

Λειτουργικά Συστήματα Υπολογιστών

2^η Εργαστηριακή Άσκηση

Συγχρονισμός

Συντάκτες: Νικόλαος Λάμπας 03121098
Γεώργιος Πνευματικός 03121058

Άσκηση 1: Συγχρονισμός σε Υπάρχοντα Κώδικα *simplesync_atomic_mutex*

Δίνεται το πρόγραμμα simplesync.c (και το Makefile του) το οποίο λειτουργεί με τον εξής τρόπο:

Αρχικοποιεί μια μεταβλητή `int val` στο 0 και έπειτα δημιουργεί δύο νήματα (T_{increase} και T_{decrease}) τα οποία εκτελούνται ταυτόχρονα όπου το T_{increase} αυξάνει την μεταβλητή `val` κατά 1, N φορές, ενώ το T_{decrease} ελαττώνει την μεταβλητή `val` κατά 1, N φορές. Αυτό το οποίο θέλουμε να πετύχουμε είναι η μεταβλητή `val` μετά το πέρας των δύο νημάτων να έχει την τιμή 0.

Θα χρησιμοποιήσουμε το Makefile για να μεταγλωττίσουμε τον κώδικα

```
oslab044@os-node1:~/second_ex$ make simplesync
gcc -Wall -O2 -pthread simplesync.c -o simplesync
simplesync.c:30:3: error: You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
  30 | # error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
      | ^~~~~~
make: *** [<builtin>: simplesync] Error 1
oslab044@os-node1:~/second_ex$
```

Παρατηρούμε πως αν επιχειρήσουμε να μεταγλωττίσουμε το αρχείο χωρίς να προσδιορίσουμε αν θα είναι `SYNC_ATOMIC` ή `SYNC_MUTEX` τότε λαμβάνουμε σφάλμα κατά την μεταγλώττιση. Πρέπει λοιπόν να προσδιορίσουμε αν το εκτελέσιμο θα είναι `atomic` ή `mutex`.

```
#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
```

Η ρύθμιση αυτή εντός του αρχείου `simplesync.c` επιτρέπει σε μέρη του κώδικα να μεταγλωττιστούν ανάλογα με το εάν πρέπει να χρησιμοποιηθούν ατομικές λειτουργίες ή `mutexes` για την επίτευξη του συγχρονισμού, βάσει των `SYNC_ATOMIC` ή `SYNC_MUTEX`.

Επιχειρούμε επομένως να τρέξουμε τα δύο εκτελέσιμα `simplesync-atomic` και `simplesync-mutex`

```
[oslab044@os-node1:~/second_ex$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 1166147.
[oslab044@os-node1:~/second_ex$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -29350.
oslab044@os-node1:~/second_ex$
```

Παρατηρούμε πώς αν τρέξουμε τα δύο αυτά εκτελέσιμα δεν λαμβάνουμε το επιθυμητό αποτέλεσμα καθώς και στις δύο περιπτώσεις η μεταβλητή `val` στο τέλος λαμβάνει τιμή διαφορετική του μηδενός. Αυτό συμβαίνει γιατί τα δύο νήματα δεν συγχρονίζουν την εκτέλεσή τους. Πιο αναλυτικά, σε επίπεδο εντολών `assembly` στο οποίο μεταφράζεται ο κώδικας, η αύξηση και μείωση μιας μεταβλητής αντιστοιχούν σε πολλές εντολές οι οποίες αν δεν συγχρονιστούν (όπως ακριβώς συμβαίνει στην περίπτωσηή μας) οδηγούν σε λανθασμένα μη επιθυμητά αποτελέσματα (`race conditions`).

Για να επιλύσουμε λοιπόν αυτό το πρόβλημα συγχρονισμού θα ενσωματώσουμε στον κώδικά μας κρίσιμα τμήματα (`critical sections`) τα οποία αποτελούν τμήματα κώδικα στα οποία μπορεί να βρίσκεται το πολύ μια διεργασία (μια διεργασία ή καμία) με αποτέλεσμα να επιτρέπουν την ατομική εκτέλεση τμημάτων του κώδικα. Στην περίπτωσηή μας, θα επιτύχουμε την ατομική εκτέλεση των τμημάτων κώδικα με ατομικές εντολές (`atomic operations`) και κλειδώματα αμοιβαίου αποκλεισμού (`mutexes`).

Επεκτείνουμε, λοιπόν, τον δοθέντα κώδικα ώστε να επιτύχουμε τον συγχρονισμό των δύο νημάτων και εντέλει η τιμή της μεταβλητής `val` να έχει την επιθυμητή μηδενική τιμή. Θα χρησιμοποιήσουμε, όπως ακριβώς αναφέραμε και παραπάνω, `atomic operations` και `mutexes` για την ατομική εκτέλεση των τμημάτων. Επισημαίνουμε, ότι οι ατομικές εντολές προσφέρονται από τον GCC μεταγλωττιστή και χρησιμοποιούνται για την ασφαλή μεταβολή κοινών πόρων μεταξύ διεργασιών. Δηλαδή, μέσω των ατομικών εντολών τα κρίσιμα τμήματα του κώδικα εκτελούνται ατομικά για κάθε διεργασία και χωρίς διακοπές με αποτέλεσμα να αποφεύγονται τα `race conditions`.

Παράλληλα, με τα `mutexes` όπως θα δούμε και στον κώδικά μας εφαρμόζουμε ένα κλείδωμα ώστε μόνο η μια διεργασία να έχει πρόσβαση στο κρίσιμο τμήμα του κώδικα και να μπορεί

να προκαλέσει μεταβολή στον κοινό πόρο. Το άλλο νήμα θα αποκτήσει πρόσβαση μόνο όταν βρει ξεκλείδωτο το κλείδωμα, δηλαδή όταν δεν υπάρχει άλλη διεργασία στο κρίσιμο τμήμα

Παραθέτουμε τις μεταβολές στον κώδικα

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *increase_fn(void *arg)
{
    int i, ret;
    volatile int *ip = arg;
    fprintf(stderr, "About to increase variable %d times\n", N);
    for (i = 0; i < N; i++) {
        if (USE_ATOMIC_OPS) {
            /* You can modify the following line */
            __sync_add_and_fetch(&ip, 1);
        } else {
            /* You cannot modify the following line */
            ret = pthread_mutex_lock(&lock);
            if (ret)
                perror_pthread(ret, "lock");
            ++(*ip);
            pthread_mutex_unlock(&lock);
            if (ret)
                perror_pthread(ret, "unlock");
        }
    }
    fprintf(stderr, "Done increasing variable.\n");
    return NULL;
}

