

Λειτουργικά Συστήματα
6ο Εξάμηνο
Άσκηση 3: Διαχείριση Διεργασιών και Διαδιεργασιακή Επικοινωνία
Αναφορά

Χαρδούβελης Γεώργιος-Ορέστης
el15100
6ο Εξάμηνο

Κυριάκου Αθηνά
el17405
6ο Εξάμηνο

Άσκηση 1:

Στο 1ο μέρος της εργαστηριακής άσκησης, ζητήθηκε ο συγχρονισμός 2 νημάτων σε υπάρχοντα κώδικα τα οποία μεταβάλλουν την τιμή της κοινής τους μεταβλητής `val`. Η μεταβλητή `val` αρχικοποιείται στο 0 και το ένα νήμα αυξάνει την τιμή της `N` φορές κατά 1 κάθε φορά που εκτελείται ενώ το άλλο τη μειώνει `N` φορές. Αν επιτυγχάνεται ο συγχρονισμός, μετά την εκτέλεση του προγράμματος θα πρέπει να ισχύει `val=0`.

Τρέχοντας το αρχικό πρόγραμμα χωρίς συγχρονισμό παρατηρούμε ότι:

```
oslabd14@os-node1:~/lab3$ ./simplesyncinitial
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -5350176.
oslabd14@os-node1:~/lab3$
```

Και άρα ο συγχρονισμός δεν είναι επιτυχημένος εφόσον `val!=0`.

Ζητήθηκε το συγχρονισμός του πηγαίου κώδικα στο αρχείο **`simplesync.c`** με δύο τρόπους: με χρήση κλειδωμάτων (POSIX mutexes) και ατομικών λειτουργιών του GCC. Με τον πρώτο τρόπο προκύπτει το εκτελέσιμο **`simplesync-mutex`** ενώ με τον δεύτερο το **`simplesync-atomic`**.

Και τα δύο προκύπτουν από το ίδιο αρχείο πηγαίου κώδικα `simplesync.c` και αυτό επιτυγχάνεται με τη χρήση της σημαίας `-DSYNC` στο Makefile κατά το compilation του αρχείου `simplesync.c` για τη δημιουργία των δύο object files `.o`. Για να προκύψει το mutex αρχείο χρησιμοποιείται το flag `-DSYNC_MUTEX` ενώ για το atomic το `-DSYNC_ATOMIC` τα οποία προσδιορίζουν ποιο κομμάτι κώδικα θα εκτελεστεί στο κοινό αρχείο πηγαίου μέσω της boolean μεταβλητής `USE_ATOMIC_OPS`.

```

# Makefile
#

CC = gcc

# CAUTION: Always use '-pthread' when compiling POSIX threads-based
# applications, instead of linking with "-lpthread" directly.
CFLAGS = -Wall -O2 -pthread
LIBS =

all: pthread-test simplesync-mutex simplesync-atomic kgarten mandel simplesyncinitial

## Pthread test
pthread-test: pthread-test.o
$(CC) $(CFLAGS) -o pthread-test pthread-test.o $(LIBS)

pthread-test.o: pthread-test.c
$(CC) $(CFLAGS) -c -o pthread-test.o pthread-test.c

## 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)

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

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

```

Ο πηγαίος κώδικας του αρχείου simplesync.c όπως τροποποιήθηκε για να επιτευχθεί ο συγχρονισμός φαίνεται παρακάτω. Η χρήση mutexes γίνεται όταν δεν αληθεύει η boolean μεταβλητή USE_ATOMIC_OPS.

```

/*
 * 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;
#define NUM 1

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

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

    return NULL;
}

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

```

```

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

    return NULL;
}

```

```

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

    /*
     * Initial value
     */
    val = 0;

    //create mutex
    if(pthread_mutex_init(&lock, NULL) != 0){
        printf("Mutex init failed!\n");
        exit(1);
    }

    /*
     * 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");

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

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

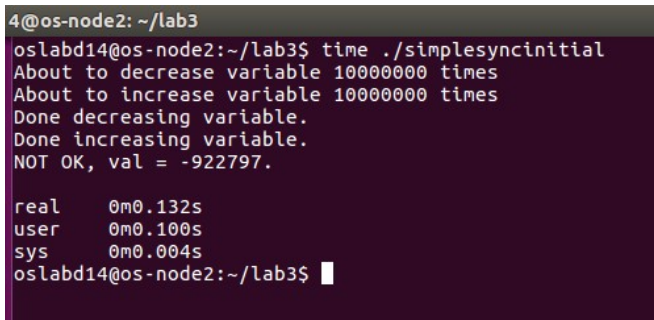
return ok;
}

```

Ερωτήσεις 1.1:

1. Για τη σύγκριση του χρόνου εκτέλεσης των εκτελέσιμων χρησιμοποιήθηκε η εντολή `time(1)`.

