Λειτουργικά Συστήματα Υπολογιστών

6ο εξάμηνο, Ακαδημαϊκή περίοδος 2023-2024

2η Εργαστηριακή Άσκηση

Συγχρονσισμός

Δημήτρης Βατάλας: 03121093

Αγγελική Ζέρβα: 03121101

Άσκηση 1

Στη συγκεκριμένη άσκηση, μας δίνεται το πρόγραμμα **simplesync.c**, το οποίο έχει την εξής λειτουργία. Αφού αρχικοποιεί μια μεταβλητή val = 0, δημιουργεί δύο νήματα, τα οποία εκτελούνται ταυτόχρονα και το πρώτο αυξάνει N φορές την τιμή της μεταβλητής val κατά 1, ενώ το δεύτερο μειώνει N φορές την τιμή της μεταβλητής val κατά 1. Σκοπός μας είναι να συγχρονίσουμε τα δύο νήματα, έτσι ώστε η τελική τιμή της μεταβλητής val να είναι ίση με το μηδέν.

Αρχικά, χρησιμοποιούμε το περιεχόμενο Makefile για να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα. Κατά τη μεταγλώττιση παρατηρούμε πως παράγονται δύο διαφορετικά εκτελέσιμα : **simplesync-atomic** και **simplesync-mutex**. Παρακάτω παρατηρούμε τη δομή του Makefile για την επίτευξη της παραπάνω λειτουργίας.

```
## Simple sync (two versions)
simplesync-mutex: simplesync-mutex.o
	$(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o $(LIBS)

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o $(LIBS)

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
```

Κατά τη δημιουργία των object files **simplesync-atomic.o** και **simplesync-mutex.o** χρησιμοποιούνται τα flags **DSYNC_MUTEX** και **DSYNC_ATOMIC**, μέσω των οποίων γίνονται predefined τα macros **SYNC_MUTEX** και **SYNC_ATOMIC**. Έτσι, ανάλογα με το macro που έχει γίνει predefined, αλλάζει ο τρόπος λειτουργίας και επιλέγεται η αντίστοιχη μέθοδος συγχρονισμού που έχουμε υλοποιήσει. Για να συμβεί αυτό, γίνεται χρήση του παρακάτω κώδικα στην αρχή του **simplesync.c**, μέσω του οποίου, η μεταβλητή **USE_ATOMIC_OPS** γίνεται 1 ή 0, ανάλογα με το macro που έχει οριστεί και στη συνέχεια καθορίζει αν θα γίνει συγχρονισμός με mutexes (αν είναι 0) ή με atomic operations (αν είναι 1).

```
#if defined(SYNC_ATOMIC)
# define USE_ATOMIC_OPS 1
#else
# define USE_ATOMIC_OPS 0
#endif
```

Στη συνέχεια, τρέχουμε τα simplesync-atomic και simplesync-mutex και παρατηρούμε ότι και στις δύο περιπτώσεις παράγεται λάθος αποτέλεσμα για την τιμή της μεταβλητής val.

```
oslab093@os-node2:~/ex2/original/sync$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -9037964.
oslab093@os-node2:~/ex2/original/sync$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 2389751.
oslab093@os-node2:~/ex2/original/sync$
```

Το λάθος αποτέλεσμα οφείλεται στην απουσία συγχρονισμού των δύο νημάτων, με αποτέλεσμα να υπάρχει ασυνέπεια και αβεβαιότητα ως προς την τελική τιμή της μεταβλητής val. Πιο συγκεκριμένα, η πρόσθεση και αφαίρεση μεταφράζονται σε παραπάνω από μια εντολές σε assembly, με αποτέλεσμα να υπάρχει ένα κρίσιμο τμήμα (**critical part**) για την εκτέλεση των συγκεκριμένων λειτουργιών από το κάθε νήμα. Έτσι, αν ενώ ένα νήμα βρίσκεται εντός του κρίσιμου τμήματος για την εκτέλεση της πρόσθεσης ή της αφαίρεσης γίνει αλλαγή του νήματος στον επεξεργαστή, τότε διακόπτεται η εκτέλεση του αρχικού νήματος και το αποτέλεσμα είναι απρόβλεπτο. Παρατηρούμε λοιπόν, πως πρόκειται για μια κατάσταση συναγωνισμού (**race condition**), κατά την οποία το αποτέλεσμα του υπολογισμού, δηλαδή της τελικής τιμής της μεταβλητής val, εξαρτάται από τη σειρά με την οποία γίνονται οι προσπελάσεις.

