την Ενότητα 2.1, δεν είναι πάντα προφανές πώς ένα στοιχείο στο οποίο αναφέρεται ένας ανγνώστης θα παραμείνει ανέπαφο ενώ ένας εγγραφεάς ταυτόχρονα το τροποποιεί. Θα παρουσιάσουμε και θα αναλύσουμε παραδείγματα με την ελπίδα ότι μετά το πέρας αυτού του κεφαλαίου, θα υπάρχει μια επαρκής κατανόηση του τρόπου που λειτουργεί το RCU, τουλάχιστον διαισθητικά.

Πρώτα απ' όλα, ας θεωρήσουμε την τυπική ακολουθία ενημερώσεων με χρήση του RCU, όπως παρουσιάστηκε στην Ενότητα 2.2.2. Το παράδειγμα αυτό μπορεί να τροπο-

ποιηθεί για να δείξει την διαγραφή ενός στοιχείου μιας λίστα, με μικρές μόνο αλλαγές και τη χρήση της `list_del_rcu()`. Ενώ το παράδειγμα αυτό μπορεί να μοιάζει απλό με μια πρώτη ματιά, υπάρχουν κάποιες λεπτομέρειες που είναι εύκολο να αγνοηθούν.

Γενικά, πρέπει να είναι προφανές ότι υπάρχουν δύο εκδόσεις ενός στοχείου ορατές στους ταυτόχρονους αναγνώστες. Για παράδειγμα, ένας αναγνώστης μπορεί να διαβάσει την τιμή ενός δείκτη στο στοιχείο και μετά να εξυπηρετήσει μια διακοπή. Στην περίπτωση αυτή, αν ένας εγγραφέας αντιγράψει και ενημερώσει το στοιχείο, ο αναγνώστης μπορεί να βλέπει την παλιά έκδοση της λίστας για αρκετό χρονικό διάστημα μετά την αφαίρεση του στοιχείου. Έτσι, σε μία τέτοια περίπτωση έχουμε δύο εκδόσεις της λίστας.

Αυτό που δεν είναι τόσο προφανές όμως είναι *πώς* η `list_replace_rcu()` αντικαθιστά ένα στοιχείο, χωρίς να ενοχλεί τους ταυτόχρονους αναγνώστες. Στην περίπτωση μιας διπλά συνδεδεμένης λίστας, η `list_replace_rcu()` παρουσιάζεται παρακάτω (έκδοση v3.19 του πυρήνα):

```
1  static inline void list_replace_rcu(struct list_head *old, struct list_head *new)
2  {
3          new->next = old->next;
4          new->prev = old->prev;
5          rcu_assign_pointer(list_next_rcu(new->prev), new);
6          new->next->prev = new;
7          old->prev = LIST_POISON2;
8  }
```

Listing 2.7: Ορισμός για την `list_replace_rcu()`.

όπου το `LIST_POISON2` είναι ένας δείκτης διάφορος του `NULL` που θα οδηγήσει σε σφάλμα σελιδοποίησης (page fault) υπό κανονικές συνθήκες. Οι γραμμές 4-5 απλά θέτουν τους δείκτες `->prev` και `->next` του νεοεισαχθέντος στοιχείου στις κατάλληλες τιμές. Η γραμμή 6 θέτει τον δείκτη `->next` του προηγηθέντος του νεοεισαχθέντος στοιχείου να δείχνει στο νεοεισαχθέν στοιχείο. Η γραμμή 7 θέτει τον δείκτη `->prev` του διαδόχου του νεοεισαχθέντος στοιχείου να δείχνει στο νεοεισαχθέν στοιχείο, και η γραμμή 8 "δηλητηριάζει" τον δείκτη `->prev` του στοιχείου που αντικαταστάθηκε. Τι θα συμβεί όμως αν υπήρχαν αναγνώστες που διέσχιζαν τη λίστα τόσο προς τα εμπρός όσο και προς τα πίσω; Δεν θα προκαλούσε η τελευταία γραμμή ένα σφάλμα κατάτμησης (segmentation fault);

Η απάντηση στο ερώτημα αυτό είναι ότι οι αναγνώστες που διασχίζουν μία προστατευόμενη από το RCU λίστα πρέπει να χρησιμοποιούν συναρτήσεις που παρέχονται από το RCU για τον σκοπό αυτό (π.χ. `list_for_each_entry_rcu()`, κλπ). Οι συναρτήσεις αυτές διασχίζουν τη λίστα με μία συγκεκριμένη φορά (προς τα εμπρός μόνο) και αποφεύγουν τέτοιου είδους ζητήματα. Ουσιαστικά, ο "δηλητηριασμός" του δείκτη παραπάνω βοηθά στον εντοπισμό αναγνωστών που διασχίζουν τη λίστα με αντίθετη φορά. Κι επειδή το RCU επιτρέπει πάνω από μία εκδόσεις της λίστας να είναι ενεργές κάθε στιγμή και οι εγγραφείς τρέχουν ταυτόχρονα με τους αναγνώστες, δεν υπάρχει ιδιαίτερη χρησιμότητα να προσπαθήσουμε να κρατήσουμε τη λίστα συνεπή για αναγνώστες που διασχίζουν τη λίστα και προς τις δύο κατευθύνσεις (π.χ. με χρήση ατομικών ενημερώσεων δύο δεικτών – atomic two-pointer updates). Ωστόσο, αν κάποτε υπάρξει ανάγκη για διάσχιση λιστών προστατευόμενων από το RCU και προς τις δύο κατευθύνσεις, η υλοποίηση του RCU θα μπορούσε να αλλάξει εύκολα έτσι ώστε να εξυπηρετήσει την ανάγκη αυτή.

2.3 Συμπεράσματα που Αντλούνται από τη Χρήση του RCU

Μετά από την εισαγώγη στο RCU και την παρουσίαση των εσωτερικών μηχανισμών του, θα συζητηθούν ορισμένα πλεονεκτήματα και μειονεκτήματα που προκύπτουν από τη χρήση του. Και παρ' όλο που ορισμένα από αυτά μπορεί να είναι ήδη προφανή και να μην εισάγουν κάτι νέο, ο τρόπος με τον οποίο κάποιος σκέφτεται για το RCU και τους μηχανισμούς που προσφέρει είναι κρίσιμος για αυτά που θα πραγματοποιήσουμε στα επόμενα κεφάλαια. Εξάλλου, το RCU είναι πολλά παραπάνω από ένας απλός μηχανισμός συγχρονισμού, και αυτό θα γίνει εμφανές στη συνέχεια.

2.3.1 Μερικές Παρατηρήσεις

Όπως παρατηρήθηκε στην Ενότητα 2.2.2, το RCU είναι ένας τρόπος να περιμένουμε για πράγματα να ολοκληρωθούν [McKe08c]. Άρα, το RCU μπορεί να θεωρηθεί ένας μηχανισμός μετρήματος αναφορών. Ας θεωρήσουμε τον Κώδικα 2.8. Εδώ, η `synchronize_rcu()` αναμένει την απελευθέρωση όλων των προηγουμένως αποκτημένων αναφορών στο `p`, αφού όλες αυτές οι αναφορές γίνονται μέσα σε κρίσιμα τμήματα ανάγνωσης RCU. Δεν γίνεται να υπάρξουν νέες αναφορές στο `p`, λόγω της ανάθεσης στο `head`. Όταν κανένα νήμα δεν έχει αναφορά στο `p` πια, ο εγγραφέας μπορεί να το διαγράψει. Φυσικά, αυτός ο τρόπος μέτρησης αναφορών είναι κάπως περιορισμένος αφού τα κρίσιμα τμήματα ανάγνωσης RCU δεν επιτρέπουν τον φραγμό του νήματος. Επιπρόθετα, οι αναφορές δεν μπορούν να δοθούν από ένα νήμα σε ένα άλλο.

```
1  spin_lock(&lock);
2  p = head;
3  rcu_assign_pointer(head, NULL);
4  spin_unlock(&lock);
5  synchronize_rcu();
6  kfree(p);
```

Listing 2.8: Το RCU είναι ένας μηχανισμός μετρήματος αναφορών.

Ωστόσο, συγκριτικά με παραδοσιακούς μηχανισμούς μετρήματος αναφορών, το RCU δεν σχετίζεται με μία συγκεκριμένη δομή δεδομένων ή με μία συγκεκριμένη ομάδα δομών, κι έτσι δεν χρειάζεται να διατηρεί έναν καθολική μετρητή αναφορών για την εκάστοτε δομή/ομάδα. Σαν αποτέλεσμα, το RCU μπορεί να θεωρηθεί ένας μαζικός μηχανισμός μετρήματος αναφορών με πολύ μικρή επιβάρυνση, λόγω των εξαιρετικά αποδοτικών και κλιμακώσιμων αρχέγονων συνάρτησεων ανάγνωσης που προσφέρει. Έτσι, όπου ένας παραδοσιακός μηχανισμός συγχρονισμού θα υπέφερε από χειροτέρευση της απόδοσης όταν οι αναγνώσεις για μία συγκεκριμένη ομάδα δομών δεδομένων θα αυξανόταν, το RCU εγγυάται ότι η επιβάρυνση από πλευράς αναγνωστών θα είναι ελάχιστη. Και αυτός είναι ο κύριος λόγος για τον οποίο χρησιμοποιείται το RCU έναντι παραδοσιακών μηχανισμών μετρήματος αναφορών, όπου φυσικά οι συνθήκες το επιτρέπουν (π.χ. αν δεν υπάρχει ανάγκη για πέρασμα μιας αναφοράς από ένα νήμα σε ένα άλλο).

Όμως, αφού οι μηχανισμοί μετρήματος αναφορών χρησιμοποιούνται για την κατασκευή συλλεκτών σκουπιδιών, και το RCU μπορεί να θεωρηθεί ένας μηχανισμός μετρήματος αναφορών, μπορεί το RCU να θεωρηθεί και συλλέκτης σκουπιδιών; Το RCU, φυσικά μοιάζει με έναν συλλέκτη σκουπιδιών, όμως υπάρχουν κάποιες μικρές αλλά σημαντικές διαφορές: ο προγραμματιστής πρέπει να ορίσει το τμήμα μέσα στο οποίο μπορούν να κρατούνται αναφορές (δηλαδή τα κρίσιμα τμήματα ανάγνωσης RCU), καθώς και να ορίσει πότε μία δομή δεδομένων μπορεί να συλλεχθεί. Αυτό σημαίνει ότι, παρ' όλο που υπάρχουν αρκετές ομοιότητες, ένας συλλέκτης σκουπιδιών βασισμένος στο RCU δεν θα

ήταν αυτόματος. Εκτός αυτού όμως, το RCU μπορεί να θεωρηθεί συλλέκτης σκουπιδιών, και μπορεί να παρέχει εγγυήσεις είτε για το ότι δεν υπάρχουν αναφορές προς ένα αντικείμενο πλέον, είτε για το ότι ένα αντικείμενο υπάρχει ακόμα (για τη διάρκεια ενός κρίσιμου τμήματος ανάγνωσης RCU).

Τέλος, θα παρουσιάσουμε τις αλληλεπιδράσεις μεταξύ των αναγνωστών και των εγγραφέων, καθώς και τους περιορισμούς στη διάταξη που επιβάλλει το RCU. Το πρώτο σημείο που πρέπει να ληφθεί υπ' όψιν είναι ότι οι αναγνώστες αλληλεπιδρούν με τους εγγραφείς μόνο μέσω του API της περιόδου χάριτος. Αυτό σημαίνει ότι οι αναγνώστες δεν μπορούν να εμποδίσουν έναν εγγραφέα από το να τροποποιήσει μία δομή δεδομένων. Αλλά μπορούν να εμποδίσουν μία περίοδο χάριτος από το να ολοκληρωθεί. Εξάλλου, έχει ήδη επισημανθεί ότι οι εγγραφείς τρέχουν ταυτόχρονα με τους αναγνώστες. Ένα άλλο σημείο που πρέπει να ληφθεί υπ' όψιν είναι ότι οι αρχέγονες συναρτήσεις των κρίσιμων τμημάτων ανάγνωσης RCU δεν παρέχουν καμία απολύτως εγγύηση όσον αφορά στην διάταξη των εντολών. Είδαμε ότι σε πολλές περιπτώσεις δεν γεννούν καθόλου κώδικα. Αυτό όμως σημαίνει ότι οποιοιδήποτε περιορισμοί εντός ενός κρίσιμου τμήματος ανάγνωσης RCU πρέπει να επιβάλλονται από τον προγραμματιστή. Αλλιώς, ο κώδικας είναι επιρρεπής τόσο σε βελτιστοποιήσεις του μεταγλωττιστή, όσο και σε πιθανές δυσμενείς επιπτώσεις που μπορεί να έχει το εφαρμοζόμενο μοντέλο μνήμης.

2.3.2 Σύγκριση με Άλλους Μηχανισμούς Συγχρονισμού

Υπάρχουν πολλοί τρόποι για την αναμονή της ολοκλήρωσης πραγμάτων. Ήδη αναφέραμε ότι το RCU προσομοιάζει έναν μηχανισμό μετρήματος αναφορών. Και υπάρχουν κι άλλοι τρόπο όπως κλειδώματα αναγνωστών-εγγραφέων, κατακερματισμένα κλειδώματα (hashed locks), γεγονότα (events), hazard pointers, κλπ. Ωστόσο, δεν είναι πάντα προφανές πότε μπορεί να χρησιμοποιηθεί το RCU, ούτε ποια είναι τα πλεονεκτήματα και οι περιορισμοί, ειδικά απέναντι σε παραδοσιακούς μηχανισμούς συγχρονισμού. Έτσι, η ενότητα αυτή στοχεύει στο ξεκαθάρισμα των παραπάνω ζητημάτων.

Μπορεί λοιπόν το RCU να χρησιμοποιηθεί σε μέρη όπου χρησιμοποιόντουσαν άλλοι μηχανισμοί συγχρονισμού; Οι περισσότεροι παραδοσιακοί μηχανισμοί συγχρονισμού χρησιμοποιούν κάποιου είδους αμοιβαίο αποκλεισμό μεταξύ αναγνωστών και εγγραφέων, ή επιτρέπουν ταυτόχρονες αναγνώσεις, αλλά όχι ταυτόχρονες εγγραφές και αναγνώσεις. Φυσικά, υπάρχουν μηχανισμοί συγχρονισμού που επιτρέπουν σε αναγνώστες και εγγραφείς να λειτουργούν ταυτόχρονα πάνω σε ένα αντικείμενο (π.χ. ακολουθιακά κλειδώματα – seqlocks), αλλά οι μηχανισμοί αυτοί αναγκάζουν τους εγγραφείς να επαναλάβουν την ενέργεια που πραγματοποίησαν κάθε φορά που συμβαίνει αυτό. Ωστόσο, σε περιπτώσεις που επιτρέπονται ταυτόχρονες αναγνώσεις (για παράδειγμα, στα κλειδώματα αναγνωστών-εγγραφέων), θεωρητικά είναι δυνατό να χρησιμοποιηθεί το RCU. Τα μόνα ερωτήματα που μένει να απαντηθούν είναι *γιατί να το κάνουμε αυτό* και *τι συνέπειες* έχει η χρήση του RCU.

Τελικά, αυτό που κάνει το RCU να ξεχωρίζει είναι τα πλεονεκτήματά του από άποψη κλιμακωσιμότητας και αποδοτικότητας, δηλαδή *η απόδοσή του*. Το RCU δεν εκτελεί τις ενημερώσεις πάνω στα ίδια τα δεδομένα, και εκμεταλλεύεται το γεγονός ότι οι μοντέρνοι επεξεργαστές εγγυώνται ότι οι εγγραφές σε μονούς ευθυγραμμισμένους δείκτες (single aligned pointers) είναι ατομικές, κάτι που επιτρέπει την τροποποίηση δομών δεδομένων χωρίς να ενοχληθούν ταυτόχρονοι αναγνώστες. Με τον τρόπο αυτό, οι αναγνώστες αποφεύγουν το ομολογουμένως μεγάλο κόστος των φραχτών μνήμης, των ατομικών εντολών και των αποτυχιών κρυφής μνήμης, και συνεχίζουν να προσπελαύνουν μία παλιά έκδοση μιας δομής δεδομένων (αν φυσικά αυτό είναι αποδεκτό αλγοριθμικά). Το RCU εξαιρετικά κλιμακώσιμο σε περιπτώσεις που τα κρίσιμα τμήματα ανάγνωσης είναι μικρά (και άρα εισάγουν μικρή επιβάρυνση), αφού οι αναγνώστες δεν ανταγωνίζονται για κάποιο κλείδωμα.

Επιπρόσθετα, τα κρίσιμα τμήματα ανάγνωσης RCU έχουν ανοσία σε αδιέξοδα (deadlocks) αφού οι αρχέγονες συναρτήσεις τους δεν φράσσουν (block) το νήμα που τις καλεί, δεν περιμένουν την ικανοποίηση κάποιας συνθήκης (spinning), ούτε κάνουν κάποιο άλμα προς τα πίσω οποιουδήποτε είδους. Αυτό ακριβώς το γεγονός σημαίνει ότι οι αρχέγονες συναρτήσεις των κρίσιμων τμημάτων ανάγνωσης παρέχουν καθυστέρηση πραγματικού χρόνου (real-time latency) και έχουν ντετερμινιστικό χρόνο εκτέλεση, κάτι που καθιστά το RCU πιο κλιμακώσιμο από άλλους μηχανισμούς συγχρονισμού.

Τέλος, το γεγονός ότι οι αναγνώστες και οι εγγραφείς ενεργούν ταυτόχρονα με το RCU σημαίνει ότι οι αναγνώστες μπορούν να δουν νέες εκδόσεις των δεδομένων σύντομα. Αντίθετα, αν χρησιμοποιούνταν κάποιος διαφορετικός μηχανισμός συγχρονισμού που επέτρεπε μόνο ταυτόχρονες αναγνώσεις, ένας εγγραφέας δεν θα μπορούσε να έχει ενημερώσει την τιμή μέχρις ότου όλοι οι προϋπάρχοντες αναγνώστες να είχαν τελειώσει το κρίσιμο τμήμα τους, και αναγνώστες οι οποίοι θα προσπαθούσαν να διαβάσουν την τιμή αφού ένας εγγραφέας είχε ήδη προσπαθήσει να πάρει το κλείδωμα θα αποτύγχαναν, μιας και θα έπρεπε να περιμένουν μέχρις ότου ο εγγραφέας να έχει τελειώσει το κρίσιμο τμήμα του. Ωστόσο, οι επακόλουθοι αυτοί αναγνώστες στην περίπτωση ενός μηχανισμού που χρησιμοποιεί κλειδώματα αναγνωστών-εγγραφέων είναι εγγυημένο ότι θα δουν τη νέα τιμή, ενώ στην περίπτωση του RCU οι αναγνώστες δεν είναι σίγουρο ότι θα δουν την καινούργια τιμή. Αξίζει να σημειωθεί εδώ ότι ακόμη και προϋπάρχοντες αναγνώστες μπορεί να δουν τη νέα τιμή, αν το κρίσιμο τμήμα τους είναι μεγάλο, και πάρουν μια αναφορά στα δεδομένα αφού η τιμή έχει ενημερωθεί. Από την άλλη όμως, μόνο οι αναγνώστες που ξεκινούν το κρίσιμο τμήμα ανάγνωσης RCU τους μετά τον εγγραφέα είναι εγγυημένο ότι θα δουν τη νέα τιμή.

Στο σημείο αυτό είναι προφανές ότι η χρήση του RCU έχει ένα τίμημα: θα υπάρχουν εσωτερικές ασυνέπειες και οι αναγνώστες μπορεί να διαβάσουν μη-ενημερωμένα δεδομένα [Arca03, McKe04]. Αντίθετα, άλλοι μηχανισμοί συγχρονισμού όπως τα κλειδώματα αναγνωστών-εγγραφέων εγγυώνται εσωτερική συνέπεια (αν και σε αυτές τις περιπτώσεις μπορεί να υπάρχει αυξημένη ασυνέπεια με τον έξω κόσμο). Επιπρόσθετα, στο RCU οι αναγνώστες εμποδίζουν τους εγγραφείς με την έννοια ότι οι εγγραφείς πρέπει να περιμένουν την ολοκλήρωση των προϋπαρχόντων αναγνωστών. Αυτό σημαίνει ότι είναι δυνατό για μία διεργασία χαμηλής προτεραιότητας να εμποδίζει μία διεργασία υψηλής προτεραιότητας για μεγάλο χρονικό διάστημα, δοθέντος ότι οι περίοδοι χάριτος κρατούν ορισμένα χιλιοστά του δευτερολέπτου. Για να αποφευχθεί αυτό, οι εγγραφείς πρέπει να χρησιμοποιήσουν ώθηση προτεραιότητας RCU (RCU priority boosting), ή ασύγχρονες συναρτήσεις όπως η `call_rcu()`, οι οποίες δεν φράσσουν το νήμα που τις καλεί.

Όπως φαίνεται, το RCU ταιριάζει καλύτερα σε περιπτώσεις που έχουμε φορτία εργασίας που περιλαμβάνουν κυρίως αναγνώσεις. Όμως το ερώτημα παραμένει: πότε είναι οι συνθήκες ιδανικές για χρήση του RCU; Το ερώτημα αυτό θα απαντηθεί στις επόμενες παραγράφους.

Περιοχές που Εφαρμόζεται το RCU

Το RCU εφαρμόζεται καλύτερα σε φορτία εργασίας που περιλαμβάνουν κυρίως αναγνώσεις, και η ανάγνωση δεδομένων που είναι ασυνεπή είναι αποδεκτή. Ένα κλασικό παράδειγμα στον πυρήνα του Linux είναι ο δικτυακός πίνακας δρομολόγησης (networking routing table). Εφ' όσον οι ενημερώσεις χρειάζονται ένα σημαντικό χρονικό διάστημα να φτάσουν σε ένα δοθέν σύστημα (δευτερόλεπτα ή ακόμη και λεπτά, αφού πρέπει να διαδοθούν στο Διαδίκτυο), το σύστημα θα στέλνει πακέτα μέσω μιας λάθος διαδρομής για αρκετό χρονικό διάστημα. Οπότε μερικά επιπλέον χιλιοστά του δευτερολέπτου συνήθως δεν αποτελούν πρόβλημα. Και έχουμε ήδη σημειώσει ότι οι αναγνώστες που χρησιμοποιούν RCU μπορεί να δουν τις ενημερωμένες τιμές γρηγορότερα απ' ό,τι οι αναγνώστες που χρησιμοποιούν κλειδώματα αναγνωστών-εγγραφέων.

Αν το φορτίο εργασίας περιλαμβάνει κυρίως αναγνώσεις, ή αναγνώσεις και εγγραφές, αλλά χρειάζονται συνεπή δεδομένα, το RCU εφαρμόζεται, αλλά πρέπει να χρησιμοποιηθεί μαζί με κάποιον άλλον μηχανισμό συγχρονισμού όπως κλειδώματα ανά δομή δεδομένων, κλειδώματα ανά νήμα, ακολουθιακά κλειδώματα, ατομικές εντολές, κλπ. Για παράδειγμα, αν χρησιμοποιείται μία δομή δεδομένων που απεικονίζει κλειδιά σε δομές δεδομένων και αυτές οι δομές δεδομένων πρέπει να είναι συγχρονισμένες, το RCU μπορεί να χρησιμοποιηθεί για την προστασία της δομής της απεικόνισης μόνο, και ένα κλείδωμα για την κάθε δομή στις οποίες απεικονίζονται τα κλειδιά θα έπρεπε να χρησιμοποιηθεί επίσης. Μία σημαία (flag) για τη διαγραφή θα έπρεπε να χρησιμοποιηθεί ανά δομή επίσης, για να δείξει αν η επιθυμητή ενέργεια μπορεί να πραγματοποιηθεί ή όχι. Η ενέργεια θα γίνει υπό την προστασία του κλειδώματος της εκάστοτε δομής, και οι εγγραφείς θα πρέπει να περιμένουν να παρέλθει μία περίοδος χάριτος προτού απελευθερώσουν τη δομή.

Δυστυχώς, σε περιπτώσεις που το φορτίο εργασίας περιλαμβάνει κυρίως ενημερώσεις, το RCU μάλλον δεν εφαρμόζεται (βλ. και [Arca03]). Σε τέτοιες περιπτώσεις, το RCU χρησιμοποιείται συνήθως για να παρέχει εγγυήσεις ύπαρξης και μόνο (όσο κάποιος αναφέρεται σε κάποια δομή δεδομένων, αυτή δεν θα διαγραφεί).

2.4 Προδιαγραφές του RCU

Στην ενότητα αυτή θα παρουσιάσουμε ορισμένες απαιτήσεις οι οποίες πρέπει να ικανοποιούνται από κάθε υλοποίηση του RCU. Οι απαιτήσεις αυτές δεν παρουσιάζονται φορμαλιστικά, και έχουν παρουσιαστεί ξανά στο παρελθόν [McKe15a, McKe15b, McKe15c, Kernd]. Η ενότητα αυτή θα χρησιμεύσει σαν οδηγός όσον αφορά στον τύπο των ελέγχων που θα πρέπει να πραγματοποιηθούν για να επαληθευτούν ορισμένες ιδιότητες του RCU.

2.4.1 Εγγύηση Περιόδου Χάριτος

Το γεγονός ότι στο RCU οι εγγραφές περιμένουν την ολοκλήρωση των κρίσιμων τμημάτων όλων των προϋπαρχόντων αναγνωστών αποτελεί τη μοναδική αλληλεπίδραση μεταξύ εγγραφέων και αναγνωστών. Η εγγύηση Περιόδου Χάριτος είναι αυτό που επιτρέπει στους εγγραφείς να περιμένουν την ολοκλήρωση όλων των προϋπαρχόντων κρίσιμων τμημάτων ανάγνωσης RCU. Υπενθυμίζουμε ότι τα κρίσμα τμήματα ανάγνωσης RCU ξεκινούν με `rcu_read_lock()` και τελειώνουν με `rcu_read_unlock()`.

Η εγγύηση αυτή λοιπόν σημαίνει ότι κάθε υλοποίηση του RCU πρέπει να διασφαλίζει ότι κάθε κρίσιμο τμήμα ανάγνωσης που έχει ήδη ξεκινήσει κατά την αρχή μιας περιόδου χάριτος θα έχει ολοκληρωθεί (συμπεριλαμβανομένων εργασιών στη μνήμη, κλπ) πριν το τέλος αυτής της περιόδου χάριτος. Το γεγονός αυτό κάνει την επαλήθευση του RCU να μπορεί να είναι εστιασμένη. Κάθε υλοποίηση πρέπει να υπακούει στον ακόλουθο κανόνα:

> Αν μία εντολή σε ένα δοθέν κρίσιμο τμήμα ανάγνωσης RCU προηγείται μιας περιόδου χάριτος, τότε όλες οι εντολές -συμπεριλαμβανομένων εργασιών στη μνήμη- σε αυτό το κρίσιμο τμήμα ανάγνωσης RCU πρέπει να ολοκληρωθούν πριν τελειώσει αυτή η περίοδος χάριτος

Οι εργασίες στη μνήμη συμπεριλαμβάνονται έτσι ώστε να εμποδιστεί ο μεταγλωττιστής ή ο επεξεργαστής από το να αναιρέσουν δουλειά που έγινε από το RCU.

Για να κατανοήσουμε τι υπονοεί η εγγύηση αυτή, ας θεωρήσουμε το ακόλουθο κομμάτι κώδικα όπου όλες οι μεταβλητές αρχικά είναι μηδέν:

```
int x;
int y;

int r_x;
int r_y

void thread_reader(void)
{
  rcu_read_lock();
  r_x = READ_ONCE(x);
  r_y = READ_ONCE(y);
  rcu_read_unlock();
}

void thread_update(void)
{
  WRITE_ONCE(x, 1);
  synchronize_rcu();
  WRITE_ONCE(y, 1);
}
```

Listing 2.9: Παράδειγμα της εγγύησης Περιόδου Χάριτος του RCU.