Για το εκτελέσιμο χωρίς συγχρονισμό έχουμε:



```

4@os-node2: ~/lab3
oslabd14@os-node2:~/lab3$ time ./simplesyncinitial
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -922797.

real    0m0.132s
user    0m0.100s
sys     0m0.004s
oslabd14@os-node2:~/lab3$

```

Για το εκτελέσιμο με συγχρονισμό με χρήση κλειδωμάτων (mutexes):

```

oslabd14@os-node2:~/lab3$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m1.371s
user    0m1.056s
sys     0m0.108s
oslabd14@os-node2:~/lab3$

```

Και για αυτό με χρήση ατομικών λειτουργιών:

```

oslabd14@os-node2:~/lab3$ time ./simplesync-atomic
About to decrease variable 10000000 times
About to increase variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m0.403s
user    0m0.388s
sys     0m0.000s
oslabd14@os-node2:~/lab3$

```

Από τους παραπάνω υπολογισμούς, παρατηρείται ότι το εκτελέσιμο χωρίς συγχρονισμό απαιτεί τον λιγότερο πραγματικό χρόνο καθώς για την αύξηση ή τη μείωση της μεταβλητής val καλεί μόνο την αντίστοιχη συνάρτηση. Αυτό είναι λογικό αφού όταν έχουμε συγχρονισμό παρατηρούνται κλειδώματα του κρίσιμο σημείο στον κώδικα.

Συγκεκριμένα, το εκτελέσιμο με συγχρονισμό με χρήση mutexes απαιτεί τον περισσότερο πραγματικό χρόνο καθώς πριν από την αύξηση ή τη μείωση της τιμής της μεταβλητής πρέπει να πραγματοποιήσει το κλείδωμα και το ξεκλείδωμα.

Ο συγχρονισμός με ατομικές λειτουργίες απαιτεί λιγότερο χρόνο από αυτόν με mutexes γιατί καλείται μόνο μία built-in συνάρτηση του GCC για αύξηση ή μείωση. Ωστόσο, απαιτεί περισσότερο χρόνο σε σχέση με το εκτελέσιμο χωρίς συγχρονισμό διότι οι συναρτήσεις αυτές κλειδώνουν όλο το pipeline δημιουργώντας full memory barrier με αποτέλεσμα να μην εκτελούνται άλλες εντολές. Οι συναρτήσεις δηλαδή αυτές εκτελούνται ατομικά.

2. Βάσει της παραπάνω αιτιολόγησης, γρηγορότερη είναι η χρήση ατομικών λειτουργιών για τον συγχρονισμό σε σχέση με τη χρήση mutexes.

3. Χρησιμοποιώντας την εντολή `gcc -O2 -S -pthread -DSYNC_ATOMIC -c simplesync.c`, παράγεται ο κώδικας assembly του `simpesync-atomic` προγράμματος στο αρχείο `simplesync.s`. Παρακάτω βλέπουμε το κομμάτι στο οποίο μεταφράζεται η χρήση ατομικών λειτουργιών, συγκεκριμένα για την `__sync_add_and_fetch`.

```

increase_fn:
.LFB22:
    .cfi_startproc
    pushq    %rbx
    .cfi_def_cfa_offset 16
    .cfi_offset 3, -16
    movq     %rdi, %rbx
    movq     stderr(%rip), %rdi
    movl     $10000000, %edx
    movl     $.LC0, %esi
    xorl     %eax, %eax
    call     fprintf
    movl     $10000000, %eax
    .p2align 4,,10
    .p2align 3
.L2:
    lock addl    $1, (%rbx)
    subl     $1, %eax
    jne      .L2
    movq     stderr(%rip), %rcx
    movl     $26, %edx
    movl     $1, %esi
    movl     $.LC1, %edi
    call     fwrite
    xorl     %eax, %eax
    popq     %rbx
    .cfi_def_cfa_offset 8
    ret

```

Αρα βλέπουμε πως το μόνο που αλλάζει από την υλοποίηση δίχως συγχρονισμό είναι η εντολή `lock addl $1, (%rbx)`, με την οποία κλειδώνει το pipeline και εκτελείται μόνο η εντολή `add`.

