**ΕΘΝΙΚΟ ΜΕΤΣΟΒΕΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

**ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ**

**ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΤΩΝ**

**ΑΝΑΦΟΡΑ 2ης ΕΡΓΑΣΤΗΡΙΑΚΗΣ ΑΣΚΗΣΗΣ**

A black background with a black square

Description automatically generated with medium confidence

**Στοιχεία Ομάδας**

* Αναγνωριστικό: oslab005
* Μέλος 1ο: Πέππας Μιχαήλ – Αθανάσιος, Α.Μ: 03121026
* Μέλος 2ο: Σαουνάτσος Ανδρέας, Α.Μ: 03121197
* Ημερομηνία Παράδοσης Αναφοράς: 15.05.2024
* **Ενότητα 1 – Συγχρονισμός σε υπάρχοντα κώδικα**

***Εισαγωγικές ερωτήσεις***

Χρησιμοποιούμε το διαθέσιμο Makefile, προκειμένου να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα simplesync. Παρατηρούμε, ότι ενώ έχουμε αρχικοποιήσει μία μεταβλητή (int val) στην τιμή 0 και την αυξάνουμε κατά 1 όσες φορές τη μειώνουμε κατά 1 – 10.000.000 φορές – το τελικό αποτέλεσμα που εμφανίζεται στην οθόνη δεν είναι 0, όπως αναμενόταν. Η εξήγηση για αυτό είναι ότι οι διεργασίες που αυξάνουν και μειώνουν την τιμή της μεταβλητής εκτελούνται ταυτόχρονα – και μάλιστα το κρίσιμο τμήμα τους που κάνει (val++) ή (val--) – δεν είναι συγχρονισμένες, με αποτέλεσμα να μοιράζονται κοινή μνήμη και να αντλούν / αποθηκεύουν την νέα τιμή της σε λάθος στιγμές: πριν ολοκληρωθεί και αποθηκευτεί η αύξηση έχει εκτελεστεί αφαίρεση ή αντιστρόφως.

Ακόμα, παρατηρούμε ότι παράγονται δύο διαφορετικά εκτελέσιμα από το simplesync.c : το simplesync-atomic και το simplesync-mutex, μέσω του Makefile. Ο λόγος που συμβαίνει αυτό είναι αφενός μεν τα #defined που έχουμε κάνει μέσα στο simplesync.c και φαίνονται παρακάτω:

A screenshot of a computer program

Description automatically generated

αφετέρου δε, τα flags κατά τη μεταγλώττιση στο Makefile (-DSYNC\_MUTEX και -DSYNC\_ATOMIC), τα οποία δίνουν τιμές 1 στις mutex και atomic παραλλαγές της simplesync, αντίστοιχα.

Κατόπιν, επεκτείνουμε τον κώδικα της simplesync.c στις δύο παραλλαγές της και το νέο αρχείο .c φαίνεται ακολούθως. Οι αλλαγές εντοπίζονται στην \_\_sync\_add\_and\_fetch και \_\_sync\_sub\_and\_fetch που εξασφαλίζουν το atomicity των προσθέσεων και αφαιρέσεων, αντίστοιχα, και στα mutex lock/unlocks στο κρίσιμο τμήμα της αύξησης/μείωσης της μεταβλητής val.

Η αλλαγμένη συνάρτηση που υλοποιεί την αύξηση φαίνεται ακολούθως, ενώ όμοια λογική υπάρχει και στη συνάρτηση που υλοποιεί τη μείωση της μεταβλητής val.

A screen shot of a computer program

Description automatically generated

Ο συνολικός – αλλαγμένος κώδικας φαίνεται ακολούθως:

Το αποτέλεσμα, όπως φαίνεται και παρακάτω, είναι 0, όπως πρέπει:

A computer screen with text

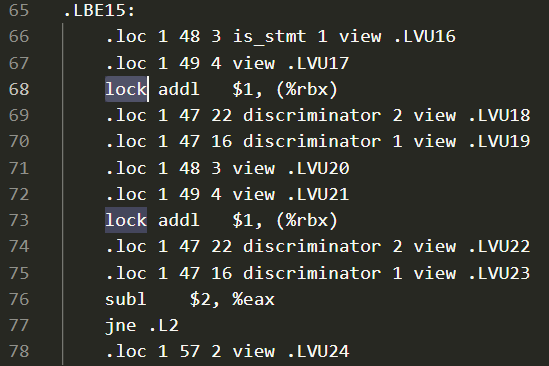
Description automatically generated

Αξίζει να σημειωθεί πως το πρόγραμμά μας, όπως φαίνεται και από τον κώδικα που παραθέσαμε, ελέγχει την ορθότητα των εισαγόμενων δεδομένων κατά τον χρόνο εκτέλεσης και σε περίπτωση λάθους, τυπώνει το αντίστοιχο μήνυμα στον χρήστη, καθοδηγώντας τον προς την εισαγωγή μιας ορθής εισόδου.