Αφού η `synchronize_rcu()` πρέπει να περιμένει την ολοκλήρωση όλων των προϋπαρχόντων κρίσιμων τμημάτων ανάγνωσης RCU, το αποτέλεσμα:

```
r_x == 0 && r_y == 1
```

είναι αδύνατο. Αυτή είναι και η ουσία της εγγύησης Περιόδου Χάριτος. Είναι η πιο σημαντική εγγύηση που παρέχει το RCU, και πρακτικά αποτελεί τον πυρήνα του RCU.

Ένα ακόμη πρόγραμμα ελέγχου για την εγγύηση Περιόδου Χάριτος του RCU παρουσιάζεται στον Κώδικα 2.10.

```
int r_x;
int r_y;

int x = 0;
int y = 0;

void thread0(void)
{
        WRITE_ONCE(x, 1);
        synchronize_rcu();
        r_y = READ_ONCE(y);
}

void thread1(void)
{
        rcu_read_lock();
        WRITE_ONCE(y, 1);
        r_x = READ_ONCE(x);
        rcu_read_unlock();
}
```

Listing 2.10: Παράδειγμα 2 της εγγύησης Περιόδου Χάριτος του RCU.

Στο παράδειγμα αυτό, αν μετά τον τερματισμό και των δύο νημάτων το αποτέλεσμα:

```
r_x == 0 && r_y == 0
```

ήταν δυνατό, αυτό θα έδειχνε μία αποτυχία του RCU.

Τέλος, ας σημειωθεί ότι στα δύο παραπάνω παραδείγματα η συνάρτηση `call_rcu()` θα μπορούσε να έχει χρησιμοποιηθεί στη θέση του `synchronize_rcu()` αν δεν επιθυμούσαμε να φράσσεται ο εγγραφέας. Φυσικά, τα κρίσιμα τμήματα ανάγνωσης RCU δεν πρέπει να καλούν την `synchronize_rcu()` (ή να περιμένουν την ολοκλήρωση της συνάρτησης αυτής), αφού αυτό μπορεί να οδηγήσει σε αδιέξοδα (deadlocks).

2.4.2 Εγγύηση Έκδοσης-Συνδρομής

Η εγγύηση Έκδοσης-Συνδρομής χρησιμοποιείται για τον συντονισμό των προσβάσεων ανάγνωσης στις δομές δεδομένων. Στην Ενότητα 2.2.1 αναφέρθηκε ότι ο μηχανισμός Έκδοσης-Συνδρομής χρησιμοποιείται για την εισαγωγή δεδομένων σε δομές δεδομένων (π.χ. λίστες) χωρίς να διαταράσσονται ταυτόχρονοι αναγνώστες.

Ουσιαστικά η εγγύηση αυτή έχει εξηγηθεί στην Ενότητα 2.2.1. Παρουσιάσθηκε τόσο ο μηχανισμός Έκδοσης όσο και ο μηχανισμός Συνδρομής (Κώδικες 2.2 και 2.5 αντίστοιχα), μαζί με παραδείγματα του τί θα μπορούσε να συμβεί αν οι μηχανισμοί αυτοί δεν χρησιμοποιούνταν. Με άλλα λόγια, η εγγύηση Έκδοσης-Συνδρομής παρέχεται από τον μηχανισμό Έκδοσης-Συνδρομής, βασίζεται στα `rcu_assign_pointer()` και `rcu_dereference()`, και διασφαλίζει ότι οι αναγνώστες θα έχουν μία συνεπή οπτική των νεοεισαχθέντων δεδομένων. Το παράδειγμα στον Κώδικα 2.11 δείχνει ένα απλό σενάριο έκδοσης-συνδρομής όπου είναι εγγυημένο ότι ο αναγνώστης δεν θα δει μη-αρχικοποιημένες τιμές. Στο παράδειγμα αυτό, ο αναγνώστης θα δει τις αρχικοποιημένες τιμές για τα πεδία του `p`, δηλαδή:

```
p->a == 42 && p->b == 42
```
(2.1)

Η εγγύηση Έκδοσης-Συνδρομής υποδεικνύει ότι ένα αποτέλεσμα διαφορετικό από το παραπάνω είναι αδύνατο.

2.4.3 Άνευ Όρων Αναβάθμιση Αναγνωστών σε Εγγραφείς

Γενικά, το RCU δεν επιβάλλει κανενός είδους περιορισμό στους αναγνώστες, κάτι που σημαίνει ότι πρέπει να είναι πάντα δυνατό ένας αναγνώστης να πραγματοποιήσει αλλαγές εντός του κρίσιμου τμήματος ανάγνωσής του. Φυσικά, οι αναγνώστες δεν πρέπει να καλούν τη συνάρτηση `synchronize_rcu()` εντός του κρίσιμου τμήματός τους, καθώς αυτό μπορεί να προκαλέσει προβλήματα όπως αδιέξοδα (ένας τρόπος να αποφευχθούν τα αδιέξοδα είναι η χρήση της `call_rcu()`, όπως έχουμε ήδη αναφέρει), αλλά εκτός από αυτό, ένας αναγνώστης είναι ελεύθερος να πραγματοποιήσει ενημερώσεις εντός του κρίσιμου τμήματός του. Το αποτέλεσμα αυτής της ενέργειας αν υπάρχουν ταυτόχρονοι εγγραφείς εξαρτάται από τον μηχανισμό συγχρονισμού που χρησιμοποιείται από τους εγγραφείς (π.χ. κατά πόσον οι εγγραφείς χρησιμοποιούν ένα καθολικό ή πολλά κλειδώματα, ατομικές εντολές, κλπ). Είναι ευθύνη του προγραμματιστή να εμποδίζει τους εγγραφείς να συγκρούονται με αποδιοργανωτικό τρόπο.

2.4.4 Άνευ Όρων Εκτέλεση των Αρχέγονων Συναρτήσεων του RCU

Η τελευταία εγγύηση που παρέχεται από το RCU είναι ότι εκτελούνται άνευ όρων δηλαδή απλά κάνουν τις απαραίτητες ενέργειες κι επιστρέφουν. Με άλλα λόγια, δεν γίνεται να αποτύχουν.

Οι υλοποιήσεις για κάποιες από τις πιο συχνά χρησιμοποιούμενες συναρτήσεις του RCU βρίσκονται στο `<linux/rcupdate.h>`, ενώ κάποιες συναρτήσεις χρησιμοποιούμενες από εγγραφείς μπορούν να βρεθούν στο αρχείο της αντίστοιχης υλοποίησης.

```
bool add_gp(int x, int y)
{
        struct foo *p;

        p = kcalloc(1, sizeof(*p), GFP_KERNEL);
        if (!p)
                return -ENOMEM;
        spin_lock(&gp_lock);
        if (rcu_access_pointer(gp)) {
                spin_unlock(&gp_lock);
                return false;
        }
        p->a = x;
        p->b = x;
        rcu_assign_pointer(gp, p);
        spin_unlock(&gp_lock);
        return true;
}

bool use_gp(void)
{
        struct foo *p;

        rcu_read_lock();
        p = rcu_dereference(gp);
        if (p) {
                BUG_ON(p->a != 42 || p->b != 42);
                /* do something with p->a, p->b */
                rcu_read_unlock();
                return true;
        }
        rcu_read_unlock();
        return false;
}

void *thread_publisher(void *arg)
{
        add_gp(42, 42);
        return NULL;
}

void *thread_subscriber(void *arg)
{
        use_gp();
        return NULL;
}
```

Listing 2.11: Παράδειγμα της εγγύησης Έκδοσης-Συνδρομής του RCU.

Κεφάλαιο 3

Συστηματικός Έλεγχος Ορθότητας του Tiny RCU

Στο κεφάλαιο αυτό θα επιβεβαιώσουμε μηχανικά την ιδιότητα της Περιόδου Χάριτος για το Tiny RCU, μία έκδοση του RCU σχεδιασμένη να λειτουργεί σε μονοεπεξεργαστικά συστήματα. Αν και αυτή δεν είναι η πρώτη μηχανική απόδειξη της ιδιότητας αυτής (ο Paul McKenney παρουσίασε την πρώτη μηχανική απόδειξη για το Tiny RCU το 2015, βλ. [McKe15d]), θα αποτελέσει το πρώτο βήμα προς τον συστηματικό έλεγχο ορθότητας κάποιων κομματιών του Tree RCU για μη-προεκχωρητικούς (non-preemptible) πυρήνες, ο οποίος θα παρουσιαστεί στο Κεφάλαιο 4. Ο κώδικας για όλους τους ελέγχους που πραγματοποιήθηκαν στα Κεφάλαια 3 και 4 βρίσκεται στην τοποθεσία `https://github.com/michalis-/rcu`.

Η διάρθρωση του κεφαλαίου αυτού είναι η ακόλουθη: στην Ενότητα 3.1 παρουσιάζουμε ορισμένες λεπτομέρειες υλοποίησης για το Tiny RCU, στην Ενότητα 3.2 παρουσιάζουμε ορισμένους ορισμούς που χρησιμοποιήθηκαν γαι την μοντελοποίηση ενός μονοεπεξεργαστικού συστήματος Linux, και στην Ενότητα 3.3 παρουσιάζουμε έναν έλεγχο για την εγγύηση Περιόδου Χάριτος του Tiny RCU.

3.1 Γενικές Πληροφορίες

Το Tiny RCU είναι μία υλοποίηση του RCU για μονοεπεξεργαστικά συστήματα, σχεδιασμένο να έχει πολύ μικρή κατανάλωση μνήμης, κάτι που το καθιστά ιδανικό για ενσωματωμένα συστήματα [McKe09b]. Το κύριο χαρακτηριστικό που ξεχωρίζει το Tiny RCU από άλλες υλοποιήσεις του RCU είναι το ακόλουθο:

> Οποιαδήποτε στιγμή ο μοναδικός επεξεργαστής του συστήματος περάσει από κατάσταση ηρεμίας, μία περίοδος χάριτος έχει παρέλθει.

Το χαρακτηριστικό αυτό απλοποιεί σε μεγάλο βαθμό τον σχεδιασμό της υλοποίησης του Tiny RCU, αφού επιτρέπει απλούστερες δομές δεδομένων και μικρότερη κατανάλωση μνήμης από άλλες εκδόσεις του RCU. Το χαρακτηριστικό αυτό επίσης υπονοεί ότι εκκρεμείς επανακλήσεις ενός επεξεργαστή μπορούν να εκτελεστούν κάθε φορά που γίνεται εναλλαγή περιβάλλοντος λειτουργίας (context switch). Ωστόσο, αυτό δεν συμβαίνει ούτε για το Tiny RCU, ούτε για το Tree RCU. Ο πυρήνας του RCU και για τις δύο αυτές περιπτώσεις εκτελείται σε περιβάλλον softirq (βλ. Ενότητα 4.1), αφού είναι γενικά επιθυμητό οι επανακλήσεις να μπορούν να περιλαμβάνουν κλήσεις σε συναρτήσεις χρονοδρομολόγησης (π.χ. `wake_up()`, χωρίς προβλήματα όπως αδίεξοδα (deadlocks).

Στο σημείο αυτό, θα αποδειχθεί χρήσιμο να παρουσιάσουμε ορισμένες λεπτομέρειες υλοποίησης του Tiny RCU. Για τις επόμενες παραγράφους είναι σημαντικό να διατηρηθεί υπ' όψιν ότι το Tiny RCU προσφέρεται μόνο για μη-προεκχωρητικούς πυρήνες.

Πρώτα απ' όλα, πώς ξέρει το RCU ότι ένας επεξεραστής έχει περάσει από κατάσταση ηρεμίας; Το Tiny RCU βασίζεται στις εναλλαγές περιβάλλοντος λειτουργίας, στις διακοπές του ρολογιού και τη λειτουργία `dynticks-idle`, όπως ακριβώς και το Tree RCU,

μονολότι με έναν λίγο διαφορετικό τρόπο. Αξίζει να σημειωθεί ότι αυτός δεν είναι κανόνας αλλά μάλλον μια σχεδιαστική επιλογή. Οι καταστάσεις ηρεμίας θα μπορούσαν να σημειώνονται χειροκίνητα από τον προγραμματιστή (βλ. την έκδοση QSBR του user-space RCU [Desn09, Desn12]), αλλά αυτό δεν συμβαίνει εδώ. Μία κλιμακώσιμη και αποδοτική υλοποίηση είναι υψίστης σημασίας.

Κάθε φορά που ένας επεξεργαστής δέχεται μια διακοπή ρολογιού, καλείται η `rcu_check_callbacks()` (η συνάρτηση αυτή πρέπει να καλείται από περιβάλλον hardirq). Η `rcu_check_callbacks()` ελέγχει αν ο επεξεργαστής είναι σε κατάσταση ηρεμίας (π.χ. αν ο επεξεργαστής εκτελεί κώδικα χώρου χρήστη ή ήταν idle όταν δέχθηκε τη διακοπή), και αν ναι, καλεί την `rcu_sched_qs()` για να ενημερώσει το RCU[1]. Η `rcu_sched_qs()` σημειώνει ότι όλες οι εκκρεμείς επανακλήσεις του επεξεργαστή είναι έτοιμες να εκτελεστούν (ενημερώνοντας κατάλληλα τη λίστα επανακλήσεων), και ενεργοποιεί ένα softirq. Οι έτοιμες επανακλήσεις του επεξεργαστή θα εκτελεστούν σε περιβάλλον softirq από την `rcu_process_callbacks()`. Η λίστα επανακλήσεων χωρίζεται σε τμήματα, διότι η `rcu_process_callbacks()` πρέπει να εκτελέσει μόνο τις επανακλήσεις που είναι έτοιμες προς εκτέλεση, και όχι επανακλήσεις που πιθανώς καταχωρίστηκαν μετά την κατάσταση ηρεμίας αλλά πριν την εκτέλεση του softirq.

Από την άλλη, αυτή είναι η μισή ιστορία. Όταν ένας επεξεργαστής εισέρχεται σε λειτουργία `dynticks-idle`, η `rcu_sched_qs()` καλείται μιας και η λειτουργία αυτή σηματοδοτεί τόσο μία κατάσταση ηρεμίας για το Tiny RCU, όσο και την ολοκλήρωση μίας περιόδου χάριτος. Η `rcu_sched_qs()` πραγματοποιεί τις ενέργειες που περιγράφηκαν πραπάνω. Οι παραπάνω παράγραφοι παρουσιάζουν επιγραμματικά τον τρόπο που το Tiny RCU διαχειρίζεται τις επανακλήσεις συναρτήσεων.

Ωστόσο, η `synchronize_rcu()` δουλεύει με ελαφρώς διαφορετικό τρόπο, όπως μπορούμε να δούμε στη συνέχεια:

```
1  void synchronize_sched(void)
2  {
3          rcu_lockdep_assert(!lock_is_held(&rcu_bh_lock_map) &&
4                             !lock_is_held(&rcu_lock_map) &&
5                             !lock_is_held(&rcu_sched_lock_map),
6                             "Illegal synchronize_sched() in RCU read-side critical section");
7          cond_resched();
8  }
```

Listing 3.1: Υλοποίηση της `synchronize_sched()` για το Tiny RCU.

Η συνάρτηση αυτή δεν βασίζεται σε επανακλήσεις του RCU και στην `call_rcu()`, σε αντίθεση με το Tree RCU. Η υλοποίησή της είναι αρκετά απλή. Στην πραγματικότητα, μία κλήση στην `synchronize_rcu()` είναι κατάσταση ηρεμίας μιας και δεν επιτρέπεται η συνάρτηση αυτή να κληθεί μέσα σε κρίσιμο τμήμα ανάγνωσης του RCU. Η συνάρτηση `cond_resched()` χρησιμοποιείται για ρητή χρονοδρομολόγηση σε μέρη που αυτό είναι ασφαλές και μπορεί να οδηγήσει στην κλήση της `__schedule()`, η οποία σημειώνει κατάσταση ηρεμίας καλώντας τη συνάρτηση `rcu_note_context_switch()`, η οποία με τη σειρά της καλεί την `rcu_sched_qs()`. Το γεγονός ότι μία κλήση στην `synchronize_rcu()` αποτελεί κατάσταση ηρεμίας, συνιστά την κύρια αιτία κλήσης της `cond_resched()`.

3.2 Μοντελοποίηση του Περιβάλλοντος του Πυρήνα

Είναι προφανές ότι το RCU βασίζεται σε πολλά από τα υποσυστήματα του πυρήνα για τη σωστή λειτουργία του. Το Tiny RCU χρησιμοποιεί πολλούς ορισμούς και συ-

[1] Η `rcu_bh_qs()` επίσης καλείται από την `rcu_check_callbacks()` για την καταγραφή καταστάσεων ηρεμίας του RCU-bh. Ωστόσο, αγνοούμε όλες τις συναρτήσεις σχετικές με το RCU-bh, χάριν απλότητος.

ναρτήσεις του πυρήνα, και συμπεριλαμβάνει (`#include`) πολλά αρχεία του πυρήνα στον πηγαίο κώδικά του. Επιπρόσθετα, πρέπει να βρούμε έναν τρόπο να μοντελοποιήσουμε ένα μη-προεκχωρητικό μονοεπεξεργαστικό περιβάλλον και τους περιορισμούς που αυτό επιβάλλει. Όλα τα παραπάνω αποτελούν ενδιαφέροντα προβλήματα που θα επιλυθούν στις επόμενες παραγράφους.

3.2.1 Επεξεργαστής, Διακοπές & Χρονοδρομολόγηση

Επεξεργαστής

Αφού το Tiny RCU λειτουργεί σε μονοπύρηνα μη-προεκχωρητικά συστήματα, μόνο ένα νήμα τρέχει στον επεξεργαστή τη φορά. Συνεπώς, χρειαζόμαστε κάποιου είδους αμοιβαίο αποκλεισμό μεταξύ νημάτων που ανταγωνίζονται για τον επεξεργαστή. Ο αμοιβαίος αποκλεισμός αυτός μπορεί να προσφερθεί από ένα mutex, και αφού τα προγράμματα ελέγχου μας θα εκτελεστούν από το Nidhugg, το οποίο υποστηρίζει τη βιβλιοθήκη pthreads, ένα `pthread_mutex` χρησιμοποιήθηκε.

Αλλά πώς παίρνει ένα νήμα τον επεξεργαστή; Η απάντηση παρουσιάζεται στον Κώδικα 3.2.

```
1  void fake_acquire_cpu(void)
2  {
3          if (pthread_mutex_lock(&cpu_lock))
4                  exit(-1);
5          rcu_idle_exit();
6  }
```

Listing 3.2: Κατάληψη του επεξεργαστή από ένα νήμα για το Tiny RCU.

Στον Κώδικα 3.2 υπονοείται ότι ο επεξεργαστής ξεκινάει ανενεργός (idle). Πράγματι λοιπόν, ο μοναδικός επεξεργαστής του συστήματος ξεκινάει ανενεργός, και σταματά να είναι ανενεργός όταν ένα νήμα παίρνει το κλείδωμά του. Μπορεί αυτό να μην συμβαίνει σερ παραγματικά συστήματα, αλλά για τη μοντελοποίησή μας η προσέγγιση αυτή επαρκεί. Με παρόμοια λογική, όταν ένα νήμα τελειώσει την εκτέλεσή του καλεί την `fake_release_cpu()`, όπως φαίνεται στον Κώδικα 3.3.

```
1  void fake_release_cpu(void)
2  {
3          rcu_idle_enter();
4          if (pthread_mutex_unlock(&cpu_lock))
5                  exit(-1);
6          if (need_softirq) {
7                  need_softirq = 0;
8          }
9  }
```

Listing 3.3: Απελευθέρωση του επεξεργαστή από ένα νήμα για το Tiny RCU.

Ας σημειωθεί ότι όταν ένα νήμα απελευθερώσει τον επεξεργαστή, ο επεξεργαστής γίνεται ανενεργός. Η μεταβλητή `need_softirq` δείχνει κατά πόσον έχει ενεργοποιηθεί ένα softirq. Δεν μοντελοποιήσαμε τα softirqs για το Tiny RCU, αλλά κάτι τέτοιο δεν θα ήταν δύσκολο (βλ. Ενότητα 4.2). Έτσι, ασύγχρονες συναρτήσεις όπως η `call_rcu()` δεν μπορούν να χρησιμοποιηθούν στα προγράμματά μας. Ωστόσο, αυτό δεν αποτέλεσε πρόβλημα. Τα προγράμματα ελέγχου που χρησιμοποιήσαμε βασίζονται στην `synchronize_rcu()`, διότι το Tiny RCU έχει διαφορετική υλοποίηση για τη συνάρτηση αυτή, απ' ό,τι το Tree RCU. Αν

θέλαμε να χρησιμοποιήσουμε την `call_rcu()` στη θέση της `synchronize_rcu()`, δεν θα ήταν δύσκολο να το κάνουμε, λόγου χάρη, ακολουθώντας μια παρόμοια προσέγγιση με αυτή που περιγράφεται στην Ενότητα 4.2 για το Tree RCU.

Διακοπές

Όσο για τον χειρισμό τον διακοπών, υπάρχουν δύο κλειδώματα: το κλείδωμα `irq_lock` το οποίο δείχνει ότι ένα νήμα έχει τις διακοπές απενεργοποιημένες, πιθανώς επειδή βρίσκεται σε έναν χειριστή διακοπών, και το `nmi_lock` το οποίο δείχνει ότι ένα νήμα είναι σε έναν χειριστή Μη Απενεργοποιήσιμων Διακοπών[2] (Non Maskable Interrupts – NMIs). Τα κλειδώματα αυτά είναι απλά mutexes της βιβλιοθήκης pthreads. Ένα νήμα δεν μπορεί να πάρει το κλείδωμα του επεξεργαστή ή το κλείδωμα τον διακοπών ενώ κρατάει το κλείδωμα των Μη Απενεργοποιήσιμων Διακοπών, και ένα νήμα μπορεί να πάρει το κλείδωμα των διακοπών μόνο αν έχει ήδη πάρει το κλείδωμα του επεξεργαστή. Ένας μετρητής χρησιμοποιείται για την εξυπηρέτηση εμφωλευμένων διακοπών και συναρτήσεων όπως η `local_irq_save()` και η `local_irq_restore()`.

Χρονοδρομολόγηση

Τέλος, όπως δείξαμε στην προηγούμενη ενότητα, οι εναλλαγές περιβάλλοντος λειτουργίας είναι καταστάσεις ηρεμίας για το Tiny RCU. Επιπρόσθετα, αναφέραμε ότι η `cond_resched()` χρησιμοποιείται στην `synchronize_rcu()`. Προφανώς, η συνάρτηση αυτή πρέπει να μοντελοποιηθεί, κρατώντας υπ' όψιν ότι η `synchronize_rcu()` μπορεί να μπλοκάρει. Στον Κώδικα 3.4 παρουσιάζεται ο τρόπος που μπορεί να μοντελοποιηθεί η `cond_resched()`.

```
1 void cond_resched(void)
2 {
3         fake_release_cpu();
4         fake_acquire_cpu();
5 }
```

Listing 3.4: Η συνάρτηση `cond_resched()` για το Tiny RCU.

Στον πυρήνα η συνάρτηση αυτή καλεί τον χρονοδρομολογητή μόνο αν υπάρχει ανάγκη για χρονοδρομολόγηση. Αν ο χρονοδρομολογητής κληθεί, σημειώνεται ότι ο επεξεργαστής πέρασε από κατάσταση ηρεμίας. Για να μοντελοποιηθεί αυτό το σενάριο, η δική μας εκδοχή για την `cond_resched()` απλά αφήνει το κλείδωμα του επεξεραγστή και μετά προσπαθεί να το ξαναπάρει. Επίσης, σημειώνεται ότι ο επεξεργαστής πέρασε από κατάσταση ηρεμίας, χάρη στη συνάρτηση `rcu_idle_enter()` η οποία καλείται από την `fake_release_cpu()`.

3.2.2 Ορισμοί του Πυρήνα

Το Tiny RCU συμπεριλαμβάνει πολλά αρχεία από διαφορετικά υποσυστήματα του πυρήνα, και εμείς θέλουμε να κρατήσουμε τον πηγαίο κώδικα ανέπαφο για να πραγματοποιήσουμε συστηματικό έλεγχο ορθότητας. Ένας τρόπος να ξεπεράσουμε το πρόβλημα αυτό είναι να προσθέσουμε τον κατάλογο αρχείων των προγραμμάτων ελέγχου στους καταλόγους που ψάχνει ο προεπεξεργαστής για αρχεία που συμπεριλαμβάνονται, και να παρέχουμε κενά αρχεία στη θέση των αρχείων που συμπεριλαμβάνονται, ενώ όμως παρέχουμε τους απαραίτητους ορισμούς σε δικά μας αρχεία, τα οποία θα συμπεριληφθούν στα προγράμματα ελέγχου μας. Μερικοί από τους απαραίτητους ορισμούς αντιγράφηκαν απευθείας από την έκδοση v3.19 του πυρήνα (π.χ. `IS_ENABLED()`, `bool`, `barrier()`), ενώ

[2] Αν και με αυτόν τον τρόπο μοντελοποιούνται οι Μη Απενεργοποιήσιμες Διακοπες, δεν συμπεριλήφθηκαν στα προγράμματα ελέγχου μας.

άλλοι αντικαταστάθηκαν με κενούς (π.χ. μακροεντολές σχετικές με προεκχωρητικότητα, έλεγχοι που γίνονται κατά την αρχικοποίηση, ενεργοποίηση softirqs κλπ) ή με δικούς μας (π.χ. `cond_resched()`). Ας σημειωθεί ότι η μακροεντολή `WARN_ON_ONCE()` και οι συγγενικές της μακροεντολές (π.χ. `BUG_ON()`) μετατράπηκαν σε εντολές `assert()`, κάτι που αποδείχθηκε εξαιρετικά ωφέλιμο τόσο για τους ελέγχους αυτού του κεφαλαίου, όσο και για αυτούς του επόμενου.

3.3 Επιβεβαίωση της Εγγύησης Περιόδου Χάριτος του Tiny RCU

Για να επιβεβαιώσουμε μηχναικά την εγγύηση Περιόδου Χάριτος του Tiny RCU, μπορούμε να χρησιμοποιήσουμε ένα πρόγραμμα ελέγχου παρόμοιο με αυτό του Κώδικα 3.5.

```
1  int x, y;
2  int r_x, r_y;
3
4  void *thread_update(void *arg)
5  {
6          fake_acquire_cpu();
7
8          x = 1;
9          synchronize_rcu();
10         y = 1;
11
12         fake_release_cpu();
13 }
14
15 void *thread_reader(void *arg)
16 {
17         fake_acquire_cpu();
18
19         rcu_read_lock();
20         r_x = x;
21 #ifdef FORCE_FAILURE
22         rcu_read_unlock();
23         cond_resched();
24         rcu_read_lock();
25 #endif
26         r_y = y;
27         rcu_read_unlock();
28
29         fake_release_cpu();
30 }
```

Listing 3.5: Πρόγραμμα για την επαλήθευση της εγγύησης Περιόδου Χάριτος του Tiny RCU.

Στον κώδικα αυτόν το αποτέλεσμα:

```
r_x == 0 && r_y == 1
```
(3.1)

πρέπει να είναι αδύνατο να συμβεί. Αφού θέλουμε η παραπάνω συνθήκη να μην παραβιαστεί, μπορούμε να προσθέσουμε ένα παρόμοιο `assert()` στο τέλος του προγράμματός μας.

Το πρόγραμμα ελέγχου μας (παρέχεται στο github) μπορεί να τρέξει υπό ακολουθιακή συνέπεια με την ακόλουθη εντολή:

```
nidhuggc -I . -std=gnu99 -- --sc --disable-mutex-init-requirement fake.c
```

όπου `fake.c` είναι το αρχείο του προγράμματος ελέγχου.

Το Nidhugg χρειάζεται μόνο 0.08 δευτερόλεπτα για να ισχυριστεί ότι δεν παραβιάζεται κάποια εντολή `assert()` στο πρόγραμμά μας. Αλλά μπορούμε να το εμπιστευτούμε; Έτσι κι αλλιώς, θα μπορούσε να υπάρχει κάποιο σφάλμα στην μοντελοποίησή μας, στο Nidhugg, ή στο πρόγραμμα ελέγχου μας.

