Λειτουργικά Συστήματα

3η Σειρά Ασκήσεων

Ακαδημαϊκό έτος 2020-2021

Άσκηση 1:

Η υλοποίηση του προγράμματος είναι η παρακάτω:

```c
#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.
 */
pthread_mutex_t mutex;

#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

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

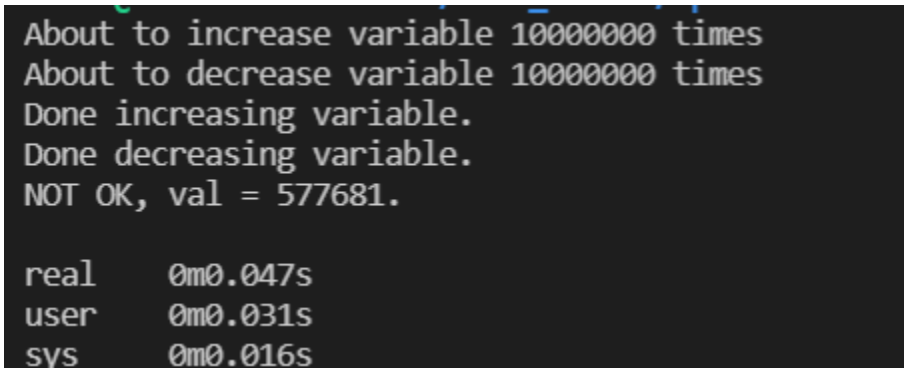
```
	fprintf(stderr, "About to increase variable %d times\n", N);
	for (i = 0; i < N; i++) {
		if (USE_ATOMIC_OPS) {
			__sync_fetch_and_add (&ip,1);
		} else {
			pthread_mutex_lock(&mutex);
			++(*ip);
			pthread_mutex_unlock(&mutex);
		}
	}
	fprintf(stderr, "Done  increasing variable.\n");

	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) {
			__sync_fetch_and_sub (&ip,1);
		} else {
			pthread_mutex_lock(&mutex);

			--(*ip);

			pthread_mutex_unlock(&mutex);
		}
	}
	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 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);
    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);

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

        return ok;
 }
```

1)

Εκτελώντας την εντολή make, παρατηρούμε ότι, από το αρχείο simplesync.c,παράγονται 2 αρχεία αντί για ένα που θα περιμέναμε. Αυτό προκύπτει από δύο παράγοντες. Αρχικά, μελετώντας προσεκτικά το αρχείο Makefile παρατηρούμε ότι γίνονται δύο μεταγλωττίσεις, με διαφορετικές επιλογές σαν

ορίσματα. Συγκεκριμένα χρησιμοποιούνται οι επιλογείς -DSYNC_ATOMIC και -DSYNC_MUTEX. Με αυτές τις εντολές καθορίζουμε ποια θα είναι η τιμή του USE_ATOMIC_OPS, που υπάρχει μέσα στο αρχείο simplesync.c, το οποίο έχει με την σειρά του ως αποτέλεσμα να εκτελείται διαφορετικό μέρος του κώδικα κάθε φορά (λόγω των παραμέτρων στις εντολές συνθήκης if-else). Παρατηρούμε επίσης ότι, όταν τρέχουμε τον κώδικα, το αποτέλεσμα δεν είναι το αναμενόμενο. Περιμένουμε να δούμε μηδενικό αποτέλεσμα στην μεταβλητη val, αντιθέτως λαμβάνει την τιμή 577681. Αυτό συμβαίνει, καθώς δεν έχει προηγηθεί συγχρονισμός των νημάτων και έτσι γίνεται εκ περιτροπής χρήση των νημάτων από τον επεξεργαστή, με αποτέλεσμα να οδηγούμαστε σε λανθασμένο αποτέλεσμα. Ειδικότερα αυτό γίνεται επειδή μπορεί κάποιο thread, να αρχίσει να αλλάζει την μεταβλητή που μας αφορά, όμως αναπάντεχα να σταματήσει, καθώς ο επεξεργαστής επέλεξε να συνεχίσει την λειτουργία, κάποιο άλλο thread. Έτσι το πρώτο thread, έχει στα “διαβασμένα” του μια παλιά εκδοχή της εν λόγω μεταβλητής και έτσι όταν συνεχιστεί η λειτουργία του, θα προκύψουν αναπάντεχα αποτελέσματα (σαν να έχουν “διαγραφεί” μερικά βήματα που έχουμε ζητήσει να γίνουν).



Τα αποτελέσματα των προγραμμάτων μετά την τροποποίηση τους είναι τα παρακάτω:

Με την χρήση mutexes:

