



## 2η Εργαστηριακή Αναφορά

### Μέλη Ομάδας:

Σοφία Σάββα ΑΜ:03121189

Ιωάννης Πολυχρονόπουλος ΑΜ:03121089

## 1. Συγχρονισμός σε υπάρχοντα κώδικα

Σκοπός αυτής της άσκησης είναι η αποφυγή race condition πάνω στη μεταβλητή val, όπου ένα νήμα αυξάνει την τιμή της κατά N φορές και το άλλο νήμα την μειώνει κατά N. Σκοπός είναι στο τέλος του προγράμματος η μεταβλητή να είναι ίση με 0. Αυτό επιτυγχάνεται με το συγχρονισμό των δυο νημάτων είτε με ατομικές λειτουργίες είτε με αμοιβαίους αποκλεισμούς (mutual exclusion – mutex)

### Ζητούμενα:

- Χρησιμοποιήστε το παρεχόμενο Makefile για να μεταγλωττίσετε και να τρέξετε το πρόγραμμα. Τι παρατηρείτε; Γιατί;

Κατά τη μεταγλώττιση του ίδιου πηγαίου αρχείου κώδικα, παρατηρούμε ότι μπορούν να παραχθούν δύο εκτελέσιμα αρχεία (simplesync-atomic, simplesync-mutex). Κάθε εκτελέσιμο χρησιμοποιεί διαφορετική τεχνική συγχρονισμού, συνεπώς αναλόγως ποια μέθοδο θέλουμε να αξιοποιήσουμε παράγουμε και το αντίστοιχο εκτελέσιμο.

- Μελετήστε πώς παράγονται δύο διαφορετικά εκτελέσιμα simplesync-atomic, simplesync-mutex, από το ίδιο αρχείο πηγαίου κώδικα simplesync.c

Αν παρατηρήσουμε την μεταγλώττιση που πραγματοποιούμε στο Makefile, βλέπουμε πως για κάθε εκτελέσιμο ορίζουμε μέσω του gcc compiler έναν προκαθορισμένο μακροορισμό (preprocessor macro) και αναλόγως ποιο macro έχουμε ορίσει θα εκτελεστεί και η αντίστοιχη μέθοδος συγχρονισμού στο πηγαίο κώδικά μας. Το macro ορίζεται μέσω της σημαίας -D και η συντακτική μορφή της είναι -D<NAME>=<VALUE>. Παρακάτω δίνεται το υπεύθυνο κομμάτι του Makefile για την δημιουργία των δύο διαφορετικών εκτελέσιμων (αρχικά Object files και έπειτα εκτελέσιμα).

```
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
```

Στον πηγαίο κώδικα ορίζουμε ποια μέθοδο συγχρονισμού θα υλοποιήσουμε ανάλογα με ποιο macro έχει οριστεί και για τον σκοπό αυτό ορίζουμε το macro `USE_ATOMIC_OPS` ίσο με `1` για να χρησιμοποιήσουμε ατομικές λειτουργίες ή `0` για να χρησιμοποιήσουμε mutexes.

```
#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
#error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif
#if defined(SYNC_ATOMIC)
#define USE_ATOMIC_OPS 1
#else
#define USE_ATOMIC_OPS 0
#endif
```

- *Επεκτείνετε τον κώδικα του `simplesync.c` ώστε η εκτέλεση των δύο νημάτων στο εκτελέσιμο `simplesync-mutex` να συγχρονίζεται με χρήση `POSIX mutexes`. Επιβεβαιώστε την ορθή λειτουργία του προγράμματος.*

