Λειτουργικά Συστήματα Υπολογιστών

2η Εργαστηριακή Άσκηση

oslab064	
Νικόλαος Γιαννόπουλος	03122086
Κωνσταντίνος Καργάκος	03122166

Πίνακας περιεχομένων

1 – Συγχρονισμός σε υπάρχοντα κώδικα

Χρησιμοποιούμε το παρεχόμενο **Makefile** για να μεταγλωττίσουμε το πρόγραμμα **simplesync.c**. Παρατηρούμε πως παράγονται δύο διαφορετικά εκτελέσιμα **simplesync-atomic** και **simplesync-mutex**, ενώ δεν παράγεται εκτελέσιμο με όνομα simplesync.

```
simplesync-mutex.o: simplesync.c
    $(CC) $(CFLAGS) -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c

simplesync-atomic.o: simplesync.c
    $(CC) $(CFLAGS) -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
```

Στο **Makefile** γίνεται compile του προγράμματος **simplesync.c** στα δύο εκτελέσιμα με διαφορετικό **Definition flag**. Το εκτελέσιμο **simplesync-mutex** παράγεται με το **[-DSYNC_MUTEX]** flag, ενώ το εκτελέσιμο **simplesync-atomic** με το **[-DSYNC_ATOMIC]**. Αυτά τα flags ορίζουν τα **SYNC_MUTEX** και **SYNC_ATOMIC** αντίστοιχα και έτσι ο ίδιος πηγαίος κώδικας έχει τη δυνατότητα να εκτελέσει διαφορετικές εντολές. Στην περίπτωσή μας υλοποιούμε διαφορετικό συγχρονισμό.

Αρχικά τα δύο εκτελέσιμα δεν περιέχουν συγχρονισμό. Τρέχουμε τα αρχεία και παρατηρούμε ότι παίρνουμε μη επιθυμητό αποτέλεσμα:

```
~/Development/NTUA-Projects/06-Operating-Systems/os-lab02 time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -3279383.
./simplesync-mutex  0.04s user 0.00s system 179% cpu 0.023 total
```

Αυτό συμβαίνει διότι χωρίς το συγχρονισμό προκύπτει το πρόβλημα race condition ανάμεσα στις δύο διεργασίες, οι οποίες επεξεργάζονται την ίδια μεταβλητή.

Υλοποιούμε τον συγχρονισμό με χρήση POSIX mutexes:

```
static pthread_mutex_t mtx;
```

Ορίζουμε ένα POSIX mutex με όνομα **mtx**.

```
pthread_mutex_lock(&mtx);
++(*ip);
pthread_mutex_unlock(&mtx);
```

Στα σημεία που θέλουμε να αυξήσουμε τη μεταβλητή, την οποία μοιράζονται τα δύο threads, προσθέτουμε τα mutex locks (lock/unlock). Με αυτά ορίζουμε ένα κρίσιμο τμήμα του κώδικα, μέσα στο οποίο “βρίσκεται” μόνο ένα thread κάθε στιγμή. Έτσι, έχουμε συγχρονισμό μεταξύ των threads και η μεταβλητή αλλάζει μόνο από ένα thread κάθε στιγμή.

```
pthread_mutex_lock(&mtx);
--(*ip);
pthread_mutex_unlock(&mtx);
```

Αντίστοιχα το ίδιο κάνουμε στα σημεία που θέλουμε να μειώσουμε τη μεταβλητή.

Επιπλέον, στο main() αρχικοποιούμε το **mtx**:

```
    if (!USE_ATOMIC_OPS) {
        ret = pthread_mutex_init(&mtx, NULL);
        if (ret) {
          perror_pthread(ret, "pthread_mutex_init");
          exit(1);
       }
    }
```

Τέλος, “καταστρέφουμε” το **mtx**:

```
    if (!USE_ATOMIC_OPS) {
       ret = pthread_mutex_destroy(&mtx);
       if (ret) {
          perror_pthread(ret, "pthread_mutex_destroy");
          exit(1);
       }
    }
```

Επιβεβαιώνουμε την ορθή λειτουργία του **simplesync-mutex**:

```
~/Development/NTUA-Projects/06-Operating-Systems/os-lab02 time ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
./simplesync-mutex  0.25s user 0.22s system 151% cpu 0.311 total
```

Υλοποιούμε τον συγχρονισμό με χρήση ατομικών λειτουργιών του GCC:

```
       if (USE_ATOMIC_OPS) {
          __sync_add_and_fetch((int *)ip, 1);
       } else {
          pthread_mutex_lock(&mtx);
          ++(*ip);
          pthread_mutex_unlock(&mtx);
       }
```