```
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m1.040s
user    0m0.984s
sys     0m1.016s
```

Με την χρήση Atomic Operations:

```
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

```
real    0m0.076s
user    0m0.125s
sys     0m0.000s
```

Οι έξοδοι των προγραμμάτων μετά την χρήση κλειδωμάτων είναι ορθοί. Το αποτέλεσμα της μεταβλητής val είναι μηδενικό, όπως άλλωστε αναμενόταν.
Οι χρόνοι είναι αυξημένοι σε σύγκριση με το αρχικό πρόγραμμα, το οποίο δεν πραγματοποιούσε κάποια διαδικασία συγχρονισμού. Στην παρούσα διεργασία βρίσκεται σε κάθε κρίσιμο τμήμα της συνάρτησης ένα μόνο νήμα, ώστε να εξασφαλίζεται η ορθότητα του αποτελέσματος. Στην αρχική διεργασία, όπου πολλαπλά νήματα βρισκόντουσαν στο ίδιο τμήμα, η διαδικασία ήταν ταχύτερη, ωστόσο αυτό οδήγησε σε λανθασμένα συμπεράσματα, καθώς δεν υπήρχε κατάλληλη επικοινωνία μεταξύ των νημάτων.

2)
Γενικότερα, το πιο βασικό σε κάθε διεργασία για την ταχεία υλοποίησης της είναι ο αλγόριθμος που ακολουθεί. Με δεδομένο, ότι εδώ διατρέχεται ο ίδιος αλγόριθμος και για τους δύο τρόπους υλοποίησης, παρατηρώντας τα αποτελέσματα των χρόνων εκτέλεσης, συγκλίνουμε στο ότι, τα posix-mutexes είναι πιο αργά σε σύγκριση με τις ατομικές λειτουργίες. Αυτό σχετίζεται με τον τρόπο υλοποίησης των δύο αυτών μεθόδων. Οι ατομικές λειτουργίες είναι αρκετά ταχύς μέθοδος, ωστόσο δεν υπάρχει μεγάλη ποικιλία, με αποτέλεσμα η χρήση τους να περιορίζεται σε μικρό εύρος δυνατοτήτων.

3)
```
.LBE17:
    .loc 1 38 17 is_stmt 1 view .LVU17
    .loc 1 39 25 view .LVU18
    lock addq    $1, (%rsp)
    .loc 1 37 28 view .LVU19

.LBE25:
    .loc 1 58 17 is_stmt 1 view .LVU47
    .loc 1 59 25 view .LVU48
    lock subq    $1, (%rsp)
    .loc 1 57 28 view .LVU49
```

Φαίνεται στις παραπάνω εντολές πως όταν ισχύει το lock, γίνεται η πρόσθεση ή η αφαίρεση της μονάδας.

Τα suffixes LBE,LVL,LBB είναι ταμπέλες. Τα LFB, LFE παράγονται στην αρχή και στο τέλος μια συνάρτησης και τα LBB, LBE παράγονται με την χρήση της επιλογής -g κατά την διάρκεια της μεταγλώττισης. Οι εντολές .loc 1 59 25 μας βοηθούν στο να καταλάβουμε σε ποιά γραμμή και ποια στήλη του κώδικα μας ανήκει το κομμάτι κώδικα που βρίσκεται ακριβώς από κάτω. Παρατηρούμε ότι εκτελούνται οι subq και addq, υπό συνθήκες lock και γίνεται η πρόσθεση και η αφαίρεση αντίστοιχα μιας μονάδας από τον καταχωρητή.
Το παραπάνω είναι ένα αρχείο γραμμένο σε ASSEMBLY γλώσσα με κατάληξη .s, όπως αυτό προέκυψε χρησιμοποιώντας την εντολή:

```
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -static -S -g -o  simplesync-atomic.s simplesync.c
```