Για να αντιμετωπίσουμε το πρόβλημα που περιγράψαμε παραπάνω, επεκτείνουμε τον κώδικα simplesync.c ώστε ο συγχρονισμός να επιτυγχάνεται με τη χρήση των **POSIX mutexes** και με τη χρήση ατομικών λειτουργιών (**atomic operations**) του GCC. Παρακάτω φαίνονται τα σημεία του κώδικα, στα οποία εκτελείται ο συγχρονισμός με καθένα από τους δύο τρόπους που αναφέραμε, ενώ υπάρχει και η αντίστοιχη επεξήγηση στα σχόλια.

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

    fprintf(stderr, "About to increase variable %d times\n", N);
    for (i = 0; i < N; i++) {
        if (USE_ATOMIC_OPS) { // Use atomic operations
            /* ... */
            /* You can modify the following line */
            __sync_add_and_fetch(&ip, 1);
            /* ... */
        } else { //Use POSIX mutexes
            /* ... */
            int ret;
            ret = pthread_mutex_lock(&mutex); //get lock, or wait for lock
            if(ret)
                perror_pthread(ret, "pthread_mutex_lock");
            /* You cannot modify the following line */
            ++(*ip); //critical part
            /* ... */
            ret = pthread_mutex_unlock(&mutex); //unlock to let another
                                                //thread get the lock
            if(ret)
                perror_pthread(ret, "pthread_mutex_unlock");
        }
    }
    fprintf(stderr, "Done increasing variable.\n");

    return NULL;
}
```

```c
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) {  // Use atomic operations
            /* ... */
            /* You can modify the following line */
            __sync_sub_and_fetch(&ip, 1); // Use atomic operations
            /* ... */
        } else { //Use POSIX mutexes
            /* ... */
            int ret;
            ret = pthread_mutex_lock(&mutex); //get lock, or wait for lock
            if(ret)
                perror_pthread(ret, "pthread_mutex_lock");
            /* You cannot modify the following line */
            --(*ip); //critical part
            /* ... */
            ret = pthread_mutex_unlock(&mutex); //unlock to let another
                                                //thread get the lock
            if(ret)
                perror_pthread(ret, "pthread_mutex_unlock");
        }
    }
    fprintf(stderr, "Done decreasing variable.\n");

    return NULL;
    }
```

Σημειώνουμε επίσης, ότι μετά την εκτέλεση των προσθέσεων και των αφαιρέσεων καταστρέφουμε το mutex με την χρήση της παρακάτω εντολής:

```c
ret = pthread_mutex_destroy(&mutex);
if(ret) perror_pthread(ret, "pthread_mutex_destroy");
```

Μεταγλωτίζοντας και εκτελόντας τα δύο προγράμματα, λαμβάνουμε τα παρακάτω αποτελέσματα, τα οποία επιβεβαιώνουν την ορθή τους λειτουργία, αφού και στις δύο περιπτώσεις val $= 0$.

