ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ - ΕΡΓΑΣΙΑ 2

Μυλωνόπουλος Δημήτριος - 1115201600112

Στην εργασία αυτή μας ζητήθηκε να υλοποιήσουμε μια προσομοίωση ενός διαχειριστή μνήμης. Στο διαχειριστή μνήμης δύο διεργασίες η PM1 και η PM2 θα στέλνουν αιτήματα αναφορών στον διαχειριστή μνήμης ο οποίος θα πρέπει να τα εξυπηρετήσει.

Για την υλοποίηση των σημαφόρων και της κοινής μνήμης έχουν υλοποιηθεί συναρτήσεις στα αρχεία **file_shared_sem_operations.c/.h** και **sem_operations.c/.h.**

Σχετικά με το **main.c:** Στο αρχείο στη main.c διαβάζουμε αρχικά ορίσματα από την γραμμή εντολών το 1ο όρισμα είναι ο αριθμός k που εκφράζει το μέγιστο πλήθος σελίδων που μπορούμε να έχουμε στο hash table από την κάθε διεργασία, και χρησιμοποιείται για την υλοποίηση του αλγορίθμου FWF που περιγράφεται στην εκφώνηση. Ως 2ο όρισμα δίνουμε τον αριθμό των πλαισίων που θα χρησιμοποιήσουμε στην κύρια μνήμη. Ως 3ο όρισμα έχουμε το q που είναι ο αριθμός των αιτημάτων που θα εξυπηρετούμε συνεχόμενα από κάθε διεργασία και τέλος το 5ο όρισμα είναι ο αριθμός των αναφορών που θα διαβάσουμε από κάθε αρχείο. Αν το όρισμα αυτό είναι κενό διαβάζουμε 1.000.000 αναφορές από το κάθε αρχείο που μας έχει δωθεί που είναι ο συνολικός αριθμός αναφορών.

Ορίζω ως διαμοιραζόμενη μνήμη μεταξύ των διεργασιών PM1, PM2 και MM μια μνήμη 2q θέσεων η οποία αποθηκεύει σε κάθε θέση της ένα **struct Trace.** Στις πρώτες q θέσεις αποθηκεύονται οι αναφορές της PM1 ενώ στις τελευταίες q αποθηκεύονται οι αναφορές της PM2. Για τον συγχρονισμό των διεργασιών εκμεταλεύομαι το πρόβλημα συγχρονισμού Producer-Consumer και γι αυτό ορίζω 2 σέτ 3ων σημαφόρων με τον 1ο να παριστάνει το αν είναι διαθέσιμο το κομματι μνήμης που αντιστοιχεί στην διεργασία που χρησιμοποιεί αυτό το σετ σημαφόρων, ο 2ος αν το κομμάτι της κοινής μνήμης που θέλει να προσπελάσει είναι άδειο και ο 3ος αν είναι γεμάτο. Αρχικοποιώ τις τιμές τους με {1,1,0} αντίστοιχα.

Στη συνέχεια δημιουργώ με **fork()** 3 διεργασίες δηλαδή τις PM1, PM2 και MM αντίστοιχα, μέσω συναρτήσεων οι οποίες θα εξηγηθούν στη συνέχεια.

Η main περιμένει τις 3 διεργασίες να τερματίσουν και έπειτα διαγράφει τους σημαφόρους και την κοινή μνήμη που χρησιμοποιήθηκε.

Για την υλοποίηση του διαχειριστή μνήμης αναγκαία ήταν η εκπόνηση ενός HashTable το οποίο περιέχεται στο HashTable.c. Το hashtable αποτελείται από έναν πίνακα από buckets και στο κάθε στοιχείο του πίνακα κρατάμε την την κεφαλή και την ουρά μιας συνδεδεμένης λίστας. Στο αντίστοιχο .h αρχείο περιέχονται οι δομές του hashtable που είναι συγκεκριμένα:

- Το **struct node** που είναι ο κόμβος που αποθηκεύει ένα στοιχείο του hashtable και αποτελείται απ' το key που είναι το page Number από εναν χαρακτήρα dirty που παίρνει την τιμή 0 ή 1 για να δείξει αν αυτή η σελίδα είναι Dirty και έναν δείκτη στο επόμενο node.
- Το **struct Trace** που αποτελείται από ένα pageNum και εναν χαρακτήρα που δηλώνει αν έχουμε 'R'ead ή 'W'rite.
- Το struct **arrayItem** περιέχει όπως αναφέρθηκε και παραπάνω δείκτες στην κεφαλή και στην ουρά τις συνδεδεμένης λίστας που αντιπροσωπευει.
- Η **struct HashTable** περιέχει έναν δυναμικά δεσμευμένο πίνακα struct arrayItem, τον αριθμό των buckets του HT τον αριθμό των καταχωρήσεων απο PM1, και PM2 (nentries1, nentries2) το πλήθος των αναγνώσεων και εγγραφών από τον δίσκο και τον συνολικό αριθμό των καταχωρήσεων που έγιναν στο HT.