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

    fprintf(stderr, "About to decrease variable %d times\n", N);
    for (i = 0; i < N; i++) {
        if (USE_ATOMIC_OPS) {
            /* You can modify the following line */
            __sync_sub_and_fetch(&ip, 1);
        } else {
            ret = pthread_mutex_lock(&lock);
            if (ret)
                perror_pthread(ret, "lock");
            /* You cannot modify the following line */
            --(*ip);
            ret = pthread_mutex_unlock(&lock);
            if (ret)
                perror_pthread(ret, "unlock");
        }
    }
    fprintf(stderr, "Done decreasing variable.\n");
    return NULL;
}
```

Επίσης στο τέλος της κύριας συνάρτησης του προγράμματος, καταστρέφουμε και το κλείδωμα που χρησιμοποιήσαμε για το mutex.

```
ret = pthread_mutex_destroy(&lock);  
if (ret)  
    perror_thread(ret, "lock destroy");
```

Όπως ακριβώς φαίνεται και στον συγκεκριμένο κώδικα, για την υλοποίηση του συγχρονισμού στην περίπτωση της μεταγλώττισης κατά atomic operations χρησιμοποιήσαμε τα built in functions της γλώσσας C που λειτουργούν όπως αναφέραμε προηγουμένως.

Συνάμα, για την υλοποίηση του συγχρονισμού με mutex χρησιμοποιήσαμε το κλείδωμα (lock) δίνοντας πρόσβαση το πολύ σε ένα νήμα στο κρίσιμο τμήμα.

Μεταγλωττίζουμε και εκτελούμε τα δύο εκτελέσιμα

```
[oslab044@os-node1:~/second_ex$ make simplesync-atomic  
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c  
gcc -Wall -O2 -pthread -o simplesync-atomic simplesync-atomic.o  
[oslab044@os-node1:~/second_ex$ make simplesync-mutex  
gcc -Wall -O2 -pthread -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c  
gcc -Wall -O2 -pthread -o simplesync-mutex simplesync-mutex.o  
[oslab044@os-node1:~/second_ex$ ./simplesync-atomic  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done decreasing variable.  
Done increasing variable.  
OK, val = 0.  
[oslab044@os-node1:~/second_ex$ ./simplesync-mutex  
About to increase variable 10000000 times  
About to decrease variable 10000000 times  
Done increasing variable.  
Done decreasing variable.  
OK, val = 0.  
oslab044@os-node1:~/second_ex$
```

Παρατηρούμε πώς και στις δύο περιπτώσεις λαμβάνουμε την επιθυμητή τιμή για την μεταβλητή val. Άρα πετύχαμε των συγχρονισμό των δύο νημάτων

Ερωτήσεις

1. Χρησιμοποιούμε την εντολή `time` προκειμένου να μετρήσουμε τον χρόνο εκτέλεσης των εκτελέσιμων προγραμμάτων. Τονίζουμε πώς στα πλαίσια του συγκεκριμένου ερωτήματος κρατήσαμε αμετάβλητο το αρχείο `simplesync.c` σε έναν ξεχωριστό φάκελο και θα το τρέξουμε εκτελώντας:
`time ./simplesync - atomic` χωρίς αυτό φυσικά να περιέχει κώδικα συγχρονισμού.

Εκτέλεση χωρίς συγχρονισμό

```
oslab044@os-node1:~/second_ex/ex_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 = 279950.

real    0m0.156s
user    0m0.257s
sys      0m0.004s
oslab044@os-node1:~/second_ex/ex_1$
```

Εκτέλεση συγχρονισμού με gcc atomic operation

```
oslab044@os-node1:~/second_ex$ 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    0m0.251s
user    0m0.312s
sys      0m0.012s
oslab044@os-node1:~/second_ex$
```

Εκτέλεση συγχρονισμού με POSIX mutexes

```
oslab044@os-node1:~/second_ex$ 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    0m1.424s
user    0m1.534s
sys     0m0.423s
oslab044@os-node1:~/second_ex$
```

Παρατηρούμε πώς ο χρόνος εκτέλεσης των εκτελέσιμων με συγχρονισμό είναι μεγαλύτερος από τον χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό. Αυτό συμβαίνει γιατί στο αρχικό πρόγραμμα δεν υπάρχει ο περιορισμός κάθε νήμα να εκτελεί ατομικά τον κώδικα στο κρίσιμο τμήμα και επομένως η εκτέλεση στο κρίσιμο τμήμα είναι παράλληλη και από τα δύο νήματα. Στην περίπτωση του συγχρονισμού, στο κρίσιμο τμήμα θα βρίσκεται το πολύ ένα από τα δύο νήματα με αποτέλεσμα η εκτέλεση να είναι σειριακή και ο συνολικός χρόνος εκτέλεσης να είναι αυξημένος.

- Μπορούμε παραπάνω να παρατηρήσουμε πώς ο χρόνος εκτέλεσης με gcc atomic operations είναι μικρότερος από τον χρόνο εκτέλεσης με POSIX mutexes. Αυτό συμβαίνει καθώς με POSIX mutexes ενδεχομένως να έχουμε περισσότερες εντολές σε assembly, αλλά κυρίως λόγω του γεγονότος ότι με τα POSIX mutexes εμπλέκεται άμεσα το λειτουργικό σύστημα το οποίο βάζει σε αναμονή και έπειτα ξυπνάει τις διεργασίες όταν πλέον πρέπει να έρθουν σε θέση να διεκδικήσουν το lock για να εισέλθουν στο κρίσιμο τμήμα. Επομένως, με το mutex πέρα των παραπάνω εντολών που απαιτούνται, προστίθεται και ο χρόνος για context switch και δρομολόγηση των διεργασιών.

3. Τροποποιούμε το Makefile για να δημιουργήσουμε τους ενδιαμέσους κώδικες σε assembly για τις δύο λύσεις συγχρονισμού που υλοποιήσαμε.

Παραθέτουμε τις προσθήκες

```
## 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

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

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

Στις δύο τελευταίες γραμμές στο συγκεκριμένο κομμάτι του Makefile παράγουμε τα .s αρχεία. Τα .s αρχεία περιέχουν τον ενδιαμέσο κώδικα assembly και εντός των αρχείων αυτών θα εξετάσουμε σε ποιες εντολές του επεξεργαστή μεταφράζεται η χρήση ατομικών λειτουργιών καθώς και η χρήση των mutexes στο επόμενο ερώτημα.