Εδώ είναι που εισέρχεται το κομμάτι κώδικα που συμπεριλαμβάνεται ορίζοντας `FORCE_FAILURE` στον Κώδικα 3.5. Αυτό το τμήμα κώδικα εισάγει ένα σφάλμα στο κρίσιμο τμήμα του αναγνώστη το οποίο μπορεί να κάνει την Επιβεβαίωση 3.1 να παραβιαστεί. Έτσι, αν υπάρχει τουλάχιστον μία διεμπλοκή (interleaving) που μπορεί να κάνει την επιβεβαίωση αυτή να παραβιαστεί, το Nidhugg πρέπει να είναι σε θέση να τη βρει.

Ξανατρέχοντας το Nidhugg, αλλά με `-DFORCE_FAILURE` σαν όρισμα του προεπεξεργαστή της `C`, το Nidhugg όντως βρίσκει μία τέτοια διεμπλοκή:

1. Ο αναγνώστης καταλαμβάνει τον επεξεργαστή και εκτελεί `r_x = x`.
2. Ο αναγνώστης εκτελεί την `cond_resched()` και απελευθερώνει τον επεξεργαστή, αλλά δεν τον επανακαταλαμβάνει ακόμη.
3. Ο εγγραφέας καταλαμβάνει τον επεξεργαστή και εκτελεί `x = 1`.
4. Ο εγγραφέας εκτελεί `cond_resched()`, απελευθερώνει τον επεξεργαστή και τον επανακαταλαμβάνει αμέσως.
5. Ο εγγραφέας εκτελεί `y = 1` και τερματίζει.
6. Ο αναγνώστης επανακαταλαμβάνει τον επεξεργαστή από το βήμα 2, εκτελεί `r_y = y`, απελευθερώνει τον επεξεργαστή και τερματίζει.

Προφανώς, η προαναφερθείσα ακολουθία γεγονότων παραβιάζει την Επιβεβαίωση 3.1.

Επιπρόσθετα, το Nidhugg χρειάζεται μόνο 0.08 δευτερόλεπτα για να βρει ένα αντιπαράδειγμα στο πρόγραμμά μας αυτό, σε αντίθεση με τον CBMC [Clar04] (έκδοση v4.9) που χρειάζεται 8.55 και 10.10 δευτερόλεπτα για να ισχυριστεί επιτυχία και αποτυχία, αντίστοιχα.

Τέλος, λόγω της αλληλεπίδρασης του RCU με τη λειτουργίας `dynticks-idle` μέσω των αντίστοιχων μετρητών, και επειδή το σχετικό τμήμα του πηγαίου κώδικα πυρήνα περιλαμβάνει εντολές `WARN_ON_ONCE()` (οι οποίες μετατράπηκαν σε `assert()`), το πρόγραμμα αυτό κάνει και μία μηχανική επαλήθευση των ιδιοτήτων των μετρητών αυτών. Γενικά, εντολές `WARN_ON_ONCE()` υπάρχουν και στο Tree RCU, και η αντικατάστασή τους με `assert()` ήταν εξαιρετικά χρήσιμη στο να διαπιστώσουμε κατά πόσο η μοντελοποίησή μας για τον πυρήνα ήταν ορθή. Φυσικά, το Tiny RCU έχει μία μάλλον απλή υλοποίηση (μόνο 389 γραμμές κώδικα για την έκδοση v3.19). Ο συστηματικός έλεγχος ορθότητας του Tree RCU παρουσιάζει πολύ σημαντικότερες δυσκολίες, αλλά περισσότερα γι' αυτό, στο επόμενο κεφάλαιο.

Κεφάλαιο 4

Συστηματικός Έλεγχος Ορθότητας του Tree RCU

Στο κεφάλαιο αυτό θα επαληθεύσουμε μηχανικά την ιδιότητα Έκδοσης-Συνδρομής (Ενότητα 4.4) και την ιδιότητα της Περιόδου Χάριτος του Tree RCU (Ενότητα 4.5). Επιπρόσθετα, θα αναπαράγουμε ένα γνωστό σφάλμα (bug) του πυρήνα (Ενότητα 4.3.1), και θα δείξουμε ότι ένα πρώην αναφερόμενο ως σφάλμα στην πραγματικότητα δεν αποτελεί σφάλμα (Ενότητα 4.3.2). Αρχικά, παρέχουμε μία υψηλού επιπέδου επεξήγηση της υλοποίησης του Tree RCU (Ενότητα 4.1) και του τρόπου που μοντελοποιήθηκε το περιβάλλον του πυρήνα για να τρέξουν τα προγράμματα ελέγχου που κατασκευάσαμε (Ενότητα 4.2).

Ο κώδικας για το αυτό το κεφάλαιο και το Κεφάλαιο 3 μπορεί να βρεθεί στη διεύθυνση `https://www.github.com/michalis-/rcu`.

4.1 Γενικές Πληροφορίες

Ο πυρήνας του Linux παρέχει πολλές διαφορετικές υλοποιήσεις του RCU, με την καθεμία να εξυπηρετεί τον δικό της σκοπό. Η πρώτη υλοποίηση του RCU (διαθέσιμη από το 2002 και την έκδοση 2.5.43 του πυρήνα) είναι το Classic RCU. Το πρόβλημα με το Classic RCU ήταν ότι ενώ οι αρχέγονες συναρτήσεις (primitives) που παρέχει για τους αναγνώστες ήταν εξαιρετικά αποδοτικές και κλιμακώσιμες (scalable), οι συναρτήσεις που χρησιμοποιούνταν για να καθοριστεί το πότε όλοι οι προϋπάρχοντες αναγνώστες έχουν ολοκληρώσει το κρίσιμο τμήμα τους, διέθεταν περιορισμένη κλιμακωσιμότητα. Πιο συγκεκριμένα, στο Classic RCU υπήρχε ένα καθολικό κλείδωμα (lock) για το οποίο ανταγωνιζόντουσαν όλοι οι επεξεργαστές, καθώς ο κάθε επεξεργαστής έπρεπε να αναφέρει ότι πέρασε από κατάσταση ηρεμίας (quiescent state) στο RCU.

Ωστόσο, αυτό δεν ήταν το μοναδικό πρόβλημα. Από τον πυρήνα 2.6.21 του Linux, η λειτουργία `dynticks-idle` ενσωματώθηκε στον πυρήνα (`CONFIG_NO_HZ_IDLE=y` ή `CONFIG_NO_HZ=y` για παλαιότερες εκδόσεις του πυρήνα). Αυτή η λειτουργία παραλείπει τους χτύπους του ρολογιού σε επεξεργαστές οι οποίοι είναι ανενεργοί, κάτι που είναι εξαιρετικά σημαντικό για εφαρμογές πραγματικού χρόνου (real-time applications) και για υπερυπολογιστές (HPC) [Kernb]. Δυστυχώς, το Classic RCU "ξύπναγε" κάθε επεξεργαστή τουλάχιστον μία φορά ανά περίοδο χάριτος, εμποδίζοντας έτσι τους επεξεργαστές να εισέλθουν σε βαθιές C-states[1], και αυξάνοντας την κατανάλωση ενέργειας [Inte]. Σε περιπτώσεις όπου πραγματοποιούνταν ενημερώσεις συχνά, από λίγους μόνο επεξεργαστές, με τους υπόλοιπους επεξεργαστές να είναι ανενεργοί (π.χ. σε συστήματα κατασκευασμένα να εξυπηρετούν πολλά αιτήματα μαζεμένα σε μικρό χρονικό διάστημα, τα οποία το υπόλοιπο διάστημα είναι σε μεγάλο βαθμό ανενεργά), ένα σημαντικό ποσό ενέργειες χανόταν άσκοπα.

Το Tree RCU προσφέρει τη λύση στα παραπάνω προβλήματα καθώς μειώνει τη ζήτηση για ένα καθολικό κλείδωμα (lock contention) και αποφεύγει το να ξυπνάει τους επεξεργαστές σε λειτουργία `dynticks-idle` [McKe08a]. Το Tree RCU είναι πολύ κλιμακώσιμο και μπορεί να εξυπηρετήσει εύκολα χιλιάδες επεξεργαστές, σε αντίθεση με το Classic RCU

[1] Οι C-states αφορούν επεξεργαστές Intel. Η AMD χρησιμοποιεί διαφορετικές τεχνικές για εξοικονόμηση ενέργειας όπως η Cool'n'Quiet [AMDC].

που μπορούσε να εξυπηρετήσει μόνο μερικές εκατοντάδες.

Στις επόμενες υποενότητες, παρέχουμε μια υψηλού επιπέδου περιγραφή του Tree RCU μαζί με μερικές λεπτομέρειες υλοποίησης (η υλοποίηση του Tree RCU μεταβάλλεται διαρκώς στις διάφορες εκδόσεις του πυρήνα), μια σύντομη περιγραφή των δομών δεδομένων που χρησιμοποιεί το Tree RCU, και μερικές περιπτώσεις χρήσης του, χρήσιμες για την κατανόηση του τρόπου που οι βασικοί μηχανισμοί του RCU υλοποιούνται στον πυρήνα.

4.1.1 Περιγραφή Υψηλού Επιπέδου του Tree RCU

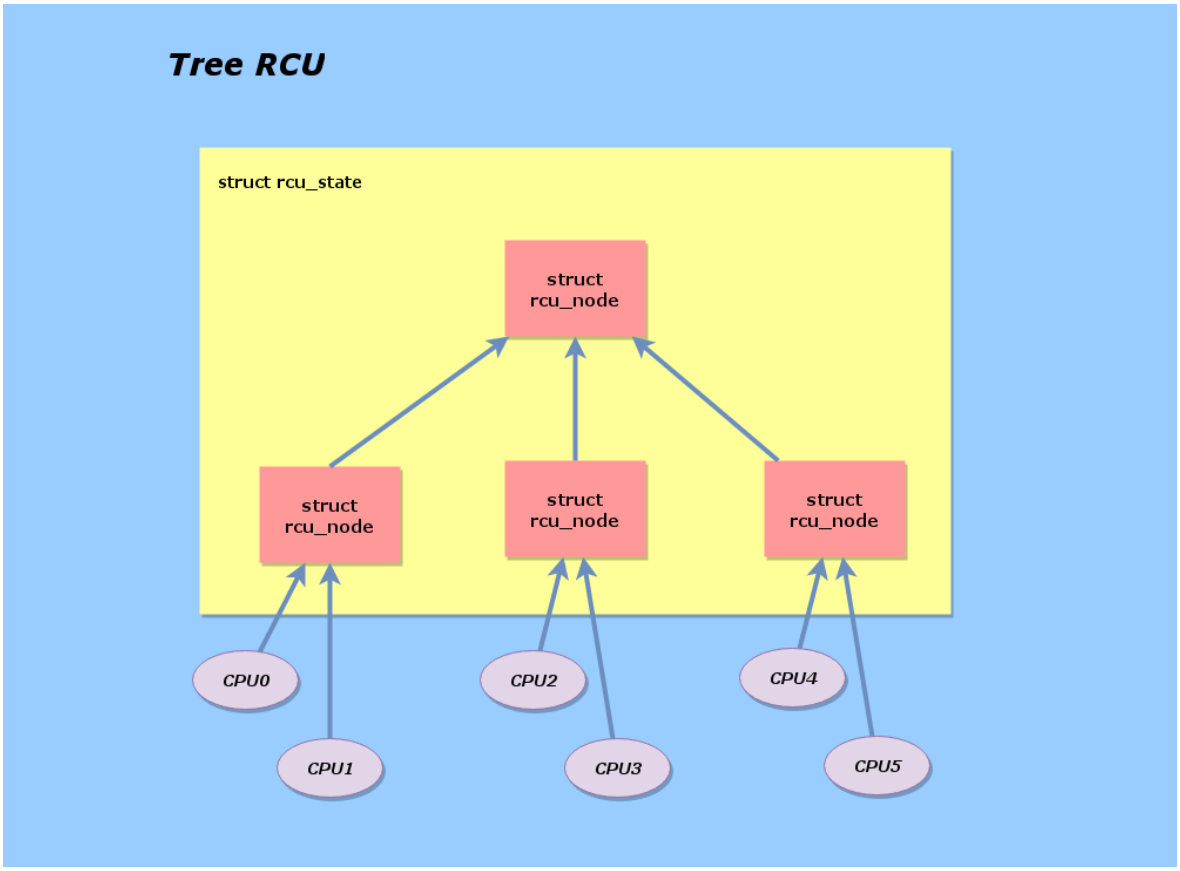
Όπως έχουμε ήδη αναφέρει, στο Classic RCU κάθε επεξεργαστής έπρεπε να μηδενίσει το bit που του αντιστοιχούσε σε ένα πεδίο μιας καθολικής δομής δεδομένων (συγκεκριμένα της δομής `rcu_ctrlblk`) όταν πέρναγε από μια κατάσταση ηρεμίας. Εφόσον οι επεξεργαστές προσπαθούσαν ταυτόχρονα να μηδενίσουν το bit τους στη δομή αυτή, ένα κλείδωμα χρησιμοποιούνταν για να προστατευθεί η μάσκα με τα bits των επεξεργαστών (`->cpumask`), κάτι που φυσικά σήμαινε ότι πολλοί επεξεργαστές ανταγωνίζονται για το ίδιο κλείδωμα.

Το Tree RCU αντιμετωπίζει το παραπάνω πρόβλημα δημιουργώντας μια ιεραρχία κόμβων με μορφή σωρού (heap), μειώνοντας έτσι τον αριθμό των επεξεργαστών που ανταγωνίζονται για ένα κλείδωμα. Το κλειδί εδώ είναι ότι οι επεξεργαστές δεν ανταγωνίζονται για το κλείδωμα του ίδιου κόμβου όταν προσπαθούν να αναφέρουν ότι πέρασαν από κατάσταση ηρεμίας στο RCU. Αντίθετα, οι επεξεργαστές χωρίζονται σε ομάδες και κάθε ομάδα θα ανταγωνιστεί για ένα διαφορετικό κλείδωμα. Κάθε επεξεργαστής πρέπει να μηδενίσει το bit του στην μάσκα του αντίστοιχου κόμβου μία φορά ανά περίοδο χάριτος. Ο τελευταίος επεξεργαστής που θα καθαρίσει το bit του (δηλαδή θα αναφέρει κατάσταση ηρεμίας στο RCU) για κάθε ομάδα επεξεργαστών, θα προσπαθήσει να παει το κλείδωμα του κόμβου-πατέρα, μέχρις ότου η μάσκα του κόμβου-ρίζα του σωρού έχει καθαριστεί. Στο σημείο αυτό μία περίοδος χάριτος μπορεί να τελειώσει (θα εξετάσουμε τον ακριβή τρόπο που συμβαίνει αυτό στην Ενότητα 4.1.3). Μία απλή ιεραρχία κόμβων για ένα σύστημα με 6 επεξεργαστές παρουσιάζεται στο Σχήμα 4.1.

Γίνεται φανερό από το Σχήμα 4.1 ότι οι επεξεργαστές CPU0 και CPU1 θα ανταγωνιστούν για το κλείδωμα του κόμβου κάτω αριστερά, οι επεξεργαστές CPU2 και CPU3 θα ανταγωνιστούν για το κλείδωμα του μεσαίου κάτω κόμβου, και οι επεξεργαστές CPU4 και CPU5 θα ανταγωνιστούν για το κλείδωμα του κάτω δεξιά κόμβου. Ο τελευταίος επεξεργαστής που θα αναφέρει κατάσταση ηρεμίας στο RCU για κάθε έναν από τους κόμβους της κάτω σειράς θα προσπαθήσει να πάρει το κλείδωμα του κόμβου-ρίζα, και αυτή η διαδικασία συμβαίνει μία φορά ανά περίοδο χάριτος.

Για να γίνει όμως πιο κατανοητή η παραπάνω διαδικασία, ας θεωρήσουμε το παράδειγμα του Σχήματος 4.2. Όπως φαίνεται στο σχήμα αυτό, οι επεξεργαστές 0, 3 και 4 περνάνε από κατάσταση ηρεμίας και το αναφέρουν στο RCU (δηλαδή μηδενίζουν το bit τους στην μάσκα του αντίστοιχου κόμβου). Μετά, ταυτόχρονα, οι επεξεργαστές 1, 2 και 5 αναφέρουν κατάσταση ηρεμίας στο RCU και, μιας και είναι όλοι οι τελευταίοι στην ομάδα τους που το κάνουν αυτό, προσπαθούν να πάρουν το κλείδωμα του κόμβου-ρίζα. Φυσικά, αυτό σημαίνει ότι οι προσβάσεις του στον κόμβο-ρίζα θα σειριοποιηθούν, και ότι θα πάρουν το κλείδωμα του κόμβου-ρίζα με μία συγκεκριμένη σειρά. Όταν όλοι οι επεξεργαστές έχουν περάσει από κατάσταση ηρεμίας τουλάχιστον μία φορά, η περίοδος χάριτος μπορεί να τελειώσει.

Συγκριτικά με το Classic RCU, η ιεραρχική φύση του Tree RCU οδηγεί σε πολύ μικρότερο ανταγωνισμό για κλειδώματα: το πολύ τρεις επεξεργαστές ανταγωνίζονται για κάποιο κλείδωμα κατά τη διάρκεια μιας περιόδου χάριτος (και μόνο δύο ανταγωνίζονται για το κλείδωμα τον κόμβων-φύλλων), ενώ στο Classic RCU θα είχαμε και τους έξι επεξεργαστές να ανταγωνίζονται για το κλείδωμα της δομής `rcu_ctrlblk` κατά τη



Σχήμα 4.1: Ιεραρχία κόμβων για το Tree RCU (σχήμα προσαρμοσμένο από [McKe08a]).

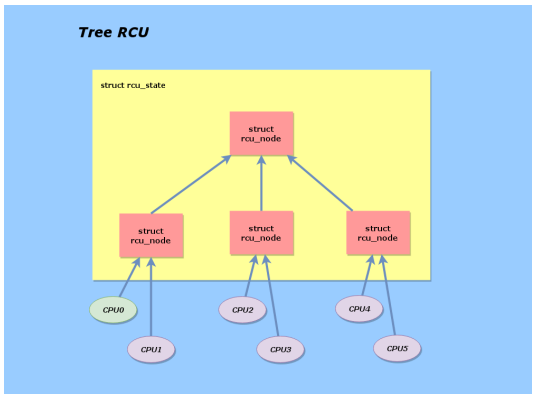
διάρκεια μιας περιόδου χάριτος. Και ενώ αυτό μπορεί να μη φαντάζει τόσο μεγάλη υπόθεση, σε συστήματα με μεγάλο αριθμό επεξεργαστών η μείωση στον ανταγωνισμό για τα κλειδώματα είναι ακόμη μεγαλύτερη. Για παράδειγμα, για ένα σύστημα με 16.384 επεξεργαστές, μόνο 16 (το πολύ) επεξεργαστές θα ανταγωνίζονται για τα κλειδώματα των κόμβων φύλλων, και το πολύ 32 επεξεργαστές θα ανταγωνίζονται για τα κλειδώματα των εσωτερικών κόμβων, υποθέτοντας ότι έχουμε μια ιεραρχία πολλών επιπέδων. Αυτό υπονοεί μια δραματική μείωση από τους 16.384 επεξεργαστές που θα διαγωνίζονταν για το κλείδωμα της δομής `rcu_ctrlblk` στην περίπτωση του Classic RCU.

Τέλος, η ιεραρχία κόμβων που δημιουργείται από το Tree RCU είναι παραμετροποιήσιμη και ελέγχεται από τις επιλογές `Kconfig`. Υπάρχουν δύο επιλογές γι' αυτό:

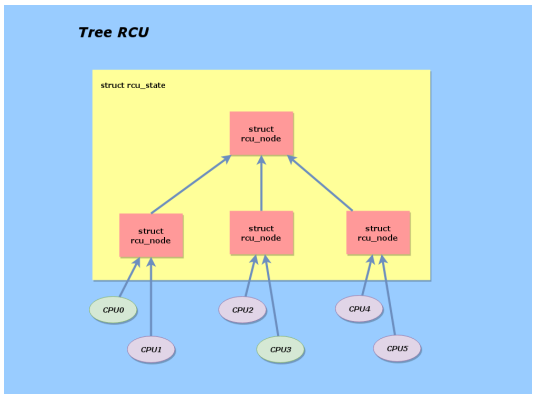
`CONFIG_RCU_FANOUT_LEAF`: Ελέγχει το πόσοι επεξεργαστές θα ανταγωνίζονται για το κλείδωμα των κόμβων-φύλλων. Η προεπιλογή είναι 16 για παλαιότερους πυρήνες, και 32/64 για 32-bit/64-bit συστήματα σε νεότερους πυρήνες.

`CONFIG_RCU_FANOUT`: Ελέγχει πόσοι επεξεργαστές θα ανταγωνίζονται για το κλείδωμα των εσωτερικών κομβων της ιεραρχίας. Η προεπιλογή είναι 32 για 32-bit συστήματα και 64 για 64-bit συστήματα.

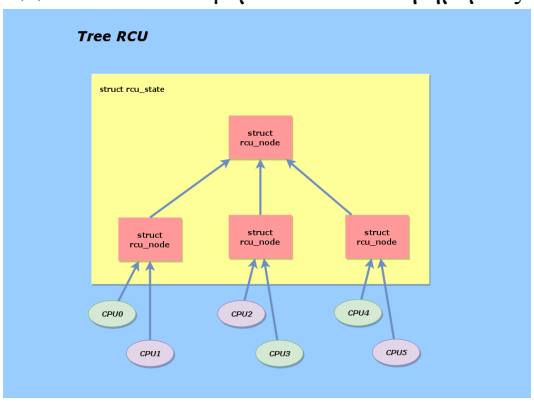
Περισσότερες πληροφορίες μπορούν να βρεθούν στο αρχείο `init/Kconfig`. Αξίζει να σημειωθεί ότι μία μεγάλη ιεραρχία χρειάζεται περισσότερο χρόνο για να αρχικοποιηθεί και να καθαριστεί (για την έναρξη και τη λήξη μιας περιόδου χάριτος, αντίστοιχα), και υπάρχουν περισσότερες δομές που πρέπει να ελεγχθούν όταν πρέπει να εξαναγκάσουμε τη λήξη μιας περιόδου χάριτος (βλ. Ενότητα 4.1.3). Αν εξαιρεθούν τα παραπάνω, ο χρήστης είναι ελεύθερος να παραμετροποιήσει την ιεραρχία με όποιο τρόπο επιθυμεί, και το RCU μπορεί να υποστηρίξει μέχρι 16.777.216 επεξεγαστές σε 64-bit συστήματα.



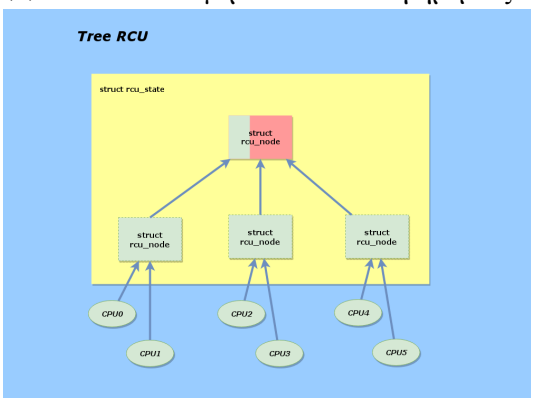
(a) Ο CPU0 αναφέρει κατάσταση ηρεμίας.



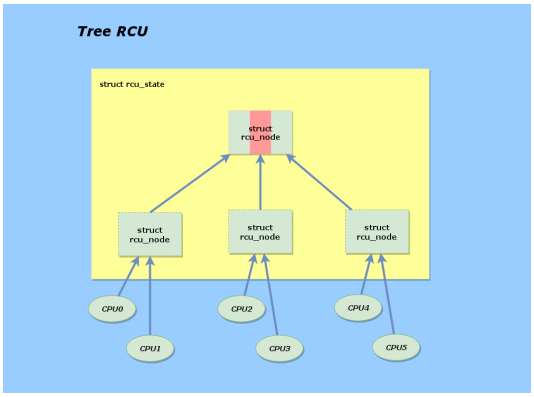
(b) Ο CPU3 αναφέρει κατάσταση ηρεμίας.



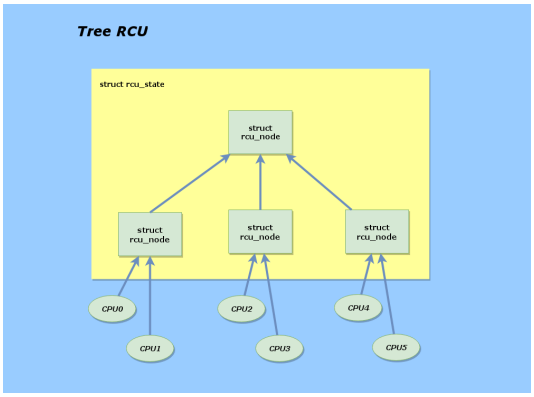
(c) Ο CPU4 αναφέρει κατάσταση ηρεμίας.



(d) Ο CPU1 παίρνει το κλείδωμα της ρίζας.



(e) Ο CPU5 παίρνει το κλείδωμα της ρίζας.



(f) Ο CPU2 παίρνει το κλείδωμα της ρίζας.

Σχήμα 4.2: Ένα παράδειγμα για μια απλή ιεραρχία (προσαρμοσμένο από [McKe08a]).

4.1.2 Δομές Δεδομένων του Tree RCU

Πέρα από την προηγούμενη περιγραφή του Tree RCU, ένας συστηματικός έλεγχος ορθότητας των θεμελιωδών μηχανισμών του RCU απαιτεί την απόλυτη κατανόηση της υλοποίησης του Tree RCU. Στην ενότητα αυτή, θα προσπαθήσουμε να παρουσιάσουμε εν συντομία μερικές σημαντικές πλευρές της υλοποίησης του Tree RCU, κρίσιμες για την ανάπτυξη μιας συλλογής προγραμμάτων ελέγχου που στοχεύουν στον συστηματικό έλεγχο ορισμένων μηχανισμών του Tree RCU.

Ο πιο σημαντικός μηχανισμός του RCU είναι ο μηχανισμός για την αναμονή των προϋπαρχόντων αναγνωστών. Και παρ' όλο που έχει προηγηθεί μία περιγραφή υψηλού επιπέδου για τον μηχανισμό αυτόν, παραμένουν αρκετά ερωτήματα σχετικά με την υλοποίηση του μηχανισμού αυτού και τις δομές δεδομένων που εμπλέκονται, προκειμένου ο

μηχανισμός αυτός να λειτουργεί όπως οφείλει. Με λίγα λόγια, η υλοποίησή του σίγουρα αξίζει την προσοχή μας.

Υπάρχουν τρεις βασικές δομές δεδομένων που χρησιμοποιούνται στο Tree RCU: η δομή `rcu_data`, η δομή `rcu_node`, και η δομή `rcu_state`.

Ας υποθέσουμε ότι ένας επεξεργαστής καταχωρεί μια συνάρτηση προς επανάκληση (callback). Φυσικά, το Tree RCU πρέπει να αποθηκεύει ορισμένες πληροφορίες σχετικά με την επανάκληση αυτή. Γι' αυτό, η υλοποίηση του Tree RCU διατηρεί μερικά δεδομένα ανά επεξεργαστή, οργανωμένα σε δομές `rcu_data`. Αυτές οι δομές είναι per-CPU (βλ. Ενότητα 4.2.2), και συμπεριλαμβάνουν, μεταξύ άλλων:

- Τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος που αυτός ο επεξεργαστής έχει δει. Χρησιμοποιείται για να εντοπιστεί το τέλος μιας περιόδου χάριτος.
- Τον τελευταίο αριθμό περιόδου χάριτος που αυτός ο επεξεργαστής γνωρίζει ότι ξεκίνησε.
- Μία `bool` μεταβλητή που δείχνει κατά πόσο αυτός ο επεξεργαστής πέρασε από κατάσταση ηρεμίας για την τρέχουσα περίοδο χάριτος.
- Έναν δείκτη στο φύλλο της ιεραρχίας που αντιστοιχεί στον επεξεργαστή αυτόν.
- Μία μάσκα που θα εφαρμοστεί στην μάσκα του κόμβου-φύλλου.
- Μεταβλητές σχετικές με την διαχείριση των επανακλήσεων του επεξεργαστή αυτού, συμπεριλαμβανομένης της λίστας επανακλήσεων του επεξεργαστή.
- Μεταβλητές σχετικές με τη διαπροσωπεία της λειτουργίας `dynticks`, καθώς και ένα δείκτη στη δομή `rcu_dynticks` του επεξεργαστή αυτού (θα εξηγηθεί στη συνέχεια).
- Μεταβλητές με στατιστικά στοιχεία.
- Μεταβλητές σχετικές με την αφαίρεση επεξεργαστών ενώ το σύστημα είναι σε λειτουργία (CPU hot plugs).
- Έναν δείκτη στην δομή καθολικής κατάστασης του RCU (`rcu_state`).