***Ερωτήσεις***

1. A black screen with white text

   Description automatically generatedA screen shot of a computer

   Description automatically generatedΑρχικά, τρέχουμε με χρήση της εντολής time τη simplesync-atomic, η οποία έχει παραχθεί από τον αρχικό κώδικα, χωρίς παραλλαγές, δηλαδή χωρίς συγχρονισμό. Το αποτέλεσμα φαίνεται ακολούθως:  
   Όπως εύκολα γίνεται αντιληπτό, το αποτέλεσμα είναι λανθασμένο, αφού δεν υπάρχει συγχρονισμός. Ακολούθως, τρέχουμε τους σωστούς κώδικες simplesync-atomic και simplesync-mutex, αντίστοιχα, και το αποτέλεσμα φαίνεται ακολούθως:  
   Παρατηρούμε, όπως εξάλλου ήταν και αναμενόμενο, ότι ο χρόνος εκτέλεσης του ασυγχρόνιστου προγράμματος είναι πολύ μικρότερος και από τα δύο προγράμματα που εκτελούν συγχρονισμό και εμφανίζουν – φυσικά – το σωστό αποτέλεσμα. Ο λόγος για αυτό είναι ότι στο αρχικό πρόγραμμα εκτελούνται όλες οι προσθέσεις και οι αφαιρέσεις ταυτόχρονα, χωρίς έτσι να παράγεται σωστό αποτέλεσμα, αφού χάνεται η σωστή αλληλουχία τους και χρησιμοποιούν κοινή μνήμη, ενώ στις άλλες περιπτώσεις εκτελούνται μόνο όταν το κρίσιμο τμήμα της προηγούμενης έχει ολοκληρωθεί, υστερώντας έτσι σε χρόνο, εμφανίζοντας όμως το σωστό αποτέλεσμα.
2. Όπως φαίνεται και από την παραπάνω εικόνα, η μέθοδος των atomic operations είναι κατά πολύ (περίπου 5 φορές) γρηγορότερη από τη μέθοδο των mutexes. Ο λόγος πίσω από αυτό είναι ότι στην περίπτωση των atomic operations οι προσθέσεις και οι αφαιρέσεις μπορούν να τρέχουν παράλληλα – ατομικά – και δεν επικαλύπτεται μόνο το κρίσιμο (σε assembly) κομμάτι τους ενώ το μη κρίσιμο εκτελείται ταυτοχρόνως, ενώ στην περίπτωση των mutexes, έχουμε κλειδώσει όλο το κομμάτι των προσθέσεων και των αφαιρέσεων, με αποτέλεσμα αυτές να εκτελούνται σειριακά, μία κάθε φορά.
3. A screenshot of a computer

   Description automatically generatedA screen shot of a computer

   Description automatically generatedΑλλάζουμε το Makefile, προσθέτοντας τα flags -S και -g, ώστε να παραγάγουμε τον αναγνώσιμο κώδικα των αρχείων μας σε assembly, ως εξής:  
   Κατόπιν, ανοίγουμε το simplesync-atomic.o και εντοπίζουμε ένα σημείο στο οποίο εκτελούνται τα atomic operations, το οποίο φαίνεται ακολούθως:
4. A screenshot of a computer

   Description automatically generatedΟμοίως, ανοίγουμε το simplesync-mutex.o και εντοπίζουμε ένα σημείο στο οποίο εκτελούνται τα pthread\_mutex\_lock / pthread\_mutex\_unlock, το οποίο φαίνεται ακολούθως:

* **Ενότητα 2 – Παράλληλος υπολογισμός του συνόλου Mandelbrot**

***Εισαγωγή***

Σημειώνουμε πως αμφότερα τα προγράμματά μας (mandel\_sema.c και mandel\_cond.c) ελέγχουν την ορθότητα των εισαγόμενων δεδομένων κατά τον χρόνο εκτέλεσης και σε περίπτωση λάθους, τυπώνουν το αντίστοιχο μήνυμα στον χρήστη, καθοδηγώντας τον προς την εισαγωγή μιας ορθής εισόδου.

