Λειτουργικά Συστήματα // Άσκηση 3:Συγχρονισμός

Ομάδα: oslabc18
Φοιτητές: Αθανασίου Ιωάννης 031117041
Καραβαγγέλης Αθανάσιος 03117022

ΑΣΚΗΣΗ 1.1

Αφού αντιγράψουμε στο directory μας τα αρχεία κώδικα που δίνονται στο directory "/home/oslab/code/sync", εκτελούμε την εντολή "make" και παράγονται τα αντίστοιχα εκτελέσιμα αρχεία.

- Τρέχουμε το εκτελέσιμο "simplesync-mutex" και παρατηρούμε λαθεμένη έξοδο. Πιο συγκεκριμένα, η τιμή της μεταβλητής val θα περιμέναμε να ισούται με 0 μετά το πέρας της διαδικασίας αφού αυξήσαμε και μειώσαμε ίσες φορές την τιμή της. Αυτό, όμως, δεν συμβαίνει επειδή από τον κώδικα απουσιάζουν εντολές που να εξασφαλίζουν ότι τα δύο κρίσιμα τμήματά του εκτελούνται από ένα μόνο νήμα κάθε στιγμή. Το αποτέλεσμα είναι να έχουμε race conditions, οι λειτουργίες των δύο νημάτων να "μπλέκονται" μεταξύ τους και το αποτέλεσμα της εκτέλεσής τους να εξαρτάται από τη σειρά εκτέλεσης τους από τα νήματα.

- Παρατηρούμε ότι με βάση το αρχείο κώδικα "simplesync.c" παράγονται δύο εκτελέσιμα αρχεία, τα "simplesync-atomic" και "simplesync-mutex". Αυτό επιτυγχάνεται μέσω του τμήματος:

"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** -g -s -c -o simplesync-atomic.o simplesync.c"

του Makefile, το οποίο "συνεργάζεται" με το τμήμα του κώδικα:

```
#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
```

Και με τη συνθήκη ελέγχου ροής:

```
if (USE_ATOMIC_OPS) {
    ...
 } else {
    ...
 }
```

- Ο κώδικας που δίνεται τροποποιείται κατάλληλα ώστε να συγχρονίζεται η εκτέλεση των δύο νημάτων με την χρήση POSIX mutexes στο εκτελέσιμο "simplesync-mutex" και με χρήση των ατομικών λειτουργιών "__sync_add_and_fetch()" και "__sync_sub_and_fetch"του GCC στο εκτελέσιμο "simplesync-atomic".

Το αρχείο κώδικα "simplesync.c":

```
/*
 * 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>
//#include <semaphore.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
#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;
        fprintf(stderr, "About to increase variable %d times\n", N);
        for (i = 0; i < N; i++) {
                if (USE_ATOMIC_OPS) {
                        __sync_add_and_fetch(ip, 1);
                }
                else {
                        ret = pthread_mutex_lock(&lock);
                        if(ret){
                                perror_pthread(ret, "pthread_mutex_lock");
                        }
                        ++(*ip);
                        ret = pthread_mutex_unlock(&lock);
                        if (ret) {
                                perror_pthread(ret, "pthread_mutex_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) {
                        __sync_sub_and_fetch(ip, 1);
                }
                else {
```

```
                ret = pthread_mutex_lock(&lock);
                if (ret) {
                        perror_pthread(ret, "pthread_mutex_lock");
                }
                --(*ip);
                ret = pthread_mutex_unlock(&lock);
                if (ret) {
                        perror_pthread(ret, "pthread_mutex_unlock");
                }
            }
        }
        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
        //THREAD_ID = t1                START_FUN = INCREASE
        ret = pthread_create(&t1, NULL, increase_fn, &val);
        if (ret) {
                perror_pthread(ret, "pthread_create");
                exit(1);
        }
        //THREAD_ID = t2                START_FUN = DECREASE
        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");
        }
        // CHECK val
        ok = (val == 0);
        printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
        return ok;
}
```

Παρατηρούμε ότι μία εκτέλεση του “simplesync-mutex” οδηγεί στην έξοδο:

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

Και επομένως επαληθεύεται η λειτουργία που θέλαμε να πετύχουμε.