Στις συναρτήσεις `increase_fn` και `decrease_fn` για να συγχρονίσουμε τα νήματα μέσω mutexes και να αποφύγουμε κάποιο race condition, που θα οδηγούσε σε απρόβλεπτα αποτελέσματα (αναφορικά για την τιμή της μεταβλητής `val`) θα κλειδώσουμε (lock) το αντίστοιχο κρίσιμο τμήμα κώδικα της συνάρτησης (critical section) επιτρέποντας την ατομική εκτέλεσή τους. Τέλος, ξεκλειδώνουμε (unlock) το mutex σε κάθε περίπτωση μόλις υλοποιήσει την αντίστοιχη πράξη (πρόσθεση ή αφαίρεση τιμής της μεταβλητής). Με αυτό τον τρόπο εξασφαλίζουμε ατομικότητα στις διαδικασίες μας.

```
void *increase_fn(void *arg):
    ret = pthread_mutex_lock(&mutex);
    if (ret){
        perror_pthread(ret, "pthread_mutex_lock");
        exit(1);
    }
    ++(*ip);
    ret = pthread_mutex_unlock(&mutex);
    if (ret){
        perror_pthread(ret, "pthread_mutex_unlock");
        exit(1);}
void *decrease_fn(void *arg):
    ret = pthread_mutex_lock(&mutex);
    if (ret){
        perror_pthread(ret, "pthread_mutex_lock");
        exit(1);
    }
    --(*ip);
    ret = pthread_mutex_unlock(&mutex);
    if (ret){
        perror_pthread(ret, "pthread_mutex_unlock");
        exit(1);
    }
```

- Επεκτείνετε τον κώδικα του `simplesync.c` ώστε η εκτέλεση των δύο νημάτων στο εκτελέσιμο `simplesync-atomic` να συγχρονίζεται με χρήση ατομικών λειτουργιών του GCC. Επιβεβαιώστε την ορθή λειτουργία του προγράμματος.

Γενικά, οι ατομικές λειτουργίες ορίζονται στο hardware level και εξάγονται από τον προγραμματιστή μέσω συναρτήσεων, builtins - Built-in functions for atomic memory access ([Atomic Builtins - Using the GNU Compiler Collection \(GCC\)](#)), του μεταγλωττιστή `gcc`. Όλες αυτές οι συναρτήσεις εξασφαλίζουν πως η πρόσβαση και η τροποποίηση της μεταβλητής θα γίνεται ατομικά (όχι από πολλά νήματα ταυτόχρονα). Συνεπώς, για να υλοποιήσουμε τον παραπάνω συγχρονισμό, αλλά με ατομικές λειτουργίες, θα αξιοποιήσουμε δύο συναρτήσεις `__sync_fetch_and_add` και `__sync_fetch_and_sub`.

```
void *increase_fn(void *arg):
__sync_fetch_and_add(ip, 1);
```

```
void *decrease_fn(void *arg):
__sync_fetch_and_sub(ip, 1);
```

Για την εξασφάλιση της ορθής λειτουργίας των δύο μεθόδων συγχρονισμού θα εξετάσουμε την τιμή της μεταβλητής `val`:

```
ok = (val == 0);
printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
return ok;
```

Εάν η τιμή είναι 0, τότε θα εκτυπώσουμε πως όλα είναι εντάξει και η `main` θα επιστρέψει με κωδικό εξόδου 0 (επιτυχία), ενώ αν η μεταβλητή έχει οποιαδήποτε άλλη τιμή, τότε θα εκτυπώσουμε πως δεν είναι όλα εντάξει και η `main` επιστρέφει με έναν «τυχαίο» κωδικό εξόδου πέρα από το 0 που υποδεικνύει αποτυχία.

```
└─$ ./original_simplesync
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -1680610.
```

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

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

## Ερωτήσεις:

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;

<pre>\$ time ./original_simplesync About to increase variable 10000000 times About to decrease variable 10000000 times Done decreasing variable. Done increasing variable. NOT OK, val = -355531.  real    0m0.047s user    0m0.054s sys     0m0.000s</pre>	<pre>\$ 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    0m1.470s user    0m2.032s sys     0m0.815s</pre>	<pre>\$ 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    0m0.213s user    0m0.372s sys     0m0.000s</pre>
---	--	---

Παρατηρούμε πως τα προγράμματα που υλοποιούν συγχρονισμό είναι πιο αργά σε σύγκριση με το πρόγραμμα χωρίς. Αυτό συμβαίνει, καθώς στο πρόγραμμα χωρίς συγχρονισμό η εκτέλεση του κρίσιμου τμήματος και από τα δύο νήματα γίνεται παράλληλα και όχι ατομικά. Αντιθέτως, όταν έχουμε συγχρονισμό νημάτων η εκτέλεση του κρίσιμου τμήματος του κάθε νήματος είναι ατομική, συνεπώς ο σειριακός υπολογισμός κοστίζει.