Γίνεται προφανές ότι όταν ένας επεξεργαστείς καταχωρεί μια επανάκληση, η επανάκληση αυτή αποθηκεύεται στην αντίστοιχη per-CPU δομή. Σημειώστε ότι, στο Tree RCU, η υλοποίηση της συνάρτησης `synchronize_rcu()` χρησιμοποιεί τη συνάρτηση `call_rcu()`.

Φυσικά, όπως έχουμε ήδη αναφέρει, όταν ένας επεξεργαστής περνάει από κατάσταση ηρεμίας, πρέπει να το αναφέρει στο RCU μηδενίζοντας το bit του στον αντίστοιχο κόμβο-φύλλο. Η ιεραρχία αποτελείται από δομές `rcu_node`, οι οποίες περιλαμβάνουν, μεταξύ άλλων:

- Ένα κλείδωμα για την προστασία του κόμβου.
- Τον αριθμό της τρέχουσας περιόδου χάριτος για αυτόν τον κόμβο.
- Τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος για τον κόμβο αυτό.
- Μια μάσκα που υποδεικνύει ποιοι επεξεργαστές πρέπει να αναφέρουν κατάσταση ηρεμίας στο RCU για αυτήν την περίοδο χάριτος. Σε κόμβους-φύλλα κάθε bit αντιστοιχεί σε μία δομή `rcu_data`, ενώ σε εσωτερικούς κόμβους κάθε bit αντιστοιχεί σε έναν κόμβο-παιδί.
- Μία μάσκα που θα εφαρμοστεί στην μάσκα του κόμβου πατέρα.

- Τον αύξων αριθμό του πρώτου και του τελευταίου επεξεργαστή ή ομάδας για τον κόμβο αυτό.
- Το επίπεδο του κόμβου αυτού στην ιεραρχία.
- Έναν δείκτη στον κόμβο-πατέρα αυτού του κόμβου.

Τέλος, η καθολική κατάσταση του RCU, καθώς και η ιεραρχία των κόμβων περιλαμβάνονται σε μία δομή `rcu_state`. Η ιεραρχία των κόμβων αναπαρίσταται με έναν πίνακα, ο οποίος δεσμεύεται στατικά κατά το χρόνο μεταγλώττισης βασει του `NR_CPUS` και ορισμένων επιλογών `Kconfig`. Να σημειωθεί ότι τα μικρά συστήματα θα έχουν μια ιεραρχία που αποτελείται μονάχα από έναν κόμβο. Η δομή `rcu_state` περιλαμβάνει, μεταξύ άλλων:

- Την ιεραρχία των κόμβων.
- Έναν πίνακα δεικτών στα επίπεδα της ιεραρχίας, καθώς και τον αριθμό των επιπέδων και την αναλογία παιδιών ανά κόμβο σε κάθε επίπεδο.
- Έναν δείκτη στην per-CPU δομή `rcu_data`.
- Τον αριθμό της τρέχουσας περιόδου χάριτος.
- Τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος.
- Έναν δείκτη στη δομή `task_struct` του νήματος της περιόδου χάριτος (το νήμα της περιόδου χάριτος θα εξηγηθεί στην Ενότητα 4.1.3).
- Την ουρά αναμονής (wait queue) όπου περιμένει το νήμα της περιόδου χάριτος.
- Μία μεταβλητή που αντιπροσωπεύει εντολές προς το νήμα της περιόδου χάριτος.
- Ένα κλείδωμα που συντονίζει τις αφαιρέσεις επεξεργαστών από το σύστημα κατά τη διάρκεια λειτουργίας (CPU hot plugs) με τις περιόδους χάριτος.
- Μία λίστα με όλες τις διαθέσιμες εκδόσεις του RCU.

Όπως θα έχει ήδη παρατηρηθεί, υπάρχουν αρκετές τιμές που διαδίδονται μεταξύ των παραπάνω διαφορετικών δομών δεδομένων, για παράδειγμα, ο αριθμός της τρέχουσας περιόδου χάριτος. Ο λόγος γι' αυτό θα γίνει προφανής στην Ενότητα 4.1.3. Ωστόσο, αυτό δεν συνέβαινε πάντα, και ήταν η ανακάλυψη σφαλμάτων που οδήγησε στις ανάλογες αλλαγές στον πηγαίο κώδικα.

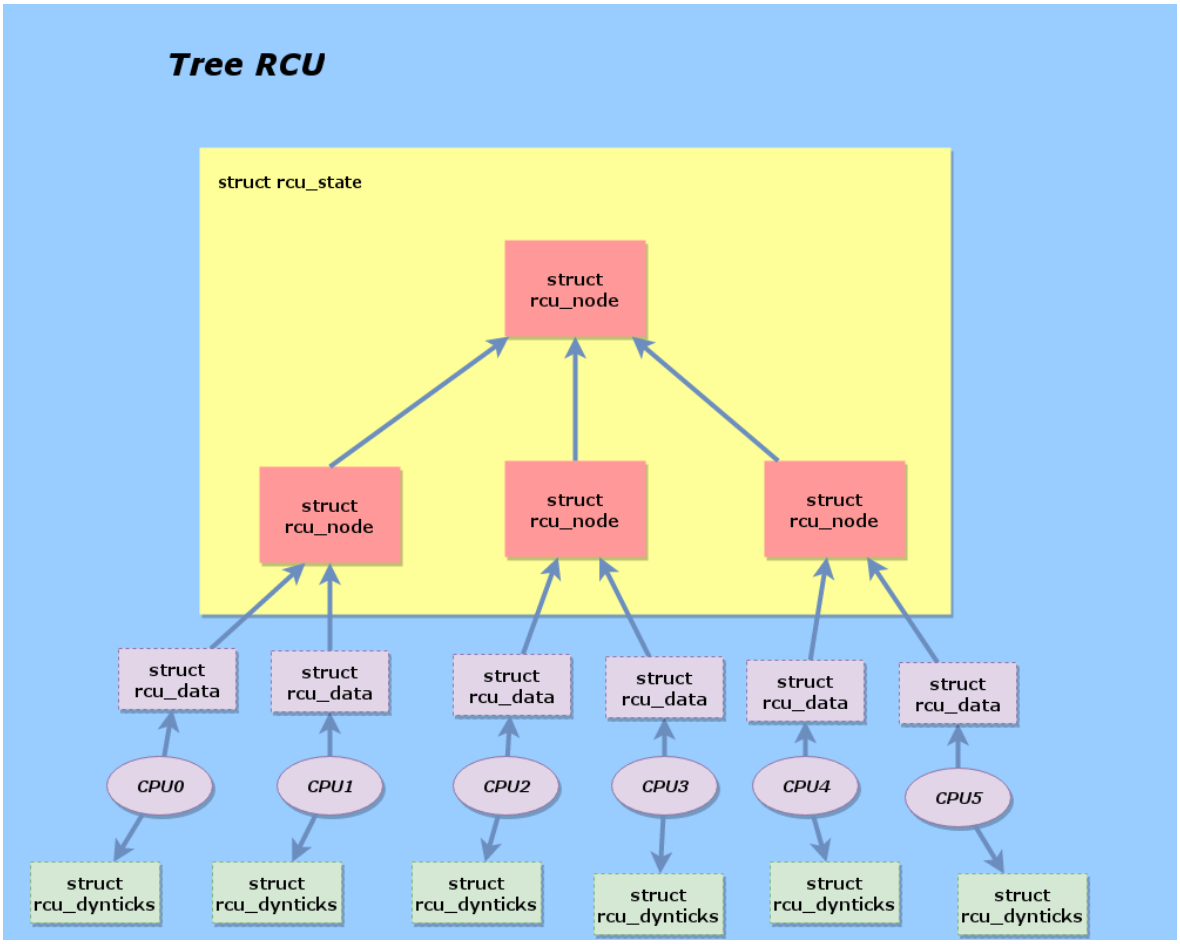
Σύμφωνα με όσα αναφέραμε στην αρχή του παρόντος κεφαλαίου, το Classic RCU δεν αλληλεπιδρούσε με τη λειτουργία `dynticks` με τον βέλτιστο τρόπο, κι αυτός ήταν ένας από τους κύριους λόγους που οδήγησαν στη δημιουργία του Tree RCU (με άλλα λόγια, η εξοικονόμηση ενέργειας).

Ευτυχώς, το Tree RCU αποφεύγει το να ξυπνάει επεξεργαστές που βρίσκονται σε λειτουργία `dynticks-idle`, εξοικονομώντας έτσι αρκετή ενέργεια. Για να είναι σε θέση όμως να το καταφέρει αυτό, το Tree RCU χρησιμοποιεί μια τέταρτη δομή δεδομένων, την per-CPU δομή `rcu_dynticks`. Η δομή αυτή περιλαμβάνει, μεταξύ άλλων:

- Έναν μετρητή που ακολουθεί το επίπεδο φωλιάσματος διεργασιών/διακοπών.
- Έναν μετρητή που ακολουθεί το επίπεδο φωλιάσματος μη απενεργοποιήσιμων διακοπών (NMIs).
- Έναν μετρητή που περιέχει άρτια τιμή όταν ο συγκεκριμένος επεξεργαστής είναι σε λειτουργία `dynticks-idle`, ή περιττή τιμή διαφορετικά.

Οι μετρητές αυτοί επιτρέπουν στο Tree RCU να περιμένει μόνο για επεξεργαστές που δεν κοιμούνται, και να αφήνει τους επεξεργαστές που κοιμούνται να κοιμούνται. Ο τρόπος που αυτό επιτυγχάνεται περιγράφεται στην Ενότητα 4.1.3.

Στο Σχήμα 4.3, παρουσιάζεται μια πληρέστερη έκδοση της ιεραρχίας που περιγράφηκε στην ενότητα 4.1.1. Ο τρόπους που οι επεξεργαστές αλληλεπιδρούν με τις δομές δεδομένων του σχήματος αυτού θα περιγραφεί στην επόμενη ενότητα.



Σχήμα 4.3: Ιεραρχία κόμβων για το Tree RCU – πλήρες σχήμα (προσαρμοσμένο από [McKe08a]).

4.1.3 Περιπτώσεις Χρήσης του Tree RCU

Η τυπική ακολουθία ενημερώσεων με χρήση του RCU περιλαμβάνει την καταχώριση μιας επανάκλησης, την αναμονή για την ολοκλήρωση όλων των προϋπαρχόντων αναγνωστών, και τέλος, την εκτέλεση της επανάκλησης. Κατά τη διαδικασία αυτή πρέπει να ληφθεί ειδική μέριμνα για τους επεξεργαστές που κοιμούνται, τους απενεργοποιημένους (offline) επεξεργαστές, τους επεξεργαστές που αφαιρούνται ενώ το σύστημα είναι σε λειτουργία (CPU hot plugs), τους επεξεργαστές που εκτελούν κώδικα χώρου χρήστη, καθώς και για τους επεξεργαστές που δεν έχουν αναφέρει κατάσταση ηρεμίας στο RCU εντός κάποιου προκαθορισμένου εύλογου διαστήματος. Ακόμη, υπάρχουν πολλές συναρτήσεις και δομές δεδομένων που εμπλέκονται στην προαναφερθείσα διαδικασία (βλ. Ενότητα 4.1.2). Στις επόμενες υποενότητες θα περιγράψουμε τόσο μερικές περιπτώσεις χρήσης του RCU[2], όσο και την αλληλεπίδραση τους RCU με τις εμπλεκόμενες συναρτήσεις και δομές δεδομένων.

[2] Οι περιπτώσεις χρήσης αυτές αφορούν τόσο στο RCU-sched, όσο και στο RCU-bh.

Καταχώριση μιας Επανάκλησης

Ένας επεξεργαστής μπορεί να καταχωρίσει μία επανάκληση καλώντας τη συνάρτηση `call_rcu()`. Η συνάρτηση αυτή καταχωρίζει σε μία ουρά μία συνάρτηση προς επανάκληση μετά από κάποια περίοδο χάριτος. Η επανάκληση τοποθετείται στην αντίστοιχη δομή `rcu_data` για αυτόν τον επεξεργαστή. Η δομή αυτή περιέχει μία λίστα με τις επανακλήσεις του συγκεκριμένου επεξεργαστή. Η λίστα επανακλήσεων είναι χωρισμένη σε τέσσερα τμήματα:

1. Το πρώτο τμήμα περιέχει καταχωρίσεις που είναι έτοιμες προς κλήση (τμήμα `DONE`).
2. Το δεύτερο τμήμα περιέχει καταχωρίσεις που αναμένουν την ολοκλήρωση της τρέχουσας περιόδου χάριτος (τμήμα `WAIT`).
3. Το τρίτο τμήμα περιέχει καταχωρίσεις που είναι γνωστό ότι έχουν φτάσει πριν το τέλος της παρούσας περιόδου χάριτος (τμήμα `NEXT_READY`).
4. Το τέταρτο τμήμα περιέχει καταχωρίσει που *ίσως* έχουν φτάσει μετά το τέλος της τρέχουσας περιόδου χάριτος (τμήμα `NEXT`).

Όταν μια καινούργια επανάκληση προστίθεται στη λίστα εισάγεται στο τέλος του τέταρτου τμήματος.

Σε παλαιότερες εκδόσεις του πυρήνα (για παράδειγμα, v2.6.x), η `call_rcu()` μπορούσε να ξεκινήσει μία νέα περίοδο χάριτος (βλ. επόμενη περίπτωση χρήσης), αλλά αυτό δεν συμβαίνει πια. Σε νεότερες εκδόσεις του πυρήνα, ο μόνος τρόπος να ξεκινήσει μία νέα περίοδος χάριτος από την `call_rcu()` είναι αν υπάρχουν πάρα πολλές καταχωριμένες επανακλήσεις και καμία εν ενεργεία περίοδος χάριτος. Διαφορετικά, οι περίοδοι χάριτος ξεκινούν σε περιβάλλον softirq (softirq context). Αλλά τι ακριβώς σημαίνει το παραπάνω;

Γενικά, κάθε softirq συσχετίζεται με μία συγκεκριμένη συνάρτηση η οποία εκτελείται κάθε φορά που εξυπηρετείται ο συγκεκριμένος τύπος softirqs. Για το Tree RCU, η συνάρτηση αυτή είναι η `rcu_process_callbacks()`. Οπότε, όταν ενεργοποιείται το softirq για το RCU, κάποια στιγμή θα κληθεί η `rcu_process_callbacks()` (αυτό θα γίνει είτε κατά την έξοδο από έναν χειριστή διακοπών, είτε από ένα νήμα ksoftirq/n), και θα ξεκινήσει μία νέα περίοδος χάριτος, αν ο αντίστοιχος επεξεργαστής χρειάζεται μία (για παράδειγμα αν δεν υπάρχει περίοδος χάριτος εν ενεργεία και ο αντίστοιχος επεξεργαστής έχει νέες καταχωρίσεις επανακλήσεων, ή υπάρχουν επακλήσεις που χρειάζονται μία επιπρόσθετη περίοδο χάριτος). Τα softirqs του RCU ενεργοποιούνται από τη συνάρτηση `rcu_check_callbacks()` η οποία καλείται από τις διακοπές του ρολογιού (πιο συγκεκριμένα, από τη συνάρτηση `update_process_times()`). Αν υπάρχει εκκρεμής δουλειά σχετική με το RCU (λόγου χάρη, κάποιος επεξεργαστής χρειάζεται μία νέα περίοδο χάριτος), η συνάρτηση `rcu_check_callbacks()` ενεργοποιεί ένα softirq.

Τέλος, αξίζει να σημειωθεί ότι στο Tree RCU η `synchronize_rcu()` υλοποιείται με χρήση της `call_rcu()`. Πιο συγκεκριμένα, η `synchronize_rcu()` καταχωρίζει μία επανάκληση που θα ξυπνήσει το καλούν νήμα όταν μία περίοδος χάριτος έχει ολοκληρωθεί. Το καλούν νήμα περιμένει σε μία μεταβλητή ολοκλήρωσης (completion variable) του πυρήνα, και για το λόγο αυτό τοποθετείται σε μία ουρά αναμονής (wait queue).

Έναρξη μίας Περιόδου Χάριτος

Η συνάρτηση `rcu_start_gp()` είναι υπεύθυνη για την έναρξη μιας νέας περιόδου χάριτος. Όπως έχουμε ήδη αναφέρει, αυτή η συνάρτηση καλείται από την `rcu_process_callbacks()` σε περιβάλλον softirq.

Ωστόσο, σε νεότερες εκδόσεις του πυρήνα, η συνάρτηση αυτή δεν πραγματοποιεί την έναρξη της νέας περιόδου χάριτος η ίδια, ούτε αρχικοποιεί τις δομές δεδομένων που

χρειάζονται αρχικοποίηση. Το μόνο που κάνει είναι να αναπροσαρμόζει κατάλληλα τα τμήματα στην λίστα επανακλήσεων του επεξεργαστή στον οποίον τρέχει, και να θέτει την κατάλληλη τιμή σε μία μεταβλητή-σημαία στη δομή `rcu_state`, για να υποδείξει ότι ένας επεξεργαστής έχει ανάγκη για μια νέα περίοδο χάριτος. Το νήμα πυρήνα για τις περιόδους χάριτος όμως είναι αυτό που θα αρχικοποιήσει την ιεραρχία των κόμβων, τις δομές `rcu_data` και τη δομή `rcu_state`, και κατ' επέκτασιν θα ξεκινήσει τη νέα περίοδο χάριτος. Σημειώστε ότι υπάρχουν δύο νήματα πυρήνα για τις περιόδους χάριτος: ένα για το RCU-sched κι ένα για το RCU-bh.

Το νήμα πυρήνα για τις περιόδους χάριτος αποκλείει ταυτόχρονες αφαιρέσεις επεξεργαστών (CPU hot plugs) κι έπειτα θέτει τα bit στις μάσκες όλων των κόμβων της ιεραρχίας που αντιστοιχούν σε επεξεργαστές που είναι σε λειτουργία (online), για να υποδείξει ότι οι αντίστοιχοι επεξεργαστές πρέπει να περάσουν από κατάσταση ηρεμίας. Επίσης αντιγράφει τον αριθμό της νέας περιόδου χάριτος και της τελευταίας ολοκληρωμένης περιόδου χάριτος σε όλους τους κόμβους της ιεραρχίας (βλ. και επόμενες περιπτώσεις χρήσης). Ταυτόχρονες προσβάσεις στις δομές των κόμβων της ιεραρχίας προφανώς αφορούν μόνο τους κόμβους-φύλλα, και άλλοι επεξεργαστές μπορεί να δουν τον κόμβο που τους αντιστοιχεί αρχικοποιημένο ή όχι. Αλλά κάθε επεξεργαστής πρέπει να εκτελέσει κώδικα που αφορά το RCU σε softirq περιβάλλον για να αντιληφθεί την έναρξη μιας περιόδου χάριτος. Αυτό σημαίνει ότι κάθε επεξεργαστής εκτός αυτού στον οποίον εκτελείται το νήμα πυρήνα για τις περιόδους χάριτος πρέπει να εισέλθει σε περιβάλλον softirq για να δει την έναρξη μιας νέας περιόδου χάριτος (μέσω της συνάρτησης `rcu_check_callbacks()`).

Το νήμα πυρήνα για τις περιόδους χάριτος έλυσε διάφορες καταστάσεις ανταγωνισμού (race conditions) παρούσες σε παλαιές εκδόσεις του πυρήνα, όπου όταν ένας επεξεργαστής χρειαζόταν μία νέα περίοδο χάριτος, προσπαθούσε να αρχικοποιήσει την ιεραρχία των κόμβων απευθείας, κάτι που μπορούσε να οδηγήσει -μέσω πολύπλοκων σεναρίων- σε σφάλματα (βλ. Ενότητα 4.3.1).

Πέρασμα από Κατάσταση Ηρεμίας

Οι καταστάσεις ηρεμίας για το Tree RCU (RCU-sched) είναι:

- Η χρονοδρομολόγηση (context switch),
- Η λειτουργία idle (είτε idle loop, είτε dynticks-idle), και
- Η εκτέλεση κώδικα χώρου χρήση

Όταν ένας επεξεργαστής περνάει από κατάσταση ηρεμίας, ενημερώνει την δομή `rcu_data` που του αντιστοιχεί καλώντας τη συνάρτηση `rcu_sched_qs` (ή `rcu_qsctr_inc()` για παλαιότερες εκδόσεις του πυρήνα). Η συνάρτηση αυτή καλείται από συναρτήσεις σχετικές με τον χρονοδρομολογητή (αλλά και από τον ίδιο τον χρονοδρομολογητή), από τη συνάρτηση `rcu_check_callbacks()` (η οποία καλείται κατά την εξυπηρέτηση διακοπών ρολογιού), και από τα νήματα πυρήνα `ksoftirq/n`.

Όμως, το γεγονός ότι ένας επεξεργαστής πέρασε από κατάσταση ηρεμίας δεν σημαίνει ότι το RCU το γνωρίζει αυτό. Εξάλλου, το γεγονός αυτό σημειώνεται τοπικά στην αντίστοιχη per-CPU δομή, και όχι στην ιεραρχία των κόμβων. Έτσι, ένας επεξεργαστής οφείλει να ενημερώσει το RCU ότι πέρασε από κατάσταση ηρεμίας, και αυτό θα γίνει σε περιβάλλον softirq, μέσω της συνάρτησης `rcu_process_callbacks()`. Η λειτουργία της συνάρτησης αυτής περιγράφεται στην επόμενη περίπτωση χρήσης.

Αναφορά Κατάστασης Ηρεμίας στο RCU

Αφού ένας επεξεργαστής περάσει από κατάσταση ηρεμίας, πρέπει να το αναφέρει στο RCU. Αυτό γίνεται μέσω της συνάρτησης `rcu_process_callbacks()`, η οποία τρέχει σε περιβάλλον softirq. Η συνάρτηση αυτή έχει πολλά καθήκοντα, τα οποία περιλαμβάνουν:

- Το ξύπνημα του νήματος πυρήνα για τις περιόδους χάριτος με κλήση της `rcu_start_gp()`, έτσι ώστε να αρχικοποιήσει και να ξεκινήσει αυτό μία νέα περίοδο χάριτος, αν υπάρχει ανάγκη για μία.
- Την αναγνώριση έναρξης/λήξης μιας νέας περιόδου χάριτος. Όπως έχουμε ήδη αναφέρει εν συντομία, κάθε επεξεργαστής εκτός αυτού στον οποίο τρέχει το νήμα πυρήνα για τις περιόδους χάριτος πρέπει να εισέλθει σε περιβάλλον softirq και να τρέξει κώδικα του RCU για να δει την έναρξη/λήξη μίας περιόδου χάριτος. Η αναγνώριση αυτή γίνεται μέσω της συνάρτησης `rcu_check_quiescent_state()`, η οποία με τη σειρά της καλεί την συνάρτηση `note_gp_changes()`. Η τελευταία αναπροσαρμόζει κατάλληλα τα τμήματα της λίστας επανακλήσεων του αντίστοιχου επεξεργαστή, και γράφει στην κατάλληλη δομη `rcu_data` όλες τις απαραίτητες πληροφορίες σχετικά με την έναρξη/λήξη της περιόδου χάριτος.
- Την αναφορά ότι ο συγκεκριμένος επεξεργαστής πέρασε από κατάσταση ηρεμίας (μέσω της συνάρτησης `rcu_check_quiescent_state()`, η οποία καλεί την `rcu_report_qs_rdp()` – `cpu_quiet()` για παλαιότερες εκδόσεις του πυρήνα). Μία περιγραφή υψηλού επιπέδου για τη διαδικασία αυτή έχει δοθεί στην Ενότητα 4.1.1. Αν ο τρέχων επεξεργαστής είναι ο τελευταίος που αναφέρει κατάσταση ηρεμίας στο RCU, το νήμα πυρήνα για τις περιόδους χάριτος ξυπνάει ξανά, με σκοπό τον καθαρισμό των διαφόρων δομών δεδομένων για τη λήξη της περιόδου χάριτος, και την διάδοση της τιμής `->completed` σε όλους τους κόμβους της ιεραρχίας.
- Την εκτέλεση όλων των επανακλήσεων των οποίων η περίοδος χάριτος τελείωσε, μέσω της `rcu_do_batch()`.

Όπως είναι φανερό, το νήμα πυρήνα για τις περιόδους χάριτος χρησιμοποιείται κατά κόρον για το συντονισμό των ενάρξεων και των λήξεων των περιόδων χάριτος. Πέρα από αυτό, τα κλειδώματα των κόμβων στην ιεραρχία χρησιμοποιούνται για την προστασία από ταυτόχρονες προσβάσεις στους κόμβους, οι οποίες μπορεί να οδηγήσουν σε προβλήματα (βλ. Ενότητα 4.3).

Είσοδος/Έξοδος σε/από λειτουργία Dynticks-Idle

Όταν ένας επεξεργαστής εισέρχεται σε λειτουργία `dynticks-idle`, καλείται η συνάρτηση `rcu_idle_enter()` (`rcu_enter_nohz()` για παλαιότερες εκδόσεις του πυρήνα). Η συνάρτηση αυτή μειώνει μία per-CPU μεταβλητή φωλιάσματος (`dynticks_nesting`) και αυξάνει έναν per-CPU μετρητή (`dynticks`), μεταβλητές που βρίσκονται στην per-CPU δομή `rcu_dynticks`. Ο μετρητής `dynticks` πρέπει να έχει άρτια τιμή όταν ο επεξεργαστής είναι σε λειτουργία `dynticks-idle`.

Όταν ένας επεξεργαστής εξέρχεται από τη λειτουργία `dynticks-idle`, καλείται η συνάρτηση `rcu_idle_exit()` (`rcu_exit_nohz()` για παλαιότερους πυρήνες). Η συνάρτηση αυτή αυξάνει τη μεταβλητή `dynticks_nesting` και τον μετρητή `dynticks`, ο οποίος τώρα πρέπει να έχει περιττή τιμή.

Φυσικά, γνωρίζουμε ότι η λειτουργία `dynticks-idle` είναι κατάσταση ηρεμίας για το Tree RCU. Ο λόγος, λοιπόν, που χρειάζονται οι δύο αυτές μεταβλητές είναι για να μπορούν να ληφθούν δείγματα αυτών από άλλους επεξεργαστές. Με τον τρόπο αυτόν, μπορούμε να αποφανθούμε με ασφάλεια αν ένας επεξεργαστής είναι σε κατάσταση ηρεμίας (αν

ο μετρητής dynticks έχει άρτια τιμή), ή αν ένας επεξεργαστής πέρασε από κατάσταση ηρεμίας κάποια στιγμή κατά τη διάρκεια της τρέχουσας περιόδου χάριτος (αν η τιμή της μεταβλητής dynticks άλλαξε κατά τη διάρκεια της περιόδου χάριτος). Η διαδικασία της δειγματοληψίας γίνεται όταν ένας επεξεργαστής δεν έχει αναφέρει κατάσταση ηρεμίας στο RCU για ένα μεγάλο χρονικό διάστημα και η περίοδος χάριτος πρέπει να τελειώσει (βλ. επόμενες περιπτώσεις χρήσης). Στην πραγματικότητα, υπάρχουν και κάποιες ακόμη μεταβλητές των οποίων δείγματα πρέπει να ληφθούν (λόγου χάρη, η μεταβλητή dynticks_nmi), ωστόσο, η διαδικασία της δειγματοληψίας δεν θα συζητηθεί εδώ.