```
oslab093@os-node2:~/ex2/sync$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
oslab093@os-node2:~/ex2/sync$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Ερώτημα 1.1

Χρησιμοποιώντας την εντολή **time(1)** βλέπουμε τον χρόνο εκτέλεσης των δύο νέων εκτελέσιμων και του αρχικού για N = 10000000

```
oslab093@os-node2:~/ex2/original/sync$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 3962614.

real    0m0.278s
user    0m0.545s
sys     0m0.000s
```

Figure 1: Χωρίς συγχρονισμό

```
oslab093@os-node1:~/ex2/sync$ 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    0m0.321s
user    0m0.629s
sys     0m0.004s
```

Figure 2: Συγχρονισμός με atomic operations

```
oslab093@os-node1:~/ex2/sync$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m6.769s
user    0m6.802s
sys     0m6.442s
```

Figure 3: Συγχρονισμός με POSIX mutexes

Παρατηρούμε αρχικά ότι ο χρόνος εκτέλεσης των εκτελέσιμων με συγχρονισμό είναι μεγαλύτερος από τον χρόνο εκτέλεσης του εκτελέσιμου χωρίς συγχρονισμό. Αυτό οφείλεται στο γεγονός ότι χωρίς τον συγχρονισμό δεν υπάρχει ο περιορισμός ότι θα υπάρχει αποκλειστικά ένα νήμα στο κύριο τμήμα, με αποτέλεσμα τα νήματα να εκτελούν τις λειτουργίες τους παράλληλα, οδηγώντας και στα αρνητικά αποτελέσματα που αναφέραμε παραπάνω.

Ερώτημα 1.2

Αναφορικά με τη διαφορά του χρόνου εκτέλεσης ανάμεσα στα δύο σχήματα παρατηρούμε ότι ο χρόνος εκτέλεσης για τα atomic operations είναι σημαντικά μικρότερος από τον χρόνο των POSIX mutexes. Στην περίπτωση των atomic operations ο συγχρονσιμός υλοποιείται από το υλικό και εγγυάται τη λειτουργία της κάθε εντολής ατομικά. Από την άλλη, τα POSIX mutexes εμπλέκουν το λειτουργικό έτσι ώστε να αναστέλλεται η λειτουργία ενός νήματος όσο αυτό περιμένει και να ενεργοποιείται όταν απελευθερωθεί ξανά το lock. Αυτή η διαδικασία σε συνδυασμό με τη μεγαλύτερη πολυπλοκότητα σε επίπεδο assembly των POSIX mutexes, αιτιολογεί τον μεγαλύτερο χρόνο εκτέλεσης με το συγκεκριμένο σχήμα συγχρονισμού.

Ερώτημα 1.3

Για να παράγουμε τον κώδικα assembly για το simplesync.c τροποποιούμε το Makefile όπως φαίνεται παρακάτω και παρουσιάζουμε την έξοδο του make:

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

simplesync-mutex.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c

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

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

```
oslab093@os-node2:~/ex2/sync$ make simplesync-atomic.s
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c
oslab093@os-node2:~/ex2/sync$ make simplesync-mutex.s
gcc -Wall -O2 -pthread -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c
```

Οι εντολές assembly στις οποίες μεταφράζονται τα atomic operations είναι οι εξής:

- Για το thread που αυξάνει τη μεταβλητή:

```
.loc 1 52 4 view .LVU12
lock addq	$1, 8(%rsp)
```

- Για το thread που μειώνει τη μεταβλητή:

```
lock subq	$1, 8(%rsp)
.loc 1 96 5 view .LVU34
```

Ερώτημα 1.4

Οι εντολές assembly στις οποίες μεταφράζονται τα POSIX mutexes είναι οι εξής:

```
.globl  mutex
.bss
.align 32
.type   mutex, @object
.size   mutex, 40
```

- Για το thread που αυξάνει τη μεταβλητή:

```
.loc 1 87 4 view .LVU83
.loc 1 88 4 view .LVU84
.loc 1 88 10 is_stmt 0 view .LVU85
movq    %r13, %rdi
call    pthread_mutex_lock@PLT
```