2. Ποια μέθοδος συγχρονισμού είναι γρηγορότερη, η χρήση ατομικών λειτουργιών ή η χρήση POSIX mutexes; Γιατί;

Από τα παραπάνω μπορούμε να συμπεράνουμε πως οι ατομικές λειτουργίες αποτελούν γρηγορότερη μέθοδο συγχρονισμού. Γενικά, όπως αναφέραμε και σε προηγούμενο ζήτημα, οι ατομικές λειτουργίες λειτουργούν στο hardware επίπεδο και μας επιτρέπουν να αξιοποιούμε hardware locking μηχανισμούς για θεμελιώδης διαδικασίες - [Mutex vs Atomic | CoffeeBeforeArch.github.io](https://CoffeeBeforeArch.github.io) - (πρόσθεση, αφαίρεση, εναλλαγή, κ.λ.π...) που κοστίζουν πολύ λίγο.

Από την άλλη, τα mutexes αποτελούν πιο σύνθετη μέθοδο συγχρονισμού και λειτουργούν σε υψηλότερο επίπεδο. Αναλυτικότερα, η χρήση mutexes εισάγει επιβάρυνση λόγω των λειτουργιών κλειδώματος και ξεκλειδώματος, που μπορεί να περιλαμβάνουν κλήσεις συστήματος και αλλαγές συμφραζομένων (context switches). Υπάρχει επίσης επιβάρυνση από πιθανές καθυστερήσεις όταν τα νήματα μπλοκάρονται περιμένοντας να αποκτήσουν έναν mutex. Παράλληλα, η κλήση των εντολών αυτών αντιστοιχούν σε περισσότερες εντολές σε επίπεδο Assembly, όπως θα δούμε και στα επόμενα δύο ερωτήματα.

Συνοψίζοντας, οι ατομικές λειτουργίες είναι πιο γρήγορη μέθοδος συγχρονισμού για θεμελιώδης διαδικασίες, ενώ τα mutexes είναι πιο αργές αλλά μπορούν να αξιοποιηθούν σε πιο περίπλοκες διαδικασίες.

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 ενδιαμέσο 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`.

Για τις ατομικές λειτουργίες, δηλαδή οι συναρτήσεις `__sync_fetch_and_add(ip, 1)` και `__sync_fetch_and_sub(ip, 1)` βλέπουμε πως αντιστοιχούν με τις εξής εντολές assembly:

```
.loc 1 58 13 view .LVU11          .loc 1 96 13 view .LVU33
lock addl    $1, (%rbx)          lock subl    $1, (%rbx)
```

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.

Αντίστοιχα οι συναρτήσεις κλειδώματος και ξεκλειδώματος των mutexes αντιστοιχούν στις παρακάτω assembly εντολές.

```
.loc 1 63 19 is_stmt 0 view .LVU16      .loc 1 71 19 view .LVU21
movq    %r13, %rdi                     movq    %r13, %rdi
call    pthread_mutex_lock@PLT          .loc 1 71 19 is_stmt 0 view .LVU24
                                           call    pthread_mutex_unlock@PLT

.loc 1 100 19 is_stmt 0 view .LVU62      .loc 1 108 19 view .LVU67
movq    %r13, %rdi                     movq    %r13, %rdi
call    pthread_mutex_lock@PLT          .loc 1 108 19 is_stmt 0 view .LVU70
                                           call    pthread_mutex_unlock@PLT
```

Παρατηρούμε, όπως αναφέραμε και στο δεύτερο ερώτημα, πως τα mutexes είναι πιο περίπλοκη μέθοδος καθώς απαιτούνται παραπάνω εντολές για τον συγχρονισμό των νημάτων. Αναλυτικότερα, για το κλείδωμα και ξεκλείδωμα ενός mutex απαιτούνται 4 εντολές σε αντίθεση με τις ατομικές λειτουργίες που απαιτείται μόνο μία εντολή.

Παρακάτω δίνεται η προσθήκη που τοποθετήσαμε στο Makefile για την σωστή μεταγλώττιση του `simplesync.c` σε assembly files.

```
## Assembly simplesync
assembly-atomic.asm: simplesync.c
    $(CC) $(CFLAGS) -DSYNC_ATOMIC -S -g -o assembly-atomic.asm simplesync.c
assembly-mutex.asm: simplesync.c
    $(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -o assembly-mutex.asm simplesync.c
```

**Σημείωση:** Οι κώδικές μας διαχειρίζονται όλα τα πιθανά errors που μπορούν να προκύψουν (error handling). Την φιλοσοφία αυτή την χρησιμοποιούμε και στην επόμενη άσκηση. Επίσης, όλοι οι κώδικες δίνονται ξεχωριστά στα αντίστοιχα source files.