4)

```
.L2:
    .loc 1 38 17 view .LVU18
    .loc 1 41 25 view .LVU19
    movq    %r12, %rdi
    call    pthread_mutex_lock@PLT
.LVL4:
    .loc 1 42 25 view .LVU20
    .loc 1 42 28 is_stmt 0 view .LVU21
    movl    0(%rbp), %eax
    .loc 1 43 25 view .LVU22
    movq    %r12, %rdi
    .loc 1 42 25 view .LVU23
    addl    $1, %eax
    movl    %eax, 0(%rbp)
    .loc 1 43 25 is_stmt 1 view .LVU24
    call    pthread_mutex_unlock@PLT
```

Το παραπάνω είναι ένα αρχείο γραμμένο σε ASSEMBLY γλώσσα με κατάληξη .s, όπως αυτό προέκυψε χρησιμοποιώντας την εντολή:

```
gcc -Wall -O2 -pthread -DSYNC_MUTEX -static -S -g -o  simplesync-mutex.s simplesync.c
```

Παρατηρούμε στο παραπάνω τμήμα κώδικα πως κατά την διάρκεια του προγράμματος καλούνται τα pthread_mutex με lock και unlock για να κλειδώσουν το κρίσιμο τμήμα του προγράμματος.

Άσκηση 2.

Η υλοποίηση του προγράμματος είναι η παρακάτω:

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include <pthread.h>
#include <signal.h>
#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg); } while (0)



sem_t* semaphore;

int*  NTHREADS;

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;
void *safe_malloc(size_t size) {
        void *p;

