3η Άσκηση στα Λειτουργικά Συστήματα

6ο Εξάμηνο, Ακαδημαϊκή περίοδος 2020 – 2021

Ονοματεπώνυμο	Αριθμός Μητρώου
Στεφανάκης Γεώργιος	el18436
Τζανακάκης Αλέξανδρος	el18431

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Σε αυτή την άσκηση κληθήκαμε να τροποποιήσουμε το αρχείο πηγαίου κώδικα `simplesync.c` ώστε τα POSIX νήματα που αναλαμβάνουν την αύξηση και τη μείωση μίας μεταβλητής κατά 10000000 φορές να συγχρονιστούν και στο τέλος του προγράμματος η μεταβλητή αυτή να έχει την τιμή μηδέν. Η μεταγλώττιση του source αρχείου δίνει δύο εκτελέσιμα, ένα για κάθε τύπο συγχρονισμού. Ο πρώτος τύπος συγχρονισμού γίνεται με χρήση των POSIX Mutexes ενώ η δεύτερη μέσω των GCC Atomic Operators. Μέσω της σημαίας `USE_ATOMIC_OPS`, ελέγχεται ο τύπος συγχρονισμού μέσα στον κώδικα και παράγεται η ανάλογη υλοποίηση. Σε πρώτη φάση τρέχοντας τα εκτελέσιμα αυτά παρατηρούμε ότι το αποτέλεσμα είναι λανθασμένο γιατί είναι διάφορο του μηδενός και προφανώς δεν παρουσιάζεται σημαντική διαφορά στο χρόνο εκτέλεσής τους. Στο αρχείο πηγαίου κώδικα που έχουμε επισυνάψει μαζί με την αναφορά έχουμε προσθέσει τις κατάλληλες εντολές ώστε ανάλογα με το εκτελέσιμο που τρέχουμε να χρησιμοποιείται ο κατάλληλος συγχρονισμός.

1)

```
$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 761149.

real    0m0,032s
user    0m0,062s
sys     0m0,000s

$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -1405590.

real    0m0,034s
user    0m0,066s
sys     0m0,000s
```

```
$ time ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m1,178s
user    0m1,632s
sys     0m0,712s

$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m0,457s
user    0m0,902s
sys     0m0,010s

$
```

Όπως αναμέναμε, όταν δεν υπάρχει συγχρονισμός, λόγω των παράλληλων εκτελέσεων των νημάτων, η εκτέλεση του προγράμματος είναι γρηγορότερη από την εκτέλεση με συγχρονισμό, όπου το ένα νήμα περιμένει το άλλο.

2) Προφανώς χωρίς συγχρονισμό δεν παρατηρείται σημαντική διαφορά στην χρονική διάρκεια των δύο εκτελέσιμων. Όμως μετά την τροποποίηση παρατηρούμε ότι οι ατομικές λειτουργίες του GCC είναι πολύ ταχύτερες από τα POSIX mutexes καθώς είναι πιο χαμηλού επιπέδου λύση (πρακτικά ένα atomic operator αντιστοιχεί σε μία μόνο εντολή assembly) σε σχέση με το API των mutexes που περιέχει σύνθετες κλήσεις βιβλιοθήκης και είναι συνεπώς πιο αργές. Επιπλέον, γνωρίζουμε ότι τα mutexes χρησιμοποιούν atomic operators για την υλοποίησή τους άρα είναι λογικό να καθυστερούν περισσότερο.

3)

```
.loc 1 52 4 view .LVU11
lock addl  $1, (%rbx)
```

```
.loc 1 75 4 view .LVU31
lock subl  $1, (%rbx)
```

Παραπάνω φαίνεται το μέρος του assembly κώδικα που παράχθηκε από το GCC και που αφορά την αύξηση ή τη μείωση της μεταβλητής `val` με χρήση των ατομικών λειτουργιών GCC. Για τη δημιουργία των αρχείων κώδικα assembly χρησιμοποιήσαμε τις παραμέτρους `gcc -S -g` τις οποίες προσθέσαμε στο `Makefile`.

