Οι FIFO ουρές αποτελούν μια από της πιο ευρέως διαδεδομένες δομές δεδομένων. Έχουν μελετηθεί σε βάθος στη βιβλιογραφία και χρησιμοποιούνται σε πολλά προγραμματιστικά περιβάλλοντα. Προκειμένου να διατηρηθεί η συνέπεια της δομής σε περιβάλλον με πολλά παράλληλα νήματα, είναι απαραίτητος ο συγχρονισμός μεταξύ τους. Είναι μια δομή που από τη φύση της παρουσιάζει χαμηλή παραλληλία, αφού όλα τα reads και writes γίνονται στη κεφαλή και την ουρά, σημεία που αποτελούν hot spots. Έτσι, αναμένουμε η δομή αυτή να μην κλιμακώνει καλά , όσο ο αριθμός των threads που επιχειρούν ταυτόχρονη πρόσβαση αυξάνεται.

Ωστόσο, σε πολλές εφαρμογές, υπάρχει η ανάγκη πολλά threads να επικοινωνούν με μια κοινή ουρά, η οποία μάλιστα να μπορεί πολλές φορές να δέχεται πολλά operations ανά δευτερόλεπτο. Για το λόγο αυτό, οι concurrent υλοποιήσεις για FIFO ουρές έχουν τόσο θεωρητικό όσο και πρακτικό ενδιαφέρον.

Δεδομένης της μικρής παραλληλίας που προσφέρει η δομή, το ζήτημα της καλής επίδοσης μετατοπίζεται στην εύρεση ενός τρόπου συγχρονισμού των threads με χαμηλή συμφόρηση, καθώς και στην, όσο γίνεται , καλύτερη συμπεριφορά προς τη cache.

H πρώτη υλοποίηση ακολούθησε μια naive, coarse grain λογική. Εισάγουμε ένα global lock , το οποίο κάθε thread προσπαθεί να πάρει στην αρχή κάθε operation που εκτελεί(enqueue ή dequeue). Η επίδοση φυσικά δεν είναι καλή και η υλοποίηση δε κλιμακώνει. Ο λόγος είναι ότι μόνο ένα thread μπορεί να εκτελεί κάποια λειτουργία στη δομή κάθε φορά, ενώ τα υπόλοιπα αναλώνονται σε busy waiting πάνω στο lock. Ακόμα και όταν υλοποιήσαμε και χρησιμοποιήσαμε TTEST(Test and Test and Set) lock, που μειώνει τη συμφόρηση στο διάδρομο μνήμης λόγω του coherence protocol, η καθυστέρηση κάθε thread ήταν υψηλή.

Ένα άλλο χαρακτηριστικό είναι ότι η καθυστέρηση αυξάνεται ραγδαία όταν, ο αριθμός των παράλληλων threads ξεπερνά των αριθμό των διαθέσιμων πυρήνων. Σε αυτή τη περίπτωση, περισσότερα του ενός threads τρέχουν στον ίδιο επεξεργαστή, με αποτέλεσμα αν το threads που κατέχει το lock ανασταλεί από τον scheduler και κάποιο άλλο πάρει τον επεξεργαστή, κανένα threads δε θα προοδεύσει μέχρι να αφήσει το lock το αρχικό thread. Αυτό είναι ένα γενικό πρόβλημα που παρουσιάζουν οι locking υλοποιήσεις : Αν κάποιο thread που βρίσκεται μέσα σε critical section δεχτεί καθυστέρηση, ο πρόοδος των υπολοίπων αναστέλλεται και η επίδοση πέφτει σημαντικά.

//global lock benchmark here??

Για το σκοπό αυτό, στις FIFO ουρές, αλλά και γενικά στις δομές δεδομένων, έχει γίνει προσπάθεια να βρεθούν βιώσιμες , lock –free υλοποιήσεις. Η πρώτη επιτυχημένη υλοποίηση για lock-free FIFO queue έγινε από τους Michael και Scott και αποτελεί μέτρο σύγκρισης για τις μετέπειτα προσπάθειες.

Η ουρά υλοποιείται σαν απλή συνδεδεμένη λίστα , με ένα δείκτη Head στην αρχή της ουράς, απ’ όπου γίνονται dequeues και ένα δείκτη Tail στο τέλος όπου προσθέτουμε καινούργιους κόμβους στην ουρά. Ο κόμβος στον οποίο δείχνει ο Head θεωρείται dummy node και χρησιμοποιείται για να εξασφαλίσουμε ότι δε θα μείνει άδεια η ουρά.

Στη βάση του, ο αλγόριθμος χρησιμοποιεί atomic operations (και συγκεκριμένα Compare-And-Swap) για να μεταβάλει ατομικά τους κατάλληλους δείκτες. Κάθε φορά που ελέγχουμε ώστε να βεβαιωθούμε ότι έχουμε συνεπή εικόνα για τις μεταβλητές που θέλουμε να μεταβάλλουμε ατομικά.

Όπως φαίνεται και από το παρακάτω σχήμα, ένα enqueue απαιτεί 2 atomic operations: ένα για να συνδέσουμε τον τελευταίο κόμβο με τον καινούργιο που προσπαθούμε να εισάγουμε, και ένα για να κάνουμε τον Tail να δείχνει στον τελευταίο κόμβο της λίστας.

// εικόνα 1 από το optimistic paper

Αντίθετα, για να αφαιρέσουμε κόμβο από την ουρά, αρκεί ένα C-A-S για να ενημερωθεί η τιμή του Head.

//κωδικας για enq-deq??

Επειδή ανά πάσα στιγμή, κατά τη διάρκεια της εκτέλεσης ενός thread, οι μεταβλητές Head και Τail μπορούν να αλλάξουν τιμή, πολλές φορές τα atomic operations αποτυγχάνουν και η εκτέλεση ξεκινάει από την αρχή : διάβασμα του νέου Head/Tail και προσπάθεια αλλαγής του με ατομικό τρόπο. Επίσης, επειδή για να έχουμε επιτυχημένο enqueue, πρέπει να είναι επιτυχημένα και τα δύο CAS που απαιτούνται, είναι πιθανόν ο κόμβος να προστεθεί στην ουρά χωρίς όμως να αλλάξει η τιμή του Tail, δηλαδή να Tail να μη δείχνει πλέον στο τελευταίο κόμβο. Για το λόγο αυτό, τόσο στο enqueue όσο και sto dequeue, γίνεται έλεγχος και διόρθωση του δείκτη Tail.

Ένα από τα προβλήματα που προκύπτει κατά την υλοποίηση αυτού του αλγορίθμου, αλλά και γενικά σε μεθόδους που χρησιμοποιούν C-A-S για συγχρονισμό , είναι το λεγόμενο ABA πρόβλημα. Συνοπτικά, το πρόβλημα περιγράφεται από το εξής σενάριο: Μια διεργασία διαβάσει τη τιμή Α από μια θέση μνήμης και στη συνέχεια δέχεται μια καθυστέρηση. Στο ενδιάμεσο, μια άλλη διεργασία γράφει Β στην ίδια θέση μνήμης και στη συνέχεια την επαναφέρει ξανά σε κατάσταση Α. Η πρώτη διεργασία θα δει τη θέση μνήμης σε κατάσταση Α και το C-A-S θα επιτύχει , χωρίς να έχει επίγνωση ότι η κατάσταση της μεταβλητής άλλαξε στο ενδιάμεσο. To πρόβλημα αυτό σχετίζεται με τη έλλειψη garbage collector ή άλλου κατάλληλου μηχανισμού που εξασφαλίζει ότι δε μπορούμε να απελευθερώσουμε θέσεις μνήμης στις οποίες υπάρχει ακόμα κάποιο reference .

Για να λύσουμε αυτό το πρόβλημα επιλέξαμε να χρησιμοποιήσουμε modifications counters για κάθε pointer που μας ενδιαφέρει, τους οποίους αυξάνουμε σε κάθε επιτυχημένο C-A-S. To atomic operation τώρα δε γίνεται μεμονωμένα στον δείκτη άλλα στο ζεύγος <δείκτης,modification counter> το οποίο το μεταχειριζόμαστε σαν ενιαία μεταβλητή. Έτσι είμαστε αναγκασμένοι να κάνουμε τις αλλαγές στους δείκτες με μη παραδοσιακό τρόπο, σαν τμήμα μιας μεταβλητής, χρησιμοποιώντας εντολές για bit shifting. Εναλλακτικά, στη βιβλιογραφία υπάρχουν άλλοι τρόποι για να λυθεί το ABA problem, όπως για παράδειγμα η χρήση reference counters που δεν απαιτεί τη συνένωση pointer και counter.

Το αποτέλεσμα είναι μια lock-free υλοποίηση, που δεν απαιτεί κεντρικό κλείδωμα της δομής, αφήνει τα threads να προχωρούν όλα μαζί και η επίδοση είναι ανεπηρέαστη από τυχαίες καθυστερήσεις που μπορεί να δεχθεί κάποιο thread. Εμφανίζει έτσι βελτιωμένη επίδοση σε σχέση με το global-lock υλοποίηση , ειδικά σε περιπτώσεις που τα threads είναι περισσότερα από τα διαθέσιμα cores.

// ίσως γραφική για msqueue

// lock-freeness/ορθότητα?

Ένα από τα μειονεκτήματα των lock-free υλοποιήσεων είναι ότι κάθε φορά που ένα atomic operation αποτυγχάνει, η εκτέλεση ξεκινάει από την αρχή. Πέρα από αυτό , τα ίδια τα atomic operation εισάγουν καθυστέρηση, αφού απαιτούν συγχρονισμό στη μνήμη. Ειδικά στη περίπτωση αυτή, που απαιτούνται 2 επιτυχή atomic operations για να ολοκληρωθεί το enqueue, συμβαίνει συχνά κάποιο thread να επαναλαμβάνει την εκτέλεση του ξανά και ξανά μέχρι να γίνει σωστά. Για το λόγο αυτό θα θέλαμε όσο γίνεται να μειώσουμε τα σημεία συγχρονισμού, δηλαδή τα σημεία στα οποία γίνεται atomic operation.