Παράγουμε λοιπόν τα δύο εκτελέσιμα αρχεία

```
[oslab044@os-node2:~/second_ex$ make simplesync-atomic.s
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c
[oslab044@os-node2:~/second_ex$ make simplesync-mutex.s
gcc -Wall -O2 -pthread -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c
```

Στις δύο φωτογραφίες που ακολουθούν βλέπουμε τις εντολές στις οποίες μεταφράζεται σε κώδικα assembly το simplesync.c με βάση τις gcc atomic operations που εξετάσαμε

- Increase

```
.L2:
    .loc 1 48 3 is_stmt 1 view .LVU11
    .loc 1 50 4 view .LVU12
    lock addq    $1, 8(%rsp)
```

- Decrease

```
.L7:
    .loc 1 74 3 is_stmt 1 view .LVU32
    .loc 1 76 4 view .LVU33
    lock subq    $1, 8(%rsp)
```

Η εντολή lock που βλέπουμε είναι αυτή που προσθέτει το επίπεδο συγχρονισμού στις πράξεις που επηρεάζουν τους κοινούς πόρους των νημάτων.

4. Στην περίπτωση του POSIX mutex θα εξετάσουμε πάλι από το simplesync-mutex.s αρχείο πώς μεταγλωττίζονται σε ενδιαμέσο κώδικα assembly οι εντολές που χρησιμοποιήσαμε για pthread_mutex_lock().

- Increase

```
.L3:
    .loc 1 48 3 is_stmt 1 view .LVU27
    .loc 1 53 4 view .LVU28
    .loc 1 53 10 is_stmt 0 view .LVU29
    movq    %r13, %rdi
    call    pthread_mutex_lock@PLT
```

```
.LVL12:
    .loc 1 55 33 view .LVU36
    .loc 1 56 4 view .LVU37
    .loc 1 56 7 is_stmt 0 view .LVU38
    movl    (%r12), %eax
    .loc 1 57 4 view .LVU39
    movq    %r13, %rdi
    .loc 1 56 4 view .LVU40
    addl    $1, %eax
    movl    %eax, (%r12)
    .loc 1 57 4 is_stmt 1 view .LVU41
    call    pthread_mutex_unlock@PLT
```

- Decrease

```
.L14:
    .loc 1 74 3 is_stmt 1 view .LVU85
    .loc 1 78 4 view .LVU86
    .loc 1 78 10 is_stmt 0 view .LVU87
    movq    %r13, %rdi
    call    pthread_mutex_lock@PLT
```

```
.L12:
    .loc 1 80 5 discriminator 1 view .LVU71
    .loc 1 82 4 discriminator 1 view .LVU72
    .loc 1 82 7 is_stmt 0 discriminator 1 view .LVU73
    movl    (%r12), %eax
    .loc 1 83 10 discriminator 1 view .LVU74
    movq    %r13, %rdi
    .loc 1 82 4 discriminator 1 view .LVU75
    subl    $1, %eax
    movl    %eax, (%r12)
    .loc 1 83 4 is_stmt 1 discriminator 1 view .LVU76
    .loc 1 83 10 is_stmt 0 discriminator 1 view .LVU77
    call    pthread_mutex_unlock@PLT
```


Άσκηση 2: Παράλληλος υπολογισμός του συνόλου Mandelbrot

Στην συγκεκριμένη άσκηση, δίνονται τα αρχεία κώδικα για την υλοποίηση του συνόλου Mandelbrot σειριακά και από εμάς ζητείται η μετατροπή της σειριακής επεξεργασίας που έχουμε στο αρχείο *mandel.c* σε παράλληλη επεξεργασία με δυο διαφορετικούς τρόπους. Ο πρώτος τρόπος ζητάει τον απαραίτητο συγχρονισμό των νημάτων με την χρήση σημαφόρων, ενώ ο δεύτερος τρόπος ζητάει τον απαραίτητο συγχρονισμό νημάτων με την χρήση μεταβλητών κατάστασης. Καλούμαστε, λοιπόν, να επεκτείνουμε τον δοθέντα κώδικα έτσι ώστε ο υπολογισμός να κατανέμεται σε NTHREADS νήματα του POSIX, με την κατανομή του υπολογιστικού φόρτου να γίνεται ανά σειρά και κυκλικά. Ακολουθούν οι επεξηγήσεις για την παράλληλη υλοποίηση του προγράμματος *mandel.c* με χρήση σημαφόρων και με χρήση μεταβλητών κατάστασης.

Στο σημείο παραθέτουμε τις διαφορές του νέου κώδικα που χρησιμοποιεί σημαφόρους για την υλοποίηση του κρίσιμου τμήματος. *mandel_semaphores*

Επιπρόσθετες global μεταβλητές και υλοποίηση σημαφόρων στην συνάρτηση *compute_and_output_mandel_line*:

```
int thrcnt;//Globalizing the total thread count
sem_t *semaphores;//Globalizing sempaphore array

void *compute_and_output_mandel_line(void *arg)
{
    /*
     * A temporary array, used to hold color values for the line being drawn
     */
    struct thread_info_struct *thread = arg;

    int color_val[x_chars];
    for (int i = thread->thread_number; i < y_chars; i += thrcnt){
        //each thread increases it's iterator by thread_count
        //Redisistributing the work in a cyclic manner

        compute_mandel_line(i, color_val);

        int number = thread->thread_number;
        if (sem_wait(&semaphores[number]) != 0){
            perror("sem_wait");
            exit(1);
        }

        output_mandel_line(1, color_val);

        if (sem_post(&semaphores[(number + 1) % thrcnt]) != 0){
            perror("sem_signal");
        }
    }
    return NULL;
}
```