Για τις συναρτήσεις του HashTable έχουμε τα εξής:

- Για το hashing χρησιμοποιήθηκε η universal hash function που οριζεται στην συνάρτηση **Hash.**
- Η **findHash(key,ht,processNum)** η οποία συνάρτηση επιστρέφει έναν δείκτη στο entry που ψάχνουμε αν αυτό υπάρχει. Αλλιώς επιστρέφει NULL
- Η **deleteBucket** διαγράφει όλο το περιεχόμενο του HashTable σε ενα συγκεκριμένο bucket
- Η **deletePages** διαγράφει σε ενα bucket μόνο τα pages τις διεργασίας που δίνεται ως όρισμα. Αν η σελίδες που διαγράφει είναι dirty τότε αυξάνει τον μετρητή των diskWrites, αφού οι σελίδες που έχουν τροποποιηθεί απ τη στιγμή που φορτώθηκαν απ τον δίσκο μόλις φύγουν απ' την μνήμη πρέπει να ενημερώσουν το περιεχόμενο τους στον δίσκο.
- η **createHash** δημιουργεί το hashTable αρχικοποιόντας τις τιμές του και επιστρέφει ένα δείκτη σε αυτό
- η **deleteHashTable** αδειάζει το HashTable και ελευθερώνει τις δυναμικά δεσμευμένες δομές
- η **insertHash** εκχωρεί μια καινούργια σελίδα μιας διεργασίας. Αν η διεργασία δεν υπάρχει στο hashTable τότε αυξάνει τον αριθμό τον DiskReads εφόσον πρέπει να την φορτώσει απ τον δίσκο και την εκχωρεί στο bucket που ορίζει το HashFunction. Αν το trace ορίζει πως στη σελίδα αυτή απαιτείται write τότε την μαρκάρουμε ως dirty. Όμοίως και αν η διεργασία υπάρχει ήδη στο HashTable και δεν είναι dirty.

Στο αρχείο **Processes.c** έχουν υλοποιηθεί οι συναρτήσεις των διεργασίων PM1,PM2, και mm.

Για τις 2 πρώτες έχει οριστεί η συνάρτηση **pm(ntraces, q, pnum)** η οποία παίρνει σαν ορίσματα τον αριθμό των αναφορών που θέλουμε να διαβάσει απ το αρχείο το μεγεθος των κομματιών που θα στέλνει στην διαμοιραζόμενη μνήμη q και τον κωδικό pnum ={0,1} ανάλογα με το αν ειναι το PM1 ή PM2 αντίστοιχα

pm(ntraces,q, pnum): Η συνάρτηση pm ουσιαστικά διαβάζει τον αριθμό των traces που θέλεις να εξυπηρετήσει η mm. Για να το κάνει αυτό διαβάζει απ' το κατάλληλο αρχείο το οποίο εξαρτάται από το αν είμαστε στην PM1 ή PM2 διεργασία. Λατ'αρχας κάνει attatch στο shared mem και φορτώνει τους σημαφόρους που της αντιστοιχούν. Σε κάθε iteration η διεργασία διαβάζει απ το αρχείο μια γραμμή απ την οποία παίρνει τον δεκαεξαδικό που παριστάνει το pageno και το offset αλλά και το action R ή W. Δεδομένου ότι το μέγεθος σελιδας είναι 4096 bytes το offset θα πρέπει να μπορεί να προσπελάσει 4096 bytes= 2^{12} bytes αρα χρειάζεται 12 bits συνεπως 3 hex digits. Γι αυτό και διαιρούμε τον αριθμό με το 4096 ώστε να μην πάρουμε τα τρια τελευταία hex digits. Το ζευγάρι pageNum και action αποθηκεύεται στο shared memory στην κατάλληλη θέση του πίνακα που αντιστοιχεί στη διεργασία. (Από 0 έως q-1 για το PM1 και από q έως 2q-1 για το PM2 αντίστοιχα). Για να συγχρονιστούν σωστά οι διεργασίες χρησιμοποιούμε σημαφόρους με τη λογική του προβλήματος Producer consumer. Που σημαίνει όταν η κάθε διεργασία "γεμίζει" τις θέσεις του πίνακα που της αντιστοιχούν δεν μπορεί η MM να διαβάσει το περιεχόμενα από τις θέσεις αυτές. Για λεπτομέριες περί συγχρονισμού αναφερθείτε στο κομμάτι του κώδικα που αντιστοιχεί στη συνάρτηση της διεργασίας. Μόλις τελειώσει την ανάγνωση κλείνει το αρχείο και η συνάρτηση επιστρέφει.