4. Ομοίως με το παραπάνω παράδειγμα, εκτελούμε την εντολή `gcc -O2 -S -pthread -DSYNC_MUTEX -c simplesync.c` και στο αρχείο `simplesync.s` έχουμε πλέον τον κώδικα assembly του `simpesync-mutex` προγράμματος.

```

.cfi_def_cfa_offset 16
.cfi_offset 6, -16
pushq   %rbx
.cfi_def_cfa_offset 24
.cfi_offset 3, -24
movq    %rdi, %rbp
movl    $10000000, %edx
movl    $.LC0, %esi
xorl    %eax, %eax
subq    $8, %rsp
.cfi_def_cfa_offset 32
movq    stderr(%rip), %rdi
movl    $10000000, %ebx
call    fprintf
.p2align 4,,10
.p2align 3
.L2:
movl    $lock, %edi
call    pthread_mutex_lock
movl    0(%rbp), %eax
movl    $lock, %edi
addl    $1, %eax
movl    %eax, 0(%rbp)
call    pthread_mutex_unlock
subl    $1, %ebx
jne     .L2
movq    stderr(%rip), %rcx
movl    $26, %edx
movl    $1, %esi
movl    $.LC1, %edi
call    fwrite
addq    $8, %rsp
.cfi_def_cfa_offset 24
xorl    %eax, %eax
popq    %rbx

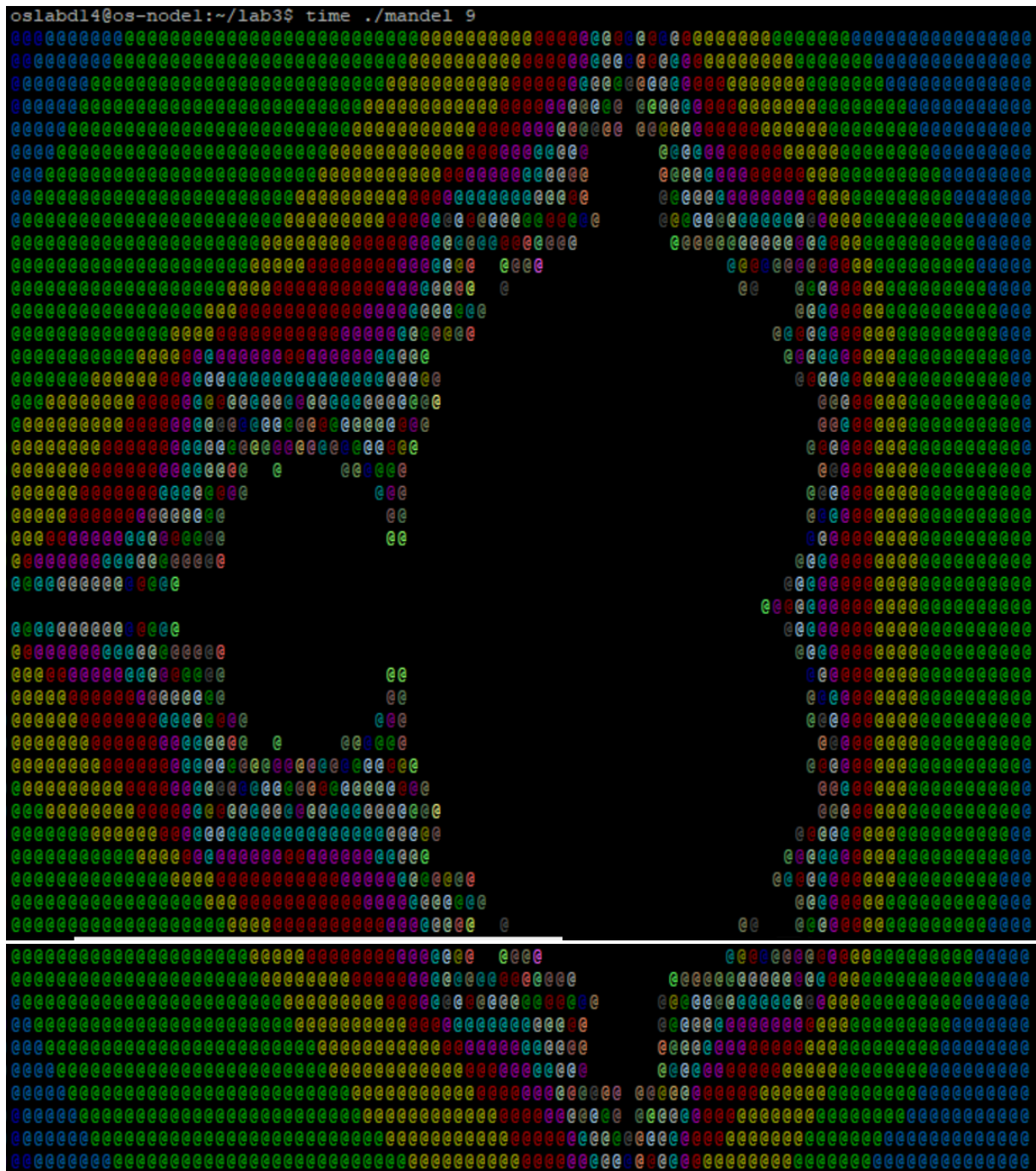
```

