Λειτουργικά Συστήματα Υπολογιστών

$2^{η}$ Εργαστηριακή Άσκηση

Συγχρονισμός

Μπίλιας Ευστάθιος el20800

Γκούντης Βασίλης el20636

13 Ιουνίου 2024

ΣΗΜΜΥ
Εθνικό Μετσόβιο Πολυτεχνείο

Περιεχόμενα

1 Συγχρονισμός σε υπάρχοντα κώδικα

Παρατηρούμε ότι όταν τρέχουμε την εντολή make για να μεταγλωττίσουμε το πρόγραμμα symplesync.c, δημιουργούνται 2 εκτελέσιμα. Βλέποντας το περιεχόμενο του Makefile, συνειδητοποιούμε ότι δημιουργούνται 2 object files από το κοινό symplesync.c αρχείο. Αυτό συμβαίνει διότι χρησιμοποιούμε τις παραμέτρους DSYNC_ATOMIC και DSYNC_MUTEX. Αυτές οι παράμετροι μεταφράζονται ως define SYNC_ATOMIC και define SYNC_MUTEX αντίστοιχα. Αξίζει να αναφερθεί ότι, ανάλογα με το ποιο από τα 2 εκτελέσιμα θα επιλέξουμε να τρέξουμε, το πρόγραμμα θα εκτελεστεί διαφορετικά. Στην πρώτη περίπτωση θα κάνει χρήση atomic operations, ενώ στη δεύτερη θα κάνει χρήση των mutexes. Ο έλεγχος για το ποια μέθοδος θα χρησιμοποιηθεί ελέγχεται μέσω if statements στα προαναφερόμενα define.

```
void *increase_fn(void *arg) {
  int i;
  volatile int *ip = arg;
  pthread_mutex_init(&mutex, NULL);

  fprintf(stderr, "About to increase variable %d times\n", N);
  for (i = 0; i < N; i++) {
    if (USE_ATOMIC_OPS) {
        __sync_add_and_fetch(ip, 1);
    } else {
      pthread_mutex_lock(&mutex);
      ++(*ip);
      pthread_mutex_unlock(&mutex);
    }
  }
  fprintf(stderr, "Done increasing variable.\n");
  return NULL;
}

void *decrease_fn(void *arg) {
  int i;
  volatile int *ip = arg;

  fprintf(stderr, "About to decrease variable %d times\n", N);
  for (i = 0; i < N; i++) {
    if (USE_ATOMIC_OPS) {
      __sync_sub_and_fetch(ip, 1);
    } else {
      pthread_mutex_lock(&mutex);
      --(*ip);
      pthread_mutex_unlock(&mutex);
    }
  }
  fprintf(stderr, "Done decreasing variable.\n");
  return NULL;
}
```

Ερώτηση 1.1 *Χρησιμοποιήστε την εντολή **time(1)** για να μετρήσετε το χρόνο εκτέλεσης των εκτελέσιμων. Πώς συγκρίνεται ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό, σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό; Γιατί;*

Χρησιμοποιώντας την εντολή time για να μετρήσουμε τους χρόνους εκτέλεσης των εκτελέσιμων παρατηρούμε ότι τα εκτελέσιμα που εκτελούν συγχρονισμό παίρνουν περισσότερο χρόνο σε σχέση με το αρχικό πρόγραμμα (χωρίς συγχρονισμό). Αυτό συμβαίνει γιατί οι λειτουργίες συγχρονισμού προσθέτουν επιπλέον φόρτο, προκαλώντας καθυστερήσεις.

Ερώτηση 1.2 *Ποια μέθοδος συγχρονισμού είναι γρηγορότερη, η χρήση ατομικών λειτουργιών ή η χρήση POSIX mutexes; Γιατί;*

Τα atomic operations φαίνεται να είναι πιο γρήγορα από τα mutexes. Ο λογός είναι οτί ο τρόπος λειτουργίας των atomic operations είναι απλός και απευθύνεται στον πυρήνα του λειτουργικού συστήματος σε αντίθεση με τα mutexes, όπου η υλοποίηση και η διαχείριση τους απαιτεί περισσότερους πόρους.

Ερώτηση 1.3 *Σε ποιες εντολές του επεξεργαστή μεταφράζεται η χρήση ατομικών λειτουργιών του GCC στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Χρησιμοποιήστε την παράμετρο **-S** του **GCC** για να παράγετε τον ενδιάμεσο κώδικα Assembly, μαζί με την παράμετρο **-g** για να συμπεριλάβετε πληροφορίες γραμμών πηγαίου κώδικα (π.χ., ".loc 1 63 0"), οι οποίες μπορεί να σας διευκολύνουν. Δείτε την έξοδο της εντολής **make** για τον τρόπο μεταγλώττισης του **simplesync.c.***

Add

Στη γραμμή 51 του .c αρχείου έχουμε το εξής:

```
__sync_add_and_fetch(ip, 1);
```

Στο .s αρχείο βρίσκουμε τον ενδιάμεσο κώδικα που αφορά αυτή την γραμμή και είναι ο εξής:

```
.loc 1 51 4
movq  -8(%rbp), %rax
lock addl  $1, (%rax)
```

Sub

Στη γραμμή 77 του .c αρχείου έχουμε το εξής:

```
__sync_sub_and_fetch(ip, 1);
```

Στο .s αρχείο βρίσκουμε τον ενδιάμεσο κώδικα που αφορά αυτή την γραμμή και είναι ο εξής:

```
.loc 1 77 4
movq  -8(%rbp), %rax
lock subl  $1, (%rax)
```

Ερώτηση 1.4 *Σε ποιες εντολές μεταφράζεται η χρήση POSIX mutexes στην αρχιτεκτονική για την οποία μεταγλωττίζετε; Παραθέστε παράδειγμα μεταγλώττισης λειτουργίας* ***pthread_mutex_lock()*** *σε Assembly, όπως στο προηγούμενο ερώτημα.*

Add

Στις γραμμές 55-59 του .c αρχείου έχουμε το εξής:

```
    pthread_mutex_lock(&mutex);
  /* You cannot modify the following line */
  ++(*ip);
  /* ... */
  pthread_mutex_unlock(&mutex);
```

Στο .s αρχείο βρίσκουμε τον ενδιάμεσο κώδικα που αφορά αυτές τις γραμμές και είναι ο εξής:

```
.loc 1 55 4
leaq  mutex(%rip), %rax
movq  %rax, %rdi
call  pthread_mutex_lock@PLT
.loc 1 57 7
movq  -8(%rbp), %rax
movl  (%rax), %eax
.loc 1 57 4
leal  1(%rax), %edx
movq  -8(%rbp), %rax
movl  %edx, (%rax)
.loc 1 59 4
leaq  mutex(%rip), %rax
movq  %rax, %rdi
call  pthread_mutex_unlock@PLT
```

Sub

Στις γραμμές 81-85 του .c αρχείου έχουμε το εξής:

```
pthread_mutex_lock(&mutex);
/* You cannot modify the following line */
--(*ip);
/* ... */
pthread_mutex_unlock(&mutex);
```

Στο .s αρχείο βρίσκουμε τον ενδιάμεσο κώδικα που αφορά αυτές τις γραμμές και είναι ο εξής:

```
.loc 1 81 4
leaq  mutex(%rip), %rax
movq  %rax, %rdi
call  pthread_mutex_lock@PLT
.loc 1 83 7
movq  -8(%rbp), %rax
movl  (%rax), %eax
.loc 1 83 4
leal  -1(%rax), %edx
movq  -8(%rbp), %rax
movl  %edx, (%rax)
.loc 1 85 4
leaq  mutex(%rip), %rax
movq  %rax, %rdi
call  pthread_mutex_unlock@PLT
```

2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Ερώτηση 2.1 *Πόσοι σημαφόροι χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;*

Για το σχήμα συγχρονισμού που υλοποιούμε χρειαζόμαστε N σημαφόρους. Δηλαδή ο αριθμός των σημαφόρων θα είναι ίδιος με τον αριθμό των νημάτων. Αυτό δεν είναι τυχαιό, διοτί εχούμε σκοπό να ”αντιστοιχίζουμε” κάθε νήμα με ένα ”δικό του” σημαφόρο.

Ερώτηση 2.2 *Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο νήματα υπολογισμού; Χρησιμοποιήστε την εντολή **time(1)** για να χρονομετρήσετε την εκτέλεση ενός προγράμματος, π.χ., **time sleep 2**. Για να έχει νόημα η μέτρηση, δοκιμάστε σε ένα μηχάνημα που διαθέτει επεξεργαστή δύο πυρήνων. Χρησιμοποιήστε την εντολή **cat /proc/cpuinfo** για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.*

Παρακάτω φαίνεται τι εκτυπώνεται όταν χρονομετράμε την εκτέλεση των προγραμμάτων μας χρησιμοποιώντας την εντολή time.

Χωρίς συγχρονισμό

```
real    0m0.480s
user    0m0.468s
sys     0m0.012s
```

Συγχρονισμός με σημαφόρους (semaphores)

```
real    0m0.250s
user    0m0.479s
sys     0m0.011s
```

Συγχρονισμός με με μεταβλητές συνθήκης (condition variables)

```
real    0m0.249s
user    0m0.472s
sys     0m0.017s
```

Ερώτηση 2.3 *Πόσες μεταβλητές συνθήκης χρησιμοποιήσατε στη δεύτερη εκδοχή του προγράμματος σας? Αν χρησιμοποιηθεί μια μεταβλητή πως λειτουργεί ο συγχρονισμός και ποιο πρόβλημα επίδοσης υπάρχει?*

Όπως και με τους σημαφόρους έτσι και με τις μεταβλητές συνθήκης έχουμε μια για κάθε νήμα (N μεταβλητές συνθήκης). Με αυτόν τον τρόπο ο συγχρονισμός γίνεται προσεκτικά ώστε να αποφευχθούν συγκρούσεις (race conditions) και να εξασφαλιστεί η ασφαλής πρόσβαση στον κρίσιμο τμήμα. Αν και σαν υλοποίηση θα ήταν, ισώς, πιο απλή η χρήση μιας μεταβλήτης θα είχαμε προβλημά επίδοσης. Και αυτό γιατί μπορεί να προκύψει καθυστέρηση λόγω ανταγωνισμού για το κλείδωμα της μεταβλητής συνθήκης. Όταν ένα νήμα κλειδώνει τη μεταβλητή συνθήκης για να εισέλθει στην κρίσιμη περιοχή, τα άλλα νήματα που επιθυμούν να εισέλθουν πρέπει να περιμένουν μέχρι να ελευθερωθεί η μεταβλητή. Αυτό μπορεί να οδηγήσει σε καθυστέρηση στην εκτέλεση και μπορεί να επηρεάσει τη γενική απόδοση του προγράμματος.

Ερώτηση 2.4 *Το παράλληλο πρόγραμμα που φτιάξατε, εμφανίζει επιτάχυνση; Αν όχι, γιατί; Τι πρόβλημα υπάρχει στο σχήμα συγχρονισμού που έχετε υλοποιήσει;* ***Υπόδειξη****: Πόσο μεγάλο είναι το κρίσιμο τμήμα; Χρειάζεται να περιέχει και τη φάση υπολογισμού και τη φάση εξόδου κάθε γραμμής που παράγεται;*

Το παράλληλο πρόγραμμα που φτιάξαμε στην αρχή δεν εμφάνιζε επιτάχυνση. Ύστερα άπο λίγη σκέψη συνειδητοποιήσαμε ότι το κρίσμο τμήμα ήταν πιο μεγάλο από ’τι χρειάζοταν. Πιο συγκεκριμένα, εμπεριείχε και τη φάση υπολογισμού καθέ γραμμής, το όποιο ουσιαστίκα δεν υπάρχει λόγος να είναι εκεί και περισσότερο κακό παρά καλό κάνει.

Ερώτηση 2.5 *Τι συμβαίνει στο τερματικό αν πατήσετε* ***Ctrl-C*** *ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το* ***mandel.c*** *σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει* ***Ctrl-C****, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;*

Όταν πατήσουμε Ctrl-C στο πληκτρολόγιο θα σταλθεί signal SIGINT με αποτέλεσμα η εκτέλεση του προγράμματος να διακοπεί. Αυτή η αποτόμη διακοπή θα εχεί ως αποτέλεσμα τα χρώματα του τερματικού να παραμείνουν στην τελευταία κατάσταση, εφόσον δεν θα έχει τρέξει η εντολή reset_xterm_color(1);
Για να το αντιμετωπίσουμε αυτό το πρόβλημα και να εξασφαλίσουμε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του αρκεί να φτιάξουμε ένα signal handler. Οπότε μέσα στην int main γράφουμε signal(SIGINT, handler); όπου handler είναι το παρακάτω void function:

```
void handler()
{
  reset_xterm_color(1);
  printf("\nReseting Terminal Color and Terminating the program\n");
  exit(1);
}
```