mm(ntraces,q,k,frames): Η συνάρτηση αυτή παίρνει ουσιαστικά ως όρισμα τον αριθμό των traces που θα διαβάσει από κάθε αρχείο, πόσα θα εξυπηρετεί εναλλάξ από κάθε αρχείο, το μέγεθος k αλλά και τον αριθμό των frames της μνήμης. Εδώ να τονίσουμε πως επειδή η μνήμη είναι χωρισμένη στα δύο, θα πρέπει k<= frames/2 για την κάθε διεργασία. Η συνάρτηση του MM κάνει iterations και στό κάθε iteration εξυπηρετεί q αναφορές απ' την κάθε διεργασία. Πριν κάνει εισαγωγή ένα στοιχείο στο hashtable ελέγχει πως αν οι σελίδες που έχουν εισαχθεί στο hashtable απ την συγκεκριμένη διεργασία είναι k αν είναι τότε ελέγχει αν η καινούργια ανορά που ζητάει υπάρχει ήδη στο hashtable. Αν δεν υπάρχει τότε εκτελούμε FWF στο hashtable για την διεργασία στόχο.

Για να το συγχρονίσουμε τις διεργασίες χρησιμοποιούμε την λογική Consumer-Producer και ουσιαστικά κλειδώνουμε τις q θέσεις που αντιστοιχούν σε κάθε διεργασία μόνο για προσπέλαση απ' το MM ώστε να εξυπηρετήσει τα δωθέντα ίχνη αναφορών. Επιπλέον κάθε φορά που καλούμε την insert ελέγχουμε αν οι εκχωρήσεις του hashtable είναι μεγαλύτερες απ' το MaxFrarmes και αν ειναι το ενημερώνουμε με την παρούσα τιμή.

Τέλος, μόλις εξυπηρετήσει όλες τις αιτήσεις εκτυπώνει τα στατιστικά στοιχεία που ζητήθηκαν για 2 καταστάσεις. Η πρώτη κατάσταση είναι όταν εξυπηρετεί όλες τις αιτήσεις και η δεύτερη είναι όταν έχει τελειώσει την εξυπηρέτηση αλλά έχουν αδειάσει και τα frames για αυτές τις διεργασίες.

Για να τρέξετε το πρόγραμμα: Εκτελέστε την εντολή **make** στο working directory της εργασίας. Και μετά την εντολή **"./Simulation k frames q <optional: max>"** .Για να καθαρίσετε το directory εκτελέστε **make clean**

Σχετικά με τις εκτελέσεις του προγράμματος:

q=10 frames = 1000 ntraces per process= 10000:

k = 500

Before closing the simulation:

Total disk Reads: 1038

Total disk Writes: 436

Total page Faults: 1038

Max frames in use: 719

Total traces examined: 20000

After closing the simulation:

Total disk Reads: 1038

Total disk Writes: 636

Total page Faults: 1038

Max frames in use: 719

Total traces examined: 20000

k=300

Before closing the simulation:

Total disk Reads: 1280

Total disk Writes: 672

Total page Faults: 1280

Max frames in use: 583

Total traces examined: 20000

After closing the simulation:

Total disk Reads: 1280

Total disk Writes: 836

Total page Faults: 1280

Max frames in use: 583

Total traces examined: 20000

k = 50:

Before closing the simulation:

Total disk Reads: 2296

Total disk Writes: 1117

Total page Faults: 2296

Max frames in use: 100

Total traces examined: 20000

After closing the simulation:

Total disk Reads: 2296

Total disk Writes: 1133

Total page Faults: 2296

Max frames in use: 100

Total traces examined: 20000

Παρατηρούμε ότι όταν το k πλησιάζει το frames 2 τόσο ο αριθμός των DiskReads όσο και ο αριθμός των DiskWrites μειώνεται δραματικά. Αυτό ειναί λογικό εφόσον μας δίνει τη δυνατότητα να κρατάμε περισσότερες σελίδες στη μνήμη με αποτέλεσμα να χρειάζονται λιγότερες αναγνώσεις και εγγραφές από τον δίσκο.