Παραθέτουμε επίσης και τις κύριες διαφορές που υλοποιήσαμε στην κύρια συνάρτηση του προγράμματος μας η οποία δημιουργεί τα νήματα και τους περνάει ως παράμετρο εκτέλεσης στην συνάρτηση που είδαμε ακριβώς παραπάνω:

```
semaphores = (sem_t*)safe_malloc(thrcnt * sizeof(sem_t)); //dynamic memory for semaphore array

if (sem_init(&semaphores[0], 0, 1) != 0) { //initializing first semaphore to one
    perror("sem init");
    exit(1);
}

for (int i = 1; i < thrcnt; i++) { //initializing the rest semaphores to 0
    if (sem_init(&semaphores[i], 0, 0) != 0) {
        perror("sem init");
        exit(1);
    }
}

struct thread_info_struct *thr;
thr = safe_malloc(thrcnt * sizeof(*thr)); //dynamic allocation of memory for array of thread information
int ret;

xstep = (xmax - xmin) / x_chars;
ystep = (ymax - ymin) / y_chars;

/*
 * draw the Mandelbrot Set, one line at a time.
 * Output is sent to file descriptor '1', i.e., standard output.
 */

for (int i = 0; i < thrcnt; i++) {
    thr[i].thread_number = i; //passing the number of the thread on the struct array
    ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
    if (ret) {
        perror("thread create");
        exit(1);
    }
}

for (int i = 0; i < thrcnt; i++) { //waiting for threads
    ret = pthread_join(thr[i].tid, NULL);
    if (ret) {
        perror("thread join");
        exit(1);
    }
}

for (int i = 0; i < thrcnt; i++) {
    if (sem_destroy(&semaphores[i]) != 0) {
        perror("sema destruction");
        exit(1);
    }
}

free (semaphores);
free (thr);
reset_xterm_color(1);
return 0;
```

Η κύρια συνάρτηση καλείται με ένα όρισμα το οποίο φυσικά είναι ο αριθμός των νημάτων που θα εκτελέσουν τον παράλληλο υπολογισμό. Ο αριθμός αυτός των νημάτων τοποθετείται στην μεταβλητή *thrcnt* που είναι global (ώστε να είναι προσβάσιμη και από τα νήματα) μέσω της συνάρτησης *safe_atoi*. Ταυτόχρονα η κατανομή του υπολογιστικού φόρτου γίνεται ανά γραμμές και κυκλικά. Δηλαδή αν έστω έχουμε συνολικά *N* νήματα τότε το *i_{οστο}* νήμα αναλαμβάνει τις γραμμές *i*, *i + N*, *i + 2N*, Θα δούμε εντός του πρώτου ερωτήματος πως εξασφαλίζουμε στην συγκεκριμένη περίπτωση την ορθή σειρά εκτύπωσης των γραμμών και εκτέλεσης των νημάτων.

Στο σημείο παραθέτουμε τις διαφορές του νέου κώδικα που χρησιμοποιεί μεταβλητές κατάστασης για την υλοποίηση του κρίσιμου τμήματος.

mandel_condition_variables_efficient

Επιπρόσθετες global μεταβλητές και υλοποίηση μεταβλητών κατάστασης στην συνάρτηση *compute_and_output_mandel_line*:

```
pthread_mutex_t mutex;
pthread_cond_t *conds;
int next_line = 0;
int thrcnt;

void *compute_and_output_mandel_line(void *arg) {
    struct thread_info_struct *thread = arg;
    int color_val[x_chars];

    for (int i = thread->thread_number; i < y_chars; i += thrcnt) {
        compute_mandel_line(i, color_val); // Compute the line first without locking.

        pthread_mutex_lock(&mutex); // This lock ensures that wait is used correctly

        // Wait until it's this thread's turn to output.
        while (i != next_line) {
            pthread_cond_wait(&conds[thread->thread_number], &mutex);
        }

        // Output the line as it's now this thread's turn.
        output_mandel_line(1, color_val);
        next_line++; // next_line is common for all threads
        pthread_cond_signal(&conds[(thread->thread_number + 1) % thrcnt]); // Wake up only one thread

        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Στον κώδικα με τις μεταβλητές κατάστασης, αυτό που κάνουμε είναι να ελέγχουμε αρχικά για τον σωστό αριθμό από arguments και να μετατρέπουμε την συμβολοσειρά *argv[1]* σε ακέραιο αριθμό, τον αριθμό που εισάγει ο χρήστης και καθορίζει τον αριθμό των νημάτων που θέλει να δημιουργηθούν. Ύστερα δεσμεύουμε δυναμικά την μνήμη τόσο για το struct που θα περιέχει τα στοιχεία κάθε νήματος, όσο και για τον δείκτη που αφορά τις μεταβλητές κατάστασης. Κατασκευάζουμε τόσα νήματα όσα θέλει ο χρήστης και άλλες τόσες μεταβλητές κατάστασης, ελέγχοντας συγχρόνως για πιθανό σφάλμα. Με κάθε κατασκευή ενός νήματος, το καθένα από αυτά ξεχωριστά, εξ ου το *&thr[i].tid*, καλεί την συνάρτηση *compute_and_output_mandel_line* βάζοντας της ως όρισμα το *&thr[i]*, το οποίο περνιέται ως δείκτης προς την δομή του νήματος που θα εκτελέσει την συνάρτηση. Η συνάρτηση έτσι που θα κληθεί, μέσω του ορίσματος *void * arg*, μετατρέπει την διεύθυνση μνήμης *&thr[i]* σε έναν δείκτη *void** και έχει πλέον η συνάρτηση πρόσβαση στα δεδομένα της δομής του νήματος *thr[i]*.

Στην συνάρτηση αυτή, όπως φαίνεται και στην προηγούμενη εικόνα, αυτό που γίνεται είναι για όσο ισχύει η συνθήκη *for (int i = thread->thread_number; i < y_chars; i += thrcnt)* κάθε νήμα αναλαμβάνει κυκλικά μια γραμμή και υπολογίζει το κατάλληλο χρώμα για κάθε χαρακτήρα κάνοντας χρήση της συνάρτησης *compute_mandel_line*. Να επισημάνουμε ότι το *i += thrcnt* καθορίζει την κυκλική ανάθεση γραμμών σε κάθε νήμα, αφού πρακτικά λέει στο εκάστοτε νήμα θα σου ανατίθενται οι γραμμές *i += thrcnt*. Στην συνέχεια υπάρχει ένα κλείδωμα με *mutex*, το οποίο καθορίζει την έναρξη του κρίσιμου

τμήματος και είναι υπεύθυνο για την ορθή χρήση της συνθήκης αναμονής. Το πρώτο νήμα που θα εισέλθει το βρίσκει ανοιχτό και μόλις μπει κλειδώνει ώστε κανένα άλλο να μην εισέλθει παράλληλα με αυτό στο κρίσιμο τμήμα. Το νήμα περιμένει την συνθήκη αναμονής μέχρις ότου είναι η σειρά του να εκτελέσει την εκτύπωση της γραμμής. Αυτό ελέγχεται από την μεταβλητή *next_line*. Μόνο όταν ικανοποιηθεί η συνθήκη του while loop καλείται η συνάρτηση *output_mandel_line* ώστε να εκτυπωθεί η κατάλληλη γραμμή. Να αναφέρουμε, επίσης, ότι χρησιμοποιήσαμε while αντί για if προκειμένου μόλις επιβεβαιωθεί η συνθήκη να επαναληφθεί μια τελευταία φορά ο έλεγχος μετά την αφύπνιση. Τέλος, ανανεώνουμε τον μετρητή *next_line* ++, και σε περίπτωση που το επόμενο νήμα περιμένει το condition variable, το ξυπνάμε ορθά ώστε να εκτυπώσει το χωρίο του και κάνουμε unlock το mutex ώστε να το βρει ξεκλειδωτο.

Παραθέτουμε επίσης και τις κύριες διαφορές που υλοποιήσαμε στην κύρια συνάρτηση του προγράμματος μας η οποία δημιουργεί τα νήματα και τους περνάει ως παράμετρο εκτέλεσης στην συνάρτηση που είδαμε ακριβώς παραπάνω:

```
int main (int argc, char **argv)
{
    if (argc != 2){
        perror("We need a positive thread count");
        exit(1);
    }

    if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0){
        fprintf(stderr, "%s' is not valid for 'thread_count'\n", argv[1]);
        exit(1);
    }

    int ret;
    struct thread_info_struct *thr = safe_malloc(thrcnt * sizeof(*thr));
    conds = safe_malloc(thrcnt * sizeof(pthread_cond_t));

    if (pthread_mutex_init(&mutex, NULL)){
        perror("mutex");
        exit(1);
    }

    for (int i = 0; i < thrcnt; i++) {
        if (pthread_cond_init(&conds[i], NULL) != 0){
            perror("condition-variables");
            exit(1);
        }
    }

    xstep = (xmax - xmin) / x_chars;
    ystep = (ymax - ymin) / y_chars;

    /*
     * draw the Mandelbrot Set, one line at a time.
     * Output is sent to file descriptor '1', i.e., standard output.
     */

    for (int i = 0; i < thrcnt; i++){
        thr[i].thread_number = i;
        ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
        if (ret){
            perror("thread create");
            exit(1);
        }
    }

    for (int i = 0; i < thrcnt; i++){
        ret = pthread_join(thr[i].tid, NULL);
        if (ret){
            perror ("thread join");
            exit(1);
        }
    }

    for (int i = 0; i < thrcnt; i++) {
        if (pthread_cond_destroy(&conds[i]) != 0){
            perror("condition variables");
            exit(1);
        }
    }

    if (pthread_mutex_destroy(&mutex) != 0){
        perror("mutex");
        exit(1);
    }

    free(conds);
    free(thr);

    reset_xterm_color(1);
    return 0;
}
```

Ερωτήσεις

1. Η απεικόνιση των γραμμών απαιτεί τον συγχρονισμό των νημάτων προκειμένου αυτές να απεικονίζονται στην σωστή σειρά. Προκειμένου να το επιτύχουμε αυτό ορίζουμε το output των γραμμών ως κρίσιμο τμήμα. Αυτό φυσικά στον συγκεκριμένο κώδικα το πετυχαίνουμε με χρήση σημαφόρων. Αξίζει να σημειωθεί πως ο υπολογισμός των γραμμών δεν βρίσκεται μέσα στο κρίσιμο τμήμα με αποτέλεσμα ο υπολογισμός να είναι αρκετά παράλληλος και να βελτιώνεται η επίδοση του προγράμματος. Προκειμένου να επιτύχουμε εξ ολοκλήρου παράλληλο υπολογισμό θα χρειαζόμασταν τόσα νήματα όσο και το σύνολο των γραμμών προκειμένου κάθε νήμα να εκτελεί τον υπολογισμό και μετά να τον εκτυπώνει. Στην συνήθη περίπτωση όπου ο αριθμός των νημάτων είναι σημαντικά μικρότερος από τον αριθμό των γραμμών δεν επιτυγχάνεται παράλληλος υπολογισμός εξ-ολοκλήρου καθώς ένα νήμα πρώτα θα περιμένει να εκτυπωθούν οι προηγούμενες N-1 γραμμές, μετά θα εκτυπώσει το χωρίο του και μετά θα προσχωρήσει στον επόμενο υπολογισμό γραμμής που του αντιστοιχεί.

Για τον συγχρονισμό χρησιμοποιούμε τόσους σημαφόρους όσα είναι τα νήματα. Αρχικά, εντός της κύριας συνάρτησης, δημιουργούμε τόσους σημαφόρους όσους καθορίζει το input από τον χρήστη. Τους αρχικοποιούμε όλους στην τιμή 0 εκτός του πρώτου σημαφόρου που βρίσκεται στην θέση `semaphores[0]`, τον οποίο αρχικοποιούμε στην τιμή 1. Έτσι, αφού τα νήματα εκτελέσουν τον υπολογισμό της γραμμής τους για 1^η φορά, μόνο το πρώτο νήμα θα καταφέρει να εισέλθει στο κρίσιμο τμήμα αφού καλώντας `sem_wait(&semaphores[0])` θα βρει τον σημαφόρο του αρχικά θετικό (θα μηδενιστεί εντός της wait). Όλα τα υπόλοιπα νήματα που βρίσκουν τον σημαφόρο τους ίσο με το 0 θα τεθούν σε κατάσταση ύπνου (αφού ένας σημαφόρος αρχικοποιημένος στην τιμή 1 είναι ισοδύναμος με το POSIX mutex). Τα νήματα που βρίσκουν τον σημαφόρο τους ίσο με το 0 ξυπνούν όταν το αμέσως προηγούμενο νήμα καλέσει `sem_post(&semaphores[(number + 1) % thrcnt]);`. Όταν το νήμα εκτυπώσει το χωρίο του τότε επιστρέφει στην λούπα όπου του ανατίθεται πάλι νέο χωρίο που βρίσκεται `thrcnt` (αριθμός νημάτων) γραμμές πιο κάτω και περιμένει το αμέσως προηγούμενο νήμα να καλέσει `signal` για τον σημαφόρο του ώστε να εισέλθει εκ νέου στο κρίσιμο τμήμα. Κατ' αυτόν τον τρόπο εξασφαλίζουμε την κυκλική ανακατανομή των νημάτων και την ορθή εκτύπωση των γραμμών.

Σημείωση 1^η: Η τιμή `number` περιέχει τον αριθμό του κάθε νήματος και το modulo χρησιμοποιείται για να μπορούμε να θεωρούμε το 1^ο νήμα ως το επόμενο του τελευταίου νήματος στον κυκλικό διαμοιρασμό.

Σημείωση 2^η: Ένας σημαφόρος δρα ανεξάρτητα από τους άλλους σημαφόρους σε κοινό πρόγραμμα. Έτσι, όταν κληθεί η `sem_post(&semaphores[i])` τότε ξυπνά μόνο εκείνο το νήμα που είχε καλέσει `wait` σε αυτόν τον σημαφόρο. Τα υπόλοιπα νήματα δεν έχουν καμία διασύνδεση με τον σημαφόρο αυτόν.

2. Για την ολοκλήρωση του σειριακού υπολογισμού του συνόλου Mandelbrot απαιτείται χρόνος ίσος με αυτόν που φαίνεται στην ακόλουθη εικόνα:

```
real    0m0.732s
user    0m0.440s
sys     0m0.012s
```

Για την ολοκλήρωση του παράλληλου υπολογισμού του συνόλου Mandelbrot με δυο νήματα υπολογισμού για τους δυο διαφορετικούς τρόπους υλοποίησης (με σημαφόρους και με μεταβλητές κατάστασης) αναμένουμε ελάττωση της συνολικής καθυστέρησης. Αυτό οφείλεται στο γεγονός ότι παρόλο που υπάρχουν καθυστερήσεις, είτε λόγω των κλειδωμάτων - όταν πρόκειται για σημαφόρους - προκειμένου να μην υπάρξουν race conditions και να τυπωθούν οι γραμμές όπως ακριβώς θέλουμε με την σειρά, είτε όταν υπάρχουν εντολές wait υπό συνθήκη πριν την εκτύπωση γραμμών – όταν πρόκειται για μεταβλητές κατάστασης – προκειμένου κάθε νήμα να ενημερώνεται μέσω ενός *cond_signal* για το πότε είναι η σειρά του να εκτυπώσει, η παραλληλία που υλοποιούμε είναι ικανή να ξεπεράσει τον σειριακό χρόνο υλοποίησης. Αυτό άλλωστε είναι το ζητούμενο της παράλληλης επεξεργασίας, η μείωση του χρόνου εκτέλεσης ενός προγράμματος.

Η εντολή που εκτελούμε για να δούμε τον χρόνο εκτέλεσης είναι η *time(./mandel 2)*.

- i. Για παραλληλία και μεταβλητές κατάστασης

```
real    0m0.018s
user    0m0.020s
sys     0m0.009s
```

Περνάμε σε έναν γρήγορο σχολιασμό των τριών χρόνων που βλέπουμε. Συγκεκριμένα, ο πρώτος χρόνος (real) είναι ο πραγματικός χρόνος που έκανε ο επεξεργαστής από την εκκίνηση της εντολής μέχρι την ολοκλήρωσή της, δηλαδή την εκτύπωση των χρόνων. Ο δεύτερος χρόνος (user) είναι ο χρόνος που δαπανήθηκε από τον επεξεργαστή για την εκτέλεση της εντολής, χωρίς να προσμετράτε ο χρόνος αναμονής κάθε νήματος. Τέλος, ο χρόνος sys αντιπροσωπεύει τον χρόνο που δαπανήθηκε προκειμένου να εκτελεστούν οι εντολές πυρήνα, όπως η *malloc* για παράδειγμα.

Όπως είναι ορατό από τις τιμές για τους χρόνους που πήραμε, βλέπουμε μια αισθητή μείωση στον χρόνο εκτέλεσης του προγράμματος με την χρήση μεταβλητών κατάστασης. Το ίδιο θα βλέπαμε και για τον χρόνο εκτέλεσης με σημαφόρους. Αυτό συμβαίνει λόγω της παραλληλίας που επιτυγχάνουμε με τις μεταβλητές κατάστασης (αντίστοιχα και με τους σημαφόρους), καθώς υπολογίζουμε παράλληλα γραμμές του Mandelbrot set ανάλογες του αριθμού των threads που θέλουμε (στην περίπτωση του ερωτήματος αυτού 2). Σε αντίθεση, ο σειριακός υπολογισμός καθυστερεί αρκετά διότι υπολογίζει μια μια τις γραμμές.

Σημείωση: Είναι σημαντικό να αναφέρουμε ότι τρέξαμε την εντολή *time(./mandel 2)* σε τερματικό που υποστηρίζεται από παραπάνω από 1 πυρήνα, τοπικά δηλαδή. Το τερματικό που υλοποιούμε τα προγράμματα στο εργαστήριο μέσω της εντολής *ssh* υποστηρίζει 8

μονοπύρηνους επεξεργαστές, επομένως χρειάζεται να το τρέξουμε τοπικά. Αν τρέχαμε τα νήματα στο ssh, αυτά δεν θα έτρεχαν παράλληλα όπως θέλουμε, αλλά το κάθε νήμα θα καταλάμβανε τον πυρήνα για κάποιο χρόνο μόνο μέχρι να πάρει τη θέση του το επόμενο κλπ. Άρα αυτό που θα συνέβαινε θα ήταν η εναλλαγή των threads στον πυρήνα και όχι η επιθυμητή παράλληλη εκτέλεση τους. Τρέχοντας τοπικά και χρησιμοποιώντας έναν πολυπύρηνο επεξεργαστή λύνουμε το παραπάνω πρόβλημα. Κάθε νήμα φυσικά βρίσκεται εντός της ίδιας διεργασίας και εκτελείται στον ίδιο επεξεργαστή, ωστόσο έχει την δυνατότητα να εκτελεστεί σε διαφορετικό πυρήνα του επεξεργαστή με αποτέλεσμα να εξασφαλίζεται η επιθυμητή παραλληλοποίηση του προγράμματος.

3. Πάλι και σε αυτή την περίπτωση υλοποίησης με μεταβλητές κατάστασης χρειαστήκαμε μια μεταβλητή συνθήκης για κάθε νήμα. Αυτό είναι σημαντικό γιατί μόνο έτσι είμαστε σε θέση να ενημερώνουμε το κατάλληλο νήμα προκειμένου να συνεχίσει αυτό – και μόνο αυτό – την υλοποίηση που του ορίζουμε. Αν από την άλλη είχαμε μόνο μια μεταβλητή συνθήκης, αυτό που θα ακολουθούσε είναι πολύ πιθανόν η σύγχυση για το ποιο νήμα έχει σειρά να εκτυπώσει για παράδειγμα, race condition για το ποιος είναι πρώτος έτοιμος να αναλάβει την εκάστοτε δουλειά. Έχουμε παρουσιάσει παραπάνω τον κώδικα που υλοποιήσαμε με condition variables. Στον κώδικα μας χρησιμοποιήσαμε τόσες μεταβλητές συνθήκης όσα είναι και τα νήματα. Όπως ακριβώς είδαμε και παραπάνω, οι μεταβλητές συνθήκης χρησιμεύουν ώστε αν εισέλθει κάποιο νήμα στο κρίσιμο τμήμα (βρει το mutex ξεκλειδωτο) του οποίου δεν έχει έρθει η σειρά να εκτυπώσει το χωρίο του ακόμη, τότε το νήμα αυτό βρίσκει πώς η γραμμή του, *i*, είναι διάφορη του κοινού πόρου για όλες τις διεργασίες *next_line* με αποτέλεσμα να εισέρχεται σε κατάσταση αναμονής (κατάσταση ύπνου) μέσω της εντολής:

```
pthread_cond_wait(&conds[thread-> thread_number], &mutex);
```

Όταν έρθει η σειρά του συγκεκριμένου νήματος να εκτυπώσει το χωρίο του τότε εκτελείτε η εντολή:

```
pthread_cond_signal(&conds[(thread-> thread_number + 1) % thrcnt]);
```

Με τον τρόπο αυτό ‘ξυπνάμε’ μόνο το νήμα εκείνο που αντιστοιχεί στο συγκεκριμένο condition variable το οποίο προχωρά στην εκτύπωση.

Ακολουθεί ένας σχολιασμός για την περίπτωση όπου είχαμε μια μόνο μεταβλητή κατάστασης για όλα τα νήματα. *mandel_condition_variables_inefficient*

```
void *compute_and_output_mandel_line(void *arg)
{
    struct thread_info_struct *thread = arg;
    int color_val[x_chars];

    for (int i = thread->thread_number; i < y_chars; i += thrcnt)
    {
        compute_mandel_line(i, color_val); // Compute the line first without locking.

        pthread_mutex_lock(&mutex); // This lock ensures that wait is used correctly

        // Wait until it's this thread's turn to output.
        while (i != next_line)
        {
            pthread_cond_signal(&cond); //if we want to use one condition variable, we must put this signal here
            //if we don't want this signal in this position, we shall only use broadcast after next_line++ instead of signal
            pthread_cond_wait(&cond, &mutex);
        }

        // Output the line as it's now this thread's turn.
        output_mandel_line(i, color_val);
        next_line++; // next_line is common for all threads
        pthread_cond_signal(&cond); // Wake up only one thread

        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

Αν είχαμε χρησιμοποιήσει μόνο ένα condition variable θα διατηρούσαμε τα ίδια αποτελέσματα στον κώδικά μας, ωστόσο, θα είχαμε μια επίπτωση στην επίδοση του προγράμματος. Συγκεκριμένα, αν ένα νήμα, του οποίου δεν ήταν η σειρά να εκτυπώσει το χωρίο του, εισερχόταν στο κρίσιμο τμήμα τότε πάλι θα καλούσε *pthread_cond_wait(&cond)* με αποτέλεσμα να εισερχόταν σε κατάσταση αναμονής (ύπνου). Τονίζουμε πως αυτήν την φορά έχουμε ένα condition variable που το ορίζουμε *cond*. Για την επανεκκίνηση του συγκεκριμένου νήματος αυτήν την φορά θα έπρεπε να κληθεί η εντολή: *pthread_cond_broadcast(&cond)* αφυπνίζοντας όλα τα νήματα, ή η εντολή *pthread_cond_signal(&cond)*, με την οποία θα αφυπνιζόταν ένα τυχαίο νήμα από όσα είναι έτοιμα να εκτυπώσουν.

Έτσι, στην μεν πρώτη περίπτωση όλα τα νήματα που είναι έτοιμα να εκτυπώσουν θα προσπαθούσαν να αποκτήσουν το mutex για να εισέλθουν στο κρίσιμο τμήμα και να συνεχίσουν την εκτέλεση από το σημείο στο οποίο βρίσκονταν προτού μπουν σε κατάσταση αναμονής. Στην μεν δεύτερη περίπτωση, μόνο το ένα νήμα που έγινε signaled θα προσπαθούσε να εισέλθει στο κρίσιμο τμήμα, αν όμως επιβεβαιώνε την συνθήκη *i == next_line*. Ειδάλλως θα συνέχιζε να γίνεται signal έως ότου βρεθεί το νήμα που την ικανοποιεί. Αν αυτό λειτουργήσει και δεν κολλήσει σε ένα ατέρμονο wait (όπως και έγινε όταν το υλοποιήσαμε εμείς για δοκιμή αρκετές φορές) δεν θα είναι καθόλου χρονικά efficient. Αυτό γιατί κάθε φορά είτε θα προσπαθούνε όλα τα νήματα να διεκδικήσουν την είσοδο και να εκτυπώσουν την γραμμή που τους αντιστοιχεί, αλλά μόνο ένα πάντα θα επιβεβαιώνει την επιθυμητή συνθήκη, είτε θα δίνεται το δικαίωμα σε ένα νήμα κάθε φορά για είσοδο προς εκτύπωση χωρίς όμως αυτό να είναι απαραίτητα το κατάλληλο (λάθος σειρά εκτύπωσης -> δεν θα τον άφηνε η συνθήκη μας).