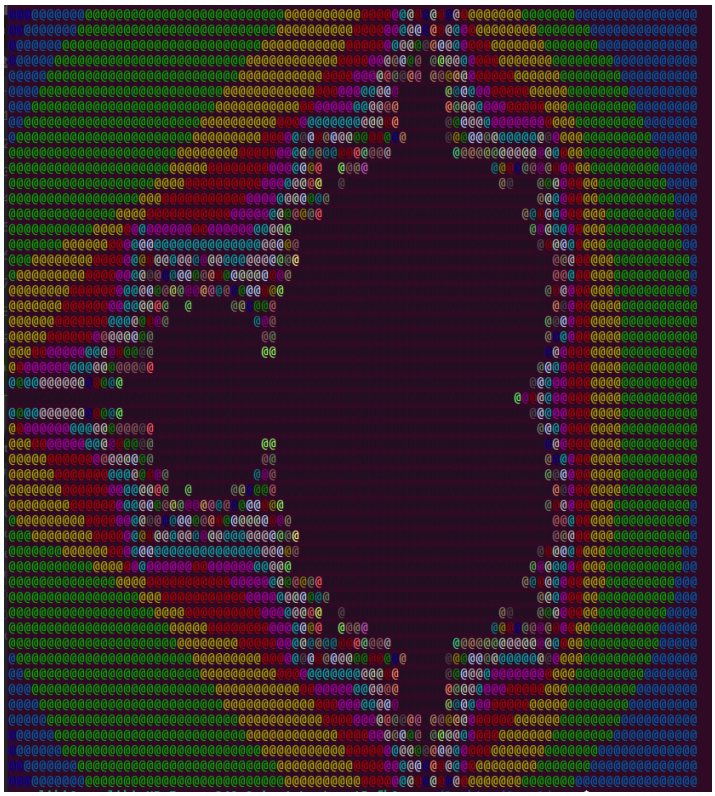
- Για το thread που μειώνει τη μεταβλητή:

```
.loc 1 90 5 discriminator 1 view .LVU68
.loc 1 92 4 discriminator 1 view .LVU69
.loc 1 92 7 is_stmt 0 discriminator 1 view .LVU70
movl    (%r12), %eax
.loc 1 94 10 discriminator 1 view .LVU71
movq    %r13, %rdi
.loc 1 92 4 discriminator 1 view .LVU72
subl    $1, %eax
movl    %eax, (%r12)
.loc 1 94 4 is_stmt 1 discriminator 1 view .LVU73
.loc 1 94 10 is_stmt 0 discriminator 1 view .LVU74
call    pthread_mutex_unlock@PLT
```

Άσκηση 2

Σκοπός της συγκεκριμένης άσκησης είναι να επεκτείνουμε το πρόγραμμα **mandel.c**, το οποίο υπολογίζει και σχεδιάζει στο τερματικό κειμένου το σύνολο Mandelbrot, με σκοπό ο υπολογισμός να κατανέμεται σε NTHREADS (θα δίνεται κατά την εκτέλεσή) νήματα POSIX. Έτσι, για n νήματα, το i-ιοστό (με i = 0, 1, 2, . . . , n-1) αναλαμβάνει τις σειρές i, i + n, i + 2 × n, i + 3 × n, . . .

Ο Συγχρονισμός των νημάτων θα γίνει με δύο τρόπους. Με τη χρήση **σεμαφόρων** από το πρότυπο POSIX και με τη χρήση μεταβλητών συνθήκης (**condition variables**) επίσης από το πρότυπο POSIX. Παρακάτω παρουσιάζεται το σχέδιο που σχηματίζεται με το δοσμένο μοντέλο Mandelbrot.