Ομοίως, μία εκτέλεση του “simplesync-atomic” οδηγεί στην έξοδο:

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

Η οποία επαληθεύει πάλι το αποτέλεσμα που επιδιώκαμε.

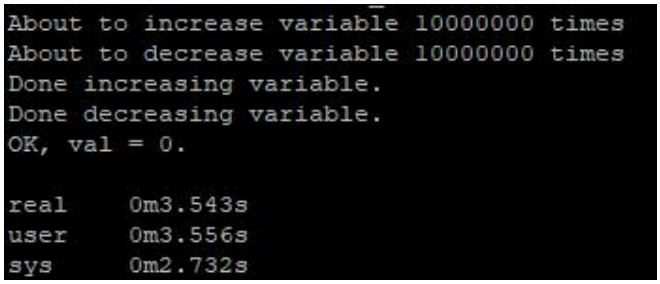
ΕΡΩΤΗΣΕΙΣ

1. Εκτελούμε στο terminal την εντολή “time ./simplesync-atomic” για να ελέγξουμε την ταχύτητα εκτέλεσης του εκτελέσιμου με χρήση των ατομικών λειτουργιών του GCC για τον συγχρονισμό των δύο νημάτων και παράγεται η έξοδος:

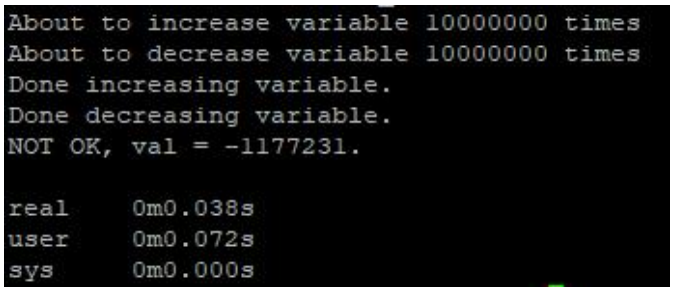
```
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m0.413s
user    0m0.816s
sys     0m0.000s
```

Ομοίως, με την εντολή “time ./simplesync-mutex” εξετάζουμε τον χρόνο εκτέλεσης του εκτελέσιμου όπου χρησιμοποιήσαμε mutexes για τον συγχρονισμό των δύο νημάτων:



Για να ελέγξουμε τον χρόνο εκτέλεσης του προγράμματος χωρίς συγχρονισμό, αφαιρούμε τις αντίστοιχες εντολές που έχουμε προσθέσει στον κώδικα και εκτελούμε και πάλι την εντολή “time ./simplesync-mutex”:



Παρατηρούμε ότι ο χρόνος εκτέλεσης στην περίπτωση που δεν γίνεται συγχρονισμός των νημάτων είναι εμφανώς μικρότερος από ότι με τον συγχρονισμό τους. Αυτό αναμένουμε ότι συμβαίνει επειδή στην πρώτη περίπτωση, το κρίσιμο τμήμα του κώδικα (αυξομείωση του ακεραίου που δεικτοδοτείται από το ip) μπορεί να εκτελείται από περισσότερα από ένα νήματα κάθε στιγμή, ενώ στην δεύτερη περίπτωση εκτελείται μόνο από ένα νήμα ακριβώς κάθε στιγμή. Έτσι, αφού ο αριθμός των συνολικών εκτελέσεων του κρίσιμου τμήματος είναι ίδιος και στις δύο περιπτώσεις, η επικάλυψη των εκτελέσεων αυτών οδηγεί σε μικρότερο συνολικό χρονικό διάστημα όταν δεν γίνεται συγχρονισμός.

2. Οι μετρήσεις του προηγούμενου ερωτήματος δείχνουν, επίσης, ότι η χρήση των ατομικών λειτουργιών του GCC είναι πιο γρήγορη μέθοδος συγχρονισμού από τα POSIX mutexes. Αυτό συμβαίνει διότι τα atomic operations αποτελούν μια μέθοδο χαμηλότερου επιπέδου απ’ ότι τα mutexes καθώς μεταφράζει τις εντολές που πρέπει να εκτελεστούν σε μία μόνο εντολή assembly. Αντίθετα, το εκτελέσιμο που κάνει χρήση mutexes εκτελεί το συγχρονισμό σε υψηλότερο επίπεδο αφού για τον συγχρονισμό γίνεται χρήση αλγορίθμου και έτσι απαιτείται μεγαλύτερος χρόνος εκτέλεσης αφού γίνεται και μετάφραση σε περισσότερες από μία εντολές assembly .