## 2. Παράλληλος υπολογισμός του συνόλου Mandelbrot

Σκοπός της άσκησης είναι να χρησιμοποιήσουμε πολυνηματισμό ( $n$  νήματα) για να υπολογίσουμε και να εκτυπώσουμε ταχύτερα το σύνολο Mandelbrot χρησιμοποιώντας από την μία σηματοδότες και από την άλλη μεταβλητές συνθήκες για να συγχρονίσουμε τα νήματα. Αναλυτικότερα, αποσκοπούμε το νήμα  $i$  να αναλαμβάνει τον υπολογισμό και την εκτύπωση των σειρών  $i, i + n, i + 2n, i + 3n, \dots$ . Ο λόγος που θέλουμε να συγχρονίζονται τα νήματα είναι για να εκτυπωθούν οι σειρές με την σωστή σειρά, ώστε να παραχθεί το σχήμα κατάλληλα. Η εκτύπωση μιας γραμμής πραγματοποιείται με την συνάρτηση `output_mandel_line` και θα το ορίσουμε ως το κρίσιμο τμήμα και στις δυο μεθόδους συγχρονισμού, δηλαδή η συνάρτηση αυτή πρέπει να εκτελείται ατομικά και κυκλικά (με την σωστή σειρά) από τα  $n$  νήματα.

Ωστόσο, ο υπολογισμός κάθε γραμμής, που υλοποιείται με την συνάρτηση `compute_mandel_line`, δεν είναι απαραίτητο να γίνεται ατομικά και επομένως δεν απαιτείται να βρίσκεται στο κρίσιμο τμήμα. Έτσι, μπορούμε να υπολογίζουμε τις γραμμές παράλληλα και να τις εκτυπώνουμε ατομικά. Η υλοποίηση έχει επεξηγηθεί αναλυτικά σε σχόλια και στους δύο κώδικες (`mandel_semaphore.c`, `mandel_condition.c`).

### Ερωτήσεις:

1. Πόσοι σηματοδότες χρειάζονται για το σχήμα συγχρονισμού που υλοποιείτε;

Για τον επιτυχή συγχρονισμό θα χρησιμοποιήσουμε τόσους σηματοδότες όσο και νήματα, εφόσον πρέπει να εκτελείται ατομικά σε κάθε νήμα η συνάρτηση `output_mandel_line`. Για να το πετύχουμε αυτό θα ορίσουμε αρχικά έναν σηματοδότη με τιμή 1 (unlocked) για το νήμα '0' και  $n - 1$  ακόμα σηματοδότες με τιμή 0 (locked) για τα υπόλοιπα. Αρχικά, εκτελείται το κρίσιμο τμήμα του νήματος '0' μετά του νήματος '1', του '2'..., του ' $n - 1$ ' και μετά πάλι του '0'. Επομένως, αφού εκτυπώσουμε την γραμμή '0' θα ξεκλειδώσουμε (δηλαδή θα δώσουμε την τιμή ένα) στον επόμενο σηματοδότη, και κλειδώνουμε τον τωρινό, κυκλικά μέχρι να εκτυπωθούν, σε σωστή σειρά, όλες οι γραμμές.

Κλειδώμα: `sem_wait(&semaphores[(int)thread_id_ptr])`

Ξεκλειδώμα: `sem_post(&semaphores[((int)thread_id_ptr + 1) % NTHREADS])`

Συνεπώς θα κατασκευάσουμε δυναμικά έναν πίνακα από `semaphores` με `NTHREADS` στοιχεία.  
(Υλοποίηση στο αρχείο `mandel_semaphore.c`)

```
sem_t *semaphores; // Array of semaphores
```

```
...
```

```
// Allocate memory for semaphores
```

```
semaphores = safe_malloc(NTHREADS * sizeof(sem_t));
```

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.

Αρχικά, το σύστημά μας ικανοποιεί την συνθήκη του να έχουμε δύο πυρήνες.

```
$ cat /proc/cpuinfo |grep "cpu cores"
cpu cores      : 2
cpu cores      : 2
cpu cores      : 2
cpu cores      : 2
```

Θα επιβλέψουμε, για κάθε μέθοδο συγχρονισμού, την απόδοση με 1, 2, 4, 8 και 16 νήματα και θα σχολιάσουμε τα αποτελέσματα χρησιμοποιώντας την εντολή `time`.