1.Συγχρονισμός με Σεμαφόρους

Αρχικά δημιουργούμε τα NTREADS και τους αναθέτουμε να υπολογίσουν το ζητούμενο τμήμα του Mandelbrot, καλώντας για το καθένα τη συνάρτηση **compute_and_output_mandel_line**.

```
/*Create threads and assign them to compute and output the needed lines*/
for(int i=0; i<thrdcnt; i++){
    ret = pthread_create(&thr[i], NULL, compute_and_output_mandel_line, (void*)i);
    if(ret){
        perror_pthread(ret, "pthread create");
    }
}
```

Στη συνέχεια χρησιμοποιούμε τη συνάρτηση safe_malloc για την δημιουργία ενός πίνακα από semaphores, έτσι ώστε να δημιουργήσουμε τόσα semaphores όσα είναι και τα νήματα. Αρχικοποιούμε όλα τα semaphores πλην του πρώτου στο 0(locked), έτσι ώστε μόνο το πρώτο thread, του οποίο τον σεμαφόρο αρχικοποιούμε στο 1 (unlocked), να μπει στο κρίσιμο τμήμα, να τυπώσει το line 0 και στη συνέχεια να αυξήσει το σεμαφόρο του επόμενου thread σε 1 (unlock), το οποίο θα εκτελέσει την ίδια λειτουργία. Η αρχικοποίηση των σεμαφόρων είναι η εξής:

```
sem = (sem_t*)safe_malloc(thrdcnt * sizeof(sem_t));
ret = sem_init(&sem[0], 0, 1); //Initialize the first semaphore to 1

if(ret!=0) {
    perror("sem_init");
    exit(1);
}
/*Initialize all other semaphores to 0*/
for(int i=1; i<thrdcnt; i++) {
    ret = sem_init(&sem[i], 0, 0);
    if(ret<0) {
        perror("sem_init");
        exit(1);
    }
}
```

Η υλοποίηση της **compute_and_output_mandel_line** στην οποία επιτυγχάνεται ο συγχρονισμός που περιγράψαμε παραπάνω φαίνεται παρακάτω και συνοδεύεται από τα απαιτητά σχόλια για την επεξήγησή της.

```
void *compute_and_output_mandel_line(void *thread_num)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars], ret;

    /*A loop that iterates over all lines for a thread (lines = y_chars)*/
    for(int line=(int)thread_num; line<=y_chars; line+=thrdcnt) {
        compute_mandel_line(line, color_val);
        /* CRITICAL PART START*/
        ret = sem_wait(&sem[(int)thread_num]); //wait on the semaphore for the
        if(ret<0) {                            //thread with number thread_num
            perror("sem_wait");
            exit(1);
        }

        output_mandel_line(1, color_val);
        /* CRITICAL PART END */
        // post semaphore for the the next thread, (use %thrdnct to be circular)
        ret = sem_post(&sem[((int)thread_num + 1)%thrdcnt]); /
        if(ret<0) {
            perror("sem_post");
            exit(1);
        }
```

```
    }
    return NULL;
}
```

Αφού ολοκληρωθεί ο υπολογισμός και η σχεδίαση του Mandelbrot από τα νήματα, περιμένουμε τα νήματα μα τερματίσουν και στη συνέχεια καταστρέφουμε τα semaphores.

```
/*Wait for threads to terminate*/
for (int i = 0; i < thrdcnt; i++) {
            ret = pthread_join(thr[i], NULL);
            if (ret) {
                    perror_pthread(ret, "pthread_join");
                    exit(1);
            }
    }

/*Destrory semaphores*/
for (int i=0; i<thrdcnt; i++){
    ret = sem_destroy(&sem[i]);
    if(ret!=0){
        perror("sem_destroy");

    }
}
```

2.Συγχρονισμός με Condition Variables

Και σε αυτή την περίπτωση δημιουργούμε τα NTREADS με τον τρόπου που περιγράψαμε παραπάνω. Στη συνέχεια χρησιμοποιούμε και πάλι τη συνάρτηση safe_malloc για τη δημιουργία ενός πίνακα από condition variables, έτσι ώστε να δημιουργήσουμε τόσα condition variables όσα είναι και τα νήματα, και τα αρχικοποιουμε.

```
//Create array for condition variables
cond = (pthread_cond_t*)safe_malloc(thrdcnt * sizeof(pthread_cond_t));

/*Initialize condition variables*/
for(int i=1; i<thrdcnt; i++) {
    if(pthread_cond_init(&cond[i], NULL) != 0) {
        perror("pthread_cond_init() error");
        exit(1);
    }
}
```