1. A screen shot of a computer

   Description automatically generatedΑρχικά, συντάσσουμε τον ακόλουθο κώδικα, ο οποίος φέρει το όνομα mandel\_sema.c και υλοποιεί το ζητούμενο με τη χρήση σημαφόρων. Το struct του καθενός thread φαίνεται ακολούθως και περιέχει το id του, τον συνολικό αριθμό των νημάτων και έναν πίνακα στον οποίο αποθηκεύονται τα δεδομένα του compute του mandel\_line και αργότερα απεικονίζονται. Το ίδιο struct χρησιμοποιείται και για το mandel\_cond.c στο επόμενο ερώτημα και για τον λόγο αυτό δεν θα παρατεθεί ξανά.  
   Η λογική πίσω από τον κώδικά μας είναι η εξής: εάν έχουμε n threads να τρέχουν το mandel, καθένα από αυτά έχει τον δικό του σημαφόρο (έχουμε δηλαδή έναν πίνακα από σημαφόρους, τον sem) αρχικοποιημένο σε 0, εκτός από το 1ο thread, του οποίου είναι αρχικοποιημένος σε 1. Έτσι, αφού το 1ο thread εκτελέσει το κρίσιμο τμήμα του, αυτό θα ξυπνήσει το 2ο, το 2ο το 3ο κ.ο.κ., μέσω της sem\_post του αντίστοιχου σημαφόρου. Επίσης, έχουμε χειριστεί το πρόγραμμά μας έτσι ώστε το κρίσιμο τμήμα να περιλαμβάνει μόνο τη φάση του output και όχι και αυτή του compute, για λόγους επίδοσης που θα σχολιαστούν αργότερα, στα αντίστοιχα ζητούμενα ερωτήματα, και άρα τα computes γίνονται ταυτόχρονα και αποθηκεύονται στον ξεχωριστό πίνακα του κάθε thread. Ο κώδικας αυτός φαίνεται ακολούθως:
2. Για το ερώτημα αυτό, συντάσσουμε τον ακόλουθο κώδικα, ο οποίος φέρει το όνομα mandel\_cond.c και υλοποιεί το ζητούμενο με τη χρήση condition variables. Επισημαίνουμε πως το κάθε thread χρησιμοποιεί το struct που παρατέθηκε προηγουμένως. Η λογική αυτού του προγράμματος είναι η εξής: έχουμε n threads, τα οποία πάλι (για λόγους επίδοσης) τρέχουν παράλληλα το compute κάθε γραμμής που έχουν αναλάβει και έχουν ως κρίσιμο τμήμα μόνο αυτό της απεικόνισης μιας γραμμής, πριν το οποίο περιμένουν σε μια κοινή κελιδαριά. Έτσι, όντας συνδεδεμένα σε μια κοινή κλειδαριά (lock), έχουμε έναν πίνακα από conditions (cond), ένα για κάθε νήμα. Με τη βοήθεια ενός μετρητή, αρχικά τρέχει μόνο το υπ’ αριθμόν 0 νήμα και ακολούθως, αφού εκτελέσει το κρίσιμο τμήμα του, κάνει broadcast στο επόμενο (στο αντίστοιχο condition) για να ξεκινήσει. Έτσι, το mandel απεικονίζεται επιτυχώς στην έξοδο, με τη βέλτιστη επίδοση.

Αξίζει να επισημάνουμε ότι τα threads μοιράζονται τις global μεταβλητές τους. Για τον λόγο αυτό, προτιμήσαμε κάθε thread να κάνει malloc έναν δικό του πίνακα, ώστε να μπορούν όλα μαζί – παράλληλα – να υπολογίζουν την έξοδο μίας γραμμής, περιορίζοντας έτσι το κρίσιμο τμήμα των threads μόνο στη διαδικασία του output, επιτυγχάνοντας βέλτιστη επίδοση. Αυτό, γίνεται φανερό στον τρόπο που αρχικοποιούμε το struct κάθε thread και στο σώμα της συνάρτησης που αυτό καλεί.