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

```
                exit(1);
        }

        return p;
}
/*Sets how the signal must act, when CNTRL + C is pressed */

void handle_sigint(int sig) {
        reset_xterm_color(1);
        exit(3);
}



/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
        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);
                }
        }//write(fd,&newline,1);
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write newline");
                exit(1);
        }
}

int fd=1;


void* compute_and_output_mandel_line(void* start)
{
        int line ;
        int st=*(int*)start;
        for(line =st; line<y_chars; line+=(*NTHREADS)){
                int color_val[x_chars];
                //sem_wait(&semaphore[line % (*NTHREADS)]);
                compute_mandel_line(line, color_val);
                sem_wait(&semaphore[line % (*NTHREADS)]);
                output_mandel_line(fd, color_val);
                sem_post(&semaphore[(line + 1) % (*NTHREADS)]);
        }
        return NULL;
}

int safe_atoi(char *s, int *val)
{
        long l;
        char *endp;
```

```
        l = strtol(s, &endp, 10);
        if (s != endp && *endp == '\0') {
                *val = l;   return 0;
        } else
                return -1;
}

int main(int argc, char **argv)
{
        if(argc!=2){
                perror("Too few args\n");
                exit (3);
        }
        NTHREADS= (int*) malloc(sizeof(int));
        safe_atoi(argv[1],NTHREADS);

        semaphore = (sem_t*) malloc((*NTHREADS)*sizeof(sem_t));

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

        /*For handling out signaL CTRL + C and restarting the colors */
        struct sigaction sa;
        sa.sa_handler = &handle_sigint;
        sa.sa_flags = SA_RESTART;
        sigaction(SIGINT, &sa,NULL);

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

        pthread_t th[(*NTHREADS)];
        int  i;
        for (i = 0; i < (*NTHREADS); i++){
                if(i!=0) sem_init(&semaphore[i],0,0);
                else sem_init(&semaphore[i], 0, 1);
                int* start = (int*)malloc(sizeof(int));
                *start = i;

                if (pthread_create(&th[i], NULL,
&compute_and_output_mandel_line, start) != 0) {
                        perror("Failed to create thread");
                        exit(1);
                }
```

```
        }

        for (i = 0; i < (*NTHREADS); i++) {
                if (pthread_join(th[i], NULL) != 0) {
                        perror("Failed to join thread");
                        exit(2);
                }
        }
        for(i=0;i<(*NTHREADS);i++){
                sem_destroy(&semaphore[i]);
        }


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

1)

Χρειάζονται όσοι σημαφόροι, όσα είναι τα threads που δίνει ο χρήστης για την εκτέλεση του προγράμματος.

2)
Σειριακά:
real 0m0.455s
user 0m0.391s
sys 0m0.047s

Παράλληλου προγράμματος:
real 0m0.283s
user 0m0.500s
sys 0m0.047s

real 0m0.310s
user 0m0.531s
sys 0m0.031s

Παρατηρούμε ότι ο πραγματικός χρόνος που υλοποιήθηκε η διεργασία με την χρήση νημάτων και σημαφόρων έχει μειωθεί στο μισό σε σύγκριση με την υλοποίηση του προγράμματος με σειριακό τρόπο. Αυτό οφείλεται στο γεγονός ότι πραγματοποιείται διαμοιρασμός της διεργασίας στα διάφορα νήματα (όσα ορίσει ο χρήστης) και έτσι μοιράζεται ο όγκος της υλοποίησης, ενώ στην πρώτη περίπτωση δεν πραγματοποιείται κάτι τέτοιο και όλα γίνονται σειριακά.

3)

Εξαρχής τροποποιώντας το αρχείο mandel.c είχαμε θέσει ως κρίσιμο τμήμα της συνάρτησης μόνο την συνάρτηση output_mandel_line() και δεν είχαμε συμπεριλάβει την compute_mandel_line(); , καθώς θεωρούμε ότι δεν υπάρχει κάποιο πρόβλημα στην ανάγνωση και καταγραφή τιμών από πολλαπλά νήματα ταυτόχρονα. Αυτό που επιθυμούμε είναι, να εκτυπώνει ένα νήμα κάθε φορά. Παρατηρούμε ότι αν θεωρήσουμε ότι στο κρίσιμο τμήμα ανήκουν και οι δύο παραπάνω συναρτήσεις που αναφέρθηκαν, τότε, το πρόγραμμα πηγαίνει αρκετά αργά και δεν φαίνεται κάποια ουσιαστική διαφορά σε σύγκριση με την σειριακή υλοποίηση του προγράμματος και αυτό, γιατί το γεγονός ότι μόνο ένα νήμα την φορά μπορεί να υπολογίζει την γραμμή για την έξοδο και μετά να την εκτυπώνει, καθυστερεί αρκετά την διαδικασία. Για αυτόν τον λόγο πρέπει να προσέχουμε στο τι ορίζουμε κρίσιμο τμήμα του προγράμματος μας. Καταλήγουμε στο ότι, το κρίσιμο τμήμα αρκεί να περιέχει μόνο την φάση εξόδου κάθε γραμμής, προκειμένου να είναι η υλοποίηση γρήγορη.

4)

Με το πάτημα, στο πληκτρολόγιο, του σήματος CTRL + C , στέλνεται στην διεργασία ένα σήμα διακοπής, και επομένως η διεργασία σταματάει να εκτελείται και το σχήμα δεν είναι ολοκληρωμένο. Επιπλέον ακριβώς μετά τα γράμματα του τερματικού, αλλάζουν χρώμα, καθώς ο κώδικας που έτρεχε, κατά πάσα πιθανότητα, βρίσκεται σε σημείο, που αλλάζει τον χρωματικό κώδικα του terminal. Για να αποφύγουμε αυτό το αποτέλεσμα, θα ορίσουμε έναν χειριστή σημάτων μέσα στην συνάρτηση, ο οποίος μετά το πάτημα των πλήκτρων CTRL + C, θα επαναφέρει τα χρώματα στην σωστή τους κατάσταση και έπειτα θα επιτρέπει την πραγματοποίηση του μηνύματος τερματισμού. Το παραπάνω επιτυγχάνεται με την χρήση του sigaction, που διαχειρίζεται το σήμα και τη συνάρτησης handle_sigint(), η οποία ορίζει τι θα γίνει όταν σταλεί το σήμα CTRL + C. Στο εσωτερικό της συνάρτησης αυτής, πραγματοποιούμε την reset_xterm_color () και οδηγούμαστε στην έξοδο του προγράμματος μέσω της exit().
Συγκεκριμένα προστέθηκαν στο πρόγραμμα τα παρακάτω:
Έξω από την main:

```
void handle_sigint(int sig) {
        reset_xterm_color(1);
        exit(3);
}
```

Στο εσωτερικό της main:

```
struct sigaction sa;
        sa.sa_handler = &handle_sigint;
```

```
        sa.sa_flags = SA_RESTART;
        sigaction(SIGINT, &sa,NULL);

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

Η έξοδος του προγράμματος είναι: