Γιώργος Δημόπουλος Α.Μ. 2964
Άγγελος Κατσαλήρος Α.Μ. 2997

1η ΑΣΚΗΣΗ ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

Οι αλλαγές που έγιναν για τον **server**:

Αρχικά στο πρόγραμμα μας προσθέσαμε τις εξής βιβλιοθήκες:

- <sys/time.h>
- <stdio.h>
- <pthread.h>
- <unistd.h>

Έπειτα βάλαμε με define τις εξής μεταβλητές για να μπορούμε να τις αλλάζουμε:

- QUEUE_SIZE
- NUM_OF_THREADS

Χρησιμοποιήσαμε 4 Κλειδαριές Αμοιβαίου Αποκλεισμού (Mutexes):

1) pthread_mutex_t for_queue = PTHREAD_MUTEX_INITIALIZER, ώστε όταν κάνουμε αλλαγές στην ουρά, αυτές τις αλλαγές να μπορεί να τις κάνει μόνο ένα νήμα για να μπορεί η ουρά να είναι συνεπής.
2) pthread_mutex_t for_var = PTHREAD_MUTEX_INITIALIZER, ώστε όταν θα πρέπει να χρησιμοποιούμε τις κοινόχρηστες μεταβλητές να μπορεί να τις αλλάζει μόνο ένα νήμα.
3) pthread_mutex_t for_ReaderWriter = PTHREAD_MUTEX_INITIALIZER, ώστε όταν θα χρειαστεί να χρησιμοποιήσουμε τις μεταβλητές reader_count και writer_count να μπορεί να το κάνει μόνο ένα νήμα.
4) pthread_mutex_t for_put = PTHREAD_MUTEX_INITIALIZER, επειδή μπορούμε να κάνουμε ένα-ένα put, πρέπει δηλαδή τα put να εκτελούνται ακολουθιακά “κλειδώνω” με ένα ξεχωριστώ mutex την βάση μου όταν θέλει να κάνει put.

Χρησιμοποιήσαμε 4 Μεταβλητές Συνθήκης (Condition variables):

1. pthread_cond_t non_empty_queue = PTHREAD_COND_INITIALIZER, όταν είναι άδεια η ουρά να μπορεί να στέλνει σήματα στα άλλα νήματα.
2. pthread_cond_t non_full_queue = PTHREAD_COND_INITIALIZER, όταν είναι γεμάτη η ουρά να μπορεί να στέλνει σήματα στα άλλα νήματα.
3. pthread_cond_t writer = PTHREAD_COND_INITIALIZER, όταν χρησιμοποιούμε το reader_count να μπορεί να στέλνει σήματα στα άλλα νήματα.
4. pthread_cond_t reader = PTHREAD_COND_INITIALIZER, όταν χρησιμοποιούμε το writer_count να μπορεί να στέλνει σήματα στα άλλα νήματα.

Υλοποίηση ουράς:

Αρχικά βάλαμε τις μεταβλητές tail,head και count.

Η ουρά θα έχει μέσα αντικείμενα τύπου Queue. Το Queue είναι ένα struct το οποίο έχει ένα int πεδίο fd για να ξέρουμε το socket και ένα πεδίο stuct timeval για να μπορούμε να μετρήσουμε τους χρόνους που κάνει μια αίτηση στην ουρά. Το μέγεθος τις ουράς το ορίζουμε με define μεταβλητή (QUEUE_SIZE). Η ουρά μας δέχεται 2 λειτουργείες(put,get). Για την put δημιουργούμε μια συνάρτηση put η οποία ελέγχει για αρχή με τον count αν είναι άδεια η γεμάτη και μετά βάζει το αντικείμενο μέσα στην ουρά. Επειδή η ουρά είναι κυκλική καθώς αυξάνουμε το tail παίρνουμε το module του tail με mod το μέγεθος της ουράς. Για την get ελέγχουμε με τον counter αν η ουρά είναι άδεια, αν δεν είναι παίρνουμε το αντικείμενο που θέλουμε, έπειτα αυξάνουμε το head κατά ένα και μετά όπως και στο tail επειδή η ουρά είναι κυκλική παίρνουμε το module του head με mod το μέγεθος της ουράς.

Υλοποίηση Παραγωγού:

Ο παραγωγός τρέχει μέσα στην main στην while(1). Για κάθε νέα αίτηση που δέχεται ο παραγωγός βάζει μέσα στην ουρά τον περιγραφέα αρχείου καθώς και τον χρόνο που μπήκε εκείνη την στιγμή, που αυτό μπορούμε να το πετύχουμε με την συνάρτηση **gettimeofday**(). Επειδή πρέπει να διατηρούμε σε συνεπή κατάσταση την ουρά όταν πειράζουμε την ουρά θα χρησιμοποιούμε τον mutex: pthread_mutex_t for_queue. Έτσι μόνο ένα νήμα μπορεί να πειράξει την ουρά κάθε φορά. Στον παραγωγό έχουμε μια ειδική περίπτωση που θέλει να βάλει μια αίτηση στην ουρά αλλά η ουρά αυτή την στιγμή είναι γεμάτη. Για αυτό τον λόγο βάζουμε τον παραγωγό να περιμένει μέχρι να του έρθει σήμα ότι άδειασε η ουρά. Αυτό γίνεται με την εντολή **pthread_cond_wait()**, μέσω την μεταβλητής συνθήκης non_full_queue. Όταν ο παραγωγός βάλει κάποιο αντικείμενο στην ουρά στέλνει σήμα μέσω της non_empty_queue στους καταναλωτές για να τους “ξυπνήσει”, αν περιμένουν να βάλουν κάποιο άλλο στοιχείο στην ουρά.

Υλοποίηση Καταναλωτή:

Έχουμε ορίσει τους καταναλωτές να τρέχουν μέσα στην συνάρτηση ***consumers()**. Εμείς θέλουμε ΜΌΝΟ ένας καταναλωτής την φορά να βγάζει μια αίτηση από την ουρά γιαι αυτό τον λόγο θα χρησιμοποιήσουμε το mutex: for_queue. Με αυτόν τον τρόπο ούτε άλλος καταναλωτής ούτε άλλος παραγωγός μπορεί να πειράξει την ουρά και έτσι πετυχαίνουμε αυτό που θέλουμε. Άμα η ουρά είναι άδεια θα δημιουργηθεί κάποιο πρόβλημα οπότε πρέπει να το λύσουμε και αυτό! Στη περίπτωση που η ουρά είναι άδεια οι καταναλωτές μέσω της συνάρτησης **pthread_cond_wait()** περιμένουν σήμα από τον παραγωγό ότι έχει βάλει ένα αίτημα στην ουρά. Όταν οι καταναλωτές παίρνουν ένα αντικείμενο από την ουρά πρέπει να στείλουν σήμα στον παραγωγό ότι άδειασε ένα στοιχείο στην ουρά. Αυτό επιτυχάνεται με την εντολή **pthread_cond_signal()** με την βοήθεια της μεταβλητής συνθήκης non_full_queue. Μόλις πάρει ο καταναλωτής πάρει την αίτηση από την ουρά καλούμε την **process_request()** με όρισμα το συγκεκριμένο περιγραφέα αρχείου για να εξυπηρετηθεί. Επίσης μας ζητείτε να υπολογίσουμε τους χρόνους εξυπηρέτησης,αναμονής και πόσες αιτήσεις εξυπηρετήθηκαν. Για αυτό λοιπόν την στιγμή που *βγάζουμε* μια αίτηση από την ουρά καλούμε την συνάρτηση **gettimeofday()** και αυτό τον χρόνο το αποθηκεύουμε στην anamoni. Μόλις ο καταναλωτής τελειώση την αίτηση (αμέσως μετά την κλήση της **process_request()**) ξανακαλώ την συνάρτηση **gettimeofday()** και αποθηκεύω το χρόνο στο support. Πρέπει τώρα να ενημερώσω τις μεταβλητές completed_request, total_waiting_time, total_service_time. Αυτές οι μεταβλητές είναι global οπότε θα χρησιμοποιήσω το *mutex: for_*var και θα λοκάρω τον κώδικα που ενημερώνω τις μεταβλητές ώστε μόνο ένα νήμα να μπορεί να τις πειράξει. Το αντικείμενο dim3 έχει την

χρονική στιγμή που μπήκε στην ουρά και η anamoni έχει τον χρόνο που βγήκε από την ουρά. Προφανώς λοιπόν ο συνολικός χρόνος αναμονής στην ουρά είναι η διαφορά τους. Ο χρόνος support είχει την χρονική στιγμή που εξυπηρετήθηκε η αίτηση. Προφανώς λοιπόν ο συνολικός χρόνος εξυπηρέτης είναι η διαφορά support-anamoni.

Αλλαγές της συνάρτησης process_request:

Τροποποιούμε αυτή την συνάρτηση ώστε να διατηρηθεί η αποθήκη κλειδιού-τιμής. Θα υλοποιήσουμε το πρόβλημα αναγνωστών-γραφέων για τα PUT και GET.

- ✔ **GET:** Όταν θέλουμε να κάνουμε get οι καναταλωτές πριν την εκτελέσουν θα πρέπει να δουν αν υπάρχουν γραφείς. Στην περίπτωση που δεν υπάρχουν γραφείς τότε το *reader_count* αυξάνεται κατά 1 έτσι ώστε αν έρθει κάποιος γράφεας να μην μπορεί να κάνει κάτι και να περιμένει να τελειώσει ο αναγνώστης. Στην περίπτωση όμως που υπάρχουν γραφείς με την εντολή **pthread_cond_wait()** οι καταναλωτές περιμένουν μέχρι να τελειώσουν οι γραφείς. Τις παραπάνω ενέργειες πρέπει να τις κάνουμε lock επειδή πειράζουμε τις κοινόχρηστες μεταβλητές. Για αυτό τον λόγο χρησιμοποιούμε το *mutex: for_ReaderWriter. Όταν τελειώσει και η ανάγνωση από την βάση χρησιμοποιούμε πάλι το mutex: for_ReaderWriter,* γιατί πρέπει να μειώσουμε το *reader_count* κατά 1 και να στείλουμε μήνυμα στους γραφείς με την συνάρτηση **pthread_cond_signal()** ότι ένας αναγνώστης τελείωσε.
- ✔ **PUT:** Όταν θέλουμε να κάνουμε put οι καταναλωτές πριν την εκτελέσουν θα πρέπει να δουν αν υπάρχουν αναγνώστες. Στην περίπτωση που δεν υπάρχουν αναγνώστες τότε το *writer_count* αυξάνεται κατά 1 έτσι ώστε αν έρθει κάποιος αναγνώστης να μην μπορεί να κάνει κάτι και να περιμένει να τελειώσει ο γραφέας. Στην περίπτωση όμως που υπάρχουν αναγνώστες με την εντολή **pthread_cond_wait()** οι καταναλωτές περιμένουν μέχρι να τελειώσουν οι αναγνώστες. Τις παραπάνω ενέργειες πρέπει να την κάνουμε lock επειδή πειράζουμε κονόχρηστες μεταβλητές. Για αυτό τον λόγο χρησιμοποιούμε το *mutex: for_ReaderWriter.* Στην put έχουμε μια ειδική περίπτωση μπορούμε να έχουμε ΜΟΝΟ ένα γραφέα την φορά. Για αυτό τον κάνουμε lock τον κώδικα:

```
if (KISSDB_put(db, request->key, request->value))
  sprintf(response_str, "PUT ERROR\n");
 else
  sprintf(response_str, "PUT OK\n");
```

με μια άλλη μεταβλητή συνθήκη: *pthread_mutex_t for_put* έτσι ώστε να μπορούμε να εκτελούμε τις put ακολουθιακά. Αμέσως μετά που τελειώνει και η γραφή στην βάση *χρησιμοποιούμε πάλι το mutex: for_ReaderWriter,* γιατί πρέπει να μειώσουμε το *writer_count* κατά 1 και να στείλουμε μήνυμα στους αναγνώστες με την συνάρτηση **pthread_cond_signal()** ότι ένας γραφέας τελείωσε.

Υλοποιήση σήματος cntl-z:

Για την υλοποίηση του σήματος cnlt-z δημιουργήσαμε την συνάρτηση **void sima(int s)** και μέσα σε αυτήν κάναμε τα εξής:

- ➔ Ορίσαμε δύο μεταβλητές όπου αποθηκεύουμε τον μέσο όρο αναμονής και τον μέσο όρο εξυπηρέτης.
- ➔ Κλείσαμε την βάση μας.
- ➔ Τυπώσαμε τις συνολικές εξυπηρετήσεις τον μέσο όρο αναμονής στην ουρά καθώς και τον μέσο όρο εξυπηρέτησης.
- ➔ Κάνουμε detach τα νήματα μας όπως μας ζητήθηκε.

Όταν ληφθεί το σήμα cntl-z που αυτό γίνεται είτε πατώντας cntl-z είτε με την εντολή kill -20 (pid του server) θα εκτελεστούν οι παραπάνω ενέργειες.

*Οι αλλαγές που έγιναν για τον **client**:*

Αρχικά βάλαμε μια global μεταβλητή την *both_requests*. Αυτή την μεταβλητή αν την έχουμε ίση με μηδέν τότε θα κάνει μόνο πολλά put ή get. Αν την αλλάξουμε χειροκίνητα και την θέσουμε ίση με 1 τότε θα κάνει ταυτόχρονα και πολλά put και πολλά get.
Για να το καταφέρουμε αυτό κάναμε αρχικά μια **fork()** πατέρα-παιδί. Στο παιδί θέτουμε το *mode=PUT_MODE* έτσι ώστε το παιδί να κάνει μόνο put. Επιπλέον στο παιδί κάνουμε δύο ακόμη εντολές **fork()** που σπάνε αυτή την διεργασία σε 2^2=4 διεργασίες που στέλνουν ταυτόχρονα αιτήσεις put. Στον πατέρα θέτουμε το *mode=GET_MODE* έτσι ώστε ο πατέρας να κάνει μόνο get. Επιπλέον στον πατέρα κάνουμε δύο ακόμη εντολές **fork()** που σπάνε αυτή την διεργασία σε 2^2=4 διεργασίες που στέλνουν ταυτόχρονα αιτήσεις get. Αυτό το κάνουμε για να πετύχουμε παράλληλη αποστολή αιτήσεων από τον client στον server. Έτσι αν έχουμε την *both_requests=1* τότε θα έχουμε 8 διεργασίες που στέλνουν ταυτόχρονα αιτήσεις στον server.

both_requests	**QUEUE_SIZE**	**NUM_OF_THREADS**	**completed_request**	**average_waiting_time**	**average_service_time**
0	10	5	516	10	78
0	10	10	441	12	70
0	10	25	516	14	84
0	3	6	490	12	76
0	15	6	393	10	78
0	25	6	463	13	78
1	10	5	1032	9	90
1	10	10	868	11	92
1	10	25	899	16	94
1	3	6	1032	8	98
1	15	6	899	11	96

1	25	6	957	9	92

Παρατηρούμε τα εξής:

- Όταν κρατάμε σταθερό το μέγεθος της ουράς αυξάνεται ο χρόνος αναμονής στην ουρά και ο χρόνος εξυπηρέτης και αυτό είναι λογικό γιατί έχουμε περισσότερα νήματα και ο καταναλωτής θέλει να παίρνει περισσότερες αιτήσεις.

- Όταν κρατάμε σταθερό το μέγεθος των νημάτων, όταν έχουμε μικρό μέγεθος ουράς ο χρόνος αναμονής στην ουρά θα είναι μικρότερος που είναι λογικό γιατί έχουμε μικρότερο μέγεθος.

- Όταν κάνουμε πολλά put και get μαζί το completed_request αυξάνεται, όπως φαίνεται και από τον πίνακα που είναι λογικό γιατί έχουμε περισσότερες αιτήσεις.

- Όταν έχουμε ταυτόχρονες put και get ο χρόνος εξυπηρέτης αυξάνεται που είναι λογικό γιατί δεν μπορούν να εξυπηρετηθούν τόσο γρήγορα.