Διακοπές και Λειτουργία dynticks-idle

Όταν ένας επεξεργαστής εισέρχεται σε έναν χειριστή διακοπών, καλείται η συνάρτηση rcu_irq_enter(). Η συνάρτηση αυτή αυξάνει τη μεταβλητή dynticks_nesting, και αν η προηγούμενη τιμή της μεταβλητής αυτής ήταν μηδέν (κάτι που σημαίνει ότι ο επεξεργαστής είναι σε λειτουργία dynticks−idle), αυξάνει και τον μετρητή dynticks (ο οποίος τώρα πρέπει να έχει περιττή τιμή).

Όταν ένας επεξεργαστής εξέρχεται από έναν χειριστή διακοπών, η συνάρτηση rcu_irq_exit() μειώνει τη μεταβλητή dynticks_nesting, και αν η νέα τιμμή είναι μηδέν (κάτι που σημαίνει ότι ο επεξεργαστής εισέρχεται σε λειτουργία dynticks−idle, αυξάνει και τον μετρητή dynticks (ο οποίος τώρα πρέπει να έχει άρτια τιμή).

Είναι προφανές ότι η είσοδος σε έναν χειριστή διακοπών από τη λειτουργία dynticks−idle σηματοδοτεί την έξοδο από τη λειτουργία dynticks−idle. Αντίστροφα, η έξοδος από έναν χειριστή διακοπών μπορεί να σηματοδοτήσει την είσοδο σε λειτουργία dynticks−idle.

Εξαναγκασμός Καταστάσεων Ηρεμίας

Αν υπάρχουν επεξεργαστές που δεν έχουν αναφέρει κατάσταση ηρεμίας στο RCU και έχουν περάσει αρκετά jiffies, τότε το νήμα πυρήνα για τις περιόδους χάριτος ξυπνάει (το νήμα αυτό ήταν σε ουρά αναμονής, αλλά υπάρχει συγκεκριμένος χρόνος που μπορεί να μείνει σε αυτήν) και θα προσπαθήσει να εξαναγκάσει καταστάσεις ηρεμίας στους επεξεργαστές που δεν έχουν ακόμη αναφέρει μία. Πιο συγκεκριμένα, καλείται η συνάρτηση rcu_gp_fqs(). Η συνάρτηση αυτή δουλεύει σε δύο φάσεις:

- Στην πρώτη φάση (RCU_SAVE_DYNTICK), συλλέγονται δείγματα από τους μετρητές dynticks όλω των επεξεργαστών, μέσω της συνάρτησης dyntick_save_progress_counter(), η οποία καλείται από τη συνάρτηση force_qs_rnp(). Η πρώτη παίρνει δείγματα από τους μετρητές των επεξεργαστών για να τους πιστώσει με "σιωπηρές" (implicit) καταστάσεις ηρεμίας. Στην περίπτωση όπου ένας επεξεργαστής είναι σε λειτουργία dynticks−idle η συνάρτηση αυτή επιστρέφει 1, έτσι ώστε η force_qs_rnp() να καλέσει την rcu_report_qs_rnp(), η οποία αναφέρει κατάσταση ηρεμίας στο RCU.

- Στη δεύτερη φάση (RCU_FORCE_QS), λαμβάνονται ξανά δείγματα από τους επεξεργαστές που δεν έχουν αναφέρει ακόμη κατάσταση ηρεμία στο RCU, μέσω της συνάρτησης rcu_implicit_dynticks_qs(), η οποία καλείται από τη συνάρτηση force_qs_rnp(). Η συνάρτηση rcu_implicit_dynticks_qs() ελέγχει τους μετρητές των επεξεργαστών στις αντίστοιχες rcu_dynticks δομές για να διαπιστώσει αν έχουν περάσει από κατάσταση ηρεμίας. Αν ένας επεξεργαστής έχει περάσει από κατάσταση ηρεμίας, ή είναι τώρα σε λειτουργία dynticks−idle, η συνάρτηση αυτή επιστρέφει 1, με σκοπό η force_qs_rnp() να καλέσει την rcu_report_qs_rnp(). Αν υπάρχουν ακόμη επεξεργαστές οι οποίοι δεν έχουν αναφέρει κατάσταση ηρεμίας, αυτοί υφίστανται αναγκαστική χρονοδρομολόγηση με σκοπό να αναφέρουν κατάσταση ηρεμίας στο RCU.

4.2 Μοντελοποίηση του Περιβάλλοντος του Πυρήνα

Ακολουθώντας μια παρόμοια λογική με αυτήν του Κεφαλαίου 3.2, στην ενότητα αυτή παρουσιάζουμε τον τρόπο που μοντελοποιήσαμε ένα μη-προεκχωρητικό περιβάλλον πυρήνα μιας πολυπύρηνης αρχιτεκτονικής. Όπως θα δούμε, οι μόνες αλλαγές στον κώδικα του Tree RCU αφορούν την αντικατάσταση των per-CPU μεταβλητών με πίνακες. Ο υπόλοιπος πηγαίος κώδικας του Tree RCU παρέμεινε ανέπαφος.

4.2.1 Επεξεργαστής, Διακοπές & Χρονοδρομολόγηση

Επεξεργαστής

Πρώτα απ' όλα, για να μοντελοποιήσουμε μία πολυπύρηνη αρχιτεκτονική χρησιμοποιήσαμε ένα κλείδωμα (`pthread_mutex_lock`) για κάθε επεξεργαστή το οποίο, όταν κρατείται, το αντίστοιχο νήμα τρέχει τον επεξεργαστή του κλειδώματος.

Όπως και στον συστηματικό έλεγχο ορθότητας του Tiny RCU, υποθέτουμε ότι όλοι οι επεξεργαστές είναι εν ενεργεία (online), οι αφαιρέσεις επεξεργαστών εν ώρα λειτουργίας του συστήματος (CPU hot plugs) απαγορεύονται, και `CONFIG_NO_HZ_FULL=n`. Όλοι οι επεξεργαστές ξεκινούν σε λειτουργία `dynticks-idle`, και όταν ένα νήμα επιθυμεί να τρέξει σε έναν επεξεργαστή καλεί τη συνάρτηση `fake_acquire_cpu()`, της οποίας ο κώδικας φαίνεται στον Κώδικα 4.1.

```
1  void fake_acquire_cpu(int cpu)
2  {
3          if (pthread_mutex_lock(&cpu_lock[cpu]))
4                  exit(-1);
5          rcu_idle_exit();
6  }
```

Listing 4.1: Κατάληψη ενός επεξεργαστή από ένα νήμα για το Tree RCU.

Υποθέτουμε ότι ο επεξεργαστής του οποίου το κλείδωμα προσπαθούμε να πάρουμε είναι σε λειτουργία `dynticks-idle`, οπότε καλείται η συνάρτηση `rcu_idle_exit()`. Φυσικά, αν ο επιθυμητός επεξεργαστής δεν είναι σε λειτουργία `dynticks-idle`, μπορούμε να πάρουμε κατευθείαν το κλείδωμά του.

Παρομοίως, όταν ένα νήμα τελειώνει την εκτέλεσή του (ή θέλει να αφήσει έναν επεξεργαστή ο οποίος και θα εισέλθει σε λειτουργία `dynticks-idle`), καλείται η συνάρτηση `fake_release_cpu()` (βλ. Κώδικα 4.2.

```
1  void fake_release_cpu(int cpu)
2  {
3          rcu_idle_enter();
4          if (pthread_mutex_unlock(&cpu_lock[cpu]))
5                  exit(-1);
6  }
```

Listing 4.2: Απελευθέρωση ενός επεξεργαστή από ένα νήμα για το Tree RCU.

Ο αριθμός των επεξεργαστών στο σύστημα μπορεί να οριστεί ορίζοντας στον προεπεξεργαστή το macro `CONFIG_NR_CPUS=x` κατά τη μεταγλώττιση ενός προγράμματος ελέγχου, όπου `x` είναι ο επιθυμητός αριθμός επεξεργαστών.

Τι γίνεται όμως με τις per-CPU μεταβλητές που υπάρχουν στον πυρήνα; Η δημιουργία μιας per-CPU μεταβλητής σημαίνει ότι ο κάθε επεξεργαστής θα έχει το δικό του αντίγραφο της μεταβλητής αυτής, αλλά ένας επεξεργαστής θα μπορεί να διαβάσει το αντίγραφο ενός άλλου επεξεργαστή (ή να αποκτήσει έναν δείκτη προς το αντίγραφο αυτό).

Θεωρητικά, υπάρχουν ορισμένες οδηγίες προς τον μεταγλωττιστή και τον συνδέτη-/φορτωτή που μπορούν να χρησιμοποιηθούν μαζί με προσεκτικά γραμμένο κώδικα προ-επεξεργαστή της c. Ωστόσο, επειδή αυτό απαιτεί σημαντική υποστήριξη από το περιβάλλον εκτέλεσης (στον πυρήνα υπάρχουν συναρτήσεις αρχικοποίησεις που φορτώνουν τους τομείς per-CPU πολλές φορές, μία ανά επεξεργαστή), χρησιμοποιήσαμε πίνακας για τη μοντελοποίηση των per-CPU μεταβλητών. Φυσικά, αυτό οδήγησε σε αρκετές μεν, ασήμαντες δε, διαφορές σε όλα τα αρχεία που χρησιμοποιούν per-CPU μεταβλητές.

Τέλος, επειδή κάθε νήμα πρέπει να γνωρίζει ανά πάσα στιγμή τον επεξεργαστή στον οποίο τρέχει, υλοποιήσαμε δύο macros, το `set_cpu()` και το `get_cpu()` τα οποία μεταχειρίζονται μία μεταβλητή τοπική στο κάθε νήμα που δείχνει τον επεξεργαστή στον οποίο τρέχει το καθένα. Με τον τρόπο αυτό, ένα νήμα μπορεί να βρει τον επεξεργαστή στον οποίο εκτελείται, αλλά η μεταβλητή αυτή πρέπει να έχει οριστεί από τον προγραμματιστή (μέσω του `set_cpu()` εκ των προτέρων.

Διακοπές & Softirqs

Σειρά έχει η μοντελοποίηση των διακοπών και των softirqs. Υπάρχουν αρκετοί τρόποι για την μοντελοποίηση των διακοπών και των softirqs. Ένας τρόπος είναι να υπάρχει ένα κλείδωμα για διακοπές και softirqs ανά επεξεργαστή το οποίο πρέπει να κρατείται κατά την εκτέλεση του χειριστή διακοπών. Φυσικά, το ίδιο κλείδωμα πρέπει να κρατείται όταν ένας επεξεργαστής απενεργοποιεί τις διακοπές ή τα softirqs. Αυτό καλύπτει ένα υπερσύνολο των περιπτώσεων που θέλουμε να ελέγξουμε αλλά επαρκεί για τους σκοπούς μας. Εξάλλου, αφού έχουμε να κάνουμε με μη-προεκχωρητικό κώδικα, δεν θα υπάρχει ανταγωνισμός για το κλείδωμα αυτό. Ένας άλλος τρόπος για τη μοντελοποίηση των διακοπών/softirqs θα ήταν να έχουμε ένα ξεχωριστό νήμα για τις διακοπές και τα softirqs. Ωστόσο, επειδή αυτή η προσέγγιση θα έκανε τον χώρο καταστάσεων αισθητά μεγαλύτερο, διαλέξαμε την πρώτη προσέγγιση.

Για τη διαχείριση του `local_irq_depth` χρησιμοποιήθηκε ένας μετρητής ανά επεξεργαστή. Με τον τρόπο αυτόν η συνάρτηση `local_irq_disable()` θέτει το `local_irq_depth` σε 1, η `irq_disabled_flags()` επιστρέφει `!!local_irq_depth` (δηλαδή την τιμή του `local_irq_depth`), και η `local_irq_enable()` θέτει το `lcoal_irq_depth` σε 0. Αντίστοιχα, η `local_irq_restore()` θέτει το `local_irq_depth` στη τιμή που δόθηκε ως παράμετρος στην `local_irq_save()` όταν η τελευταία κλήθηκε. Φυσικά, έχουμε να λύσουμε και θέματα σχετικά με τον ανταγωνισμό για το κλείδωμα των διακοπών. Ένας τρόπος να διαχειριστούμε το ζήτημα αυτό είναι να παρίνουμε το κλείδωμα ααν η προηγούμενη τιμή του `local_irq_depth` ήταν μηδέν και η νέα τιμή μη-μηδενική, και αντίστροφα. Η προσέγγιση αυτή δουλεύει μόνο για μη-προεκχωρητικό κώδικα και θα χρειαζόντουσαν τροποποιήσεις σε περίπτωση που επιθυμούσαμε να επεκταθεί για να λειτουργεί και σε προεκχωρητικό κώδικα.

Όμως γνωρίζουμε ότι το RCU βασίζεται κυρίως σε διακοπές του ρολογιού (μιας και εκεί είναι το μέρος που καλείται η `rcu_check_callbacks()`), και όχι σε διακοπές γενικά, αλλά πώς μπορεί να μοντελοποιηθεί ο χρόνος δεδομένου ότι ένα εργαλείο ελέγχου μοντέλου χωρίς αποθήκευση της κατάστασης απαιτεί τα προγράμματα ελέγχου που δέχεται ως είσοδο να είναι πεπερασμένα; Η σύντομη ερώτηση στο ερώτημα αυτό είναι ότι όλα τα προγράμματα ελέγχου που χρησιμοποιήσαμε δεν έχουν επίγνωση του χρόνου. Πιο συγκεκριμένα, η ακριβής στιγμή που εξυπηρετείται μία διακοπή δεν μας ενδιαφέρει, μιας και έτσι κι αλλιώς ένας επεξεργαστής μπορεί να τρέχει για αρκετή ώρα με τις διακοπές ενεργοοπιημένες. Επιπρόσθετα, σε πολλές περιπτώσεις αυτό που μας ενδιαφέρει δεν είναι η ακριβής στιγμή που προκύπτει μία διακοπή, αλλά οι επιπτώσεις που μπορεί να έχει το περιβάλλον διακοπών σε ένα συγκεκριμένο σημείο, δεδομένου ενός περιβάλλοντος ταυτοχρονισμού. Και ενώ υπάρχουν μερικές ειδοποιήσεις και γεγονότα που η ενεργοποίηση τους είναι βασισμένη στο χρόνο, αυτά μπορούν να απενεργοποιηθούν ή να αγνοηθούν. Έτσι γίνεται και για τις προειδοποιήσεις καθυστέρησης του RCU (RCU stall warnings),

για τα οποία οι παράμετροι αρχικοποίησης (boot parameters) `rcu_task_stall_timeout = 0` και `rcu_cpu_stall_suppress = 1` έχουν τεθεί ανάλογα.

Ένα ακόμη σημείο κλειδί για το κλείδωμα των διακοπών είναι ότι, θεωρητικά, δεν πρέπει να κρατείται όταν εκτελούνται softirqs, μιας και αυτά εκτελούνται με τις διακοπές ενεργοποιημένες. Από την άλλη όμως, αυτό δεν θα έκανε ιδιαίτερη διαφορά σε μη-προεκχωρητικό κώδικα. Φυσικά, οι διακοπές και τα softirqs θα μπορούσαν να προσομοιωθούν με ένα ξεχωριστό νήμα αλλά, όπως ήδη αναφέραμε, αυτό θα έκανε τον χώρο καταστάσεων αρκετά μεγαλύτερο.

Τέλος, υποθέτουμε ότι ο πυρήνας μεταγλωττίζεται με τη λειτουργία `dynticks-idle` ενεργοποιημένη, και ότι ένα νήμα πρέπει να πάρει το κλείδωμα του επεξεργαστή προτού πάρει το κλείδωμα των διακοπών. Ακόμη, σε πραγματικές συνθήκες υπάρχει ένα νήμα που τρέχει όταν δεν τρέχει κανένα άλλο σε κάποιον επεξεργαστή. Υποθέτοντας ότι η λειτουργία `dynticks-idle` είναι ενεργοποιημένη (προεπιλογή), ο μόνος λόγος ύπαρξης των νημάτων αυτών είναι η διατήρηση ενέργειας. Ωστόσο, αφού δεν προσπαθούμε να πραγματοποιήσουμε συστηματικό έλεγχο ορθότητας στα νήματα αυτά, δεν υπάρχει λόγος να τα μοντελοποιήσουμε και να τα συμπεριλάβουμε στους ελέγχους μας.

Χρονοδρομολόγηση

Σε ό,τι αφορά στις συναρτήσεις χρονοδρομολόγησης, η `cond_resched()` μοντελοποιήθηκε κάνοντας το τρέχον νήμα να αφήσει το κλείδωμα του επεξεργαστή στον οποίο τρέχει και μετά -πιθανώς- να το ξαναπάρει, ακριβώς όπως στην περίπτωση του Tiny RCU. Η μόνη διαφορά εδώ είναι ότι πριν αφήσει το κλείδωμα του επεξεργαστή καλείται η συνάρτηση `rcu_note_context_switch()`.

Στην πραγματικότητα, ο καλύτερος τρόπος μοντελοποίησης της παραπάνω συνάρτηση θα ήταν ένα νήμα να αφήνει το κλείδωμα ενός επεξεργαστή και μετά να παίρνει το κλείδωμα ενός τυχαίου επεξεργαστή, πιθανώς όμως και του ίδιου. Ωστόσο, όπως έχουμε αναφέρει αρκετές φορές, τα προγράμματα ελέγχου που θα αναλυθούν πρέπει να είναι ντεντερμινιστικά με την έννοια ότι σε μια συγκεκριμένη κατάσταση, η εκτέλεση ενός βήματος εκτέλεσης πρέπει να οδηγεί το σύστημα πάντα στην ίδια νέα κατάσταση. Αυτό σημαίνει ότι το Nidhugg δεν μπορεί, για παράδειγμα, να ελέγχει την ώρα, ή να βρει σφάλματα που οφείλονται σε μη-ντετερμινισμό των δεδομένων, άρα η `cond_resched()` ουσιαστικά *πρέπει* να πάρει τον ίδιο επεξεργαστή με αυτόν που άφησε.

Αλλά, όπως θα δούμε στην Ενότητα 4.3, αυτό δεν μας εμπόδισε από το να αναπαράγουμε παλαιότερα σφάλματα με χρήση του Nidhugg, αφού ο κώδικας στα προγράμματα ελέγχου μας προσανατολίζεται γύρω από τον εκάστοτε επεξεργαστή και όχι γύρω από το εκάστοτε νήμα. Εξάλλου, τα κρίσιμα τμήματα αναγνωστών RCU δεν είναι προεκχωρητικά.

Επιπρόσθετα, υπάρχει και η συνάρτηση `resched_cpu()`. Για να μοντελοποιηθεί η συνάρτηση αυτή τέλεια, το νήμα το οποίο κρατάει το κλείδωμα ενός επεξεργαστή θα έπρεπε να αναγκάζεται να εκτελέσει την `cond_resched()`, αλλά σε ένα σημείο που τόσο οι διακοπές όσο και η προεκχωρητικότητα είναι ενεργοποιημένα.

Ωστόσο, επειδή η `resched_cpu()` χρησιμοποιείται μόνο στην περίπτωση που ένας επεξεργαστής έχει αργήσει πολύ να αναφέρει κατάσταση ηρεμίας στο RCU και τα προγράμματά μας δεν έχουν επίγνωση του χρόνου, η συνάρτηση αυτή στη μοντελοποίησή μας είναι απλά κενή.

4.2.2 Ορισμοί του Πυρήνα

Στην ενότητα αυτή θα περιγράψουμε τον τρόπο που προσομοιώσαμε το περιβάλλον ενός μη-προεκχωρητικού πυρήνα μιας πολυπήρηνης αρχιτεκτονικής. Για τα προγράμματα ελέγχου που κατασκευάσαμε εστιάσαμε σε τέσσερις διαφορετικές εκδόσεις του πυρήνα:

την έκδοση v3.19, την v2.6.31.1, την v2.6.32.1 και την v3.0. Η έκδοση του πυρήνα που θα συμπεριληφθεί στον έλεγχο μπορεί να επιλεχθεί απλά εστιάζοντας την προσοχή του προεπεξεργαστή της C στον ανάλογο κατάλογο αρχείων (directory). Οι λόγοι για την επιλογή των τεσσάρων αυτών εκδόσεων θα γίνουν προφανείς στις Ενότητες 4.3-4.5, αλλά το να έχουμε πολλές εκδόσεις του πυρήνα διαθέσιμες μας επιτρέπει να τρέξουμε το ίδιο πρόγραμμα ελέγχου με διαφορετικές παραμετροποιήσεις.

Όπως στο Κεφάλαιο 3, πολλοί ορισμοί αντιγράφηκαν απευθείας από τον πυρήνα. Σε αυτούς τους ορισμούς συμπεριλαμβάνονται τύποι δεδομένων όπως `u8`, `u16` κλπ, οδηγίες προς τον μεταγλωττιστή όπως `offsetof()`, μακροεντολές όπως `ACCESS_ONCE()`, τύποι δεδομένων και συναρτήσεις λιστών, φράχτες μνήμης (memory barriers), και διάφοροι άλλοι αρχέγονοι (primitive) τύποι και συναρτήσεις του πυρήνα.

Απ' την άλλη πλευρά όμως, πολλοί αρχέγονοι τύποι και συναρτήσεις έπρεπε να αντικατασταθούν με δικούς μας, ή η υλοποίησή τους να αντικατασταθεί με μία κενή. Για ακόμη μία φορά, λοιπόν, παρείχαμε άδεια αρχεία προς συμπερίληψη από τον προεπεξεργαστή, και δώσαμε δικούς μας ορισμούς βασισμένους σε ειδικές επιλογές `Kconfig` στο αρχείο `fake_defs.c`. Σε αυτούς συμπεριλαμβάνονται ορισμοί σχετικοί με τον επεξεργαστή (π.χ. `NR_CPUS`), ορισμοί σχετικοί με το RCU (π.χ. `CONFIG_RCU_BOOST`), ειδικές οδηγίες προς τον μεταγλωττιστή, συναρτήσεις ίχνους (trace functions) κλπ. Και εδώ, η μακροεντολή `BUG_ON()` και οι συγγενικές της μακροεντολές (π.χ. `WARN_ON()`) αντικαταστάθηκαν με εντολές `assert()`. Κανονικά, η εντολή `BUG_ON()` προκαλεί ένα kernel panic, ενώ το `WARN_ON()` γράφει κάτι στα αρχεία καταγραφής (logs) του πυρήνα (μπορούμε να το δούμε με χρήση της εντολής `dmesg`). Αξιοσημείωτο είναι το γεγονός ότι το νήμα πυρήνα για τις περιόδους χάριτος του RCU-bh απενεργοποιήθηκε, αλλά μπορεί να επανεργοποιηθεί χρησιμοποιώντας την εντολή `-DENABLE_RCU_BH` στον προεπεξεργαστή. Τα προαναφερθέντα προφανώς δεν επηρεάζουν τη συμπεριφορά του κώδικα αφού αντικαταστήσαμε μόνο τύπους και συναρτήσεις άσχετους με τους ελέγχους που πραγματοποιήσαμε (π.χ. κάποιους ορισμούς σχετικούς με την επίσπευση περιόδων χάριτος) και δώσαμε δικούς μας ορισμούς για διάφορες άλλες συναρτήσεις, ούτως ώστε να δουλεύουν με τη μοντελοποίηση που έγινε στον επεξεργαστή και τις διακοπές.

Φράχτες μνήμης παρέχονται για αρχιτεκτονική x86 (προεπιλογή) και POWER (με χρήση του `-DPOWERPC`). Δυστυχώς, δεν γίνεται όλα τα προγράμματα ελέγχου να δουλέψουν για το μοντέλο μνήμης του POWER, επειδή υπάρχουν συναρτήσεις όπως η `pthread_mutex_trylock()` που δεν υποστηρίζονται από το Nidhugg για το συγκεκριμένο μοντέλο μνήμης (βλ. Ενότητα 4.2.2). Σε περίπτωση που χρειαζόντουσαν ορισμοί εξαρτώμενοι από την αρχιτεκτονική, συμπεριλάβαμε όσο περισσότερους διαφορετικούς ορισμούς μπορούσαμε. Ωστόσο, σε πολλές περιπτώσεις το Nidhugg υποστήριζε τους αντίστοιχους ορισμούς μόνο για αρχιτεκτονικές x86 (βλ. Ενότητα 4.2.2).

Μηχανισμοί συγχρονισμού

Είναι γνωστό ότι ο πυρήνας του Linux χρησιμοποιεί πληθώρα διαφορετικών μηχανισμών συγχρονισμού. Στην ενότητα αυτή θα περιγράψουμε τον τρόπο που μοντελοποιήθηκαν οι μηχανισμοί αυτοί, καθώς και ορισμένες λεπτομέρειες σχετικές με την υλοποίησή τους.

Ατομικές Εντολές

Ενώ ο ατομικός τύπος δεδομένων `atomic_t` αντιγράφηκε απευθείας από τον πυρήνα, δεν ισχύει το ίδιο και για τις ατομικές εντολές όπως η `atomic_read()`, `atomic_set()`, `atomic_add()` κλπ, αφού η υλοποίησή τους εξαρτάται από την εκάστοτε αρχιτεκτονική. Για να μοντελοποιήσουμε τις παραπάνω εντολές, χρησιμοποιήσαμε κάποιες επεκτάσης της C που προσφέρει ο gcc, οι οποίες υποστηρίζονται επίσης από τον μεταγλωττιστή clang [GCCA, Clan].

Για παράδειγμα, ο ορισμός για την `atomic_add()` είναι ο ακόλουθος: `#define atomic_add(i, v) __atomic_add_fetch(&(v)->counter, i, __ATOMIC_RELAXED)`. Γενικά, όλες οι συναρτήσεις που αφορούν ατομικούς τύπους μοντελοποιήθηκαν με χρήση των αντίστοιχων ενσωματωμένων στον gcc συναρτήσεων.

Οι συναρτήσεις αυτές χρησιμοποιούνται για ατομικές εντολές που έχουν επίγνωση του μοντέλου μνήμης στο οποίο εκτελούνται (memory model aware atomic operations): συμβαδίζουν με τις απαιτήσεις του προτύπου C++11 για το μοντέλο μνήμης. Ωστόσο, το Nidhugg υποστηρίζει τις συναρτήσεις αυτές μόνο για SC, TSO και PSO, και μόνο για το μοντέλο μνήμης `__ATOMIC_SEQ_CST` (το οποίο επιβάλλει ολική διάταξη με όλες τις άλλες `__ATOMIC_SEQ_CST` εντολές), ακόμη κι αν έχει οριστεί κάτι διαφορετικό. Αυτό σημαίνει ότι μπορούν να χρησιμοποιηθούν μόνο για την x86 αρχιτεκτονική που μοντελοποιήσαμε, αφού οι x86 επεξεργαστές κρατούν μια ολική διάταξη των ατομικών εντολών με τις υπόλοιπες εντολές [Wikia].

Spinlocks & Mutexes

Σε πυρήνες που δεν είναι RT, το `spinlock_t` είναι απλά ένα περιτύλιγμα για το `raw_spinlock_t`, το οποίο αποτελεί ουσιαστικά την υλοποίηση των spinlocks στον πυρήνα. Φυσικά, επειδή το `raw_spinlock_t` έχει υλοποίηση μη-εξαρτώμενη από την αρχιτεκτονική, αναθέτει ορισμένες εντολές χαμηλού επιπέδου στο `arch_spinlock_t`, του οποίου η υλοποίηση εξαρτάται από την αρχιτεκτονική. Εμείς, επειδή θέλαμε μία υλοποίηση που να δουλεύει ανεξαρτήτως αρχιτεκτονικής, χρησιμοποιήσαμε `pthread_mutexes` για την μοντελοποίηση των spinlocks, τα οποία υποστηρίζονται από το Nidhugg. Με παρόμοια λογική, το `pthread_mutex` χρησιμοποιήθηκε στη θέση του `struct mutex`. Ακόμη, επειδή πολλά spinlocks και mutexes αρχικοποιούνται στατικά στον πυρήνα, η επιλογή `--disable-mutex-init-requirement` του Nidhugg χρειάστηκε για όλα τα προγράμματα ελέγχου μας.