Μια λύση σε αυτό το πρόβλημα δίνει η επόμενη υλοποίηση που δοκιμάσαμε, από τους Ladan-Mozes και Shavit . H υλοποίηση αυτή ακολουθεί μια optimistic λογική(γι’ αυτό και θα αναφερόμαστε σε αυτή και σαν optimistic queue) , με την έννοια ότι λειτουργεί γρήγορα με βάση την ευνοϊκή περίπτωση που δεν υπάρχει κάποιο conflict και αφήνει τα χρονοβόρα operations μόνο για τη περίπτωση που παρατηρήθηκε ασυνέπεια στη δομή. Συγκεκριμένα, ένα από τα δύο C-A-S κατά την enqueue αντικαθίσταται με απλό local store, και στην συνέχεια φροντίζει να διορθώσει τη δομή με C-A-S αν δεν είναι πλέον συνεπής.

Πρακτικά, η συνδεδεμένη λίστα αλλάζει φορά, με τους δείκτες να δείχνουν από το Tail προς το Head. Έτσι, αρκεί ένα C-A-S στο Tail ώστε να δείχνει στον καινούργιο κόμβο για να ολοκληρωθεί επιτυχώς ένα enqueue. Βέβαια, για να μπορούμε να έχουμε πρόσβαση στους κόμβου από το Head όταν κάνουμε dequeue, χρειαζόμαστε και δείκτες κατά την αντίθετη φορά, όπως φαίνεται και στο παρακάτω σχήμα.

//figure 2 apo optimistic paper

Οι δείκτες και στις δύο κατευθύνσεις μεταβάλλονται με απλά stores, χωρίς συγχρονισμό. Αυτό εμπεριέχει τον κίνδυνο οι δείκτες κατά την κατεύθυνση prev να μην είναι συνεπείς. Για το λόγο αυτό, μόλις διαπιστωθεί ασυνέπεια, καλείται κατάλληλη συνάρτηση( FixList) που διατρέχει τη λίστα και διορθώνει τους pointers. Το πόσο συχνά όμως συμβαίνουν ασυνεπείς, έχει να κάνει με τις ατομικές καθυστερήσεις που μπορεί να δεχθεί ένα thread και όχι με τη συμφόρηση. Έτσι, ο αριθμός των κλήσεων στην FixList παραμένει χαμηλός, ακόμα και όταν ο αριθμός των παράλληλων threads μεγαλώνει.

Για να αποφύγει το ABA πρόβλημα αλλά και να εντοπίζει ασυνέπειες στους pointers, αυτή η υλοποίηση χρησιμοποιεί επίσης modification counters που ενσωματώνονται στους δείκτες και αυξάνονται κατά ένα σε κάθε επιτυχημένη C-A-S.

H διαδικασία της εισαγωγής κόμβου στη λίστα περιλαμβάνει 3 βήματα:

1. Αλλαγή του next pointer στον προς εισαγωγή κόμβο
2. C-A-S στο Tail ώστε να δείχνει στον νέο κόμβο
3. Αλλαγή του prev pointer του επόμενου κόμβου

Κάθε φορά οι αλλαγές στους pointers γίνονται μαζί με τους αντίστοιχους modification counters.

Κάποιο thread ενδέχεται να δεχθεί καθυστέρηση ανάμεσα στο βήμα 2 και 3, με αποτέλεσμα να μπουν στην ουρά και άλλοι κόμβοι και έτσι ο prev pointer να μην δείχνει όντως τον προηγούμενο κόμβο. Επειδή όμως, κάθε φορά που κάποιος κόμβος εισέρχεται στη λίστα(επιτυχημένο C-A-S) αυξάνεται κατά ένα ο modification counter που γράφεται στον κόμβο, προκύπτει ότι pointers γειτονικών κόμβων στη λίστα θα έχουν διαδοχικούς modification counters. Έτσι κατά το dequeue, αν βρεθεί κόμβος που o prev pointer του δεν έχει σωστό modification counter, καλείται η FixList που διατρέχει όλη τη λίστα και διορθώνει τους prev pointers.

Σημειώνουμε ότι πρέπει να υπάρχει ειδική φροντίδα ώστε να υπάρχει πάντα ένας dummy node στη λίστα και ο Tail να μην τον ξεπερνάει.

Το παρακάτω διάγραμμα συγκρίνει τα C-A-S operations( επιτυχημένα και μη) που χρειάστηκαν για να εκτελεστούν 1 εκατομμύριο ζεύγη enqueue/dequeue για τις δύο αυτές εκδόσεις. Παρατηρούμε ότι η optimistic υλοποίηση πραγματοποιεί αυτό που υπόσχεται , δηλαδή απαιτεί λιγότερα C-A-S και έχει συνολικά λιγότερα χρονοβόρα, αποτυχημένα atomic operations καθώς ο αριθμός των παράλληλων threads μεγαλώνει.

//διαγραμμα για failed CAS