Στα σημεία που θέλουμε να αυξήσουμε ατομικά τη μεταβλητή, την οποία μοιράζονται τα δύο threads, καλούμε την εντολή **[`__sync_add_and_fetch((int *)ip, 1)`]**, η οποία προσθέτει την τιμή 1 στη μεταβλητή που δείχνει ο pointer ip.

```
       if (USE_ATOMIC_OPS) {
          __sync_sub_and_fetch((int *)ip, 1);
       } else {
          pthread_mutex_lock(&mtx);
          --(*ip);
          pthread_mutex_unlock(&mtx);
       }
```

Στα σημεία που θέλουμε να μειώσουμε ατομικά τη μεταβλητή, την οποία μοιράζονται τα δύο threads, καλούμε την εντολή **[`__sync_sub_and_fetch((int *)ip, 1)`]**, η οποία αφαιρεί την τιμή 1 στη μεταβλητή που δείχνει ο pointer ip.

Έτσι, εξασφαλίζουμε ατομικότητα των πράξεων και λύνεται το πρόβλημα του συγχρονισμού.

Επιβεβαιώνουμε την ορθή λειτουργία του **simplesync-atomic**:

```
~/Development/NTUA-Projects/06-Operating-Systems/os-lab02 time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
./simplesync-atomic  0.74s user 0.00s system 197% cpu 0.376 total
```

Ερωτήσεις:

Με την εντολή time μετράμε το χρόνο εκτέλεσης:

Time (s)	user	system	total
Χωρίς συγχρονισμό	0.04	0.00	0.023
POSIX mutex	0.25	0.22	0.311
Atomic GCC	0.74	0.00	0.376

1. Ο χρόνος εκτέλεσης χωρίς συγχρονισμό είναι πολύ μικρότερος σε σχέση με συγχρονισμό. Αυτό συμβαίνει επειδή τα νήματα μπλοκάρονται μεταξύ τους και περιμένουν για πρόσβαση σε κοινά δεδομένα, προσθέτοντας καθυστέρηση.

2. Παρατηρούμε ότι η υλοποίηση συγχρονισμού με POSIX mutexes έχει παρόμοιο χρόνο εκτέλεσης στο user και στο system. Στην υλοποίηση συγχρονισμού με ατομικές λειτουργίες του GCC ο χρόνος εκτέλεσης στο system είναι πάρα πολύ μικρός, αφού δεν απαιτεί κλήσεις συστήματος. Αντίθετα, τα mutexes απαιτούν πολλές κλήσεις συστήματος μιας και κάθε thread προσπαθεί να πάρει lock που κατέχει ήδη άλλο.

Επίσης, παρατηρούμε ότι το total time στο atomic είναι λίγο μεγαλύτερο από το mutex στο δικό μας σύστημα, γιατί το user time είναι αρκετά μεγαλύτερο στο atomic. Παρόλο που στο atomic δεν απαιτούνται κλήσεις συστήματος και θεωρητικά θα έπρεπε ο χρόνος εκτέλεσης να είναι μικρότερος από το mutex, όμως αυτό δεν συμβαίνει στο δικό μας σύστημα. Πιθανόν, λόγω της υψηλής επεξεργαστικής δύναμης του συστήματός μας, ο χρόνος εκτέλεσης με mutex είναι αρκετά μικρός και πλησιάζει αυτόν της εκτέλεσης με atomic.

Εάν τρέξουμε τα δύο εκτελέσιμα στο orion server παρατηρούμε ότι ο χρόνος εκτέλεσης με atomic είναι πολύ μικρότερος από αυτόν με mutex.

```
oslab064@os-node2:~/lab02$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m5.093s
user    0m6.032s
sys     0m3.626s
```

```
oslab064@os-node2:~/lab02$ time ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m0.955s
user    0m1.872s
sys     0m0.005s
```

3. Παράγουμε τον ενδιάμεσο κώδικα Assembly και παρατηρούμε τις εντολές του επεξεργαστή που μεταφράζεται η χρήση **ατομικών λειτουργιών του GCC** (τρέχουμε την εντολή σε **αρχιτεκτονική ARM64**).