Τέλος, συναρτήσεις σχετικές με το εργαλείο χρόνου εκτέλεσης ελέγχου ορθότητας κλειδωμάτων (lockdep) του πυρήνα, έχουν απενεργοποιηθεί [McKe10a].

Μεταβλητές Ολοκλήρωσης (Completions)

Για τον έλεγχο που πραγματοποιήσαμε, ο ορισμός των μεταβλητών ολοκλήρωσης από τον πυρήνα κρατήθηκε, αλλά έπρεπε να μοντελοποιηθούν κάπως οι ουρές αναμονής (wait queues). Γι' αυτό, δοκιμάσαμε δύο διαφορετικές προσεγγίσεις οι οποίες και δούλεψαν όπως αναμενόταν.

Η πρώτη προσέγγιση περιλάμβανε την χρήση μεταβλητών κατάστασης (condition variables) για τη μοντελοποίηση των ουρών αναμονής. Οπότε, όταν ένα νήμα περίμενε ένα γεγονός, απλά περίμενε σε μία μεταβλητή κατάστασης. Επειδή όμως η προσέγγιση αυτή μπορεί να μεγαλώνει λίγο τον χώρο καταστάσεων, και πρέπει να ανέχεται ορισμένα γεγονότα όπως το πρώιμο ξύπνημα των νημάτων που περιμένουν σε μεταβλητές κατάστασης (αν και αυτό δεν αποτελεί ιδιαίτερο πρόβλημα), δοκιμάσαμε κι επιλέξαμε τη δεύτερη προσέγγιση.

Η δεύτερη προσέγγιση μοντελοποίησε τις μεταβλητές ολοκλήρωσης απλά αναμένοντας μία συνθήκη να πραγματοποιηθεί (spin-waiting). Αφού λοιπόν ένα νήμα που περιμένει μία μεταβλητή ολοκλήρωσης απλά περιμένει σε μία ουρά αναμονής μέχρι να ικανοποιηθεί μία συνθήκη, εμείς απλά χρησιμοποιήσαμε μια δομή επανάληψης που εκτελείται μέχρις ότου ικανοποιηθεί η αντίστοιχη συνθήκη (spin loop). Το Nidhugg μετασχηματίζει αυτόματα όλες αυτές τις δομές σε εντολές `__VERIFIER_assume()`, όπου αν η συνθήκη δεν ισχύει, η εκτέλεση σταματά [Beye15]. Πριν την αναμονή στη δομή επανάληψης, το νήμα αφήνει το κλείδωμα του επεξεργστή. Θα προσπαθήσει να το ξαναπάρει άπαξ και ικανοποιηθεί η συνθήκη. Επειδή αυτό αποτελεί κατάσταση ηρεμίας για το RCU, η συνάρτηση `rcu_note_context_switch()` (και πιθανώς και η `do_IRQ()`, έτσι ώστε να κληθεί η

rcu_process_callbacks()) θα μπορούσε να κληθεί πριν να αφεθεί το κλείδωμα του επεξεργαστή. Ωστόσο, αν το νήμα που περιμένει δεν είναι το μοναδικό που τρέχει στον συγκεκριμένο επεξεργαστή, αυτό δεν είναι απαραίτητο. Οι συναρτήσεις αυτές μπορούν να κληθούν και από άλλα νήματα που τρέχουν στον ίδιο επεξεργαστή.

4.3 Αναπαραγωγή Παλιών Σφαλμάτων του Πυρήνα

Στην ενότητα αυτή θα προσπαθήσουμε να αναπαράγουμε ορισμένα παλαιότερα σφάλματα που υπήρχαν στον πυρήνα, με λίγη γνώση των συνθηκών υπό των οποίων πραγματοποιούνταν. Τα προγράμματα ελέγχου είναι σχεδιασμένα να τρέχουν σε πολλές εκδόσεις του πυρήνα, για να διαπιστωθεί αν τα σφάλματα αυτά διορθώθηκαν.

4.3.1 Σφάλμα #1: Θέματα Συγχρονισμού για την rcu_process_gp_end()

Στην Ενότητα 4.1.3 αναφέραμε ότι το νήμα πυρήνα για τις περιόδους χάριτος κάνει κάποιο ξεκαθάρισμα μετά το τέλος μιας περιόδου χάριτος. Ωστόσο, αυτό δεν συνέβαινε πάντα. Σε παλαιότερες εκδόσεις του πυρήνα, όταν ένας επεξεργαστής εισερχόταν στον πυρήνα του RCU, ήλεγχε μήπως είχε τελειώσει κάποια περίοδος χάριτος απευθείας, συγκρίνοντας τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος στη δομή rcu_state με τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος στην αντίστοιχη δομή rcu_data (μέσω της συνάρτησης rcu_process_gp_end()). Σε νεότερες εκδόσεις του πυρήνα, η συνάρτηση note_gp_changes() συγκρίνει τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος στη δομή rcu_node που αντιστοιχεί σε έναν επεξεργαστή, με τον αριθμό της τελευταίας ολοκληρωμένης περιόδου χάριτος στην αντίστοιχη δομή rcu_data, *κρατώντας όμως το κλείδωμα του κόμβου*, αποκλείοντας με τον τρόπο αυτό ταυτόχρονες ενέργειες στον κόμβο αυτόν.

Στην έκδοση v2.6.32 του πυρήνα, το commit d09b62dfa336 έφτιαξε ένα bug που σχετιζόταν με μη-συγχρονισμένες προσβάσεις στον μετρητή ->completed στη δομή rcu_state, το οποίο προκαλούσε την προώθηση επανακλήσεων των οποίων η περίοδος χάριτος δεν είχε ακόμη ολοκληρωθεί [LKMLc, LKMLa]. Στις επόμενες παραγράφους θα δείξουμε πώς οι μη-συγχρονισμένες αυτές προσβάσεις οδηγούσαν σε περιόδους χάριτος πολύ μικρής διάρκειας.

Ο πρωταρχικός μας στόχος είναι η κατασκευή ενός προγράμματος ελέγχου που εκθέτει το συγκεκριμένο σφάλμα και μας δείχνει τις συνθήκες υπό τις οποίες προκύπτει. Πώς όμως μπορούμε να κατασκευάσουμε ένα κατάλληλο πρόγραμμα αν δεν γνωρίζουμε το σφάλμα εκ των προτέρων; Μιας και γνωρίζουμε ότι σχετίζεταιμε τη συνάρτηση rcu_process_gp_end(), μπορούμε να δούμε το ανάλογο τμήμα της συνάρτησης αυτής στον Κώδικα 4.3. Όπως μπορούμε να δούμε, οι πρόσβαση στον μετρητή ->completed δεν προστατεύται από ταυτόχρονες προσβάσεις με κάποιον τρόπο

Το πρώτο βήμα λοιπόν ήταν να βάλουμε μια κατάλληλη εντολή BUG_ON() στο σώμα του **if** για να διαπιστώσουμε αν γίνεται ένα νήμα να πάρει την τιμή για τον μετρητή ->completed και μετά να χρησιμοποιήσει το completed_snap ενώ η τιμή του ->completed έχει αλλάξει. Η απάντηση ήταν καταφατική.

Το επόμενο βήμα ήταν να διαπιστώσουμε αν αυτό μπορούσε δυνητικά να οδηγήσει στην έναρξη μιας περιόδου χάριτος από έναν επεξεργαστή, χωρίς ο επεξεργαστής αυτός να έχει αντιληφθεί τη λήξη της προηγούμενης περιόδου χάριτος. Και εδώ, ένα κατάλληλο BUG_ON() το οποίο συνέκρινε τον αριθμό της τρέχουσας περιόδου χάριτος με τον αριθμό της περιόδου χάριτος της οποίας το τέλος αντιλαμβάνεται ένας επεξεργαστής, έδειξε ότι το παραπάνω σενάριο είναι πιθανό.

Με τα στοιχεία αυτά, κατασκευάσαμε ένα απλό πρόγραμμα ελέγχου (Κώδικας 4.4) το οποίο δείχνει ότι αυτές οι μη-συγχρονισμένες προσβάσεις μπορούν να οδηγήσουν σε

```
1  completed_snap = ACCESS_ONCE(rsp->completed);  /* outside of lock. */
2
3  /* Did another grace period end? */
4  if (rdp->completed != completed_snap) {
5
6          /* Advance callbacks.  No harm if list empty. */
7          rdp->nxttail[RCU_DONE_TAIL] = rdp->nxttail[RCU_WAIT_TAIL];
8          rdp->nxttail[RCU_WAIT_TAIL] = rdp->nxttail[RCU_NEXT_READY_TAIL];
9          rdp->nxttail[RCU_NEXT_READY_TAIL] = rdp->nxttail[RCU_NEXT_TAIL];
10
11         /* Remember that we saw this grace-period completion. */
12         rdp->completed = completed_snap;
13 }
```

Listing 4.3: Unsynchronized accesses to the `->completed` counter.

πολύ σύντομες περιόδους χάριτος (αρχείο `gp_end_bug.c`). Σημειώστε ότι γενικά οι πολύ σύντομες περίοδοι χάριτος δεν μας απασχολούν ιδιαίτερα. Ωστόσο, τέτοιες περίοδοι χάριτος επιτρέπουν σε κρίσιμα τμήματα αναγνωστών που χρησιμοποιούν RCU να έχουν μεγαλύτερη διάρκεια από μία περίοδο χάριτος, κάτι που παραβιάζει την βασική εγγύηση του RCU (εγγύηση Περιόδου Χάριτος). Δεδομένων των παραπάνω, το πρόγραμμα ελέγχου που παρέχεται πρέπει να περιλαμβάνει έναν αναγνώστη ο οποίος βλέπει αλλαγές που συμβαίνουν τόσο πριν την έναρξη μιας περιόδου χάριτος, όσο και μετά τη λήξη της ίδιας περιόδου χάριτος, μέσα στο ίδιο κρίσιμο τμήμα ανάγνωσης.

Στον Κώδικα 4.4 μπορούμε να δούμε ότι υπάρχουν τρία νήματα και δύο επεξεργαστές. Το νήμα `thread_update()` τρέχει στον επεξεργαστή 0 και το νήμα `thread_reader()` τρέχει στον επεξεργαστή 1. Το νήμα `thread_helper()` αντιπροσωπεύει ένα τυχαίο νήμα που τρέχει στον επεξεργαστή 0 και μπορεί -δυνητικά- να καταλάβει τον επεξεργαστή αφού το νήμα `thread_update()` μπλοκάρει λόγω της `synchronize_rcu()`. Μια ακολουθία γεγονότων που εκθέτει το προαναφερθέν σφάλμα παρατίθεται στη συνέχεια:

0. Ο επεξεργαστής 0 καταχωρίζει μία επανάκληση, ξεκινά μία νέα περίοδο χάριτος, περνάει από κατάσταση ηρεμίας, και το αναφέρει στο RCU.

1. Ο επεξεργαστής 1 βλέπει ότι μία περίοδος χάριτος έχει ξεκινήσει (δεχόμενος μία διακοπή ρολογιού) και περνάει από μία κατάσταση ηρεμίας, αλλά δεν το αναφέρει στο RCU ακόμη.

2. Ο επεξεργαστής 1 ξεκινάει το κρίσιμο τμήμα ανάγνωσής του και εκτελεί την εντολή `r_x = x`.

3. Ο επεξεργαστής 1 δέχεται μια διακοπή ρολογιού και εισέρχεται στον πυρήνα του RCU (αλλά δεν τελειώνει ακόμα την περίοδο χάριτος).

4. Ο επεξεργαστής 0 εκτελεί την εντολή `x = 1` κι έπειτα την `synchronize_rcu()`. Η `synchcronize_rcu()` καταχωρίζει μια επανάκληση. Στις εκδόσεις του πυρήνα v2.6.31.1 και v2.6.32.1, η `call_rcu()` καλεί την `rcu_process_gp_end()`. Άρα ο επεξεργαστής 0 καλεί την `rcu_process_gp_end()`, αλλά ο επεξεργαστής 1 δεν έχει τελειώσει ακόμη την περίοδο χάριτος, και ο επεξεργαστής 0 βλέπει ότι η τρέχουσα περίοδος χάριτος δεν έχει ολοκληρωθεί ακόμη. Σημειώστε ότι ο επεξεργαστής 1 είχε ήδη ξεκινήσει το κρίσιμο τμήμα ανάγνωσής του όταν κλήθηκε η `synchronize_rcu()` από τον επεξεργαστή 0.

5. Ο επεξεργαστής 1 τελειώνει την πρώτη περίοδο χάριτος και ενημερώνει την τιμή του μετρητή `->completed`. Ο επεξεργαστής αυτός δεν χρειάζεται μια νέα περίοδο χάριτος, άρα δεν ξεκινά κάποια.

6. Ο επεξεργαστής 0 ελέγχει να δει αν υπάρχει κάποια νέα περίοδος χάριτος (συγκρίνοντας το `rdp->gpnum` με το `rsp->gpnum`), αλλά μία νέα περίοδος χάριτος δεν έχει ξεκινήσει. Αυτό γίνεται όταν ο επεξεργαστής 0 εκτελεί τη συνάρτηση `rcu_check_for_new_grace_period()`.

7. Ο επεξεργαστής 0 έχει καταχωρισμένες επανακλήσεις και βλέπει ότι δεν υπάρχει κάποια περίοδος χάριτος εν ενεργεία (αφού `rcu_gp_in_progress() == 0`), ξεκινά μία νέα περίοδο χάριτος, και προωθεί τις επανακλήσεις του από το τμήμα `NEXT` στο τμήμα `WAIT`. Ο επεξεργαστής 0 ακόμη δεν έχει δει τη λήξη της προηγούμενη περιόδου χάριτος.

8. Ο επεξεργαστής 0 δέχεται ακόμη μία διακοπή ρολογιού και εισέρχεται στον πυρήνα του RCU. Βλέπει ότι ολοκληρώθηκε μία περίοδος χάριτος (μέσω της συνάρτησης `rcu_process_gp_end()`) και προωθεί τις επανακλήσεις του από το τμήμα `WAIT` στο `DONE`. Αυτό σημαίνει ότι η επανάκληση που αντιστοιχεί στο `synchronize_rcu()` είναι έτοιμη προς εκτέλεση, και ο επεξεργαστής 0 την εκτελεί στον ίδιο χειριστή διακοπών.

9. Ο επεξεργαστής 0 εκτελεί την εντολή `y = 1`.

10. Ο επεξεργαστής 0 εκτελεί την εντολή `r_y = 1`.

Από τα παραπάνω παρατηρούμε ότι το αποτέλεσμα

```
r_x == 0 && r_y == 1
```

είναι πιθανό, κάτι που συνιστά παραβίαση της βασικής εγγύησης που προσφέρει το RCU (εγγύηση Περιόδου Χάριτος), αφού η χρονική διάρκεια μεταξύ της καταχώρισης μιας επανάκλησης και της εκτέλεσής της πρέπει να είναι μεγαλύτερη από τη διάρκεια μιας περιόδου χάριτος (και άρα ενός πλήρους συνόλου καταστάσεων ηρεμίας), ενώ αυτό δεν συμβαίνει εδώ.

Τέλος, μερικές παρατηρήσεις για το σφάλμα αυτό:

- Το σφάλμα αυτό υπήρχε τόσο σε ιεραρχίες αποτελούμενες από έναν μόνο κόμβο όσο και σε πολυεπίπεδες ιεραρχίες (μπορούν να κατασκευαστούν θέτοντας κατάλληλα `-DCONFIG_RCU_FANOUT=x`, αν και ελαφρώς διαφορετικό πρόγραμμα από αυτό του Κώδικα 4.4 χρειάζεται), αφού δεν βασίζεται σε αλληλεπιδράσεις με την ιεραρχία των κόμβων.

- Το σφάλμα δεν παρατηρείται στην έκδοση v3.0 του πυρήνα (για να τρέξει το πρόγραμμα στην έκδοση αυτή χρειάζεται να οριστεί `-DKERNEL_VERSION_3`), κάτι που σημαίνει ότι όντως διορθώθηκε. Ο λόγος που δεν παρατηρείται είναι ότι στην έκδοση v3.0 η συνάρτηση `rcu_start_gp()` καλεί την `__rcu_process_gp_end()`, κάτι που εγγυάται ότι ένας επεξεργαστής θα δει πρώτα τη λήξη μιας περιόδου χάριτος και μετά την έναρξη μίας καινούργιας, κάτι που δεν συμβαίνει στην έκδοση v2.6.32.1. Ωστόσο, το σφάλμα παρατηρείται σε προηγούμενες εκδόσεις, όπως η v2.6.31.1.

- Μόνο δύο επεξεργαστές απαιτούνται για να προκληθεί το συγκεκριμένο σφάλμα, και μόνο ένας από τους δύο πρέπει να εκτελέσει την `call_rcu()`.

- Μόνο μία περίοδος χάριτος απαιτείται για να προκληθεί το σφάλμα, με την έννοια ότι δεν βασίζεται στο γεγονός ότι κάποιοι επεξεργαστές αγνοούν ενάρξεις και λήξεις περιόδων χάριτος (π.χ. όταν κάποιος επεξεργαστής είναι σε λειτουργία `dynticks-idle`). Φυσικά, αυτό σημαίνει ότι το `CONFIG_NO_HZ=y/n` δεν πρέπει να επηρεάσει τα αποτελέσματα του ελέγχου.

- Η συνάρτηση `force_quiescent_state()` δεν απαιτείται για να προκληθεί το σφάλμα, αν και συχνές κλήσεις στη συνάρτηση αυτή θα το εξέθεταν ευκολότερα σε πραγματικές συνθήκες.

- Το σφάλμα δεν προκαλείται από κάποιο ασθενές μοντέλο μνήμης. Το πρόγραμμα ελέγχου τρέχει υπό ακολουθιακή συνέπεια.

- Το Nidhugg παρήγαγε την προαναφερθείσα ακολουθία γεγονότων σε μόλις $0.56s$ (συμπεριλαμβανομένων των χρόνων μεταγλώττισης και μετασχηματισμού), και χρησιμοποίησε 30848 kB μνήμης (η μνήμη που καταλαμβάνουν οι δομές του RCU περιλαμβάνεται).

4.3.2 Σφάλμα #2: Σφάλμα Μεταξύ του Εξαναγκασμού και της Αρχικοποίησης Περιόδων Χάριτος

Στην ενότητα αυτή θα δείξουμε ότι ένα πρώην αναφερθέν ως σφάλμα στην πραγματικότητα δεν αποτελεί σφάλμα, χρησιμοποιώντας το Nidhugg. Πιο συγκεκριμένα, θα ασχοληθούμε με την έκδοση v2.6.31 του πυρήνα του Linux και ένα υποτιθέμενο σφάλμα που προκαλούνταν σε περιόδους χάριτος μεγάλης διάρκειας, μεταξύ της επίσπευσης αυτών (grace-period forcing) και της αρχικοποίησης μιας νέας περιόδου χάριτος [LKMLb]. Το σφάλμα αυτό υποτίθεταθ ότι διορθώθηκε με το commit 83f5b01ffbba. Ωστόσο, είναι εξαιρετικά πιθανό ότι το Σφάλμα #1 που παρουσιάσαμε στην προηγούμενη ενότητα ήταν το σφάλμα πίσω από τις πολύ σύντομες περιόδους χάριτος που είχαν παρατηρηθεί τότε, αφού αυτά τα δύο σφάλματα σχετίζονται [LKMLa, McKe09a].

Η καταγραφή (log) του commit αυτού λέει ότι κρίσιμα τμήματα ανάγνωσης πολύ μεγάλης διάρκειας μπορούν να προκαλέσουν μία κατάσταση ανταγωνισμού (race condition) μεταξύ της `force_quiescent_state()` και της `rcu_start_gp()` ως εξής:

1. Ο επεξεργαστής 0 καλεί την `force_quiescent_state()`, βλέπει ότι υπάρχει ήδη μία περίοδος χάριτος εν ενεργεία, και παίρνει το κλείδωμα `->fqslock`.

2. Ο επεξεργαστής 1 τελειώνει την περίοδο χάριτος, και η συνάρτηση `cpu_quiet_msk_finish()` θέτει τη μεταβλητή `rsp->completed` σε `rsp->gpnum`. Η εντολή αυτή γίνεται υπό την προστασία του κλειδώματος `rnp->lock` του κόμβου-ρίζα, αλλά ο επεξεργαστής 0 δεν έχει πάρει ακόμα αυτό το κλείδωμα. Σημειώστε ότι η μεταβλητή `rsp->signaled` είναι ακόμα `RCU_SAVE_DYNTICK` από την τελευταία περίοδο χάριτος.

3. Ο επεξεργαστής 1 καλεί την `rcu_start_gp()`, αλλά κανείς δεν θέλει μία νέα περίοδο χάριτος, άρα αφήνει το κλείδωμα `rnp->lock` του κόμβου-ρίζα κι επιστρέφει.

4. Ο επεξεργαστής 0 παίρνει το κλείδωμα του κόμβου ρίζα, και παίρνει δείγματα για τις μεταβλητές `rsp->completed` και `rsp->signaled`. Έπειτα αφήνει το `rnp->lock`. Στη συνέχεια εισέρχεται στο `RCU_SAVE_DYNTICK` σκέλος της εντολής switch.

5. Ο επεξεργαστής 2 καλεί την `call_rcu()` και τώρα χρειάζεται μία νέα περίοδο χάριτος. Καλεί την `rcu_start_gp()`, η οποία παίρνει το κλείδωμα του κόμβου-ρίζα `rnp->lock`, θέτει τη μεταβλητή `rsp->signaled` σε `RCU_GP_INIT` (κρίμα που ο επεξεργαστής 0 είναι ήδη στο σκέλος `RCU_SAVE_DYNTICK` της εντολής switch!) και ξεκινά την αρχικοποίηση της ιεραρχίας των κόμβων `rcu_node`. Αν υπάρχουν πολλά επίπεδα στην ιεραρχία, θα αφήσει το κλείδωμα `rnp->lock` του κόμβου-ρίζα και θα αρχικοποιήσει τα κατώτερα επίπεδα της ιεραρχίας.

6. Ο επεξεραστής 0 βλέπει ότι η μεταβλητή `rsp->completed` δεν άλλαξε, κάτι που επιτρέπει τόσο στον επεξεργαστή 2 όσο και στον επεξεργαστή 0 να προσπαθήσουν να την

καταγράψουν ταυτόχρονα. Αν η καταγραφή του επεξεργαστή 0 επικρατήσει, επόμενες κλήσεις στην `force_quiescent_state()` μπορεί να μετρήσουν παλιές καταστάσεις ηρεμίας σε μια νέα περίοδο χάριτος, κάτι που μπορεί να οδηγήσει σε πρώιμη λήξη μιας περιόδου χάριτος.

Αξίζει να σημειωθεί ότι το σφάλμα αυτό υποτίθεται ότι υπάρχει μόνο σε πυρήνες παραμετροποιημένους ώστε να έχουν πολυεπίπεδη ιεραρχία κόμβων.

Όμως, προσπαθώντας να μετατρέψουμε αυτό το σενάριο σε πρόγραμμα ελέγχου, δεν προέκυπταν πολύ σύντομες περίοδοι χάριτος. Αυτό, αν και παράξενο αρχικά, ήταν απολύτως λογικό κοιτώντας τον κώδικα της συνάρτησης `force_quiescent_state()` στον Κώδικα 4.5. Όπως μπορούμε να δούμε, ο επεξεργαστής 0 δεν μπορεί να μπει στο σκέλος `RCU_SAVE_DYNTICK` της εντολής switch αφού, αφού πάρει δείγματα για τις μεταβλητές `rsp->completed` και `rsp->signaled`, η συνθήκη `lastcomp == rsp->gpnum` θα είναι αληθής (δεν υπάρχει περίοδος χάριτος εν ενεργεία), και ο επεξεργαστής 0 θα πάει στην ετικέτα `unlock_ret`. Αυτό δικαιολόγησε τα αποτελέσματα των ελέγχων μας μέχρι στιγμής, αλλά αφού υπήρχαν αναφορές για αποτυχίες του RCU εκείνη την περίοδο, μπορεί να χάνουμε κάτι.

Άρα τι γίνεται αν η καταγραφή που συνόδευε το συγκεκριμένο commit ήταν λάθος; Μπορεί το σενάριο αποτυχίας να ήταν λάθος, όμως το σφάλμα να είναι υπαρκτό. Επομένως, το επόμενο βήμα είναι να θεωρήσουμε μια ελαφρώς διαφορετική ακολουθία γεγονότων που ίσως προκαλέσει την εμφάνιση του σφάλματος:

1. Ο επεξεργαστής 0 καλεί την `force_quiescent_state()`, βλέπει ότι υπάρχει ήδη μία περίοδος χάριτος εν ενεργεία, και παίρνει το κλείδωμα `->fqslock`.

   \+ Επίσης παίρνει το κλείδωμα `rnp->lock` του κόμβου-ρίζα και δείγματα της τιμής των μεταβλητών `rsp->completed` και `rsp->signaled`, και μετά αφήνει το κλείδωμα `rnp->lock`.

2. Ο επεξεργαστής 1 τελειώνει την περίοδο χάριτος, και η συνάρτηση `cpu_quiet_msk_finish()` θέτει τη μεταβλητή `rsp->completed` σε `rsp->gpnum`. Η εντολή αυτή γίνεται υπό την προστασία του κλειδώματος `rnp->lock` του κόμβου-ρίζα, αλλά ο επεξεργαστής 0 δεν έχει πάρει ακόμα αυτό το κλείδωμα. Σημειώστε ότι η μεταβλητή `rsp->signaled` είναι ακόμα `RCU_SAVE_DYNTICK` από την τελευταία περίοδο χάριτος.

3. Ο επεξεργαστής 1 καλεί την `rcu_start_gp()`, αλλά κανείς δεν θέλει μία νέα περίοδο χάριτος, άρα αφήνει το κλείδωμα `rnp->lock` του κόμβου-ρίζα κι επιστρέφει.

4. Ο επεξεργαστής 0 εισέρχεται στο `RCU_SAVE_DYNTICK` σκέλος της εντολής switch.

5. Ο επεξεργαστής 2 καλεί την `call_rcu()` και τώρα χρειάζεται μία νέα περίοδο χάριτος. Καλεί την `rcu_start_gp()`, η οποία παίρνει το κλείδωμα του κόμβου-ρίζα `rnp->lock`, θέτει τη μεταβλητή `rsp->signaled` σε `RCU_GP_INIT` (κρίμα που ο επεξεργαστής 0 είναι ήδη στο σκέλος `RCU_SAVE_DYNTICK` της εντολής switch!) και ξεκινά την αρχικοποίηση της ιεραρχίας των κόμβων `rcu_node`. Αν υπάρχουν πολλά επίπεδα στην ιεραρχία, θα αφήσει το κλείδωμα `rnp->lock` του κόμβου-ρίζα και θα αρχικοποιήσει τα κατώτερα επίπεδα της ιεραρχίας.

6. Ο επεξεραστής 0 βλέπει ότι η μεταβλητή `rsp->completed` δεν άλλαξε, κάτι που επιτρέπει τόσο στον επεξεργαστή 2 όσο και στον επεξεργαστή 0 να προσπαθήσουν να την καταγράψουν ταυτόχρονα. Αν η καταγραφή του επεξεργαστή 0 επικρατήσει, επόμενες κλήσεις στην `force_quiescent_state()` μπορεί να μετρήσουν παλιές καταστάσεις ηρεμίας σε μια νέα περίοδο χάριτος, κάτι που μπορεί να οδηγήσει σε πρώιμη λήξη μιας περιόδου χάριτος.

Όμως η παραπάνω ακολουθία γεγονότων επίσης δεν μπορεί να οδηγήσει στην καταμέτρηση παλιών καταστάσεων ηρεμίας σε μία καινούργια περίοδο χάριτος. Αν ο επεξεργαστής 0 πάρει δείγμα της μεταβλητής `->completed` πριν τη λήξη της περιόδου χάριτος, δεν θα προσπαθήσει να ενημερώσει τη μεταβλητή `dynticks_completed` στο βήμα 6, αφού η τιμή της μεταβλητής `->completed` είναι διαφορετική από το δείγμα που λήφθηκε προηγουμένως.