Για τη χρήση των condition variables ορίζουμε τη μεταβλητή **myturn**, η οποία πρέπει κάθε φορά να είναι ίση με τον αριθμό του thread για να ικανοποιείται η αντίτσοιχη συνθήκη και να γίνει unblock το thread. Η μεταβλητή myturn αρχικοποιείται στο 0, με αποτέλεσμα να ικανοποιείται η συνθήκη για το πρώτο thread, αυτό να γίνεται unblock, να σχεδιάζει το line 0 του Mandelbrot και στη συνέχει να αυξάνει την τιμή του myturn και να κάνει signal στο condition variable του επόμενου thread. Συνολικά, κάθε thread κάνει lock στο mutex που αντιστοιχεί στο αντίτσοιχο του condition variable και στην περίπτωση που δεν ισχύει η συνθήκη που περιγράψαμε παραπάνω, κάνει unlock στο mutex και περιμένει να δεχθεί signal για το condition variable. Όσο περιμένει η λειτουργία του νήματος αναστέλλεται, με αποτέλεσμα να μην καταναλώνονται πόροι. Μόλις δεχθεί το σήμα και ισχύει η συνθήκη, κάνει πάλι lock στο mutex και ακολουθεί την διαδικασία που περιγράψαμε για το αρχικό νήμα. Σημειώνουμε ότι και εδώ το επόμενο thread προκύπτει κυκλικά.

Η υλοποίηση της **compute_and_output_mandel_line** στην οποία επιτυγχάνεται ο συγχρονισμός που περιγράψαμε παραπάνω φαίνεται παρακάτω και συνοδεύεται από τα απαιτητά σχόλια για την επεξήγησή της.

```
void *compute_and_output_mandel_line(void *thread_num)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars], ret;

    /*Loop that iteratres over lines for each thread*/
    for(int line=(int)thread_num; line<=y_chars; line+=thrdcnt) {

        compute_mandel_line(line, color_val);

        /*Use mutex to lock*/
        pthread_mutex_lock(&lock);

        /*Each thread checks if my_turn = thread_num*/
        while(myturn != (int)thread_num)
            pthread_cond_wait(&cond[(int)thread_num], &lock);
        /* CRITICAL PART START */
        output_mandel_line(1, color_val);
        /* CRITICAL PART END */
        myturn++;
        /*Used for circular function*/
        if(myturn == thrdcnt)
            myturn = 0;
        /*Send signal to next thread*/
        pthread_cond_signal(&cond[((int)thread_num+1)%thrdcnt]);
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

Αφού ολοκληρωθεί ο υπολογισμός και η σχεδίαση του Mandelbrot από τα νήματα, περιμένουμε τα νήματα μα τερματίσουν και στη συνέχεια καταστρέφουμε τα condition variables και το mutex.

```
/* Destroy condition variables*/
for (int i=0; i<thrdcnt; i++){
    if(pthread_cond_destroy(&cond[i]) != 0) {
        perror("pthread_cond_destroy() error");
        exit(2);
    }
}
/*Destroy mutex*/
ret = pthread_mutex_destroy(&lock);
if(ret) {
    perror_pthread(ret, "mutex_destroy");
    exit(1);
}
```

Ερώτημα 2.1

Όπως περιγράψαμε και παραπάνω, χρησιμοποιούμε τόσους σεμαφόρους όσο είναι και τα threads, έτσι ώστε να πετύχουμε το σχήμα συγχρονισμού που αναλύσαμε στο σημείο που περιγράψαμε και την αρχικοποίηση των σεμαφόρων. Έτσι, κάθε thread περιμένει το προηγούμενο να ξεκλειδώσει τον σεμαφόρο που του αντιστοιχεί και στη συνέχεια αφού τυπώσει την αντίστοιχη γραμμή, κάνει wait (κλειδώνει) στον σεμαφόρο που του αντιστοιχεί και κάνει post (ξεκλειδώνει) τον σεμαφόρο που αντιστοιχεί στο επόμενο, κυκλικά, thread. Αξίζει να σημειώσουμε πως στο κρίσιμο τμήμα βρίσκεται μόνο η εκτύπωση της αντίστοιχης γραμμής του Mandelbrot και όχι ο υπολογισμός της. Με τον τρόπο αυτό, επιτυγχάνουμε να γίνεται παράλληλα ο υπολογισμός της κάθε γραμμής και σειριακά η σχεδίαση, διασφαλίζοντας έτσι την εκτύπωση των γραμμών με σωστή σειρά και σε λιγότερο χρόνο.

Ερώτημα 2.2

Χρησιμοποιώντας την εντολή **cat /proc/cpuinfo** βλέπουμε πως ο server που χρησιμοποιούμε έχει 6 cores, οπότε συνεχίζουμε με τις μετρήσεις χρόνου που αναλύονται παρακάτω.

- Σειριακός Υπολογισμός (1 νήμα):