Ο κώδικας του mandel\_cond.c φαίνεται ακολούθως:

***Ερωτήσεις***

1. Για το σύστημα συγχρονισμού που υλοποιήσαμε, χρειαστήκαμε αριθμό σημαφόρων ίσο με τον αριθμό των threads που τρέχουν συγχρόνως και που εισάγονται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματός μας. Ειδικότερα, χρησιμοποιήσαμε έναν πίνακα από σημαφόρους (τον sem), ο οποίος είναι αρχικοποιημένος στο 0, εκτός από την 1η θέση του, που είναι αρχικοποιημένη στο 1. Έτσι, θα εκτελεστεί μόνο το 1ο thread (αφού τα άλλα θα περιμένουν) και τα επόμενα θα καλούνται διαδοχικά, καθώς το προηγούμενο θα κάνει sem\_post το επόμενο. Αυτό γίνεται φανερό στον κώδικα mandel\_sema.c που παραθέσαμε παραπάνω.
2. Αρχικά, χρησιμοποιούμε την εντολή cat /proc/cpuinfo και διαπιστώνουμε ότι το μηχάνημά μας διαθέτει 4 πυρήνες (σε αντίθεση με τον orion που διαπιστώσαμε ότι έχει μόνο 1), καθιστώντας τη δοκιμή πολλών threads εφικτή. Αυτό, φαίνεται ακολούθως:  
   ***A screenshot of a computer program

   Description automatically generated***

A screen shot of a computer

Description automatically generatedΕπομένως, τρέχουμε το mandel πρώτα και το mandel\_sema μετά, με 2 threads και με κατάλληλη χρήση της time, και το αποτέλεσμα φαίνεται ακολούθως:  
A screen shot of a computer

Description automatically generated

Όπως ήταν αναμενόμενο, η εκτέλεση του mandel διήρκησε πολύ περισσότερο (σχεδόν τον διπλάσιο πραγματικό χρόνο) από αυτή του mandel\_sema με 2 threads, αφού η διαδικασία των computes γίνεται πλέον παράλληλα από τα 2 threads και έχουμε απομονώσει μόνο το πραγματικό κρίσιμο τμήμα τους. Παρατηρούμε, έτσι, μια μεγάλη βελτίωση στην επίδοση του προγράμματός μας.

1. Όπως φαίνεται και στον κώδικα mandel\_cond.c που παραθέσαμε παραπάνω, έχουμε χρησιμοποιήσει αριθμό condition variables ίσο με τον αριθμό των threads που χρησιμοποιούμε και που εισάγονται από τον χρήστη κατά τον χρόνο εκτέλεσης του προγράμματός μας. Ο λόγος για αυτό είναι ο εξής: τα threads είναι συνδεδεμένα σε μια κοινή κλειδαριά (lock) και έχουμε έναν πίνακα από conditions (cond), ένα για κάθε νήμα. Με τη βοήθεια ενός μετρητή (counter), αρχικά τρέχει μόνο το υπ’ αριθμόν 0 νήμα και ακολούθως, αφού εκτελέσει το κρίσιμο τμήμα του, κάνει broadcast στο επόμενο (στο αντίστοιχο δηλαδή condition) για να ξεκινήσει. Έτσι, τα computes γίνονται παράλληλα και τα outputs με τη σειρά που πρέπει, επιτυγχάνοντας βέλτιστη επίδοση.  
     
   Έστω, τώρα, ότι χρησιμοποιούμε μόνο ένα condition variable, έστω cond και ότι – χωρίς βλάβη της γενικότητας – έχουμε 2 threads. Στην περίπτωση αυτή, θα δημιουργηθεί το εξής πρόβλημα συγχρονισμού: αρχικά, θα τρέξει το 1ο thread τη διαδικασία του compute του, όπως και το 2ο, παράλληλα, ενώ στην συνέχεια, το 1ο θα τρέξει και το κρίσιμο τμήμα του (output), ενώ το 2ο θα κάνει wait, περιμένοντας το 1ο να τελειώσει και να κάνει broadcast στο cond, ώστε να ξεκινήσει κι αυτό. Όμως, μπορεί το 1ο thread να τελειώσει το κρίσιμο τμήμα του και να κάνει broadcast πριν το 2ο κάνει wait και αυτό θα έχει ως αποτέλεσμα το 2ο να κάνει wait για πάντα, αφού το 1ο δεν θα ξανακάνει broadcast ποτέ. Άρα, το 2ο thread δεν θα κάνει ποτέ output και το πρόγραμμά μας θα κολλήσει στο σημείο αυτό. Αντίστοιχο πρόβλημα θα παρατηρηθεί και εάν έχουμε παραπάνω από 2 threads σε κοινό cond.  
     