Στην πραγματικότητα, φαίνεται ότι δεν υπάρχει καμία κατάσταση ανταγωνισμού μεταξύ της επίσπευσης και της αρχικοποίησης περιόδων χάριτος. Για να το δείξουμε αυτό, παρέχουμε προγράμματα ελέγχου της ακόλουθης λογικής: πρέπει να είναι αδύνατο για έναν επεξεργαστή που προσπαθεί να επισπεύσει μία περίοδο χάριτος να πάρει δείγματα των τιμών των μεταβλητών `->completed` και `->signaled` πριν τη λήξη μιας περιόδου χάριτος και μετά να ενημερώσει τη μεταβλητή `->dynticks_completed` αφού μια νέα περίοδος χάριτος έχει ξεκινήσει. Πιο συγκεκριμένα, είναι αδύνατον να εισέλθουμε στο `RCU_SAVE_DYNTICK` σκέλος της εντολής switch στην `force_quiescent_state()` πριν τη λήξη μιας περιόδου χάριτος και να γράψουμε την τιμή `dynticks_completed` μετά την έναρξη μιας νέας περιόδου χάριτος. Τα αποτελέσματα που πήραμε επαλήθευσαν το παραπάνω, μαζί με το γεγονός ότι είναι αδύνατον για τον επεξεργαστή 0 να είναι στο `RCU_SAVE_DYNTICK` σκέλος της εντολής switch, ενώ `rsp->signaled == RCU_GP_INIT` και `lastcomp == rsp->gpnum`.

Αφού αυτό το υποτιθέμενο σφάλμα χρειάζεται -τουλάχιστον- τρεις επεξεργαστές για να προκληθεί, τα προγράμματά μας πρέπει να τρέχουν με `-DCONFIG_NR_CPUS=3`. Επιπρόσθετα, πρέπει να οριστεί `-DCONFIG_RCU_FANOUT=2` για να δημιουργηθεί μια πολυεπίπεδη ιεραρχία. Τέλος, πρέπει να οριστεί `-DFQS_NO_BUG` για να εισαχθούν όλες οι απαραίτητες εντολές `assert()` στον πηγαίο κώδικα του Tree RCU.

4.4 Επιβεβαίωση της Εγγύησης Έκδοσης-Συνδρομής του Tree RCU

Στην ενότητα αυτή θα επαληθεύσουμε μηχανικά την εγγύηση Έκδοσης-Συνδρομής του Tree RCU, για την έκδοση v3.19 του πυρήνα. Το πρόγραμμα του Κώδικα 4.6 αποτελεί ένα πρόγραμμα ελέγχου για την εγγύηση αυτή.

Αυτό το πρόγραμμα έχει σχεδιασθεί να τρέχει υπό τα μοντέλα μνήμης TSO και POWER. Για το μοντέλο POWER, πρέπει να οριστεί η επιλογή `-DPOWERPC` στον προεπεξεργαστή, έτσι ώστε να συμπεριληφθούν οι κατάλληλοι φράχτες μνήμης από το αρχείο `fake_defs.h`. Όταν ορίζεται το `-DORDERING_BUG` στον προεπεξεργαστή, η μακροεντολή `rcu_assign_pointer()` δεν χρησιμοποιείται, κάτι που μπορεί να οδηγήσει σε σφάλματα υποθέτοντας ένα μοντέλο μνήμης που αναδιατάσσει τις εγγραφές στη μνήμη. Μία απλή εντολή `BUG_ON()` στη συνάρτηση `use_gp()` χρησιμοποιείται για να ελέγξουμε κατά πόσο ένας αναγνώστης μπορεί να δει μη-αρχικοποιημένες τιμές.

Όπως αναμενόταν, η μακροεντολή `rcu_assign_pointer()` ήταν απολύτως απαραίτητη για το μοντέλο μνήμης POWER, ενώ το μοντέλο μνήμης στον x86 δεν χρειαζόταν στην μακροεντολή (κάτι αναμενόμενο αφού οι x86 επεξεργαστές δεν αναδιατάσσουν τις εγγραφές). Από την άλλη βέβαια, η μακροεντολή `rcu_assign_pointer()` είναι ένας απλός φράχτης για τον μεταγλωττιστή σε x86 αρχιτεκτονικές (εκτός ίσως από την οικογένεια επεξεργαστών Pentium Pro). Φυσικά, επιθετικές βελτιστοποιήσεις από τον μεταγλωττιστή μπορούν να επηρεάσουν το αποτέλεσμα επίσης, αλλά το Nidhugg δεν μπορεί να βρει σφάλματα σε τέτοιες περιπτώσεις, αφού το σφάλμα δεν σχετίζεται με τον χρονοδρομολογητή ή κάποιο ασθενές μοντέλο μνήμης.

Τέλος, το πρόγραμμα αυτό πραγματοποιεί ελέγχους από την πλευρά του εκδότη. Ένας ανάλογος έλεγχος από την πλευρά του συνδρομητή θα απαιτούσε υποστήριξη για ένα μοντέλο μνήμης που αναδιατάσσει αλληλοεξαρτώμενες αναγνώσεις (dependent loads reordering), όπως το αντίστοιχο μοντέλο μνήμης στους επεξεργαστές DEC Alpha.

4.5 Επιβεβαίωση της Εγγύησης Περιόδου Χάριτος του Tree RCU

Στην ενότητα αυτή θα επαληθεύσουμε μηχανικά την εγγύηση της Περιόδου Χάριτος του Tree RCU για ένα μη-προεκχωρητικό περιβάλλον πυρήνα (έκδοση v3.19). Στον Κώδικα 4.7 παρέχουμε τη δομή για ένα πρόγραμμα ελέγχου για την εγγύηση αυτή. Ωστόσο, πριν παρουσιάσουμε τα αποτελέσματα του ελέγχου αυτού, θα συζητήσουμε εν συντομία τη μοντελοποίηση μας για την αρχιτεκτονική του πυρήνα. Στον Κώδικα 4.7 το αποτέλεσμα:

```
r_x == 0 && r_y == 1
```
(4.1)

παραβιάζει την εγγύηση αυτή.

Πρώτα απ' όλα, έχουμε ένα σύστημα με δύο πυρήνες οι οποίοι αναπαρίστανται από δύο mutexes. Επίσης, έχουμε τρία βασικά νήματα: τον εγγραφέα, τον αναγνώστη και το νήμα πυρήνα για τις περιόδους χάριτος του RCU-sched. Το νήμα πυρήνα για τις περιόδους χάριτος του RCU-bh έχει απενεργοποιηθεί έτσι ώστε να μειωθεί ο χώρος καταστάσεων, αλλά μπορεί να ενεργοποιηθεί ορίζοντας `-DENABLE_RCU_BH`. Μπορούμε να υποθέσουμε ότι ο εγγραφέας και το νήμα για τις περιόδους χάριτος του RCU τρέχουν στον ίδιο επεξεργαστή (π.χ. στον επεξεργαστή 0), και ότι ο αναγνώστης τρέχει στον άλλον επεξεργαστή (π.χ. στον επεξεργαστή 1). Διαφορετικοί συνδυασμοί δοκιμάστηκαν επίσης, και δεν επηρέασαν το αποτέλεσμα των ελέγχων. Η συγκεκριμένη παραμετροποίηση επιλέχθηκε επειδή με τον τρόπο αυτό ο εγγραφέας και το νήμα για τις περιόδους χάριτος του RCU εκμεταλλεύονται ο ένας τις χρονοδρομολογήσεις του άλλου. Ας σημειωθεί ότι θα μπορούσαμε να αγνοήσουμε εντελώς το νήμα για τις περιόδους χάριτος του RCU και να καλούμε χειροκίνητα τις συναρτήσεις `rcu_gp_init()` και `rcu_gp_cleanup()` για να μειωθεί ο χώρος καταστάσεων. Ωστόσο, αυτό αποτελεί μία προσέγγιση του τρόπου που λειτουργεί ένας πραγματικός πυρήνας, και γι' αυτό κρατήσαμε το νήμα πυρήνα για τις περιόδους χάριτος του RCU στους ελέγχους που πραγματοποιήσαμε. Για την αρχικοποίηση του RCU, καλείται η συνάρτηση `rcu_init()`. Επειδή υπάρχουν μόνο δύο επεξεργαστές στη μοντελοποίησή μας, δημιουργείται μια ιεραρχία ενός κόμβου μόνο. Όλοι οι επεξεργαστές ξεκινούν σε λειτουργία `dynticks-idle` (η συνάρτηση `rcu_idle_enter()` καλείται για κάθε επεξεργαστή του συστήματος), και η συνάρτηση `rcu_spawn_gp_kthread()` καλείται για να δημιουργηθεί το νήμα πυρήνα για τις περιόδους χάριτος του RCU.

Φυσικά, το περιβάλλον διακοπών έπρεπε κι αυτό με τη σειρά του να μοντελοποιηθεί. Γι' αυτό, υπάρχουν κλήσεις στη συνάρτηση `do_IRQ()` σε διάφορα σημεία του προγράμματος ελέγχου. Για παράδειγμα, δοκιμάσαμε να εισάγουμε μία κλήση στη `do_IRQ()` ακριβώς μετά τη στιγμή που ο εγγραφέας καταχωρίζει μια επανάκληση και περιμένει στη μεταβλητή ολοκλήρωσης (δηλαδή αφού καλέσει την `synchronize_rcu()`). Αν και αυτό δεν συμβαίνει πια στο πρόγραμμα ελέγχου μας, η κλήση αυτή αντιπροσώπευε μία διακοπή που συνέβη *αφού* ο εγγραφέας έχει καταχωρίσει μια επανάκληση. Γενικά, αν και δεν μας νοιάζει η ακριβής στιγμή που προκύπτει μια διακοπή, η εξυπηρέτηση μιας διακοπής μέσα σε ένα περιβάλλον ταυτοχρονισμού είναι αυτό που κάνει τις περιόδους χάριτος να τελειώνουν.

Έτσι, κλήσεις στη συνάρτηση `do_IRQ()` έχουν εισαχθεί σε σημεία που επιτρέπουν να σημειωθεί πρόοδος για μια περίοδο χάριτος. Η πρόοδος αυτή μπορεί να μην συμβαίνει πάντα (π.χ. μια περίοδος χάριτος μπορεί να μην τελειώνει για ορισμένες εξερευνημένες εκτελέσεις), αλλά γενικά θέλουμε και τα δύο αυτά σενάρια να μπορούν να συμβούν. Ένα σύνολο με κλειδώματα για διακοπές και απενεργοποίηση διακοπών (ένα ανά επεξεργαστή), και η μοντελοποίηση των διακοπών με ένα ξεχωριστό νήμα (βλ. Ενότητα 4.2) θα ήταν επίσης καλή προσέγγιση, αλλά αυτό θα δημιουργούσε έναν πολύ μεγάλο χώρο καταστάσεων.

Αφού λοιπόν τρέξαμε το πρόγραμμα ελέγχου (ξεδιπλώνοντας τις δομές επανάληψης,

έτσι ώστε το πρόγραμμα ελέγχου να είναι πεπερασμένο), το Nidhugg αποφάνθηκε ότι δεν υπάρχουν σφάλματα. Χρειάστηκε περίπου 17 λεπτά (1051.83 δευτερόλεπτα), υπό ακολουθιακή συνέπεια (βλ. πρώτη γραμμή, Πίνακας 4.1). Αλλά μπορούμε να το εμπιστευτούμε; Εξάλλου, μπορεί να υπάρχει κάποιο σφάλμα στη μοντελοποίησή μας του περιβάλλοντος του πυρήνα, ή μπορεί να υπάρχει κάποιο σφάλμα στο Nidhugg.

Για να ενδυναμώσουμε το αποτέλεσμά μας, βάλαμε εμείς κάποια σφάλματα στον κώδικα του προγράμματος ελέγχου μας, αλλά και στον πηγαίο κώδικα του RCU (για τις αντίστοιχες τροποποιήσεις στον πηγαίο κώδικα του RCU, βλέπε Παράρτημα A). Υπάρχουν δύο είδη εισαγόμενων σφαλμάτων:

1. Σφάλματα που κάνουν την περίοδο χάριτος πολύ σύντομη, επιτρέποντας έτσι στο κρίσιμο τμήμα του αναγνώστη να είναι μεγαλύτερης διάρκειας από την περίοδο χάριτος.

2. Σφάλματα που κάνουν την περίοδο χάριτος να μην μπορεί να τελειώσει.

Και οι δύο παραπάνω τύποι σφαλμάτων αντιπροσωπεύουν αποτυχίες του RCU. Ο πρώτος τύπος οδηγεί σε αποτυχία στον συστηματικό έλεγχο ορθότητας, αφού η Επιβεβαίωση 4.1 παραβιάζεται. Ο δεύτερος τύπος πρέπει να χρησιμοποιηθεί μαζί με μία εντολή `assert(0)` μετά τη `synchronize_rcu()` για να δειχθεί ότι η περίοδος χάριτος δεν τελειώνει, και οδηγεί σε επιτυχημένο συστηματικό έλεγχο ορθότητας.

Στη συνέχεια, παρουσιάζουμε μία λίστα με εντολές προς τον προεπεξεργαστή που ενεργοποιούν σενάρια όπως τα παραπάνω, μαζί με μία επεξήγηση για την κάθε εντολή και το αποτέλεσμα του ελέγχου:

`-DASSERT_0`: Βάζει μια εντολή `assert(0)` μετά τη `synchronize_rcu()`. Αυτό οδηγεί σε αποτυχημένο έλεγχο ορθότητας, όπως αναμενόταν. Αυτό το `assert()` στην ουσία δείχνει ότι η περίοδος χάριτος μπορεί να τελειώσει, και ότι υπάρχουν μερικές εξερευνούμενες εκτελέσεις στις οποίες τελειώνει. Με άλλα λόγια, παρέχει εγγυήσεις ζωτικότητας (liveness). Θα χρησιμοποιήσουμε αυτή την εντολή σε συνδυασμό με κάποιες από τις επόμενες εντολές για να διαπιστώσουμε αν η περίοδος χάριτος τελειώνει ή όχι.

`-DFORCE_FAILURE_1`: Αναγκάζει τον αναγνώστη να περάσει από κατάσταση ηρεμίας και να το αναφέρει στο RCU κατά τη διάρκεια του κρίσιμου τμήματός του. Όπως αναμένεται, αυτό οδηγεί σε αποτυχία του ελέγχου ορθότητας με παρόμοιο τρόπο όπως στην περίπτωση του Tiny RCU (βλ. Ενότητα 3.3).

`-DFORCE_FAILURE_2`: Κάνει την `synchronize_rcu()` να επιστρέφει αμέσως. Αυτό οδηγεί σε αποτυχία του ελέγχου ορθότητας αφού ο εγγραφέας δεν περιμένει την ολοκλήρωση των κρίσιμων τμημάτων των προϋπαρχόντων αναγνωστών.

`-DFORCE_FAILURE_3`: Κάνει την `rcu_gp_init()` να μηδενίζει τις μεταβλητές `->qsmask` αντί να τους θέτει μη μηδενικές τιμές. Αυτή είναι μια πιο περίπλοκη εκδοχή του σεναρίου #2 παραπάνω. Αφού οι μεταβλητές `->qsmask` μηδενίζονται στην αρχή της περιόδου χάριτος, η περίοδος χάριτος μπορεί να τελειώσει κατευθείαν. Αυτό το σενάριο οδηγεί σε αποτυχία του συστηματικού ελέγχου ορθότητας.

`-DFORCE_FAILURE_4`: Κάνει την `rcu_gp_fqs()` να μηδενίζει τις μεταβλητές `->qsmask` αντί να περιμένει τους επεξεργαστές να τις μηδενίσουν. Αυτή είναι μία ακόμη πιο περίπλοκη εκδοχή του σεναρίου #2 παραπάνω, και οδηγεί επίσης σε αποτυχημένο έλεγχο ορθότητας. Στον κώδικάς μας, ο επεξεργαστής 0 καλεί την `rcu_gp_fqs()`, ενώ ο επεξεργαστής 1 εισέρχεται σε λειτουργία `dynticks-idle` μέσα στο κρίσιμο τμήμα του, κάτι που επιτρέπει στον επεξεργαστή 0 να τελειώσει πρώιμα την περίοδο χάριτος.

-DFORCE_FAILURE_5: Κάνει την `__note_gp_changes()` να μηδενίσει το bit του επεξεργαστή στη μεταβλητή `rnp->qsmask` (`rnp->qsmask &= ~rdp->grpmask`). Αυτό γίνεται στο σώμα του τελευταίου **if** (αυτό που εκτελείται όταν ένας επεξεργαστής εντοπίσει την έναρξη μιας νέας περιόδου χάριτος), και οδηγεί σε αποτυχία του συστηματικού ελέγχου ορθότητας.

-DFORCE_FAILURE_6: Διαγράφει το **if** που ελέγχει για μηδενική τιμή της `->qsmask` και καλεί τη συνάρτηση `rcu_preempt_blocked_readers_cgp()`, στη συνάρτηση `rcu_report_qs_rnp()`. Σε έναν πραγματικό πυρήνα, αυτό θα οδηγούσε σε πολύ σύντομες περιόδους χάριτος, μιας και ένα σήμα θα ξύπναγε πρώιμα το νήμα πυρήνα για τις περιόδους χάριτος (τουλάχιστον υποθέτοντας ότι υπάρχουν πολλοί επεξεργαστές στο σύστημα). Στην περίπτωσή μας, ωστόσο, δεν οδηγεί σε πολύ σύντομες περιόδους χάριτος, όπως θα εξηγήσουμε στις επόμενες παραγράφους. Αν αυτό το **if** διαγραφεί, και επειδή η συνθήκη `rnp->parent == NULL` ισχύει, η `rcu_report_qs_rsp()` θα κληθεί. Όμως η συνάρτηση `wait_event_interruptible_timeout()` απλά περιμένει μέχρι μία συνθήκη να γίνει αληθής. Άρα, ακόμη και με την `rcu_report_qs_rsp()` να καλείται, μέχρι να καθαρίσουν το bit τους και οι δύο επεξεργαστές, είναι αδύνατον για το νήμα πυρήνα για τις περιόδους χάριτος του RCU να βγει από την `wait_event_interruptible_timeout()` και να τελειώσει την περίοδο χάριτος, αφού δεν έχουν καθαριστεί τα bit και των δύο επεξεργαστών. Ας σημειωθεί ότι, στην μοντελοποίησή μας, η `wake_up()` δεν γεννά καθόλου κώδικα – δεν υπάρχει ανάγκη να ξυπνήσει κάποιος που απλά περιμένει να γίνει αληθής μια συνθήκη.

Ωστόσο, αν είχαμε μία ιεραρχία δύο επιπέδων, αυτός που καλεί την `rcu_report_qs_rnp()` θα ανέβαινε ένα επίπεδο και θα πυροδοτούσε την εντολή `WARN_ON_ONCE()` που ελέγχει κατά πόσον τα bit του κόμβου-παιδιού έχουν καθαριστεί. Έτσι, το σενάριο αυτό θέτει αυτόματα τον αριθμό των επεξεργαστών σε `CONFIG_RCU_FANOUT_LEAF + 1` (δηλαδή 17, αφού η προεπιλογή του `CONFIG_RCU_FANOUT_LEAF` είναι 16 στην έκδοση v3.19 του πυρήνα), και χρησιμοποιεί τιμή αναδίπλωσης των δομών επανάληψης ίση με 19. Αυτό γίνεται επειδή υπάρχουν μερικές δομές επανάληψης που πρέπει να εκτελεστούν τουλάχιστον τόσες φορές όσες ο αριθμός των επεξεργαστών που χρησιμοποιεί το πρόγραμμα ελέγχου συν μία. Υπό αυτές τις συνθήκες, το σενάριο αυτό οδηγεί σε αποτυχία του ελέγχου ορθότητας.

-DLIVENESS_CHECK_1: Μηδενίζει τη μεταβλητή `rdp->qs_pending` στην `__note_gp_changes()`. Αυτό πρέπει να εμποδίζει τις περιόδους χάριτος απ' το να τελειώσουν. Όταν χρησιμοποιείται σε συνδυασμό με το `-DASSERT_0`, οδηγεί σε επιτυχία του συστηματικού ελέγχου ορθότητας. Αυτό σημαίνει ότι δεν παραβιάζεται κανένα `assert()`, άρα το `assert(0)` μετά την `synchronize_rcu()` αποτελεί μία κατάσταση στην οποία δεν φτάνει ποτέ ο έλεγχος, άρα η περίοδος χάριτος δεν τελειώνει.

-DLIVENESS_CHECK_2: Βάζει ένα **return** στην αρχή της `rcu_sched_qs()`. Αυτό επίσης εμποδίζει τις περιόδους χάριτος απ' το να τελειώσουν, και πρέπει να χρησιμοποιηθεί σε συνδυασμό με το `-DASSERT_0`. Οδηγεί σε επιτυχία του ελέγχου ορθότητας.

-DLIVENESS_CHECK_3: Βάζει ένα **return** στην αρχή της `rcu_report_qs_rnp()`. Αυτό εμποδίζει τις περιόδους χάριτος από το να ολοκληρωθούν, πρέπει να χρησιμοποιείται μαζί με το `ASSERT_0`, και οδηγεί σε επιτυχή συστηματικό ελέγχο ορθότητας.

Για τον έλεγχο `-DFORCE_FAILURE_6` χρησιμοποιήθηκε αναδίπλωση `unroll=19` και η τιμή του `CONFIG_NR_CPUS` τέθηκε ανάλογα. Για όλους τους υπόλοιπους ελέγχους χρησιμοποιήθηκε `unroll=5`. Όλοι οι έλεγχοι πραγματοποιήθηκαν σε περιβάλλον Debian Linux 64-bit με έναν επεξεργαστή Intel E8400 δύο πυρήνων, και 2GB RAM. Όπως φαίνεται στον Πίνακα 4.1, όλοι οι έλεγχοι είχαν το επιθυμητό αποτέλεσμα, κάτι που ενισχύει σημαντικά

Preprocessor Options	*Expected*	*Nidhugg*	*SC*			*TSO*		
			Time	*E/B*	*Memory*	*Time*	*E/B*	*Memory*
-	Success	Success	1051.83	24 740/20	64.34	1130.94	24 740/20	64.84
`-DASSERT_0`	Failure	Failure	2.08	31/6	34.42	2.29	31/6	34.40
`-DFORCE_FAILURE_1`	Failure	Failure	2.20	35/6	34.43	2.55	35/6	34.37
`-DFORCE_FAILURE_2`	Failure	Failure	0.80	3/0	34.24	0.89	3/0	34.28
`-DFORCE_FAILURE_3`	Failure	Failure	567.86	13 256/8	47.92	612.75	13 256/8	49.74
`-DFORCE_FAILURE_4`	Failure	Failure	4.40	73/6	34.50	4.97	73/6	34.54
`-DFORCE_FAILURE_5`	Failure	Failure	1.34	8/1	34.62	1.46	8/1	34.86
`-DFORCE_FAILURE_6`	Failure	Failure	4.58	2/0	109.75	4.93	2/0	109.87
`-DLIVENESS_CHECK_1 -DASSERT_0`	Success	Success	15.62	588/20	34.38	17.08	588/20	34.29
`-DLIVENESS_CHECK_2 -DASSERT_0`	Success	Success	15.52	588/20	34.82	17.09	588/20	34.82
`-DLIVENESS_CHECK_3 -DASSERT_0`	Success	Success	17.70	668/20	34.52	21.26	668/20	34.68

Πίνακας 4.1: Αποτελέσματα για τον βασικό έλεγχο του Tree RCU (χρόνος σε δευτερόλεπτα, μνήμη σε MB).

τα αποτελέσματά μας.

Στον Πίνακα 4.1, η δεύτερη στήλη δείχνει το αναμενόμενο αποτέλεσμα και η τρίτη στήλη δείχνει το αποτέλεσμα που πήραμε από το Nidhugg. Οι στήλες "Time" δείχνουν τον συνολικό χρόνο (wall-clock time) σε δευτερόλεπτα, ο οποίος περιλαμβάνει τον χρόνο μεταγλώττισης και τον χρόνο μετασχηματισμού του Nidhugg. Οι στήλες με την ετικέτα E/B παρουσιάζουν τον αριθμό των εξερευνημένων και των σταματημένων (λόγω της παρουσίας τους στο sleep-set) εκτελέσεων υπό SC και TSO. Οι σταματημένες εκτελέσεις είναι εκτελέσεις που εξερευνήθηκαν μερικώς και χωρίς να υπάρχει ανάγκη. Η εξερεύνησή τους αυτή θα μπορούσε να αποφευθχεί με χρήση ενός καλύτερου αλγορίθμου DPOR, όπως ο optimal-DPOR [Abdu14].

Όπως φαίνεται επίσης από τον Πίνακα 4.1 δεν υπάρχει σοβαρό τίμημα όταν πηγαίνουμε από SC σε TSO, κάτι που είναι αναμενόμενο όταν χρησιμοποιούμε χρονολογικά ίχνη (chronological traces) [Abdu15]. Αυτό που παρουσιάζει ενδιαφέρον όμως, είναι το γεγονός ότι ο συνολικός αριθμός εξερευνούμενων εκτελέσεων για SC και TSO είναι ο ίδιος, κάτι που οφείλεται στο ότι υπάρχουν πολλοί φράχτες μνήμης στον πηγαίο κώδικα του Tree RCU οι οποίοι εμποδίζουν πιθανές αναδιατάξεις των εντολών. Αλλά, ακόμη κι αν συνέβαιναν αναδιατάξεις, αυτά τα σενάρια εισαγωγής σφαλμάτων δεν βασίζονται σε κάποιο ασθενές μοντέλο μνήμης που εφαρμόζεται, αλλά παραβιάζουν κάποιες εγγυήσεις αλγοριθμικά.