Άρα με τα πολλά condition-variables που χρησιμοποιήσαμε, κάθε νήμα έχει το δικό του condition-variable με αποτέλεσμα να μπορούμε να το ειδοποιήσουμε να ξεκινήσει (σε περίπτωση που έχει βρεθεί σε κατάσταση αναμονής) ανεξάρτητα από τα άλλα νήματα. Αποφεύγουμε έτσι τις άσκοπες αφυπνίσεις.

Σημείωση: Είναι πολύ κρίσιμο να κατανοήσουμε πώς όταν ξυπνάμε νήματα που περιμένουν σε condition variable (είτε μέσω signal όπου ξυπνάμε μόνο 1 νήμα είτε μέσω της broadcast όπου ξυπνάμε όλα τα νήματα) τα νήματα αυτά, που ξύπνησαν, τότε προσπερνούν εξ-ολοκλήρου το condition variable και περιμένουν στο mutex που έχουμε χρησιμοποιήσει το οποίο άλλωστε είναι στενά συνδεδεμένο με το condition variable. Όλα αυτά τα νήματα διεκδικούν το κλείδωμα μαζί και με τα υπόλοιπα νήματα που περίμεναν το κλείδωμα σε κάποιο άλλο σημείο του κώδικα. Το νήμα το οποίο θα πάρει το mutex καθορίζεται από εξωγενείς παράγοντες και σε περίπτωση που είναι το λάθος νήμα (δηλαδή $i \neq \text{next_line}$) τότε το νήμα αυτό θα περιμένει στο condition variable και ταυτόχρονα θα ελευθερώσει και το mutex επιτρέποντας στα υπόλοιπα νήματα που αναμένουν το mutex να το διεκδικήσουν. Η διαδικασία που περιγράψαμε είναι ο κύριος λόγος για την οποία το παραπάνω σχήμα που έχουμε υλοποίηση δεν πέφτει σε deadlock.

4. Το κρίσιμο τμήμα είναι σημαντικό να περιέχει την φάση εξόδου κάθε γραμμής, δηλαδή να υπάρχει ένα κλείδωμα (κάποιο wait) πριν κληθεί η συνάρτηση `output_mandel_line()`. Αυτό επιβάλλεται γιατί μόνο έτσι θα τυπώνονται όπως θέλουμε εμείς οι γραμμές με την σωστή σειρά. Ειδικά χωρίς κάποιο κλείδωμα, όποιο νήμα τελειώνει πρώτο με το υπολογισμό του χωρίου του, αυτό θα προλάβει να τυπώσει πρώτο το αποτέλεσμα του, με αποτέλεσμα να έχουμε (αν όχι όλες τις φορές, τις περισσότερες) λάθος αλληλουχία εκτύπωσης γραμμών.

Σχετικά με το αν θα έπρεπε το κρίσιμο τμήμα να περιέχει και την φάση υπολογισμού, αυτό εξαρτάται. Όσο μεγαλύτερο είναι το κρίσιμο σημείο, τόσο μεγαλύτερη καθυστέρηση προσθέτουμε στην υλοποίησή μας, γιατί τόσο περισσότερο περιμένει κάθε επόμενο νήμα να αναλάβει δουλειά. Πρακτικά όμως, προσθέτοντας και την φάση υπολογισμού και την φάση εξόδου στην κρίσιμη περιοχή, μετατρέπουμε την υλοποίησή μας από παράλληλη που θέλουμε να επιτύχουμε πάλι σε σειριακή και αυτό δεν το θέλουμε.

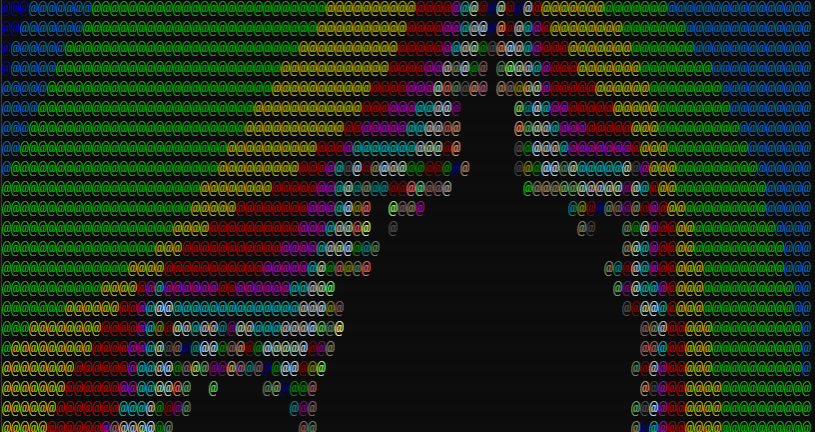
Όπως διαπιστώσαμε προηγουμένως, τα παράλληλα προγράμματα που υλοποιήσαμε πράγματι εμφανίζουν επιτάχυνση σε σχέση με το αρχικό πρόγραμμα. Αυτό συμβαίνει καθώς εφαρμόζουμε παράλληλο υπολογισμό μεταξύ των νημάτων όπως ακριβώς αναφέραμε. Επαναλαμβάνουμε ότι το κρίσιμο τμήμα περιέχει μόνο την φάση εξόδου με αποτέλεσμα ο υπολογισμός που εκτελούν τα νήματα κατά την i -οστή επανάληψή τους να είναι παράλληλος. Έτσι, έχουμε επιτάχυνση σε σχέση με το σειριακό δοθέντα κώδικα.

- [illegible]

Την λειτουργία αυτή την εκτελεί η ακόλουθη συνάρτηση:

```
void sigint_handler(int sig)
{
    printf("\033[0m");
    exit(sig);
}
```

```
oslab044@os-node1:~/second_ex/ex_2$ ./mandel
```



```
oslab044@os-node1:~/second_ex/ex_2$
```