   Το πρόβλημα αυτό, αποτυπώνεται κάτωθι, όπου παραθέτουμε τον αλλαγμένο κώδικα στο σημείο της κλήσης της συνάρτησης από το κάθε thread και το αντίστοιχο output, το οποίο παραμένει παγωμένο για πάντα:  
   ***A computer screen shot of a program code

   Description automatically generated***

***A screen shot of a computer

Description automatically generated***

1. Για τους λόγους που ήδη έχουμε αναφέρει, αμφότερα τα προγράμματα που υλοποιήσαμε για το mandel με πολλαπλά threads, εμφανίζουν σημαντική επιτάχυνση, η οποία (έπειτα από δοκιμές) είναι ολοένα και μεγαλύτερη αυξανομένου του αριθμού των νημάτων, αν και όχι με τον ίδιο ρυθμό, δηλαδή από ένα σημείο threads και πάνω (που φυσικά εξαρτάται από τον αριθμό των πυρήνων του μηχανήματος) η μείωση στον χρόνο είναι μικρή, έως και αμελητέα. Ο λόγος πίσω από αυτό είναι ο εξής: στην περίπτωση που ορίσουμε ως κρίσιμο τμήμα σε κάθε thread τις φάσεις τόσο του υπολογισμού όσο και της εμφάνισης μια γραμμής, αυτό θα έχει ως αποτέλεσμα να τρέχει μόνο ένα νήμα κάθε φορά και η όλη διαδικασία να εκτελείται σειριακά, όπως στο αρχικό mandel! Ένα όμως φροντίσουμε να έχουμε ξεχωριστές δομές δεδομένων για κάθε νήμα (όπως και κάναμε), τότε η φάση του compute – που όπως τελικά διαπιστώνουμε είναι αρκετά χρονοβόρα – δύναται να βγει εκτός του κρίσιμου τμήματος και σε αυτό να παραμείνει μόνο η φάση της αποτύπωσης. Έτσι, ο χρόνος εκτέλεσης μειώνεται αισθητά και γίνεται πραγματική αξιοποίηση του μεγάλου αριθμού νημάτων.  
   Για να γίνει αυτό φανερό, αλλάζουμε το κρίσιμο τμήμα (έστω του mandel\_sema.c), ώστε αυτό να περιλαμβάνει αμφότερες τις φάσεις που ***A screen shot of a computer code

   Description automatically generated***περιγράψαμε, ως εξής:Ο χρόνος εκτέλεσης (έστω για 4 νήματα) φαίνεται ακολούθως και παρατηρούμε ότι είναι ο ίδιος με αυτόν του mandel:  
   ***A black background with white text

   Description automatically generated***
2. Παρατηρούμε ότι στην περίπτωση που πατήσουμε Ctrl+C πριν το πρόγραμμά μας τερματίσει, το χρώμα των γραμμάτων του τερματικού αλλάζει: από άσπρο γίνεται το χρώμα που είχε ο χαρακτήρας του mandel που ζωγραφιζόταν όταν το πρόγραμμα σταμάτησε. Αυτό, φαίνεται ακολούθως:

A screen shot of a computer screen

Description automatically generated

Στο σημείο αυτό, επισημαίνουμε ότι τα δύο προγράμματα που παραθέσαμε στην αρχή έχουν την τελική τους μορφή και για τον λόγο αυτό περιέχουν από την αρχή τον κατάλληλο χειρισμό του σήματος Ctrl+C που θα εξηγήσουμε τώρα.  
  
Προκειμένου, επομένως, το χρώμα του τερματικού να επαναφέρεται στο λευκό (default) πριν το πρόγραμμά μας τερματίσει, προσθέτουμε κατάλληλο χειρισμό του σήματος Ctrl+C, όπως φαίνεται ακολούθως:

A screen shot of a computer code

Description automatically generated

***A screen shot of a computer program

Description automatically generated***

**Σ.Η.Μ.Μ.Υ. Ε.Μ.Π.  
Μάϊος 2024**