Αύξηση του *ip:

```
mov w8, #1                          ; =0x1
.loc    0 49 4 is_stmt 1                ; simplesync.c:49:4
ldaddal w8, w9, [x19]
```

Η εντολή ldaddal φορτώνει τη μεταβλητή στο w9, της προσθέτει την w8 (+1) και τέλος την αποθηκεύει πίσω. Αυτό γίνεται σε μια εντολή, όποτε καμία άλλη εντολή διαφορετικού thread δεν επηρεάζει τη μεταβλητή ενδιάμεσα.

Μείωση του *ip:

```
mov w8, #-1                         ; =0xffffffff
.loc    0 69 4 is_stmt 1                ; simplesync.c:69:4
ldaddal w8, w9, [x19]
```

Η εντολή ldaddal φορτώνει τη μεταβλητή στο w9, της προσθέτει την w8 (-1) και τέλος την αποθηκεύει πίσω. Αυτό γίνεται σε μια εντολή, όποτε καμία άλλη εντολή διαφορετικού thread δεν επηρεάζει τη μεταβλητή ενδιάμεσα.

4. Παράγουμε τον ενδιάμεσο κώδικα Assembly και παρατηρούμε τις εντολές του επεξεργαστή που μεταφράζεται η χρήση **POSIX mutexes** (τρέχουμε την εντολή σε **αρχιτεκτονική ARM64**).

```
    .loc   0 51 4 is_stmt 1             ; simplesync.c:51:4
    mov    x0, x20
    bl _pthread_mutex_lock
Ltmp4:
    .loc   0 52 4                       ; simplesync.c:52:4
    ldr    w8, [x19]
    add    w8, w8, #1
    str    w8, [x19]
    .loc   0 53 4                       ; simplesync.c:53:4
    mov    x0, x20
    bl _pthread_mutex_unlock
```

Παρατηρούμε ότι καλούνται οι συναρτήσεις **[_pthread_mutex_lock]** και **[_pthread_mutex_unlock]** για να κλειδώσει και να ξεκλειδώσει το mutex αντίστοιχα. Ενδιάμεσα αυτών πραγματοποιείται η πρόσθεση (και η αφαίρεση αντίστοιχα) της μεταβλητής. Εάν κάποιο άλλο thread έχει κάνει lock το mutex τότε η **[_pthread_mutex_lock]** θα περιμένει μέχρι να γίνει unlock.

2 – Παράλληλος υπολογισμός του συνόλου Mandelbrot

2.1 - Με σημαφόρους

Για το συγχρονισμό των N threads φτιάχνουμε N semaphores, έναν για κάθε thread.

```
static sem_t *sems;
static int nthreads;
```

Καταλαμβάνουμε δυναμικά θέση στη μνήμη για τους N σημαφόρους με τη malloc.

```
sems = malloc(nthreads * sizeof(sem_t));
if (!sems) {
    perror("malloc sems");
    exit(1);
}
```

Αρχικοποιούμε τους σημαφόρους με τιμή 0, εκτός από τον πρώτο που το αρχικοποιούμε με τιμή 1, ώστε το πρώτο thread να ξεκινήσει να τυπώνει κατευθείαν στην έξοδο. Τα υπόλοιπα threads περιμένουν μέχρι να τελειώσει το προηγούμενο από αυτά thread.

```
   for (int i = 0; i < nthreads; i++) {
       if (sem_init(&sems[i], 1, i == 0 ? 1 : 0) == -1) {
           perror("sem_init");
           exit(1);
       }
   }
```

Δημιουργούμε N threads, τα οποία εκτελούν τη συνάρτηση **compute_and_output()** και δίνουμε ως argument στο κάθε thread τον αριθμό της i (μέσω του δείκτη p). Έτσι θα μπορούν τα threads να τυπώνουν με την κατάλληλη σειρά στην έξοδο και να εναλλάσσονται κυκλικά με βάση την επόμενη γραμμή που έχει σειρά να τυπωθεί.

```
pthread_t *tids = malloc(sizeof(pthread_t) * nthreads);
if (!tids) {
    perror("malloc tids");
    exit(1);
}

for (int i = 0; i < nthreads; i++) {
    int *p = malloc(sizeof *p);
    if (!p) {
       perror("malloc p");
       exit(1);
    }
    *p = i;
    int ret = pthread_create(&tids[i], NULL, compute_and_output, p);
    if (ret) {
       perror_pthread(ret, "pthread_create");
       exit(1);
    }
}

for (int i = 0; i < nthreads; i++) {
```

```
    int ret = pthread_join(tids[i], NULL);
    if (ret) {
       perror_pthread(ret, "pthread_join");
       exit(1);
    }
}
```