Εδώ έχουμε κάποιες παραπάνω διαφορές σε σχέση με τον κώδικα δίχως συγχρονισμό. Βλέπουμε πως οι συναρτήσεις `pthread_mutex_lock` και `pthread_mutex_unlock` καλούνται όπως και στον κώδικα σε C, όπου στην μεταβλητή `lock` έχουμε πλέον το περιεχόμενο του καταχωρητή `edi`.

Άσκηση 2:

Στη 2η άσκηση, ζητήθηκε η τροποποίηση του κώδικα `mandel.c` που εκτελείται σειριακά, έτσι ώστε να μπορεί να εκτελεστεί παράλληλα. Αυτό επιτεύχθηκε με τη χρήση `threads` και σημαφόρων ως μέθοδο συγχρονισμού έτσι ώστε να εκτυπωθεί στην οθόνη το ίδιο σύνολο Mandelbrot που προέκυψε με τον σειριακό κώδικα.

Με οποιοδήποτε όρισμα προκύπτει η ίδια έξοδος, το μόνο που αλλάζει όπως θα δούμε παρακάτω είναι ο χρόνος εκτέλεσης.



```
/*
 * mandel.c
```

```

*
* A program to draw the Mandelbrot Set on a 256-color xterm.
*
*/

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>

#include "mandel-lib.h"
#include <pthread.h>
#include <semaphore.h>

#define MANDEL_MAX_ITERATION 100000

/*****
 * Compile-time parameters *
 *****/

/*
 * Output at the terminal 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;

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

int NTHREADS;
sem_t *semarray;    //to make an array of semaphores

```

```

pthread_t *thrarray;

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 = '@';
    char newline = '\n';

    for (i = 0; i < x_chars; i++) {
        /* Set the current color, then output the point */
        set_xterm_color(fd, color_val[i]);
        if (write(fd, &point, 1) != 1) {
            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) {
        perror("compute_and_output_mandel_line: write newline");
        exit(1);
    }
}

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

    compute_mandel_line(line, color_val);

    sem_wait(&semarray[num_of_thread]);
    output_mandel_line(fd, color_val);
    sem_post(&semarray[(num_of_thread+1)%NTHREADS]);
}

void *mandel_line_aux(void *nthread){
    int numth=(int)nthread;
    int line;
    for(line=numth; line<y_chars; line=line+NTHREADS)
        compute_and_output_mandel_line(1,line,numth);
}

int main(int argc, char **argv) //serial to parellel, NTHREADS passed as argument in main
{
    int i,ret;

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

    NTHREADS=atoi(argv[1]);

    //malloc array of semaphores
    semarray=malloc(NTHREADS*sizeof(sem_t));

    sem_init(&semarray[0],0,1); //initialize the first semaphore
    for(i=1; i<NTHREADS; i++)
        sem_init(&semarray[i],0,0); //initialize the rest of the semaphores

    //malloc array of threads

