

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

3η Σειρά Ασκήσεων

Μάθημα: Λειτουργικά Συστήματα (Τμήμα 1^ο)

Εξάμηνο: 6^ο

Ονοματεπώνυμο: Αλεξοπούλου Γεωργία (ΑΜ: 03120164), Γκενάκου Ζωή (ΑΜ: 03120015)

Άσκηση 3.1:

Έχουμε το πρόγραμμα `simplesync.c`, το οποίο λειτουργεί ως εξής: Αφού αρχικοποιήσει μια μεταβλητή `val = 0`, δημιουργεί δύο νήματα τα οποία εκτελούνται ταυτόχρονα: το πρώτο νήμα αυξάνει `N` φορές την τιμή της μεταβλητής `val` κατά 1, το δεύτερο τη μειώνει `N` φορές κατά 1. Τα νήματα δεν συγχρονίζουν την εκτέλεσή τους.

Αρχικά, εκτελούμε το δοθέν `Makefile` για τη μεταγλώττιση και την εκτέλεση του προγράμματος:

```
oslab33@orion:~/ex3/examples$ make
gcc -Wall -O2 -pthread -c -o pthread-test.o pthread-test.c
gcc -Wall -O2 -pthread -o pthread-test pthread-test.o
gcc -Wall -O2 -pthread -DSYNC MUTEX -c -o simplesync-mutex.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-mutex simplesync-mutex.o
gcc -Wall -O2 -pthread -DSYNC ATOMIC -c -o simplesync-atomic.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-atomic simplesync-atomic.o
gcc -Wall -O2 -pthread -c -o kgarten.o kgarten.c
gcc -Wall -O2 -pthread -o kgarten kgarten.o
gcc -Wall -O2 -pthread -c -o mandel-lib.o mandel-lib.c
gcc -Wall -O2 -pthread -c -o mandel.o mandel.c
gcc -Wall -O2 -pthread -o mandel mandel-lib.o mandel.o
oslab33@orion:~/ex3/examples$ ls
kgarten  kgarten.o  mandel  mandel-lib.c  mandel-lib.o  pthread-test  pthread-test.o  simplesync-atomic  simplesync.c  simplesync-mutex.o
kgarten.c  Makefile  mandel.c  mandel-lib.h  mandel.o  pthread-test.c  rand-fork.c  simplesync-atomic.o  simplesync-mutex
oslab33@orion:~/ex3/examples$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 766691.
oslab33@orion:~/ex3/examples$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = 4001197.
```

Εκτελώντας το Makefile που μας δίνεται, λαμβάνουμε στο terminal δύο διαφορετικά simplesync εκτελέσιμα αρχεία: το simplesync-atomic και το simplesync-mutex. Ο gcc καθορίζει προκαθορισμένες μακροεντολές SYNC_ATOMIC και SYNC_MUTEX μέσω του -Dmacro option για τον preprocessor. Έτσι, ανάλογα με την προκαθορισμένη μακροεντολή, το USE_ATOMIC_OPS ισούται είτε με 0 είτε με 1. Ανάλογα με την τιμή αυτή, λαμβάνουμε και διαφορετική λύση στο πρόβλημα συγχρονισμού: για USE_ATOMIC_OPS = 0 επιλύουμε το πρόβλημα συγχρονισμού με POSIX mutexes, ενώ για USE_ATOMIC_OPS = 1 επιλύουμε το ίδιο πρόβλημα με atomic operations.

Καθώς τρέχουμε καθένα από τα δύο προγράμματα, το αποτέλεσμα που λαμβάνουμε είναι λανθασμένο λόγω έλλειψης συγχρονισμού μεταξύ των νημάτων. Εξαιτίας αυτού, η αυξομείωση τιμών στον πηγαίο κώδικα εκτελείται με λάθος τρόπο, διότι οι εντολές που αντιστοιχούν στις συναρτήσεις αυτές δεν συγχρονίζονται και συγχέονται μεταξύ τους. Ο μεταγλωττιστής, δηλαδή, αναδιατάσσει τις εντολές, και η εκτος-σειράς εκτέλεσή τους ευθύνεται για τα λάθος αποτελέσματα (race conditions). Επιπλέον, ο preemptive scheduling ενός Λειτουργικού Συστήματος μπορεί να οδηγήσει στη διακοπή της εκτέλεσης μια διεργασίας-νήματος.

Σε αυτό το σημείο αξίζει να αναφερθούμε στα κρίσιμα τμήματα του κώδικα: critical sections ονομάζουμε τα κομμάτια κώδικα στα οποία έχει πρόσβαση το πολύ μια διεργασία-νήμα τη φορά. Η επίτευξη αυτής της ανά-μονάδα πρόσβασης στον κώδικα μπορεί να γίνει με την αξιοποίηση διαφόρων μηχανισμών, όπως atomic operations, mutexes, semaphores etc.

Επεκτείνουμε τον κώδικα του αρχείου simplesync.c, έτσι ώστε η εκτέλεση των δύο νημάτων τόσο του αρχείου simplesync-atomic, όσο και του αρχείου simplesync-mutex να συγχρονίζονται με τη χρήση atomic operations του gcc και με mutexes αντίστοιχα. Παρακάτω φαίνεται ο ζητούμενος κώδικας:

```
/*
 * simplesync.c
 *
 * A simple synchronization exercise.
 *
 * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
 * Operating Systems course, ECE, NTUA
 *
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
```

```

/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */
#define perror_pthread(ret, msg) \
do { errno = ret; perror(msg); } while (0)

#define N 10000000

/* Dots indicate lines where you are free to insert code at will */
/* ... */
#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

pthread_mutex_t lock;

void *increase_fn(void *arg) {
    int i, ret;
    volatile int *ip = arg; // declare a volatile integer pointer and initialize
                             // it with the argument passed to the function

    fprintf(stderr, "About to increase variable %d times\n", N);
    // print a message to stderr indicating the number of times the variable will
    // be increased
    for (i = 0; i < N; i++) { // loop N times
        if (USE_ATOMIC_OPS) { // if the USE_ATOMIC_OPS flag is defined, use an
                               // atomic operation to increment the variable
            __sync_fetch_and_add(ip, 1); // sync with gcc atomic operations
        }
        else { // otherwise, use a mutex to synchronize access to the variable
            ret = pthread_mutex_lock(&lock); // acquire the lock

```

```

        if (ret) { // if the lock cannot be acquired, print an error message
                    // and exit
                    perror_thread(ret, "pthread_mutex_lock");
                    exit(1);
                }
        ++(*ip); // increment the variable
        ret = pthread_mutex_unlock(&lock); // release the lock
        if (ret) { // if the lock cannot be released, print an error message
                    // and exit
                    perror_thread(ret, "pthread_mutex_unlock");
                    exit(1);
                }
    }
}

fprintf(stderr, "Done increasing variable.\n");
// print a message to stderr indicating that the variable has been
// incremented N times

return NULL;
}

void *decrease_fn(void *arg) {
    int i, ret;
    volatile int *ip = arg; // declare a volatile integer pointer and initialize
                             // it with the argument passed to the function

    fprintf(stderr, "About to decrease variable %d times\n", N);
    // print a message to stderr indicating the number of times the variable will
    // be increased
    for (i = 0; i < N; i++) { // loop N times
        if (USE_ATOMIC_OPS) { // if the USE_ATOMIC_OPS flag is defined, use an
                               // atomic operation to increment the variable
            __sync_fetch_and_sub(&ip, 1); // sync with gcc atomic operations
        }
        else { // otherwise, use a mutex to synchronize access to the variable
            ret = pthread_mutex_lock(&lock); // acquire the lock
            if (ret) { // if the lock cannot be acquired, print an error message
                        // and exit
                        perror_thread(ret, "pthread_mutex_lock");

```

```

        exit(1);
    }
    --(*ip); // decrement the variable
    ret = pthread_mutex_unlock(&lock); // release the lock
    if (ret) { // if the lock cannot be released, print an error message
        // and exit
        perror_pthread(ret, "pthread_mutex_unlock");
        exit(1);
    }
}

}

fprintf(stderr, "Done decreasing variable.\n");
// print a message to stderr indicating that the variable has been decreased
// N times

return NULL;
}

int main(int argc, char *argv[]) {
    int val, ret, ok;
    pthread_t t1, t2;

    // Initialize mutex
    pthread_mutex_init(&lock, NULL);

    /*
     * Initial value
     */
    val = 0;

    /*
     * Create threads
     */
    ret = pthread_create(&t1, NULL, increase_fn, &val);
    if (ret) {
        perror_pthread(ret, "pthread_create");
        exit(1);
    }
    ret = pthread_create(&t2, NULL, decrease_fn, &val);

```

```

if (ret) {
    perror_pthread(ret, "pthread_create");
    exit(1);
}

/*
 * Wait for threads to terminate
 */
ret = pthread_join(t1, NULL);
if (ret)
    perror_pthread(ret, "pthread_join");
ret = pthread_join(t2, NULL);
if (ret)
    perror_pthread(ret, "pthread_join");

/*
 * Is everything OK?
 */
ok = (val == 0);

// Destroy mutex
ret = pthread_mutex_destroy(&lock);
if (ret) {
    perror_pthread(ret, "pthread_mutex_destroy");
}

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

return ok;
}

```

Τρέχουμε τον κώδικα έτσι ώστε να βεβαιωθούμε ότι τα αποτελέσματα που λαμβάνουμε είναι σωστά. Εκτελώντας εκ νέου το προηγούμενο Makefile, έχουμε την εξής έξοδο στο τερματικό μας:

```

oslab33@orion:~/ex3/3.1$ make
gcc -O2 -pthread -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c
gcc -O2 -pthread -o simplesync-mutex simplesync-mutex.o
gcc -O2 -pthread -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
gcc -O2 -pthread -o simplesync-atomic simplesync-atomic.o
oslab33@orion:~/ex3/3.1$ ls
Makefile  simplesync-atomic  simplesync-atomic.o  simplesync.c  simplesync-mutex  simplesync-mutex.o
oslab33@orion:~/ex3/3.1$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
oslab33@orion:~/ex3/3.1$ ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

```

Πράγματι, έχουμε το επιθυμητό αποτέλεσμα, δηλαδή η τιμή της μεταβλητής val ισούται με 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;

Χωρίς χρονισμό:

```

oslab33@orion:~/ex3/3.1$ time ./simplesync
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m1.753s
user    0m0.868s
sys     0m0.008s

```

Συγχρονισμός με χρήση gcc atomic operations:

```
oslab33@orion:~/ex3/3.1$ 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    0m1.144s
user    0m0.616s
sys     0m0.000s
```

Συγχρονισμός με χρήση mutexes:

```
oslab33@orion:~/ex3/3.1$ time ./simplesync-mutex
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m1.876s
user    0m0.872s
sys     0m0.004s
```

Παρατηρούμε πως ο χρόνος εκτέλεσης των προγραμμάτων με χρονισμό είναι μεγαλύτερος από τον χρόνο εκτέλεσης του προγράμματος χωρίς χρονισμό. Αυτό συμβαίνει γιατί, όπως εξηγήσαμε παραπάνω, η διαδικασία χρονισμού των νημάτων συνεπάγεται πως το κρίσιμο σημείο του κώδικα γίνεται accessed από ένα νήμα κάθε φορά, πρακτική η οποία είναι περισσότερο χρονοβόρα. Αντίθετα, όταν τα νήματα δεν συγχρονίζονται, εισέρχονται στο εκτελέσιμο κομμάτι του κώδικα παράλληλα.

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

Η μέθοδος χρονισμού με χρήση POSIX mutexes είναι πιο αργή, σε σχέση με τη σχήμα χρονισμού gcc atomic operations. Πιο συγκεκριμένα, η χρήση POSIX mutexes συνεπάγεται πως, εκτός από τις ατομικές λειτουργίες που εκτελούνται ούτως ή άλλως για τον συγχρονισμό των νημάτων, εκτελούνται και κάποιες επιπλέον ατομικές λειτουργίες, που σχετίζονται με τις εντολές lock, unlock, wait. Το Λειτουργικό Σύστημα εμπλέκεται προκειμένου να “κλειδώσει”, να “ξεκλειδώσει” και να “βάζει σε αναμονή” τα νήματα, προκειμένου ένα μόνο να έχει πρόσβαση στο shared resource. Από την άλλη, η μέθοδος χρονισμού με χρήση gcc atomic operations εγγυάται τον συγχρονισμό των νημάτων με την ατομική εκτέλεση των εντολών του κρίσιμου τμήματος του κώδικα. Η μεγαλύτερη χρονική αποδοτικότητα των ατομικών λειτουργιών οφείλεται στο γεγονός πως αυτές εκτελούνται άμεσα πάνω στο Λειτουργικό Σύστημα, χωρίς αυτό να “κλειδώνει” και να “ξεκλειδώνει” μέρος του κώδικα, στο οποίο πρέπει οι ατομικές λειτουργίες να σπαταλούν χρόνο για να μπαίνουν κάθε φορά

που εκτελούνται. Συνεπώς, ο χρόνος εκτέλεσης με gcc atomic operations είναι μικρότερος, σε σχέση με τον χρόνο εκτέλεσης με POSIX mutexes.