Αφού τελειώσουν τα threads και τυπωθεί το αποτέλεσμα, καταστρέφουμε τους N σημαφόρους.

```
   for (int i = 0; i < nthreads; i++) {
       if (sem_destroy(&sems[i]) == -1) {
           perror("sem_destroy");
           exit(1);
       }
   }
```

Συνάρτηση **void *compute_and_output(void *arg)**:

```
static void *compute_and_output(void *arg) {
    int tid = *(int*)arg;
    free(arg);

    int color_val[x_chars];

    for (int line = tid; line < y_chars; line += nthreads) {
       compute_mandel_line(line, color_val);

       if (sem_wait(&sems[tid]) == -1) {
          perror("sem_wait");
          exit(1);
       }

       output_mandel_line(1, color_val);

       int next_tid = (tid + 1) % nthreads;
       if (line + 1 < y_chars) {
          if (sem_post(&sems[next_tid]) == -1) {
             perror("sem_post");
             exit(1);
          }
       }
    }
    return NULL;
}
```

Κάθε thread μπαίνει στο for loop, ώστε να υπολογίσει και να εκτυπώσει τις γραμμές *i, i+n, i+2*n, ...* . Ο υπολογισμός **compute_mandel_line()** δεν γίνεται με συγχρονισμό και μπορεί να γίνεται ταυτόχρονα σε όλα τα threads μιας και ο συγχρονισμός μας ενδιαφέρει τη στιγμή που τα threads τυπώνουν τη γραμμή τους.

Αφού τα threads υπολογίσουν τη γραμμής τους, τότε περιμένουν μέχρι ο δικός τους σημαφόρος να πάρει τιμή 1. Όταν ένα thread τυπώσει τη γραμμή του στην έξοδο, στη

συνέχεια θέτει την τιμή 1 στον σημαφόρο που αντιστοιχεί στο thread που πρέπει να τυπώσει την επόμενη γραμμή. Αυτό υπολογίζεται στη μεταβλητή **[`next_tid = (tid + 1) % nthreads`]**.

2.2 - Με condition variables

Αντίστοιχα, για τον συγχρονισμό των N threads φτιάχνουμε N condition variables.

```
static pthread_mutex_t mtx;
static pthread_cond_t *cond;
static int nthreads;
static int next_line;
```

Αρχικοποιούμε το mutex [**mtx**] αλλά και όλα τα condition variables [**cond**].

```
int ret = pthread_mutex_init(&mtx, NULL);
if (ret) {
    perror_pthread(ret, "pthread_mutex_init");
    exit(1);
}

cond = malloc(sizeof(*cond) * nthreads);
if (!cond) {
    perror("malloc cond");
    exit(1);
}
for (int i = 0; i < nthreads; i++) {
    ret = pthread_cond_init(&cond[i], NULL);
    if (ret) {
       perror_pthread(ret, "pthread_cond_init");
       exit(1);
    }
}
```

Όπως και στην περίπτωση με τους σημαφόρους, δημιουργούμε N threads, τα οποία εκτελούν τη συνάρτηση **compute_and_output()** και δίνουμε ως argument στο κάθε thread τον αριθμό της i (μέσω του δείκτη p). Έτσι θα μπορούν τα threads να τυπώνουν με την κατάλληλη σειρά στην έξοδο και να εναλλάσσονται κυκλικά με βάση την επόμενη γραμμή που έχει σειρά να τυπωθεί.

Αφού τελειώσουν τα threads και τυπωθεί το αποτέλεσμα, καταστρέφουμε τα N condition variables αλλά και το mutex.

```
reset_xterm_color(1);
free(tids);
pthread_mutex_destroy(&mtx);
for (int i = 0; i < nthreads; i++) {
    ret = pthread_cond_destroy(&cond[i]);
    if (ret) {
       perror_pthread(ret, "pthread_cond_destroy");
       exit(1);
    }
}
```

Συνάρτηση **void *compute_and_output(void *arg)**:

```
static void *compute_and_output(void *arg) {
    int tid = *(int*)arg;
    free(arg);

    int color_val[x_chars];

    for (int line = tid; line < y_chars; line += nthreads) {
       compute_mandel_line(line, color_val);

       pthread_mutex_lock(&mtx);
       while (line != next_line) {
            pthread_cond_wait(&cond[tid], &mtx);
       }
       output_mandel_line(1, color_val);

       next_line++;
       int next_tid = next_line % nthreads;
       pthread_cond_signal(&cond[next_tid]);
       pthread_mutex_unlock(&mtx);
    }
    return NULL;
}
```

Ξανά ο υπολογισμός της γραμμής **compute_mandel_line()** γίνεται χωρίς συγχρονισμό.