```

```

thrarray=malloc(NTHREADS*sizeof(pthread_t));

//create the threads
for(i=0; i<NTHREADS; i++){
    pthread_create(&thrarray[i],NULL,mandel_line_aux,(void *)i);
    //we can add an error report
}

for(i=0; i<NTHREADS; i++){
    pthread_join(thrarray[i],NULL);
    //we can add an error report
}

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

reset_xterm_color(1);
return 0;
}

```

Ερωτήσεις 1.2:

1. Το πλήθος των σημαφόρων που χρησιμοποιήθηκαν είναι ένας για κάθε νήμα για το συγκεκριμένο σχήμα συγχρονισμού.
2. Για την εύρεση του χρόνου που απαιτείται για την εκτέλεση του σειριακού και του παράλληλου προγράμματος με 2 νήματα υπολογισμού, χρησιμοποιήθηκε και πάλι η εντολή `time(1)`.

Σε μηχάνημα με 4 πυρήνες, για το σειριακό πρόγραμμα, δηλαδή τρέχοντας το πρόγραμμά μας με ένα νήμα, έχουμε:

```

real    0m1.025s
user    0m0.968s
sys     0m0.028s

```

Και για το παράλληλο με 2 νήματα:

```

real    0m0.517s
user    0m0.984s
sys     0m0.016s

```

Από τα παραπάνω προκύπτει ότι ο καταμερισμός του υπολογισμού σε επιμέρους νήματα και η παράλληλη εκτέλεσή τους με συγχρονισμό απαιτεί λιγότερο χρόνο. Συγκεκριμένα για την εκτέλεση του σειριακού προγράμματος απαιτούνται 1.025 sec ενώ για την εκτέλεση του

παράλληλου προγράμματος 0.571 sec, μείωση σχεδόν στο μισό, αφού τη εκτέλεση του κρίσιμου σημείο, στο οποίο υπήρχε καθυστέρηση, μοιράζεται σε δύο νήματα.

3. Το πρόγραμμα εμφανίζει επιτάχυνση όσο αυξάνονται και τα νήματα, καθώς ο χρόνος αυξάνεται σχεδόν γραμμικά, τουλάχιστον μέχρι και τα 4 νήματα.

Το κρίσιμο τμήμα αφορά μόνο την φάση εξόδου κάθε γραμμής που παράγεται και όχι την φάση υπολογισμού, αφού η τελευταία πραγματοποιείται σε διαφορετικό -προωρινό- πίνακα που δημιουργείται κάθε φορά που καλείται η αντίστοιχη συνάρτηση.

Μετά τα 4 νήματα ο χρόνος μειώνεται με πολύ αργότερος ρυθμούς και από ένα σημείο και μετά μένει πρακτικά σταθερός. Αυτό συμβαίνει γιατί χρησιμοποιούμε μηχανήμα με 4 πυρήνες και, αν και έχουμε περισσότερα νήματα, κάθε πυρήνας μπορεί να εξυπηρετήσει ένα νήμα την φορά. Έτσι, αν και η φάση εξόδου κάθε γραμμής έχει χωριστεί σε παραπάνω νήματα κάθε φορά, την ίδια στιγμή εκτελείται ο ίδιος αριθμός νημάτων.

4. Αν πατήσουμε το πλήκτρο CTRL-C ενώ το πρόγραμμα εκτελείται, προκαλείται μία διακοπή υλικού. Αυτό έχει ως αποτέλεσμα το χρώμα της γραμματοσειράς του τερματικού μετά τη διακοπή να αφήνεται στο χρώμα που το εκάστοτε νήμα του προγράμματος τύπωνε στην οθόνη όταν πραγματοποιήθηκε η διακοπή.

Για να επανέρχεται το τερματικό στην κατάσταση που ήταν πριν γίνει η διακοπή, μπορούμε να ορίσουμε μια συνάρτηση handler για το χειρισμό της συγκεκριμένης διακοπής που όταν πατήσει ο χρήστης CTRL-C θα καλείται και θα εκτελεί την εντολή `reset_xterm_color(1)`; για την επαναφορά του τερματικού.