Τέλος, όπως αναμενόταν, η μνήμη που χρησιμοποιήθηκε σε όλους τους ελέγχους ήταν πολύ μικρή, δεδομένου του μεγέθους του κώδικα που ελέγχθηκε. Επιπρόσθετα, παρατηρούμε ότι το τίμημα όταν πηγαίνουμε από SC σε TSO είναι πρακτικά μηδενικό. Αυτό οφείλεται στο ότι ο αριθμός των εξερευνούμενων εκτελέσεων σε SC και TSO είναι ο ίδιος. Στο σενάριο `-DFORCE_FAILURE_6`, η κατανάλωση μνήμης είναι αυξημένη λόγω του `unroll=19` και του αυξημένου αριθμού των δομών δεδομένων του RCU (`CONFIG_NR_CPUS=17`).

```
void *thread_update(void *arg)
{
        set_cpu(cpu0);
        fake_acquire_cpu(get_cpu());

        call_rcu(&cb1, dummy);
        cond_resched();
        do_IRQ();

        x = 1;
        synchronize_rcu();
        y = 1;

        fake_release_cpu(get_cpu());
        return NULL;
}

void *thread_helper(void *arg)
{
        set_cpu(cpu0);
        fake_acquire_cpu(get_cpu());

        do_IRQ();

        fake_release_cpu(get_cpu());
        return NULL;
}

void *thread_reader(void *arg)
{
        set_cpu(cpu1);
        fake_acquire_cpu(get_cpu());

        do_IRQ();
        cond_resched();

        rcu_read_lock();
        r_x = x;
        do_IRQ();
        r_y = y;
        rcu_read_unlock();

        fake_release_cpu(get_cpu());
        return NULL;
}

int main()
{
        /* ... */
        /* Initializations and creation of threads */
        /* ... */

        BUG_ON(r_x == 0 && r_y == 1);

        return 0;
}
```

Listing 4.4: Πολύ σύντομη περίοδος χάριτος που οφείλεται σε θέματα συγχρονισμού στην `rcu_process_gp_end()`.

```
1057  if (ACCESS_ONCE(rsp->completed) == ACCESS_ONCE(rsp->gpnum))
1058          return;  /* No grace period in progress, nothing to force. */
1059  if (!spin_trylock_irqsave(&rsp->fqslock, flags)) {
1060          rsp->n_force_qs_lh++; /* Inexact, can lose counts.  Tough! */
1061          return;       /* Someone else is already on the job. */
1062  }
1063  if (relaxed && (long)(rsp->jiffies_force_qs - jiffies) >= 0)
1064          goto unlock_ret; /* no emergency and done recently. */
1065  rsp->n_force_qs++;
1066  spin_lock(&rnp->lock);
1067  lastcomp = rsp->completed;
1068  signaled = rsp->signaled;
1069  rsp->jiffies_force_qs = jiffies + RCU_JIFFIES_TILL_FORCE_QS;
1070  if (lastcomp == rsp->gpnum) {
1071          rsp->n_force_qs_ngp++;
1072          spin_unlock(&rnp->lock);
1073          goto unlock_ret;  /* no GP in progress, time updated. */
1074  }
1075  spin_unlock(&rnp->lock);
1076  switch (signaled) {
1077  case RCU_GP_INIT:
1078
1079          break; /* grace period still initializing, ignore. */
1080
1081  case RCU_SAVE_DYNTICK:
1082
1083          if (RCU_SIGNAL_INIT != RCU_SAVE_DYNTICK)
1084                  break; /* So gcc recognizes the dead code. */
1085
1086          /* Record dyntick-idle state. */
1087          if (rcu_process_dyntick(rsp, lastcomp, dyntick_save_progress_counter))
1088                  goto unlock_ret;
1089
1090          /* Update state, record completion counter. */
1091          spin_lock(&rnp->lock);
1092          if (lastcomp == rsp->completed) {
1093
1094                  rsp->signaled = RCU_FORCE_QS;
1095                  dyntick_record_completed(rsp, lastcomp);
1096          }
1097          spin_unlock(&rnp->lock);
1098          break;
1099
1100  case RCU_FORCE_QS:
```

Listing 4.5: Απόσπασμα της `force_quiescent_state()` στον πυρήνα v2.6.31.1. Κώδικας από το αρχείο `kernel/rcutree.c`.

```
bool add_gp(int x, int y)
{
        struct foo *p;

        p = calloc(1, sizeof(*p));
        if (!p)
                return -ENOMEM;
        spin_lock(&gp_lock);
        if (rcu_access_pointer(gp)) {
                spin_unlock(&gp_lock);
                return false;
        }
        p->a = x;
        p->b = x;
#ifdef ORDERING_BUG
        gp = p;
#else
        rcu_assign_pointer(gp, p);
#endif
        spin_unlock(&gp_lock);
        return true;
}

bool use_gp(void)
{
        struct foo *p;

        rcu_read_lock();
        p = rcu_dereference(gp);
        if (p) {
                BUG_ON(p->a != 42 || p->b != 42);
                /* do something with p->a, p->b */
                rcu_read_unlock();
                return true;
        }
        rcu_read_unlock();
        return false;
}

void *thread_publisher(void *arg)
{
        add_gp(42, 42);
        return NULL;
}

void *thread_subscriber(void *arg)
{
        use_gp();
        return NULL;
}
```

Listing 4.6: Πρόγραμμα για την επιβεβαίωση της εγγύησης Έκδοσης-Συνδρομής του Tree RCU.

```
int r_x;
int r_y;

int x;
int y;

void *thread_reader(void *arg)
{
        set_cpu(cpu1);
        fake_acquire_cpu(get_cpu());

        rcu_read_lock();
        r_x = x;
        do_IRQ();
        r_y = y;
        rcu_read_unlock();
        cond_resched();
        do_IRQ();

        fake_release_cpu(get_cpu());
        return NULL;
}

void *thread_update(void *arg)
{
        set_cpu(cpu0);
        fake_acquire_cpu(get_cpu());

        x = 1;
        synchronize_rcu();
        y = 1;

        fake_release_cpu(get_cpu());
        return NULL;
}
```

Listing 4.7: Πρόγραμμα για την επαλήθευση της εγγύησης Περιόδου Χάριτος του Tree RCU.

Κεφάλαιο 5

Συμπεράσματα

Στην εργασία αυτή χρησιμοποιήσαμε το εργαλείο ελέγχου μοντέλου χωρίς αποθήκευση της κατάστασης Nidhugg για τον συστηματικό έλεγχο ορθότητας των βασικών εγγυήσεων τόσο του Tree RCU, όσο και του Tiny RCU, παίρνοντας ως είσοδο ολόκληρο τον πηγαίο κώδικα από τις υλοποιήσεις αυτές.

Συγκεκριμένα, περιγράψαμε την βασική εγγύηση που κάθε υλοποίηση του RCU οφείλει να προσφέρει (εγγύηση Περιόδου Χάριτος), καθώς και ορισμένες επιπλοκές που υπάρχουν λόγω των σημερινών μεταγλωττιστών κι επεξεργαστών, καθώς και τον τρόπο που το RCU τις ξεπερνά (εγγύηση Έκδοσης-Συνδρομής). Τέλος, κατασκευάσαμε προγράμματα που ελέγχουν κατά πόσον οι εγγυήσεις αυτές τηρούνται ή παραβιάζονται.

Επαληθεύσαμε την ιδιότητα της Περιόδου Χάριτος τόσο για το Tiny RCU, όσο και για το Tree RCU, υπό ένα ακολουθιακά συνεπές και ένα ασθενές μοντέλο μνήμης με ολική διάταξη των εγγραφών (TSO). Για το Tree RCU, ένας έλεγχος παρόμοιας λογικής πραγματοποιήθηκε για τις εκδόσεις v3.0 και v3.19 του πυρήνα του Linux, με τον αντίστοιχο έλεγχο για την έκδοση v3.19 να είναι αρκετά πιο σημαντικός λόγω μεγάλων διαφορών στην υλοποίηση του RCU στις δύο αυτές εκδόσεις (π.χ. το νήμα πυρήνα για τις περιόδους χάριτος). Επιπρόσθετα, πραγματοποιήσαμε επαληθεύσαμε την εγγύηση Έκδοσης-Συνδρομής για το Tree RCU υπό ένα μοντέλο μνήμης TSO και ένα μοντέλο μνήμης POWER.

Για να δείξουμε ότι η μοντελοποίηση μας του περιβάλλοντος του πυρήνα είναι ορθή και για να ενδυναμώσουμε περαιτέρω τα αποτελέσματά μας, αναπαρηγάγαμε ένα γνωστό σφάλμα (bug) του πυρήνα και δείξαμε ότι ένα παλαιά αναφερόμενο ως σφάλμα στην πραγματικότητα δεν αποτελεί σφάλμα [LKMLb, LKMLc]. Είναι εξαιρετικά πιθανό οι διορθώσεις που έγιναν για το δεύτερο να έφτιαξαν το πρώτο, καθώς μόνο ένα μικρό διάστημα μεσολαβούσε μεταξύ των υποβληθεισών επιδιορθώσεων [McKe09a, LKMLa].

Η εργασία αυτή επισημαίνει με τον καλύτερο τρόπο ότι το Nidhugg και ο έλεγχος μοντέλου χωρίς αποθήκευση της κατάστασης μπορούν να χρησιμοποιηθούν για τον συστηματικό έλεγχο *πραγματικών* συστημάτων βιομηχανικών προδιαγραφών μεγάλης κλίμακας. Στους ελέγχους που πραγματοποιήσαμε, χρησιμοποιήσαμε τον πηγαίο κώδικα από τέσσερις διαφορετικές εκδόσεις του πυρήνα του Linux σχεδόν ατόφιο, ενώ οι ελάχιστες αλλαγές που έγιναν θα μπορούσαν να πραγματοποιηθούν αυτόματα από κάποιο πρόγραμμα (script). Σημειώνεται ότι οι μόνες αλλαγές που έγιναν στον κώδικα του πυρήνα αφορούσαν στη χρήση per-CPU μεταβλητών. Το γεγονός ότι το Nidhugg ήταν παρείχε αντιπαραδείγματα σφαλμάτων στους ελέγχους που πραγματοποιήσαμε μέσα σε λίγα χιλιοστά του δευτερολέπτου ή σε λίγα δευτερόλεπτα τονίζει την αποτελεσματικότητα της προσέγγισής μας.

Μελλοντικές Επεκτάσεις

Η διπλωματική εργασία αυτή μπορεί να επεκταθεί με πολλούς τρόπους:

1. Περισσότεροι και πιο πολύπλοκοι έλεγχοι που αφορούν τον εξαναγκασμό κατάστασης ηρεμίας (quiescent state forcing) θα μπορούσαν να προστεθούν.

2. Διαφορετικοί έλεγχοι για την επίσπευση περιόδων χάριτος (grace-period expediting) θα μπορούσαν επίσης να πραγματοποιηθούν.

3. Έλεγχοι που περιλαμβάνουν τη δυναμική αφαίρεση επεξεργαστικών μονάδων από το σύστημα εν ώρα λειτουργίας θα μπορούσαν να συμπεριληφθούν.

4. Η πλήρης λειτουργία χωρίς διακοπές του πυρήνα (full-dynticks mode) θα μπορούσε να ενεργοποιηθεί.

5. Οι διακοπές και τα softirqs θα μπορούσαν να μοντελοποιηθούν με ξεχωριστά νήματα.

6. Έλεγχοι παρόμοιας λογικής για προεκχωρητικούς πυρήνες θα παρουσίαζαν επίσης μεγάλο ενδιαφέρον.

Βιβλιογραφία

[Abdu14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Optimal Dynamic Partial Order Reduction", in *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pp. 373–384, New York, NY, USA, 2014, ACM.

[Abdu15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson and Konstantinos Sagonas, "Stateless Model Checking for TSO and PSO", in *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pp. 353–367, New York, NY, USA, 2015, Springer-Verlag New York, Inc.

[Ahme15] Iftekhar Ahmed, Alex Groce, Carlos Jensen and Paul E. McKenney, "How Verified is My Code? Falsification-Driven Verification", in *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 737–748, November 2015.

[Algl13] Jade Alglave, Daniel Kroening and Michael Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software", in *Proceedings of the 25th International Conference on Computer Aided Verification*, pp. 141–157, 2013.

[AMDC] "Cool'n'Quiet". Available: https://en.wikipedia.org/wiki/Cool%27n%27Quiet.

[Arca03] Andrea Arcangeli, Mingming Cao, Paul E. McKenney and Dipankar Sarma, "Using Read-Copy Update Techniques for System V IPC in the Linux 2.5 Kernel", in *Proceedings of the 2003 USENIX Annual Technical Conference (FREENIX Track)*, pp. 297–310, USENIX Association, June 2003.

[Beye15] Dirk Beyer, "Rules for 4th Intl. Competition on Software Verification", in *21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 4 2015. Available: https://sv-comp.sosy-lab.org/2015/rules.php.

[Bier03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman and Yunshan Zhu, "Bounded Model Checking", *Advances in Computers*, vol. 58, 2003.

[Bove05] Daniel P. Bovet and Marco Cesati, *Understanding the Linux kernel*, O' Reilly, 3rd edition, 2005.

[Call15] John Callaham, "Google says there are now 1.4 billion active Android devices worldwide". Available: http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide, September 2015.

[Chri13] Maria Christakis, Alkis Gotovos and Konstantinos Sagonas, "Systematic Testing for Detecting Concurrency Errors in Erlang Programs", in *Sixth IEEE International Conference on Software Testing, Verification and Validation (ICST 2013)*, pp. 154–163, Los Angeles, CA, USA, 2013, IEEE Computer Society.

[Clan] "LLVM Atomic Instructions and Concurrency Guide". Available: http://llvm.org/docs/Atomics.html#libcalls-atomic.

[Clar99] Edmund M. Clarke, Orna Grumberg, Marius Minea and Doron A. Peled, "State Space Reduction Using Partial Order Techniques", *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 3, pp. 279–287, 1999.

[Clar04] Edmund Clarke, Daniel Kroening and Flavio Lerda, "A tool for checking ANSI-C programs", in *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 168–176, Springer, 2004.

[Desn09] Mathieu Desnoyers, *Low-Impact Operating System Tracing*, Ph.D. thesis, Ecole Polytechnique de Montréal, December 2009. Available: http://www.lttng.org/pub/thesis/desnoyers-dissertation-2009-12.pdf.

[Desn12] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais and Jonathan Walpole, "User-Level Implementations of Read-Copy Update", *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 2, pp. 375–382, February 2012.

[Desn13] Mathieu Desnoyers, Paul E. McKenney and Michel R. Dagenais, "Multi-core Systems Modeling for Formal Verification of Parallel Algorithms", *SIGOPS Oper. Syst. Rev.*, vol. 47, no. 2, pp. 51–65, July 2013.

[Dijk] Edsger W. Dijkstra, "Over de sequentialiteit van procesbeschrijvingen". circulated privately.

[Dugg10] Abhinav Duggal, *Stopping Data Races Using Redflag*, Ph.D. thesis, Stony Brook University, 2010.

[Flan05] Cormac Flanagan and Patrice Godefroid, "Dynamic Partial-order Reduction for Model Checking Software", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pp. 110–121, New York, NY, USA, 2005, ACM.

[GCCA] "Built-in Functions for Memory Model Aware Atomic Operations". Available: https://gcc.gnu.org/onlinedocs/gcc/_005f_005fatomic-Builtins.html.

[Gode96] Patrice Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[Gode97] Patrice Godefroid, "Model checking for programming languages using VeriSoft", in *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 147–186, 1997.

[Gode05] Patrice Godefroid, "Software Model Checking: The VeriSoft Approach", *Formal Methods in System Design*, vol. 26, no. 2, pp. 77–101, 2005.

[Gots13] Alexey Gotsman, Noam Rinetzky and Hongseok Yang, "Verifying Concurrent Memory Reclamation Algorithms with Grace", in *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP'13, pp. 249–269, Berlin, Heidelberg, 2013, Springer-Verlag.

[Howe] David Howells, Paul E. McKenney, Will Deacon and Peter Zijlstra, "Linux Kernel Memory Barriers". Available: https://www.kernel.org/doc/Documentation/memory-barriers.txt.

[Inte] "Power Management States: P-States, C-States, and Package C-States". Available: https://software.intel.com/en-us/articles/power-management-states-p-states-c-states-and-package-c-states.

[Kerna] "Linux Kernel Documentation". Available: https://www.kernel.org/doc/.

[Kernb] "NO_HZ: Reducing Scheduling-Clock Ticks". Available: https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt.

[Kernc] "RCU Concepts". Available: https://www.kernel.org/doc/Documentation/RCU/rcu.txt.

[Kernd] "RCU Requirements". Available: https://www.kernel.org/doc/Documentation/RCU/Design/Requirements/.

[Linu] "The Linux kernel". https://www.kernel.org/.

[LKMLa] "rcu: clean up locking for `->completed` and `->gpnum` fields". https://lkml.org/lkml/2009/10/30/212.

[LKMLb] "rcu: fix long-grace-period race between forcing and initialization". https://lkml.org/lkml/2009/10/28/196.

[LKMLc] "rcu: Fix synchronization for `rcu_process_gp_end()` uses of `->completed` counter". https://lkml.org/lkml/2009/11/4/69.

[Love10] Robert Love, *Linux Kernel Development*, Addison-Wesley, 3rd edition, 2010.

[McKe] Paul E. McKenney, "RCU Linux Usage". Available: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html.

[McKe98] Paul E. McKenney and John D. Slingwine, "Read-Copy Update: Using Execution History to Solve Concurrency Problems", in *Parallel and Distributed Computing and Systems*, pp. 509–518, Las Vegas, NV, October 1998.

[McKe04] Paul E. McKenney, *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*, Ph.D. thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf.

[McKe06] Paul E. McKenney, "Sleepable RCU". Available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf, October 2006.

[McKe07a] Paul E. McKenney, "The design of preemptible read-copy-update". Available: http://lwn.net/Articles/253651/, October 2007.

[McKe07b] Paul E. McKenney, "Using Promela and Spin to verify parallel algorithms". Available: http://lwn.net/Articles/243851/, August 2007.

[McKe07c] Paul E. McKenney and Jonathan Walpole, "What is RCU, Fundamentally?". Available: http://lwn.net/Articles/262464/, December 2007.

[McKe08a] Paul E. McKenney, "Hierarchical RCU". Available: http://lwn.net/Articles/305782/, November 2008.

[McKe08b] Paul E. McKenney, "RCU part 3: the RCU API". Available: http://lwn.net/Articles/264090/, January 2008.

[McKe08c] Paul E. McKenney, "What is RCU? Part 2: Usage". Available:, http://lwn.net/Articles/263130/, January 2008.

[McKe09a] Paul E. McKenney, "Hunting Heisenbugs". Available: http://paulmck.livejournal.com/14639.html, 11 2009.

[McKe09b] Paul E. McKenney, "RCU: The Bloatwatch Edition". Available: http://lwn.net/Articles/323929/, March 2009.

[McKe10a] Paul E. McKenney, "Lockdep-RCU". Available: https://lwn.net/Articles/371986/, February 2010.

[McKe10b] Paul E. McKenney, "The RCU API, 2010 Edition". Available: http://lwn.net/Articles/418853/, December 2010.

[McKe13] Paul E. McKenney, "Structured Deferral: Synchronization via Procrastination", *ACM Queue*, vol. 11, no. 5, May 2013. Available: https://queue.acm.org/detail.cfm?id=2488549.

[McKe14] Paul E. McKenney, "The RCU API, 2014 Edition". Available: http://lwn.net/Articles/609904/, September 2014.

[McKe15a] Paul E. McKenney, "Requirements for RCU part 1: the fundamentals". Available: http://lwn.net/Articles/652156/, July 2015.

[McKe15b] Paul E. McKenney, "Requirements for RCU part 2 – parallelism and software engineering". Available: http://lwn.net/Articles/652677/, August 2015.

[McKe15c] Paul E. McKenney, "Requirements for RCU part 3". Available: http://lwn.net/Articles/652677/, August 2015.

[McKe15d] Paul E. McKenney, "Verification Challenge 4: Tiny RCU". Available: http://paulmck.livejournal.com/39343.html, 3 2015.

[Moln] Ingo Molnar and Arjan van de Ven, "Runtime locking correctness validator". Available: https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.

[Musu08] Mandanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerald Basler, Piramanayagam Arumuga Nainar and Iulian Neamtiu, "Finding and Reproducing Heisenbugs in Concurrent Programs", in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08)*, pp. 267–280, Berkeley, CA, USA, 2008, USENIX Association.

[Pele93] Doron Peled, "All from One, One for All: On Model Checking Using Representatives", in *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pp. 409–423, London, UK, UK, 1993, Springer-Verlag.

[Rela] "Relaxed-Memory Concurrency". Available: http://www.cl.cam.ac.uk/~pes20/weakmemory/.

[Seys12] Justin Seyster, *Runtime Verification of Kernel-Level Concurrency Using Compiler-Based Instrumentation*, Ph.D. thesis, Stony Brook University, 2012.

[Spar] "Sparse - a Semantic Parser for C". Available: https://sparse.wiki.kernel.org/index.php/Main_Page.

[Tass15] Joseph Tassarotti, Derek Dreyer and Viktor Vafeiadis, "Verifying Read-copy-update in a Logic for Weak Memory", in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pp. 110–120, New York, NY, USA, 2015, ACM.

[Trip09] Josh Triplett, Paul E. McKenney and Jonathan Walpole, "Relativistic Programming". Available: http://wiki.cs.pdx.edu/rp/, September 2009.

[Valm91] Antti Valmari, "Stubborn Sets for Reduced State Space Generation", in *Proceedings of the 10th International Conference on Applications and Theory of Petri Nets: Advances in Petri Nets 1990*, pp. 491–515, London, UK, UK, 1991, Springer-Verlag.

[Wikia] "Memory ordering". Available: https://en.wikipedia.org/wiki/Memory_ordering.

[Wikib] "Model checking". Available: https://en.wikipedia.org/wiki/Model_checking.

[Wikic] "Read-copy-update". Available: https://en.wikipedia.org/wiki/Read-copy-update.

Παράρτημα Α

Τροποποιημένες Συναρτήσεις του Tree RCU

Στη συνέχεια παρατίθενται ορισμένες συναρτήσεις από το αρχείο `kernel/rcu/tree.c` οι οποίες τροποποιήθηκαν για τη διαδικασία εισαγωγής σφαλμάτων (βλ. Ενότητα 4.5). Ο σχετικός με τη διαδικασία εισαγωγής σφαλμάτων κώδικας έχει χρωματιστεί μπλε.

```
void rcu_sched_qs(void)
{
#ifdef LIVENESS_CHECK_2
        return;
#endif
        if (!rcu_sched_data[get_cpu()].passed_quiesce) {
                trace_rcu_grace_period(TPS("rcu_sched"),
                                       rcu_sched_data[get_cpu()].gpnum,
                                       TPS("cpuqs"));
                rcu_sched_data[get_cpu()].passed_quiesce = 1;
        }
}

static bool __note_gp_changes(struct rcu_state *rsp, struct rcu_node *rnp,
                              struct rcu_data *rdp)
{
        bool ret;

        /* Handle the ends of any preceding grace periods first. */
        if (rdp->completed == rnp->completed) {

                /* No grace period end, so just accelerate recent callbacks. */
                ret = rcu_accelerate_cbs(rsp, rnp, rdp);

        } else {

                /* Advance callbacks. */
                ret = rcu_advance_cbs(rsp, rnp, rdp);

                /* Remember that we saw this grace-period completion. */
                rdp->completed = rnp->completed;
                trace_rcu_grace_period(rsp->name, rdp->gpnum, TPS("cpuend"));
        }

        if (rdp->gpnum != rnp->gpnum) {
                /*
                 * If the current grace period is waiting for this CPU,
                 * set up to detect a quiescent state, otherwise don't
                 * go looking for one.
                 */
                rdp->gpnum = rnp->gpnum;
                trace_rcu_grace_period(rsp->name, rdp->gpnum, TPS("cpustart"));
                rdp->passed_quiesce = 0;
#ifdef LIVENESS_CHECK_1
                rdp->qs_pending = 0;
#else
```

```
                rdp->qs_pending = !!(rnp->qsmask & rdp->grpmask);
#endif
#ifdef FORCE_FAILURE_5
                rnp->qsmask &= ~rdp->grpmask;
#endif
                zero_cpu_stall_ticks(rdp);
        }
        return ret;
}

static int rcu_gp_init(struct rcu_state *rsp)
{
        struct rcu_data *rdp;
        struct rcu_node *rnp = rcu_get_root(rsp);

        rcu_bind_gp_kthread();
        raw_spin_lock_irq(&rnp->lock);
        smp_mb__after_unlock_lock();
        if (!ACCESS_ONCE(rsp->gp_flags)) {
                /* Spurious wakeup, tell caller to go back to sleep.  */
                raw_spin_unlock_irq(&rnp->lock);
                return 0;
        }
        ACCESS_ONCE(rsp->gp_flags) = 0; /* Clear all flags: New grace period. */

        if (WARN_ON_ONCE(rcu_gp_in_progress(rsp))) {
                /*
                 * Grace period already in progress, don't start another.
                 * Not supposed to be able to happen.
                 */
                raw_spin_unlock_irq(&rnp->lock);
                return 0;
        }

        /* Advance to a new grace period and initialize state. */
        record_gp_stall_check_time(rsp);
        /* Record GP times before starting GP, hence smp_store_release(). */
        smp_store_release(&rsp->gpnum, rsp->gpnum + 1);
        trace_rcu_grace_period(rsp->name, rsp->gpnum, TPS("start"));
        raw_spin_unlock_irq(&rnp->lock);

        /* Exclude any concurrent CPU-hotplug operations. */
        mutex_lock(&rsp->onoff_mutex);
        smp_mb__after_unlock_lock(); /* ->gpnum increment before GP! */

        /*
         * Set the quiescent-state-needed bits in all the rcu_node
         * structures for all currently online CPUs in breadth-first order,
         * starting from the root rcu_node structure, relying on the layout
         * of the tree within the rsp->node[] array.  Note that other CPUs
         * will access only the leaves of the hierarchy, thus seeing that no
         * grace period is in progress, at least until the corresponding
         * leaf node has been initialized.  In addition, we have excluded
         * CPU-hotplug operations.
         *
         * The grace period cannot complete until the initialization
         * process finishes, because this kthread handles both.
         */
        rcu_for_each_node_breadth_first(rsp, rnp) {
                raw_spin_lock_irq(&rnp->lock);
                smp_mb__after_unlock_lock();
                rdp = &rsp->rda[get_cpu()];
                rcu_preempt_check_blocked_tasks(rnp);
```

```
#ifdef FORCE_FAILURE_3
                rnp->qsmask &= ~rdp->grpmask;
#else
                rnp->qsmask = rnp->qsmaskinit;
#endif
                ACCESS_ONCE(rnp->gpnum) = rsp->gpnum;
                WARN_ON_ONCE(rnp->completed != rsp->completed);
                ACCESS_ONCE(rnp->completed) = rsp->completed;
                if (rnp == rdp->mynode)
                        (void)__note_gp_changes(rsp, rnp, rdp);
                rcu_preempt_boost_start_gp(rnp);
                trace_rcu_grace_period_init(rsp->name, rnp->gpnum,
                                            rnp->level, rnp->grplo,
                                            rnp->grphi, rnp->qsmask);
                raw_spin_unlock_irq(&rnp->lock);
                cond_resched_rcu_qs();
        }

        mutex_unlock(&rsp->onoff_mutex);
        return 1;
}

static void
rcu_report_qs_rnp(unsigned long mask, struct rcu_state *rsp,
                  struct rcu_node *rnp, unsigned long flags)
        __releases(rnp->lock)
{
        struct rcu_node *rnp_c;

#ifdef LIVENESS_CHECK_3
        return;
#endif
        /* Walk up the rcu_node hierarchy. */
        for (;;) {
                if (!(rnp->qsmask & mask)) {

                        /* Our bit has already been cleared, so done. */
                        raw_spin_unlock_irqrestore(&rnp->lock, flags);
                        return;
                }
                rnp->qsmask &= ~mask;
                trace_rcu_quiescent_state_report(rsp->name, rnp->gpnum,
                                                 mask, rnp->qsmask, rnp->level,
                                                 rnp->grplo, rnp->grphi,
                                                 !!rnp->gp_tasks);
#ifndef FORCE_FAILURE_6
                if (rnp->qsmask != 0 || rcu_preempt_blocked_readers_cgp(rnp)) {

                        /* Other bits still set at this level, so done. */
                        raw_spin_unlock_irqrestore(&rnp->lock, flags);
                        return;
                }
#endif
                mask = rnp->grpmask;
                if (rnp->parent == NULL) {

                        /* No more levels.  Exit loop holding root lock. */

                        break;
                }
                raw_spin_unlock_irqrestore(&rnp->lock, flags);
                rnp_c = rnp;
                rnp = rnp->parent;
```

```
                raw_spin_lock_irqsave(&rnp->lock, flags);
                smp_mb__after_unlock_lock();
                WARN_ON_ONCE(rnp_c->qsmask);
        }

        /*
         * Get here if we are the last CPU to pass through a quiescent
         * state for this grace period.  Invoke rcu_report_qs_rsp()
         * to clean up and start the next grace period if one is needed.
         */
        rcu_report_qs_rsp(rsp, flags); /* releases rnp->lock. */
}

static int __init rcu_spawn_gp_kthread(void)
{
        unsigned long flags;
        struct rcu_node *rnp;
        struct rcu_state *rsp;
        struct task_struct *t;

        rcu_scheduler_fully_active = 1;
#ifdef ENABLE_RCU_BH
        for_each_rcu_flavor(rsp) {
                t = kthread_run(rcu_gp_kthread, rsp, "%s", rsp->name);
                BUG_ON(IS_ERR(t));
                rnp = rcu_get_root(rsp);
                raw_spin_lock_irqsave(&rnp->lock, flags);
                rsp->gp_kthread = t;
                raw_spin_unlock_irqrestore(&rnp->lock, flags);
        }
#else
        t = kthread_run(rcu_gp_kthread, &rcu_sched_state, "%s",rcu_sched_state.name);
        rnp = rcu_get_root(&rcu_sched_state);
        raw_spin_lock_irqsave(&rnp->lock, flags);
        rcu_sched_state.gp_kthread = t;
        raw_spin_unlock_irqrestore(&rnp->lock, flags);
#endif
      rcu_spawn_nocb_kthreads();
      rcu_spawn_boost_kthreads();
      return 0;
}
early_initcall(rcu_spawn_gp_kthread);
```