Original mandel

Semaphore mandel

Conditional mandel:

1

```
real    0m1.807s
user    0m1.058s
sys     0m0.031s
```

```
real    0m1.478s
user    0m0.956s
sys     0m0.050s
```

```
real    0m1.762s
user    0m0.993s
sys     0m0.050s
```

2

```
real    0m1.807s
user    0m1.058s
sys     0m0.031s
```

```
real    0m0.619s
user    0m1.088s
sys     0m0.051s
```

```
real    0m0.604s
user    0m1.074s
sys     0m0.039s
```

4

```
real    0m1.807s
user    0m1.058s
sys     0m0.031s
```

```
real    0m0.522s
user    0m1.375s
sys     0m0.229s
```

```
real    0m0.582s
user    0m1.267s
sys     0m0.378s
```

8

```
real    0m1.807s
user    0m1.058s
sys     0m0.031s
```

```
real    0m0.469s
user    0m1.211s
sys     0m0.250s
```

```
real    0m0.464s
user    0m1.321s
sys     0m0.179s
```

16

```
real    0m1.807s
user    0m1.058s
sys     0m0.031s
```

```
real    0m0.421s
user    0m1.286s
sys     0m0.115s
```

```
real    0m0.424s
user    0m1.199s
sys     0m0.150s
```



Από τις παραπάνω μετρήσεις παρατηρούμε πως ο χρόνος εκτέλεσης του παράλληλου προγράμματος με δύο νήματα είναι ο μισός περίπου σε σύγκριση με τον χρόνο εκτέλεσης του σειριακού προγράμματος. Αυτό συμβαίνει καθώς ο υπολογισμός των γραμμών διαμοιράζεται στους δυο πυρήνες, όπου εκτελούν το αντίστοιχο νήμα και οι υπολογισμοί γίνονται παράλληλα και όχι σειριακά.

Ακόμα, μπορούμε να παρατηρήσουμε πως το πρόγραμμά μας, και στις δυο περιπτώσεις συγχρονισμού, βελτιώνεται με τον διπλασιασμό των νημάτων, αλλά όλο και λιγότερο καθώς η κατανομή των νημάτων στους πυρήνες και ο διαμοιρασμός των υπολογισμών από ένα σημείο και μετά δεν θα επηρεαστεί ιδιαίτερω.

Ωστόσο, εάν αυξήσουμε υπερβολικά τον αριθμό των νημάτων θα παρατηρήσουμε πως η απόδοση των προγραμμάτων μειώνεται και ο χρόνος εκτέλεσης αυξάνεται. Αυτό οφείλεται στο γεγονός πως η δημιουργία νημάτων και η αναμονή στο main νήμα για τον τερματισμό τους κοστίζουν.

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

Στο πρόγραμμά μας χρησιμοποιήθηκε μία μεταβλητή συνθήκη:

```
pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER;
```

Ο συγχρονισμός στην περίπτωση αυτή θα λειτουργεί ως εξής:

- Αρχικά, ορίζουμε ένα global variable (`current_line`) που θα συμβάλλει στη συνθήκη για το εάν το νήμα θα εκτελέσει το κρίσιμο τμήμα του (δηλαδή να εκτυπώσει την γραμμή που του αντιστοιχεί και να δηλώσει πως πάμε στην επόμενη γραμμή (`current_line++`)).
- Εάν βρισκόμαστε σε νήμα που δεν είναι η σειρά του να εκτυπώσει (`i != current_line`) τότε καλούμε την `pthread_cond_wait`. Έτσι, «μπλοκάρουμε» όλα τα νήματα που δεν είναι η σειρά τους να εκτυπώσουν την αντίστοιχη γραμμή και περιμένουν για κάποιο σήμα πάνω στη μεταβλητή συνθήκη, ώστε να συνεχίσουν και να ελέγξουν εάν η συνθήκη `i != current_line` ικανοποιείται ή όχι (δηλαδή θα ξαναπεριμένει ή θα περάσει στο κρίσιμο τμήμα)
- Σε επόμενο βήμα εάν βρισκόμαστε σε νήμα που δεν ικανοποιεί την συνθήκη, σημαίνει πως δεν θα περιμένει («μπλοκάρει») και θα συνεχίσει στην εκτέλεση του υπόλοιπου κώδικα. Συγκεκριμένα θα εκτυπώσει την γραμμή που του αντιστοιχεί και θα δηλώσει πως περνάμε στην επόμενη γραμμή αυξάνοντας την τιμή της μεταβλητής `current_line`. Τέλος, θα καλέσουμε την συνάρτηση `pthread_cond_broadcast` για να «ξεμπλοκάρουμε» όλα τα νήματα που περιμένουν στο condition variable για να συνεχίσουν και να ελέγξουν εάν είναι η σειρά τους να εκτυπώσουν την γραμμή τους.