4)

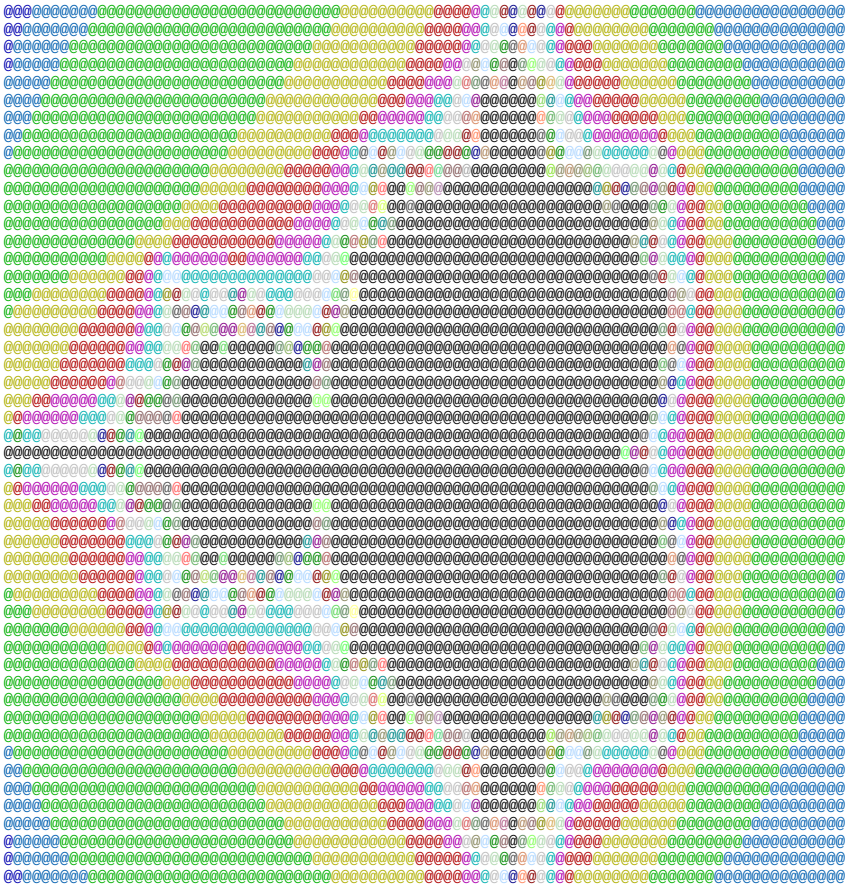
```
  .loc 1 54 4 view .LVU13
       movq  %r12, %rdi
       call  pthread_mutex_lock@PLT
  .LVL4:
       .loc 1 56 4 view .LVU14
       .loc 1 56 7 is_stmt 0 view .LVU15
       movl  0(%rbp), %eax
       .loc 1 57 4 view .LVU16
       movq  %r12, %rdi
       .loc 1 56 4 view .LVU17
       addl  $1, %eax
       movl  %eax, 0(%rbp)
       .loc 1 57 4 is_stmt 1 view .LVU18
       call  pthread_mutex_unlock@PLT
```

```
       .loc 1 77 4 view .LVU41
       movq  %r12, %rdi
       call  pthread_mutex_lock@PLT
  .LVL14:
       .loc 1 79 4 view .LVU42
       .loc 1 79 7 is_stmt 0 view .LVU43
       movl  0(%rbp), %eax
       .loc 1 80 4 view .LVU44
       movq  %r12, %rdi
       .loc 1 79 4 view .LVU45
       subl  $1, %eax
       movl  %eax, 0(%rbp)
       .loc 1 80 4 is_stmt 1 view .LVU46
       call  pthread_mutex_unlock@PLT
```

Παραπάνω φαίνεται το μέρος του assembly κώδικα που παράχθηκε από το GCC και που αφορά την αύξηση ή τη μείωση της μεταβλητής `val` με χρήση mutex. Παρατηρούμε ότι όπως αναφέρθηκε στο προηγούμενο ερώτημα, στο atomic εκτελέσιμο, ο κώδικας περιέχει μόνο μία εντολή σε αντίθεση με το mutex εκτελέσιμο στο οποίο η διαδικασία αυτή είναι πιο σύνθετη και απαιτεί περισσότερες από μία εντολές.

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Ο παράλληλος υπολογισμός του συνόλου Mandelbrot επιτυγχάνεται με επικοινωνία μεταξύ των νημάτων με χρήση σημαφόρων του προτύπου POSIX.



1) Απαιτούνται συνολικά NTHREADS σημαφόροι, δηλαδή όσοι και το πλήθος των νημάτων που ορίζει ο χρήστης. Το i – οστό νήμα θα καλεί την post για τον σημαφόρο που είναι υπεύθυνος για το i+1 – οστό νήμα που περιμένει. Οπότε πρέπει να ορίσουμε έναν πίνακα κυκλικής λογικής από NTHREADS σημαφόρους ο καθένας από τους οποίους είναι αρχικοποιημένος στο 0 εκτός από τον πρώτο που είναι αρχικοποιημένος στο 1, για τον λόγο που εξηγήθηκε παραπάνω.

2)

```
$ time ./mandel 1

real   0m0,514s
user   0m0,492s
sys    0m0,021s


$ time ./mandel 2
real   0m0,267s
user   0m0,513s
sys    0m0,011s

$ time ./mandel 4
real   0m0,168s
user   0m0,547s
sys    0m0,010s


$ time ./mandel 8

real   0m0,184s
user   0m0,551s
sys    0m0,020s

$
```

Παραπάνω βλέπουμε τους χρόνους εκτέλεσης του προγράμματος για αριθμό νημάτων 1, 2, 4 και 8 αντίστοιχα. Όπως περιμέναμε, ο χρόνος εκτέλεσης του παράλληλου προγράμματος με 2 νήματα είναι μικρότερος από εκείνον του μονονηματικού προγράμματος. Παρ' όλα αυτά δεν παρατηρούμε σημαντική διαφορά ανάμεσα στα 4 και στα 8 νήματα πράγμα που οφείλεται στο γεγονός ότι ο υπολογιστής στον οποίο τρέξαμε το πρόγραμμα έχει 4 υπολογιστικούς πυρήνες δηλαδή μπορεί να υποστηρίξει μέχρι 4 παράλληλους υπολογισμούς. Προφανώς μπορούμε να ορίσουμε όσα POSIX threads επιθυμούμε όμως δεν μπορούμε να έχουμε ανάλογη κλιμάκωση στο χρονικό κέρδος λόγω των φυσικών περιορισμών του μηχανήματος καθώς μόλις ξεπεράσουμε τον αριθμό των πραγματικών πυρήνων ο χρονοδρομολογητής του λειτουργικού συστήματος εναλλάσσει συνεχώς τα ενεργά threads χωρίς ουσιαστικό κέρδος.

3) Το πρόγραμμα έχει τροποποιηθεί ώστε να γίνεται παράλληλος υπολογισμός του χρώματος των χαρακτήρων κάθε γραμμής από τα νήματα και για αυτό το λόγο είναι γρηγορότερο από το αρχικό. Παρόλο που η εκτύπωση κάθε γραμμής θέλουμε να συνεχίσει να γίνεται με σειριακό τρόπο, ο υπολογισμός μίας γραμμής μέσω της συνάρτησης `compute_mandel_line` εκτελείται παράλληλα για όλα τα νήματα εφόσον δεν αποτελεί μέρος του κρίσιμου τμήματος κώδικα. Δηλαδή, ο συγχρονισμός των νημάτων με χρήση σημαφόρων περιορίζεται στη φάση της εκτύπωσης.

4) Όταν ο χρήστης κατά τη διάρκεια της εκτέλεσης του προγράμματος πατήσει Ctrl+C, στέλνει στο πρόγραμμα σήμα SIGINT. Επιπλέον, το χρώμα του τερματικού παραμένει το ίδιο με το χρώμα του τελευταία τυπωμένου χαρακτήρα κατά τη στιγμή της διακοπής. Για αυτό το λόγο κατασκευάσαμε έναν handler στον οποίο μεταφέρεται η ροή του προγράμματος εάν ο χρήστης στείλει αυτό το σήμα διακοπής. Ο handler `handleCtrlC` επαναφέρει το default χρώμα των χαρακτήρων του τερματικού, τυπώνει κατάλληλο μήνυμα που περιλαμβάνει τον κωδικό του σήματος διακοπής και τερματίζει το πρόγραμμα.

Πηγαίος Κώδικας

Μαζί με την παρούσα αναφορά επισυνάπτουμε και τον φάκελο `sync` στον οποίο συμπεριλαμβάνονται όλα τα αρχεία που μας δόθηκαν από την εκφώνηση της άσκησης, μεταρρυθμισμένα ώστε να λειτουργούν με τον επιθυμητό τρόπο. Επιπλέον έχουμε τροποποιήσει το αρχείο `Makefile` ώστε να παράγει και τα αρχεία κώδικα assembly που ζητούνται στο πρώτο ερώτημα.