```
real    0m1.725s
user    0m1.693s
sys     0m0.028s
```

- Παράλληλος Υπολογισμός με Semaphores (2 νήματα):

```
real    0m0.868s
user    0m1.654s
sys     0m0.052s
```

- Παράλληλος Υπολογισμός με Condition Variables (2 νήματα):

```
real    0m0.869s
user    0m1.679s
sys     0m0.037s
```

Παρατηρούμε πως ο χρόνος που απαιτείται για την ολοκλήρωση του σειριακού προγράμματος είναι μεγαλύτερος από τον αντίστοιχο χρόνο του παράλληλου προγράμματος με 2 νήματα (και με τους δύο τρόπους συγχρονισμού). Το γεγονός αυτό ήταν αναμενόμενο, καθώς στην περίπτωση των παράλληλων προγραμμάτων υπολογίζονται παράλληλα οι γραμμές του Mandelbrot που αντιστοιχούν σε κάθε thread και μειώνεται έτσι ο συνολικός χρόνος εκτέλεσης. Παράλληλα, παρατηρούμε πως τα δύο σχήματα συγχρονισμού παρουσιάζουν ελάχιστες αποκλίσεις στον χρόνο εκτέλεσης τους. Αξίζει να σημειωθεί πως αυξάνοντας τον αριθμό τον threads πετυχαίνουμε και μικρότερο χρόνο, καθώς υπολογίζονται παράλληλα περισσότερες γραμμές. Παρ'όλα αυτά αν αυξήσουμε τα threads περισσότερο από τον αριθμό των πηρύνων του μηχανήματος θα παρατηρούμε ελάχιστες αυξομοιώσεις στον χρόνο εκτέλεσης, καθώς δε θα μπορούν να εκτελούνται πραγματικά παράλληλα όλα τα threads.

Ερώτημα 2.3

Στη δεύτερη εκδοχή του προγράμματος χρησιμοποιήσαμε τόσα condition variables όσα και τα νήματα, με αποτέλεσμα να έχουμε το σχήμα συγχρονισμού που περιγράψαμε πιο πάνω. Αν χρησιμοποιήσουμε μια μεταβλητή θα πρέπει να τροποποιήσουμε το σχήμα συγχρονισμού έτσι ώστε πλέον κάθε thread να μην κάνει signal στο condition variable του επόμενου thread, αλλά να κάνει **broadcast**, έτσι ώστε να "ξυπνάνε" όλα τα νήματα και να γίνεται τελικά unblock μόνο το νήμα για το οποίο ισχύει η αναγκαία συνθήκη. Η τροποποιημένη εκδοχή του προγράμματος παρουσιάζεται παρακάτω.

```
void *compute_and_output_mandel_line(void *thread_num)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    int color_val[x_chars], ret;

    /*Loop that iterates over necessary lines for each thread*/

    for(int line=(int)thread_num; line<=y_chars; line+=thrdcnt) {

        compute_mandel_line(line, color_val);
        /*Lock mutex*/
        pthread_mutex_lock(&lock);

        /*Each thread waits for its turn*/
        while(myturn != (int)thread_num)
            pthread_cond_wait(&cond[0], &lock);
        /* CRITICAL PART START */
        output_mandel_line(1, color_val);
        /* CRITICAL PART END */
        myturn++;
        if(myturn == thrdcnt)
            myturn = 0;
        /*Use pthread_cond_broadcast to signal all threads*/
        pthread_cond_broadcast(&cond[0]);
        pthread_mutex_unlock(&lock);

    }
    return NULL;
}
```

Στην περίπτωση της μιας μεταβλητής γίνεται, όπως αναφέραμε, broadcast σε όλα τα νήματα, αντί για signal στο ακριβώς επόμενο. Αυτό έχει ως αποτέλεσμα να ενεργοποιούνται και να αδρανοποιούνται πολύ πιο συνχά οι διεργασίες και να επεμβαίνει έτσι περισσότερο το λειτουργικό, οδηγόντας έτσι στην αυξημένη κατανάλωση υπολογιστικών πόρων και στην καθυστέρηση της ολοκλήρωσης της εκτέλεσης του προγράμματος σε σχέση με τη χρήση πολλών condition variables.

Ερώτημα 2.4

Τα παράλληλα προγράμματα που φτιάξαμε παρουσιάζουν και τα δύο επιτάχυνση, καθώς έχει υλοποιηθεί σωστά το σχήμα συγχρονισμού και το κρίσιμο τμήμα περιέχει μόνο τις απαραίτητες ενέργειες, δηλαδή την εμφάνιση της γραμμής, έτσι ώστε να γίνει η εκτύπωση με τη σειρά και να παραχθεί το σωστό αποτέλεσμα. Η φάση υπολογισμού της κάθε γραμμής μπορεί να βρίσκεται έξω από το κρίσιμο τμήμα, καθώς δεν είναι αναγκαίος ο σειριακός υπολογίσιμος της από τα διαφορετικά νήματα. Στην περίπτωση που είχαμε συμπεριλάβει στο κρίσιμο τμήμα τη φάση υπολογισμού, δε θα παρατηρούσαμε μεγάλες διαφορές στους χρόνους εκτέλεσης του παράλληλου και του σειριακού προγράμματος. Αυτό θα συνέβαινε γιατί ο υπολογισμός της κάθε γραμμής δε θα γινόταν παράλληλα για κάθε thread αλλά σειριακά, με αποτέλεσμα να μην εκμεταλλευόμαστε σωστά τη δυνατότητα των νημάτων για παράλληλη επεξεργασία.

Ερώτημα 2.5

Στην περίπτωση που πατήσουμε Ctrl-C ενώ το πρόγραμμα εκτελείται, το τερματικό μένει σε λάθος χρώμα. Για να εξασφαλίσουμε την επαναφορά του τερματικού στην αρχική του κατάσταση ακόμα και όταν συμβεί το παραπάνω θα προσθέσουμε στο πρόγραμμά μας έναν signal handler για τον χειρισμό του SIGINT, δηλαδή του Ctrl-C. Για τον χειρισμό του SIGINT θα χρησιμοποιήσουμε τη sigaction, ενώ στο signal handler θα χρησιμοποιείται η εντολή reset για την επαναφορά του terminal και στη συνέχεια θα τερματίζει το πρόγραμμα με τη χρήση του system call exit().

```
struct sigaction sa;
        sigset_t sigset;
        sa.sa_handler = sighandler;
        sa.sa_flags = SA_RESTART;
        sa.sa_mask = sigset;

        /*Call sigaction*/
        if(sigaction(SIGINT, &sa, NULL) < 0) {
                perror("sigaction");
                exit(1);
        }

/*Signal Handler*/
void sighandler(int signum) {
        reset_xterm_color(1);
        exit(1);
}
```