Αφού τα threads υπολογίσουν την γραμμή τους, τότε περιμένουν στο while loop μέχρι να δοθεί σήμα στο αντίστοιχο condition variable. Όταν ένα thread τυπώσει τη γραμμή του στην έξοδο, στη συνέχεια δίνει σήμα στο condition variable που αντιστοιχεί στο thread που πρέπει να τυπώσει την επόμενη γραμμή. Αυτό υπολογίζεται στη μεταβλητή **[`next_tid = next_line % nthreads`]**.

Για να προχωρήσει ένα thread και να τυπώσει την γραμμή του θα πρέπει να ικανοποιεί τη συνθήκη **[`line != next_line`]** μιας και μόνο τότε είναι η σειρά του να τυπώσει την γραμμή που έχει υπολογίσει.

Ερωτήσεις:

1. Στο σχήμα συγχρονισμού που υλοποιήσαμε χρειάζονται N σημαφόροι, δηλαδή όσα και τα threads που δημιουργούνται.
2. Τρέχουμε τα δύο εκτελέσιμα (σειριακό και παράλληλο με συγχρονισμό):

   ```
   ./mandel  0.41s user 0.01s system 97% cpu 0.424 total
   ./mandel-semaphores 2  0.42s user 0.01s system 191% cpu 0.226 total
   ./mandel-condition-var 2  0.42s user 0.01s system 191% cpu 0.223 total
   ```

   Παρατηρούμε ότι η εκτέλεση παράλληλου προγράμματος με συγχρονισμό σε μηχάνημα με 10 πυρήνες είναι πολύ πιο γρήγορη από αυτή του σειριακού προγράμματος.

Όσο αυξάνουμε τον αριθμό των threads αναμένουμε να αυξάνεται η επίδοση του προγράμματος μέχρι ο αριθμός τους να φτάσει το πλήθος των διαθέσιμων πυρήνων.

3. Στη δεύτερη εκδοχή του προγράμματος χρησιμοποιήσαμε N condition variables, δηλαδή όσα και τα threads που δημιουργούνται.
   Αν χρησιμοποιήσουμε μια μεταβλητή τότε θα πρέπει να αντικαταστήσουμε την εντολή **[pthread_cond_signal(&cond[next_tid])]** με την εντολή **[pthread_cond_broadcast(&cond)]** μιας και τώρα χρειάζεται να δίνεται σήμα σε όλα τα threads που περιμένουν. Σε αυτή την περίπτωση που δίνεται σήμα σε όλα τα threads, ελέγχει κάθε thread τη συνθήκη **[line != next_line]** του while loop, και βγαίνει από το loop μόνο το thread που είναι η σειρά του να τυπώσει στην έξοδο. Ο έλεγχος, όμως, αυτής της συνθήκης γίνεται σειριακά από τα threads λόγω του mutex. Οι συνεχείς και σειριακοί έλεγχοι αυτής της συνθήκης κάθε φορά που ένα thread τελειώνει την εκτύπωση μιας γραμμής προσθέτει επιπλέον καθυστέρηση και μειώνει την επίδοση.
4. Το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση, εφόσον όλα τα threads εκτελούν το τμήμα του υπολογισμού των γραμμών παράλληλα (χωρίς συγχρονισμό), ενώ εφαρμόζουμε συγχρονισμό μόνο στο κρίσιμο τμήμα που είναι η εκτύπωση των γραμμών στην έξοδο.
5. Αν πατήσουμε Ctrl-C ενώ το πρόγραμμα εκτελείται τότε η εκτέλεση σταματάει αμέσως και το χρώμα των γραμμάτων του terminal δεν αλλάζει όπως πριν την εκτέλεση, αφού δεν προλαβαίνει να εκτελεστεί η εντολή **[reset_xterm_color(1);]**.
   Επεκτείνουμε το **mandel.c**, ώστε να εξασφαλίσουμε ότι ακόμη και αν ο χρήστης πατήσει Ctrl-C το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του. Προσθέτουμε το παρακάτω signal handler:

```
signal(SIGINT, handle_signint);
```

το οποίο εκτελεί τη συνάρτηση:

```
static void handle_signint(int sign) {
    reset_xterm_color(1);
    fflush(stdout);
    printf("\n");
    exit(0);
}
```