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

Παρακάτω παραθέτουμε τον κώδικα για το αρχείο Makefile που ζητείται:

```
CC = gcc
CFLAGS = -O2 -pthread

all: simplesync-mutex simplesync-atomic simplesync

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

simplesync-mutex: simplesync-mutex.o
    $(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o

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

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

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

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

```

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

clean:
rm -f *.o *.s simplsync-mutex simplsync-atomic simplsync

```

Το αποτέλεσμα που λαμβάνουμε στο τερματικό μας μετά την εκτέλεση του Makefile είναι το ακόλουθο:

```

oslab33@orion:~/ex3/3.1$ ls
Makefile simplsync.c
oslab33@orion:~/ex3/3.1$ make simplsync.s
gcc -O2 -pthread -DSYNC_MUTEX -S -g -o simplsync.s simplsync.c
oslab33@orion:~/ex3/3.1$ make simplsync-atomic.s
gcc -O2 -pthread -DSYNC_ATOMIC -S -g -o simplsync-atomic.s simplsync.c
oslab33@orion:~/ex3/3.1$ make simplsync-mutex.s
gcc -O2 -pthread -DSYNC_MUTEX -S -g -o simplsync-mutex.s simplsync.c

```

Στη συνέχεια, για να εντοπίσουμε πού ακριβώς στον επεξεργαστή μεταφράζεται η χρήση των ατομικών λειτουργιών του gcc θα αξιοποιήσουμε την εντολή `'objdump -d -S simplsync-atomic'`. Έτσι, λαμβάνουμε τις παρακάτω εξόδους:

α) Για την ατομική λειτουργία `'__sync_fetch_and_add'` για την αύξηση της τιμής της μεταβλητής `'val'`: Παρακάτω φαίνεται το κομμάτι κώδικα που αντιστοιχεί στη συνάρτηση `'increase_fn'`. Έχουμε απομονώσει το τμήμα που αναφέρεται στην εντολή `'__sync_fetch_and_add'`.

```

0000000004009f0 <increase_fn>:
4009f0: 48 83 ec 18          sub    $0x18,%rsp
4009f4: ba 80 96 98 00      mov    $0x989680,%edx
4009f9: be 38 0b 40 00      mov    $0x400b38,%esi
4009fe: 48 89 7c 24 08      mov    %rdi,0x8(%rsp)
400a03: 48 8b 3d 96 08 20 00 mov    0x200896(%rip),%rdi    # 6012a0 <stderr@GLIBC_2.2.5>
400a0a: 31 c0              xor    %eax,%eax
400a0c: e8 4f fd ff ff      callq  400760 <fprintf@plt>
400a11: b8 80 96 98 00      mov    $0x989680,%eax
400a16: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
400a1d: 00 00 00
400a20: f0 48 83 44 24 08 01 lock addq $0x1,0x8(%rsp)
400a27: 83 e8 01          sub    $0x1,%eax
400a2a: 75 f4            jne    400a20 <increase_fn+0x30>
400a2c: 48 8b 0d 6d 08 20 00 mov    0x20086d(%rip),%rcx    # 6012a0 <stderr@GLIBC_2.2.5>
400a33: ba 1a 00 00 00      mov    $0x1a,%edx
400a38: be 01 00 00 00      mov    $0x1,%esi
400a3d: bf 88 0b 40 00      mov    $0x400b88,%edi
400a42: e8 69 fd ff ff      callq  4007b0 <fwrite@plt>
400a47: 31 c0              xor    %eax,%eax
400a49: 48 83 c4 18          add    $0x18,%rsp
400a4d: c3              retq
400a4e: 66 90            xchg   %ax,%ax

400a20: f0 48 83 44 24 08 01 lock addq $0x1,0x8(%rsp)

```

β) Για την ατομική λειτουργία `'__sync_fetch_and_sub'` για τη μείωση της τιμής της μεταβλητής `'val'`: Παρακάτω φαίνεται το κομμάτι κώδικα που αντιστοιχεί στη συνάρτηση

‘*decrease_fin*’. Έχουμε απομονώσει το τμήμα που αναφέρεται στην εντολή ‘*_sync_fetch_and_sub*’.

```
000000000400a50 <decrease_fn>:
400a50: 48 83 ec 18      sub    $0x18,%rsp
400a54: ba 80 96 98 00   mov    $0x989680,%edx
400a59: be 60 0b 40 00   mov    $0x400b60,%esi
400a5e: 48 89 7c 24 08   mov    %rdi,0x8(%rsp)
400a63: 48 8b 3d 36 08 20 00 mov    0x200836(%rip),%rdi    # 6012a0 <stderr@GLIBC_2.2.5>
400a6a: 31 c0           xor    %eax,%eax
400a6c: e8 ef fc ff ff   callq  400760 <fprintf@plt>
400a71: b8 80 96 98 00   mov    $0x989680,%eax
400a76: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
400a7d: 00 00 00
400a80: f0 48 83 6c 24 08 01 lock subq $0x1,0x8(%rsp)
400a87: 83 e8 01        sub    $0x1,%eax
400a8a: 75 f4          jne    400a80 <decrease_fn+0x30>
400a8c: 48 8b 0d 0d 08 20 00 mov    0x20080d(%rip),%rcx    # 6012a0 <stderr@GLIBC_2.2.5>
400a93: ba 1a 00 00 00   mov    $0x1a,%edx
400a98: be 01 00 00 00   mov    $0x1,%esi
400a9d: bf a3 0b 40 00   mov    $0x400ba3,%edi
400aa2: e8 09 fd ff ff   callq  4007b0 <fwrite@plt>
400aa7: 31 c0           xor    %eax,%eax
400aa9: 48 83 c4 18      add    $0x18,%rsp
400aad: c3             retq
400aae: 66 90          xchgb  %ax,%ax

400a80: f0 48 83 6c 24 08 01 lock subq $0x1,0x8(%rsp)
```

Και στις δύο παραπάνω περιπτώσεις, παρατηρούμε ότι τόσο πριν την εντολή ‘*addq*’ όσο και πριν την εντολή ‘*subq*’ συναντούμε το πρόθημα ‘*lock*’. Πρόκειται για ένα πρόθεμα οδηγίων στη γλώσσα assembly x86/x86-64 που υποδεικνύει τη χρήση ενός μηχανισμού κλειδώματος για μια λειτουργία μνήμης. Στη συγκεκριμένη περίπτωση, το πρόθεμα «κλειδώμα» χρησιμοποιείται σε συνδυασμό με τις εντολές ‘*addq*’ και ‘*subq*’ για την εκτέλεση μιας λειτουργίας ατομικής προσθαφαίρεσης. Η εντολή ‘*addq \$0x1, 0x8(%rsp)*’, για παράδειγμα, προσθέτει την τιμή ‘1’ στη θέση μνήμης σε μετατόπιση ‘0x8’ από τον δείκτη στοίβας (‘*%rsp*’) και το πρόθεμα ‘*lock*’ διασφαλίζει ότι αυτή η λειτουργία εκτελείται ατομικά. Αυτή η ατομικότητα που περιγράψαμε διασφαλίζει ότι η θέση μνήμης δεν τροποποιείται ταυτόχρονα από πολλά νήματα ή επεξεργαστές κατά τη διάρκεια της λειτουργίας, αποτρέποντας τα ‘*race conditions*’ και διασφαλίζοντας την ακεραιότητα των δεδομένων σε περιβάλλοντα πολλαπλών νημάτων.

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.

Θα ακολουθήσουμε την ίδια διαδικασία με το ερώτημα 3. Για να εντοπίσουμε τις εντολές της αρχιτεκτονικής assembly στις οποίες μεταφράζεται η χρήση POSIX mutexes θα αξιοποιήσουμε την εντολή ‘*objdump -d -S simplesync-mutex*’. Έτσι, λαμβάνουμε τις παρακάτω εξόδους:

α) Για την POSIX mutex λειτουργία για την αύξηση της τιμής της μεταβλητής ‘*val*’: Παρακάτω φαίνεται το κομμάτι κώδικα που αντιστοιχεί στη συνάρτηση ‘*increase_fin*’.

```

00000000004009f0 <increase_fn>:
 4009f0: 48 83 ec 18      sub    $0x18,%rsp
 4009f4: ba 80 96 98 00   mov    $0x989680,%edx
 4009f9: be 38 0b 40 00   mov    $0x400b38,%esi
 4009fe: 48 89 7c 24 08   mov    %rdi,0x8(%rsp)
 400a03: 48 8b 3d 96 08 20 00 mov    0x200896(%rip),%rdi    # 6012a0 <stderr@@GLIBC_2.2.5>
 400a0a: 31 c0            xor    %eax,%eax
 400a0c: e8 4f fd ff ff   callq 400760 <fprintf@plt>
 400a11: b8 80 96 98 00   mov    $0x989680,%eax
 400a16: 66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
 400a1d: 00 00 00
 400a20: f0 48 83 44 24 08 01 lock addq $0x1,0x8(%rsp)
 400a27: 83 e8 01         sub    $0x1,%eax
 400a2a: 75 f4           jne    400a20 <increase_fn+0x30>
 400a2c: 48 8b 0d 6d 08 20 00 mov    0x20086d(%rip),%rcx    # 6012a0 <stderr@@GLIBC_2.2.5>
 400a33: ba 1a 00 00 00   mov    $0x1a,%edx
 400a38: be 01 00 00 00   mov    $0x1,%esi
 400a3d: bf 88 0b 40 00   mov    $0x400b88,%edi
 400a42: e8 69 fd ff ff   callq 4007b0 <fwrite@plt>
 400a47: 31 c0            xor    %eax,%eax
 400a49: 48 83 c4 18      add    $0x18,%rsp
 400a4d: c3              retq
 400a4e: 66 90           xchq   %ax,%ax

```

β) Για την POSIX mutex λειτουργία για τη μείωση της τιμής της μεταβλητής ‘val’: Παρακάτω φαίνεται το κομμάτι κώδικα που αντιστοιχεί στη συνάρτηση ‘decrease_fn’.

```

0000000000400a50 <decrease_fn>:
 400a50: 48 83 ec 18      sub    $0x18,%rsp
 400a54: ba 80 96 98 00   mov    $0x989680,%edx
 400a59: be 60 0b 40 00   mov    $0x400b60,%esi
 400a5e: 48 89 7c 24 08   mov    %rdi,0x8(%rsp)
 400a63: 48 8b 3d 36 08 20 00 mov    0x200836(%rip),%rdi    # 6012a0 <stderr@@GLIBC_2.2.5>
 400a6a: 31 c0            xor    %eax,%eax
 400a6c: e8 ef fc ff ff   callq 400760 <fprintf@plt>
 400a71: b8 80 96 98 00   mov    $0x989680,%eax
 400a76: 66 2e 0f 1f 84 00 00 nopw   %cs:0x0(%rax,%rax,1)
 400a7d: 00 00 00
 400a80: f0 48 83 6c 24 08 01 lock subq $0x1,0x8(%rsp)
 400a87: 83 e8 01         sub    $0x1,%eax
 400a8a: 75 f4           jne    400a80 <decrease_fn+0x30>
 400a8c: 48 8b 0d 0d 08 20 00 mov    0x20080d(%rip),%rcx    # 6012a0 <stderr@@GLIBC_2.2.5>
 400a93: ba 1a 00 00 00   mov    $0x1a,%edx
 400a98: be 01 00 00 00   mov    $0x1,%esi
 400a9d: bf a3 0b 40 00   mov    $0x400ba3,%edi
 400aa2: e8 09 fd ff ff   callq 4007b0 <fwrite@plt>
 400aa7: 31 c0            xor    %eax,%eax
 400aa9: 48 83 c4 18      add    $0x18,%rsp
 400aad: c3              retq
 400aae: 66 90           xchq   %ax,%ax

```

Τα POSIX mutexes κάνουν χρήση των λειτουργιών ‘pthread_mutex_lock’ και ‘pthread_mutex_unlock’. Συγκεκριμένα, η μεταγλώττιση σε assembly της εντολής ‘pthread_mutex_lock’ φαίνεται στο παρακάτω τμήμα της εξόδου του τερματικού μας:

```

0000000000400870 <pthread_mutex_lock@plt>:
 400870: ff 25 0a 0a 20 00 jmpq   *0x200a0a(%rip)    # 601280 <_GLOBAL_OFFSET
T_TABLE+0x80>
 400876: 68 0d 00 00 00   pushq $0xd
 40087b: e9 10 ff ff ff   jmpq   400790 <_init+0x28>

```

Άσκηση 3.2:

Σε αυτή την άσκηση μας δίνεται το πρόγραμμα mandel.c που υπολογίζει και σχεδιάζει το σύνολο Mandelbrot σε τερματικό κειμένου, χρησιμοποιώντας χρωματιστούς χαρακτήρες. Ζητείται να επεκτείνουμε το πρόγραμμα mandel.c έτσι ώστε ο υπολογισμός να κατανέμεται σε NTHREADS νήματα POSIX. Συγκεκριμένα, ζητούνται 2 εκδοχές του προγράμματος όπου ο απαραίτητος συγχρονισμός των νημάτων θα γίνεται αφενός (i) με σημαφόρους και αφετέρου (ii) με μεταβλητές συνθήκης.

Ο πηγαίος κώδικας για κάθε υλοποίηση φαίνεται παρακάτω:

1. Με Semaphores

```
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <errno.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

#define perror_pthread(ret, msg) \                                //macro is defined to
handle errors related to pthread functions.
do{ errno=ret; perror(msg); } while(0)

int safe_atoi(char *s, int *val){                                //The safe_atoi
function converts a string to an integer.

    long l;
    char *endp;

    // Convert string to long
    l = strtol(s, &endp, 10);
```

```

    // Check if conversion was successful and there is no extra characters
    if (s != endp && *endp == '\\0') {
        *val = 1;
        return 0; // Success
    }
    else
        return -1; // Error
}

```

`void *safe_malloc(size_t size){` //The safe_malloc function is a wrapper for the malloc function. It allocates memory of the specified size and checks if the allocation was successful.

```

    void *p;

    // Allocate memory
    if ((p = malloc(size)) == NULL) {
        fprintf(stderr, "Out of memory, failed to allocate %zd bytes\\n",
            size);
        exit(1);
    }

    return p;
}

```

//Output at the terminal is is x_chars wide by y_chars long.

```

int y_chars = 50;
int x_chars = 90;

```

```

/*
 * The part of the complex plane to be drawn:
 * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin).
 */

```

```

double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

```

```

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */

double xstep;
double ystep;

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */

void compute_mandel_line(int line, int color_val[]){

    /*
     * x and y traverse the complex plane.
     */

    double x, y;

    int n;
    int val;

    // Find out the y value corresponding to this line
    y = ymax - ystep * line;

    // and iterate for all points on this line
    for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

        //Compute the point's color value

        val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
        if (val > 255){
            val = 255;
        }
    }
}

```

```

        //And store it in the color_val[] array

        val = xterm_color(val);
        color_val[n] = val;
    }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */

void output_mandel_line(int fd, int color_val[]){

    int i;

    char point = '@';      // Character representing a point in the
Mandelbrot set
    char newline = '\n';    // Newline character

    for (i = 0; i < x_chars; i++) {

        //Set the current color, then output the point

        set_xterm_color(fd, color_val[i]);    // Set the color for the
current point

        if (write(fd, &point, 1) != 1) { // Write the point to the file
descriptor
            perror("compute_and_output_mandel_line: write point");
            exit(1);
        }
    }

    //Now that the line is done, output a newline character

    if (write(fd, &newline, 1) != 1) {        // Write the newline
character to the file descriptor

```



```

        perror("compute_and_output_mandel_line: write newline");
        exit(1);
    }
}

/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt is not
 * drawn in a funny colour if the user "terminates" the execution with Ctrl-C.
 */

void signal_handler(int sign){

    reset_xterm_color(1); // Reset the color of the xterm
    exit(1);
}

sem_t *mySem; //globalize the address so it can be visible from all threads
int thread_count; //globalize the thread_count so it can be visible from all
threads

void *compute_and_output_mandel_line(void *current_thread){

    int line, color_val[x_chars];          //color_val is not shared among
threads

    for(line=(int)current_thread; line<y_chars; line+=thread_count) {

        compute_mandel_line(line, color_val);          //
Perform the computation for the current line (parallel computation)
        if(sem_wait(&mySem[(int)current_thread])<0) {          // Wait
for my turn to output by waiting on my semaphore
            perror("sem_wait");
            exit(1);
        }

        output_mandel_line(1, color_val);          // Critical
section: Output the computed line
        if(sem_post(&mySem[((int)current_thread+1)%thread_count])<0) { //

```

```

Unlock the next thread to output (circularly)
    perror("sem_post");
    exit(1);
}
}

return NULL;
}

int main(int argc, char **argv){ //Main thread

    int line, ret;

    // compute the step size for each pixel

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

    //Check if arguments are OK

    if((argc!=2)|| (safe_atoi(argv[1], &thread_count)<0)|| (thread_count<=0))
{
        fprintf(stderr, "Usage: %s thread_count\n\n"
            "Exactly one argument required:\n"
            "    thread_count: The number of threads to create.\n",
            argv[0]);
        exit(1);
    }

    //signal handling

    struct sigaction sa;
    sa.sa_handler=signal_handler;    //set the function to call when a signal
is received
    sa.sa_flags=0;
    sigemptyset(&sa.sa_mask);
    if(sigaction(SIGINT, &sa, NULL)<0) { //register the signal handler for
SIGINT

```

```

        perror("sigaction");
        exit(1);
    }

    mySem=(sem_t*)safe_malloc(thread_count*sizeof(sem_t)); //allocate memory
for the semaphores

                                                    //we use one semaphore per thread

    if(sem_init(&mySem[0], 0, 1)<0) {        //initialize first semaphore to
value 1 (unlocked)
        perror("sem_init");                //(thread[0] outputs first the 0 line)
        exit(1);
    }
    for(line=1; line<thread_count; line++) { //initialize the other
semaphores to value 0 (locked)
        if(sem_init(&mySem[line], 0, 0)<0) {
            perror("sem_init");
            exit(1);
        }
    }

    //create threads

    pthread_t thread[thread_count];
    for(line=0; line<thread_count; line++) { //create the threads and pass
the thread index as an argument
        ret=pthread_create(&thread[line], NULL,
compute_and_output_mandel_line, (void*)line);
        if(ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }

    /*
    * wait for all threads to terminate
    */

```

```

    for(line=0; line<thread_count; line++) {
        ret=pthread_join(thread[line], NULL);
        if(ret)
            perror_pthread(ret, "pthread_join");
    }

    /*
     * destroy the semaphores
     */
    for(line=0; line<thread_count; line++) { //destroy all the semaphores
        if(sem_destroy(&mySem[line])<0) {
            perror("sem_destroy");
            exit(1);
        }
    }

    free(mySem); //free the semaphore memory

    reset_xterm_color(1); //reset the terminal color
    return 0;
}

```

2. Me Condition Variables

```

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include <errno.h>

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

```

```

#define perror_pthread(ret, msg) \                                //macro is defined to
handle errors related to pthread functions.
    do{ errno=ret; perror(msg); } while(0)

int safe_atoi(char *s, int *val){                                //The safe_atoi
function converts a string to an integer.

    long l;
    char *endp;

    // Convert string to long
    l = strtol(s, &endp, 10);

    // Check if conversion was successful and there is no extra characters
    if (s != endp && *endp == '\0') {
        *val = l;
        return 0; // Success
    }
    else
        return -1; // Error
}

void *safe_malloc(size_t size){                                //The safe_malloc
function is a wrapper for the malloc function. It allocates memory of the
specified size and checks if the allocation was successful.

    void *p;

    // Allocate memory
    if ((p = malloc(size)) == NULL) {
        fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
            size);
        exit(1);
    }

    return p;
}

```

```

//Output at the terminal is is x_chars wide by y_chars long.

int y_chars = 50;
int x_chars = 90;

/*
 * The part of the complex plane to be drawn:
 * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin).
 */

double xmin = -1.8, xmax = 1.0;
double ymin = -1.0, ymax = 1.0;

/*
 * Every character in the final output is
 * xstep x ystep units wide on the complex plane.
 */

double xstep;
double ystep;

/*
 * This function computes a line of output
 * as an array of x_char color values.
 */

void compute_mandel_line(int line, int color_val[]){

    /*
     * x and y traverse the complex plane.
     */

    double x, y;

    int n;
    int val;

    // Find out the y value corresponding to this line

```

```

    y = ymax - ystep * line;

    // and iterate for all points on this line
    for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

        //Compute the point's color value

        val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
        if (val > 255){
            val = 255;
        }

        //And store it in the color_val[] array

        val = xterm_color(val);
        color_val[n] = val;
    }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */

void output_mandel_line(int fd, int color_val[]){

    int i;

    char point = '@';      // Character representing a point in the
Mandelbrot set
    char newline='\n';      // Newline character

    for (i = 0; i < x_chars; i++) {

        //Set the current color, then output the point

        set_xterm_color(fd, color_val[i]);    // Set the color for the
current point

```

```

        if (write(fd, &point, 1) != 1) { // Write the point to the file
descriptor
            perror("compute_and_output_mandel_line: write point");
            exit(1);
        }
    }

    //Now that the line is done, output a newline character

    if (write(fd, &newline, 1) != 1) { // Write the newline
character to the file descriptor
        perror("compute_and_output_mandel_line: write newline");
        exit(1);
    }
}

/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt is not
 * drawn in a funny colour if the user "terminates" the execution with Ctrl-C.
 */

void signal_handler(int sign){

    reset_xterm_color(1); // Reset the color of the xterm
    exit(1);
}

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; // Initialize mutex
variable
pthread_cond_t cond = PTHREAD_COND_INITIALIZER; // Initialize condition
variable
int thread_count; // Total number of threads
int next_thread = 0; // Index of the next thread to
execute

void *compute_and_output_mandel_line(void *current_thread){

```



```

    int line, color_val[x_chars];    // Array to store color values for each
pixel in a line

    for (line = (int)current_thread; line < y_chars; line += thread_count) {

        compute_mandel_line(line, color_val);    // Compute the Mandelbrot
line for the current thread

        pthread_mutex_lock(&mutex);    // Acquire the lock on the mutex

        // Wait until it's the turn of the current thread to execute
        while ((int)current_thread != next_thread) {
            pthread_cond_wait(&cond, &mutex);    // Release the lock and wait
for a signal
        }

        output_mandel_line(1, color_val);    // Output the computed
Mandelbrot line
        next_thread = (next_thread + 1) % thread_count;    // Update the
index of the next thread
        pthread_cond_broadcast(&cond);    // Signal all threads
that they can check the condition again
        pthread_mutex_unlock(&mutex);    // Release the lock on
the mutex
    }

    return NULL; // Exit the thread
}

int main(int argc, char **argv){

    int line, ret;

    xstep = (xmax - xmin) / x_chars;    // Calculate the step size for the
x-axis

```

```

    ystep = (ymax - ymin) / y_chars;           // Calculate the step size for the
y-axis

    // Check command-line arguments and thread count
    if((argc!=2)|| (safe_atoi(argv[1], &thread_count)<0)|| (thread_count<=0)) {
        fprintf(stderr, "Usage: %s thread_count\n\n"
            "Exactly one argument required:\n"
            "    thread_count: The number of threads to create.\n",
            argv[0]);
        exit(1);
    }

    struct sigaction sa;
    sa.sa_handler=signal_handler;               // Set the signal handler function
    sa.sa_flags=0;
    sigemptyset(&sa.sa_mask);
    if(sigaction(SIGINT, &sa, NULL)<0) {        // Set up the signal handler
for SIGINT (Ctrl+C)
        perror("sigaction");
        exit(1);
    }

    pthread_t thread[thread_count];             // Array to hold thread
identifiers

    for(line=0; line<thread_count; line++) {
        ret=pthread_create(&thread[line], NULL,
compute_and_output_mandel_line, (void*)line);
        if(ret) {
            perror_pthread(ret, "pthread_create"); // Print an error
message if thread creation fails
            exit(1);
        }
    }

    for(line=0; line<thread_count; line++) {
        ret=pthread_join(thread[line], NULL);    // Wait for each thread
to finish its execution
    }

```

```

        if(ret)
            perror_thread(ret, "pthread_join");          // Print an error
message if thread joining fails
        }

        pthread_mutex_destroy(&mutex);                  // Destroy the mutex variable
        pthread_cond_destroy(&cond);                    // Destroy the condition variable

        reset_xterm_color(1); // Reset the xterm color settings
        return 0;
    }
}

```

Η λειτουργία των προγραμμάτων φαίνεται από τα επεξηγηματικά σχόλια που περιέχουν.

Επιπλέον το Makefile που έχουμε φτιάξει για την ταυτόχρονη μεταγλώτιση είναι το εξής:

```

CC = gcc
CFLAGS = -Wno-pointer-to-int-cast -Wno-int-to-pointer-cast -O2 -pthread

.PHONY: all clean
all: mandel_sem mandel_cond
mandel_sem: mandel-lib.o mandel_sem.o
    $(CC) $(CFLAGS) -o mandel_sem mandel-lib.o mandel_sem.o

mandel_cond: mandel-lib.o mandel_cond.o
    $(CC) $(CFLAGS) -o mandel_cond mandel-lib.o mandel_cond.o

mandel-lib.o: mandel-lib.h mandel-lib.c
    $(CC) $(CFLAGS) -c -o mandel-lib.o mandel-lib.c

mandel_sem.o: mandel_sem.c mandel-lib.h
    $(CC) $(CFLAGS) -c -o mandel_sem.o mandel_sem.c

mandel_cond.o: mandel_cond.c mandel-lib.h
    $(CC) $(CFLAGS) -c -o mandel_cond.o mandel_cond.c

clean:
    rm -f mandel_sem.o mandel_sem mandel-lib.o mandel_cond.o mandel_cond

```


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

Το πρώτο πρόγραμμα χρησιμοποιεί μια παράλληλη υπολογιστική προσέγγιση για τη δημιουργία και την έξοδο του Mandelbrot set. Δημιουργεί πολλαπλά νήματα, με κάθε νήμα υπεύθυνο για τον υπολογισμό και την έξοδο ενός υποσυνόλου γραμμών στην τελική έξοδο. Ο αριθμός των νημάτων που δημιουργούνται καθορίζεται από τη μεταβλητή `thread_count`.

Για να διασφαλιστεί ότι η έξοδος είναι συγχρονισμένη και οι γραμμές εξάγονται με τη σωστή σειρά, το πρόγραμμα χρησιμοποιεί σημαφόρους για συντονισμό μεταξύ των νημάτων. Κάθε νήμα έχει τον δικό του σημαφόρο, που αντιπροσωπεύεται από τον πίνακα `mySem`.

Ακολουθεί μια ανάλυση του σχήματος συγχρονισμού:

1. Semaphore Initialization:

- Το πρόγραμμα αρχικοποιεί τον πρώτο σηματοφόρο (`mySem[0]`) με αρχική τιμή 1 (unlocked), χρησιμοποιώντας τη `sem_init()`.
- Για τους υπόλοιπους σηματοφόρους (`mySem[1]` έως `mySem[thread_count-1]`), αρχικοποιούνται με αρχική τιμή 0 (locked) χρησιμοποιώντας `sem_init()` σε έναν βρόχο.

2. Δημιουργία νήματος:

- Το πρόγραμμα δημιουργεί `thread_count` αριθμό νημάτων χρησιμοποιώντας το `pthread_create()`. Σε κάθε νήμα εκχωρείται ένα `index` από το 0 στο `thread_count-1`.

3. Συνάρτηση νήματος (`compute_and_output_mandel_line`):

- Κάθε νήμα εκτελεί τη συνάρτηση `compute_and_output_mandel_line`, η οποία υπολογίζει και εξάγει ένα υποσύνολο γραμμών στην τελική έξοδο.
- Το νήμα επαναλαμβάνεται πάνω από τις γραμμές, ξεκινώντας από τον εκχωρημένο δείκτη του και αυξάνοντας κατά `thread_count`.
- Για κάθε γραμμή, υπολογίζει το σύνολο Mandelbrot χρησιμοποιώντας `compute_mandel_line()` και αποθηκεύει τις τιμές χρώματος στον πίνακα `color_val`.
- Στη συνέχεια περιμένει να βγει η σειρά του καλώντας `sem_wait()` στον αντίστοιχο σηματοφόρο του (`mySem[(int)current_thread]`).
- Μετά την απόκτηση του σηματοφόρου, βγάζει την υπολογισμένη γραμμή χρησιμοποιώντας `output_mandel_line()`.
- Τέλος, ξεκλειδώνει το επόμενο νήμα προς έξοδο καλώντας τη `sem_post()` στον σηματοφόρο του επόμενου νήματος (`mySem[((int)current_thread+1)%thread_count]`).

4. Σύνδεση νημάτων:

- Μετά τη δημιουργία όλων των νημάτων, το κύριο νήμα περιμένει να ολοκληρωθεί κάθε νήμα χρησιμοποιώντας το `pthread_join()`.

5. Καταστροφή σηματοφόρου:

- Μόλις ολοκληρωθούν όλα τα νήματα, το πρόγραμμα καταστρέφει όλους τους σηματοφόρους σε έναν βρόχο χρησιμοποιώντας το `sem_destroy()`.

Ο υπολογισμός κατανέμεται σε N νήματα εκτέλεσης. Η κατανομή του υπολογιστικού φόρτου γίνεται ανά γραμμές: Για N νήματα, το i -οστό (με $i=0, 1, 2, \dots, N-1$) αναλαμβάνει τις γραμμές $i, i+N, i+2N, \dots$ κλπ. Η εκτύπωση των γραμμών απαιτεί τον συγχρονισμό των νημάτων, έτσι ώστε αυτές να εμφανίζονται με την σωστή σειρά. Επομένως, ορίζουμε το output των γραμμών ως κρίσιμο τμήμα, όπως φαίνεται και στον κώδικα. Το πολύ ένα thread μπορεί να είναι στο κρίσιμο τμήμα μία χρονική στιγμή, οπότε η εκτύπωση των γραμμών γίνεται ατομικά. Ο υπολογισμός των γραμμών δεν βρίσκεται μέσα στο κρίσιμο τμήμα και επομένως είναι παράλληλος, όπως και θέλαμε να πετύχουμε. Για τον συγχρονισμό χρησιμοποιούμε τόσους σηματοφόρους όσα και τα νήματα εκτέλεσης που έχουμε στο πρόγραμμά μας.

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.

Τώρα, θα συγκρίνουμε τους χρόνους εκτέλεσης του σειριακού και του παράλληλου προγράμματος. Οι μετρήσεις γίνονται στο δικό μας μηχάνημα με τα παρακάτω specs.

```
oslab33@orion:~/ex3/3.2$ cat /proc/cpuinfo
processor       : 0
vendor_id      : AuthenticAMD
cpu family     : 6
model          : 2
model name     : QEMU Virtual CPU version 1.0
stepping       : 3
microcode      : 0x1000065
cpu MHz        : 2400.028
cache size     : 512 KB
physical id    : 0
siblings       : 1
core id        : 0
cpu cores      : 1
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 4
wp             : yes
```

- Σειριακός υπολογισμός (1 νήμα εκτέλεσης)

```
time ./mandel sem 1
```

```
real    0m5.153s
user    0m1.456s
sys     0m0.040s
```

- Παράλληλος υπολογισμός (2 νήματα εκτέλεσης)

```
time ./mandel sem 2
```

```
real    0m1.437s
user    0m0.744s
sys     0m0.032s
oslab33@orion:~/ex3/3.2$
```

- Παράλληλος υπολογισμός (11 νήματα εκτέλεσης)

```
time ./mandel sem 11
```

```
real    0m0.909s
user    0m1.068s
sys     0m0.008s
oslab33@orion:~/ex3/3.2$
```

Παρατηρούμε τα εξής:

Ο χρόνος εκτέλεσης του σειριακού υπολογισμού είναι σχεδόν τετραπλάσιος σε σχέση με τον χρόνο εκτέλεσης του παράλληλου υπολογισμού με 2 νήματα. Αυτό είναι λογικό, αφού γενικά όταν ένα πρόγραμμα εκτελείται με πολλά νήματα, ο φόρτος εργασίας μπορεί να διαιρεθεί μεταξύ των νημάτων, επιτρέποντας την ταυτόχρονη εκτέλεση πολλαπλών υπολογισμών.

Αυξάνοντας τον αριθμό των νημάτων εκτέλεσης μέχρι κάποιο σημείο, παρατηρούμε κάποια μετρήσιμη διαφορά στον χρόνο εκτέλεσης, ενώ από εκεί και πέρα η διαφορά είναι αμελητέα.

Βέβαια, αυξάνοντας κατά πολύ τον αριθμό των νημάτων, παρατηρείται ολοένα και μεγαλύτερη αύξηση στον χρόνο εκτέλεσης. Αυτό οφείλεται στο γεγονός ότι γενικά δεν είναι αμελητέο το

“κόστος” δημιουργίας των νημάτων, όπως και επίσης θα είναι μεγάλος ο χρόνος που αναμένει το main-thread για τον τερματισμό όλων των νημάτων με την pthread_join().

Επιπλέον σε θεωρητικό επίπεδο, καλύτερος χρόνος εκτέλεσης θα είναι για 50 νήματα, όσες είναι δηλαδή και οι γραμμές που εκτυπώνονται από το πρόγραμμα.

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

Στη δεύτερη έκδοση του προγράμματος, χρησιμοποιείται μόνο μία μεταβλητή συνθήκης, η οποία είναι η «cond». Χρησιμοποιείται σε συνδυασμό με τη μεταβλητή mutex «mutex» για συγχρονισμό μεταξύ των νημάτων.

Ο συγχρονισμός λειτουργεί ως εξής:

1. Κάθε νήμα αποκτά το κλείδωμα mutex (`pthread_mutex_lock(&mutex)`) πριν αποκτήσει πρόσβαση στους κοινόχρηστους πόρους ή στο κρίσιμο τμήμα.
2. Όταν ένα νήμα ολοκληρώσει τον υπολογισμό του και την έξοδο μιας γραμμής Mandelbrot, ελέγχει εάν είναι το επόμενο νήμα στη γραμμή που θα εκτελεστεί συγκρίνοντας τον δείκτη νήματος του (`current_thread`) με το `next_thread`.
3. Εάν το `index` του νήματος δεν είναι ίσο με το `next_thread`, απελευθερώνει το κλείδωμα mutex και περιμένει (`pthread_cond_wait(&cond, &mutex)`) για να μεταδοθεί ένα σήμα από το νήμα που εκτελείται αυτήν τη στιγμή.
4. Μόλις το νήμα λάβει το σήμα, αποκτά ξανά το κλείδωμα mutex και ελέγχει ξανά εάν είναι το επόμενο νήμα που θα εκτελεστεί. Αν όχι, επιστρέφει στην αναμονή.
5. Όταν ένα νήμα καθορίζει ότι είναι το επόμενο νήμα προς εκτέλεση, εκτελεί την έξοδο και ενημερώνει το «`next_thread`» στο ευρετήριο του επόμενου νήματος στη σειρά. Στη συνέχεια εκπέμπει ένα σήμα (`pthread_cond_broadcast(&cond)`) για να ξυπνήσει όλα τα νήματα σε αναμονή.
6. Τέλος, το νήμα απελευθερώνει το κλείδωμα mutex (`pthread_mutex_unlock(&mutex)`) πριν από την έξοδο.

Το πρόβλημα απόδοσης σε αυτόν τον κώδικα προκύπτει λόγω της χρήσης απασχολημένης αναμονής ή περιστροφής. Όταν ένα νήμα περιμένει τη σειρά του να εκτελεστεί, ελέγχει συνεχώς τη συνθήκη σε έναν βρόχο (`while ((int)current_thread != next_thread)`). Αυτό έχει ως αποτέλεσμα το νήμα να καταναλώνει κύκλους της CPU ακόμα και όταν δεν έχει κάτι να κάνει οδηγώντας σε περιττή σπατάλη πόρων και μειωμένη συνολική απόδοση του προγράμματος.

```
time ./mandel_cond 2
```



```
real    0m2.485s
user    0m1.244s
sys     0m0.024s
oslab33@orion:~/ex3/3.2$
```

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

Γενικά, τα παράλληλα προγράμματα μπορούν να εμφανίσουν επιτάχυνση όταν εκτελούνται σε πολλούς επεξεργαστές ή πυρήνες. Η επιτάχυνση, είναι ένα μέτρο του πόσο πιο γρήγορα εκτελεί ένα παράλληλο πρόγραμμα σε σύγκριση με τη σειριακή του εκδοχή.

Όσον αφορά το σχήμα συγχρονισμού, το μέγεθος του κρίσιμου τμήματος είναι ένας σημαντικός παράγοντας που πρέπει να ληφθεί υπόψη. Εάν το κρίσιμο τμήμα είναι πολύ μεγάλο, μπορεί να οδηγήσει σε αυξημένο synchronization overhead, οδηγώντας σε υποβάθμιση της απόδοσης. Επιπλέον, εάν το κρίσιμο τμήμα περιλαμβάνει τόσο τις φάσεις υπολογισμού όσο και τις φάσεις εξόδου, μπορεί να οδηγήσει σε αυξημένη διαμάχη για τους πόρους εξόδου, επιδεινώνοντας περαιτέρω τα γενικά έξοδα συγχρονισμού.

Στους δεδομένους κώδικες, το κρίσιμο τμήμα είναι το τμήμα του κώδικα όπου καλείται η συνάρτηση `output_mandel_line()`. Συγκεκριμένα, αυτή η ενότητα βρίσκεται στη συνάρτηση `compute_and_output_mandel_line()`:

```
output_mandel_line(1, color_val); // Critical section: Output the
computed line
```

Κατά την παράλληλη εκτέλεση του προγράμματος, πολλαπλά νήματα εκτελούνται ταυτόχρονα, καθένα από τα οποία είναι υπεύθυνο για τον υπολογισμό και την έξοδο μιας συγκεκριμένης γραμμής του συνόλου Mandelbrot. Η κρίσιμη ενότητα διασφαλίζει ότι μόνο ένα νήμα μπορεί να έχει πρόσβαση στη συνάρτηση `output_mandel_line()` κάθε φορά για να αποτρέψει τις διενέξεις εγγραφής στην έξοδο.

Το κρίσιμο τμήμα προστατεύει τον κοινόχρηστο πόρο εξόδου, ο οποίος είναι το τερματικό όπου εμφανίζεται το σύνολο Mandelbrot. Χωρίς τον κατάλληλο συγχρονισμό, πολλαπλά νήματα μπορεί να επιχειρήσουν να γράψουν στο τερματικό ταυτόχρονα, με αποτέλεσμα μπερδεμένη ή εσφαλμένη έξοδο.

Στον πρώτο κώδικα, ένας σηματοφόρος (`mySem`) χρησιμοποιείται για τον έλεγχο της πρόσβασης στο κρίσιμο τμήμα. Κάθε νήμα περιμένει στον δικό του σηματοφόρο πριν εισέλθει

Επομένως και στις δύο υλοποιήσεις, υπάρχει επιτάχυνση (φαίνεται και από την εκτέλεση των προγραμμάτων, όταν συγκρίνουμε την σειριακή και την παράλληλη υλοποίηση). Το κρίσιμο τμήμα του κώδικα είναι όσο το δυνατόν πιο μικρό και εμπεριέχει μόνο το κομμάτι της εξόδου και όχι του υπολογισμού της κάθε γραμμής.

Παραπάνω και στους δύο κώδικες, έχουμε ορίσει έναν signal handler για το σήμα SIGINT (Ctrl-C). Έστω για το πρόγραμμα mandel_cond.c, αν αφαιρέσουμε αυτό το κομμάτι του κώδικα και πατήσουμε Ctrl-C κατά την εκτέλεση του προγράμματος θα έχουμε το εξής:

[illegible]

Παρατηρούμε λοιπόν ότι το χρώμα των γραμμμάτων στο τερματικό μένει το ίδιο χρώμα με το χρώμα του τελευταίου χαρακτήρα που τυπώθηκε (από τις συναρτήσεις `set_xterm_color()` και `output_mandel_line()`). Αν το πρόγραμμα εκτελεστεί χωρίς διακοπή, τότε το χρώμα των γραμμμάτων επανέρχεται στο default, λόγω της συνάρτησης `reset_xterm_color()` που καλείται στο τέλος της `main` πριν το `return`.


```

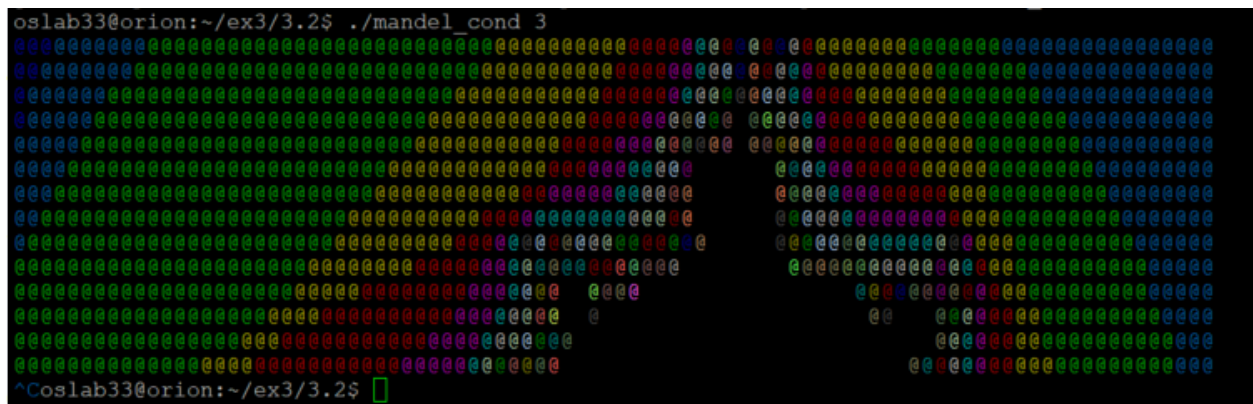
/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt
 is not
 * drawn in a funny colour if the user "terminates" the execution
 with Ctrl-C.
 */

void signal_handler(int sign){

    reset_xterm_color(1); // Reset the color of the xterm
    exit(1);
}

```

Επομένως τώρα με αυτό το κομμάτι κώδικα, αν ξαναδιακόψουμε την εκτέλεση του προγράμματος μας με Ctrl-C, λαμβάνουμε το παρακάτω:



```

oslab33@orion:~/ex3/3.2$ ./mandel_cond 3
^C
oslab33@orion:~/ex3/3.2$

```

Παρατηρούμε δηλαδή, ότι έχει γίνει reset του χρώματος όπως έχουμε ορίσει δηλαδή στην void signal_handler().