Συνεπώς, κάθε φορά θα εκτυπώνει το νήμα που η τωρινή γραμμή συμβαδίζει με την γραμμή που υπολόγισε. Η υλοποίηση επεξηγείται στον πηγαίο κώδικα **mandel\_condition.c** με σχόλια.

Το πρόβλημα επίδοσης που υπάρχει εμφανίζεται όταν έχουμε πολλά νήματα. Αναλυτικότερα, όταν υλοποιούμε το broadcast για να στείλουμε σήμα σε όλα τα «μπλοκαρισμένα» νήματα για να συνεχίσουν (πλην το τωρινό), θα στείλουμε σήμα σε πολλαπλά νήματα, διαδικασία που σίγουρα θα κοστίζει.

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;

Όπως είδαμε και στο Ερώτημα 2 το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση. Αυτό συμβαίνει, καθώς ο υπολογισμός των γραμμών γίνεται παράλληλα και όχι ατομικά.

Στην περίπτωση που επεκτείναμε το κρίσιμο τμήμα μας και προσθέταμε και τον υπολογισμό των γραμμών τότε δεν θα παρατηρούσαμε επιτάχυνση αφού ο υπολογισμός θα ήταν σειριακός, όπως και στο πρόγραμμα χωρίς πολυνηματισμό και συγχρονισμό. Αντιθέτως, μπορεί να είχαμε μέχρι και μεγαλύτερο χρόνο εκτέλεσης καθώς η δημιουργία νημάτων και ο συγχρονισμός τους κοστίζουν όπως έχουμε αναφέρει και σε προηγούμενα ερωτήματα.

Ανακεφαλαιώνοντας, το παράλληλο πρόγραμμα (και με τις δύο μεθόδους συγχρονισμού) εμφανίζει επιτάχυνση καθώς το **output\_mandel\_line** βρίσκεται στο κρίσιμο τμήμα, ενώ το **compute\_mandel\_line** γίνεται παράλληλα αποφεύγοντας σειριακό υπολογισμό. Εάν τοποθετούσαμε στο κρίσιμο τμήμα και το **compute\_mandel\_line** τότε δεν θα εμφάνιζε επιτάχυνση το παράλληλο πρόγραμμα.

### Χρησιμοποιώντας σημαφόρους:

```
compute_mandel_line(i, color_val); // Parallel computation
if (sem_wait(&semaphores[(int)thread_id_ptr]))
{
    perror("sem_wait");
    exit(1);
}
output_mandel_line(1, color_val); // Critical section
if (sem_post(&semaphores[((int)thread_id_ptr + 1) % NTHREADS]))
{
    perror("sem_post");
    exit(1);
}
```

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;

[illegible]

Συνεπώς, προτού επιστρέψει και τερματιστεί το πρόγραμμα πρέπει να εκτελέσουμε την συνάρτηση `reset_xterm_color`. Παρακάτω φαίνεται η συνάρτηση signal handler που θα εκτελεστεί όταν πατηθεί Ctrl + C στη μέση του προγράμματός μας και θα επαναφέρει το σωστό χρώμα στο τερματικό μας.

*$M\varepsilon$  signal handler:*

```
oslab019@os-node2:~/2nd_lab_report$ ./mandel_semaphore 3
```

The visualization consists of a large rectangular area filled with small squares. Each square's color represents the number of iterations required for a point in the complex plane to escape a certain radius during the Mandelbrot iteration process. The colors are primarily black, white, and gray, with some scattered blue, green, and red squares, particularly along the boundaries of the fractal structures. The overall pattern shows the characteristic self-similar, fractal-like boundaries of the Mandelbrot set.

```
^Coslab019@os-node2:~/2nd_lab_report$ |
```