3. Η χρήση των ατομικών λειτουργιών του GCC μεταφράζεται στις εξής εντολές στην assembly του επεξεργαστή μας:

C		ASSEMBLY
`__sync_fetch_and_add(ip, 1);`	->	`lock addl $1, (%rax)`
`__sync_fetch_and_add(ip, -1);`	->	`lock subl $1, (%rax)`

4. Η χρήση των POSIX mutexes μεταφράζεται στις εξής εντολές στην assembly του επεξεργαστή μας:

C	ASSEMBLY
`pthread_mutex_t mutex;`	`.comm mutex,40,32`

`ret = pthread_mutex_lock(&mutex);`	`movl $mutex, %edi`
	`call pthread_mutex_lock`
	`movl %eax, -20(%rbp)`

`ret = pthread_mutex_unlock(&mutex);`	`movl $mutex, %edi`
	`call pthread_mutex_unlock`
	`movl %eax, -20(%rbp)`

`pthread_mutex_lock(&mutex);`	`movl $mutex, %edi` `call pthread_mutex_lock`
`pthread_mutex_unlock(&mutex);`	`movl $mutex, %edi` `call pthread_mutex_unlock`
`pthread_mutex_init(&mtx, NULL);`	`movl $mutex, %edi` `call pthread_mutex_init`

ΑΣΚΗΣΗ 1.2

Ο κώδικας που λύνει το πρόβλημα είναι στο αρχείο κώδικα "mandel2.c":

```
/*
* 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 <pthread.h>
#include <semaphore.h>
#include <signal.h>
#include "mandel-lib.h"
#define MANDEL_MAX_ITERATION 100000
/***************************
 * Compile-time parameters *
 ***************************/
pthread_t *thread;
sem_t *sem;
int NTHREADS = 3;
typedef struct thread_info_struct {
        pthread_t tid; /* POSIX thread id, as returned by the library */
        int thrid; /* Application-defined thread id */
        int N;
}thread_str;
/*
 * 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 ='@';
		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 N)
{
		/*
		 * 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(&sem[((line)%N)]);
		output_mandel_line(fd, color_val);
		sem_post(&sem[((line+1)%N)]);
}
```

```
void usage(char *argv0) //function that prints info if argc!=2
{
	fprintf(stderr, "Usage: %s NTHREADS \n\n"
				"Exactly one argument required:\n"
				" NTHREADS: The number of threads to create.\n",
				argv0);
	exit(1);
}
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;
}
int safe_th(char *s, int *val) //takes a string and separates the int and char parts
{
	long l;
	char *endp;
	l = strtol(s, &endp, 10);
	if (s != endp && *endp == '\0') {
		*val = l;
		return 0;
	} else
		return -1;
}
void *thread_compute(void *arg)
{
	thread_str *thr = arg;
	int line;
	for (line = thr->thrid; line < y_chars; line+= thr->N) {
		compute_and_output_mandel_line(1, line, thr->N);
	}
	return NULL;
}
void handle_interrupt() //function to handle interrupts with Ctrl-C
{
	reset_xterm_color(1);
	printf("\n");
	exit(1);
}
int main(int argc , char **argv)
{
	int ist_thread,ret;
	thread_str *thr;
	if(argc!=2)
```

```
                usage(argv[0]);
        if(safe_th(argv[1] , &NTHREADS) < 0 || NTHREADS <=0) {
                fprintf(stderr, "`%s' is not valid for `NTHREADS'\n", argv[1]);
                exit(1);
        }
        //allocating memory for threads and semaphores
        thr = safe_malloc(NTHREADS * sizeof(*thr));
        sem = safe_malloc(NTHREADS * sizeof(*sem));
        //initializing semaphores
        for(ist_thread=0; ist_thread<NTHREADS; ist_thread++){
                sem_init(&sem[ist_thread], 0, 0);
        }
        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.
        */
        //incrementing the first semaphore
        sem_post(&sem[0]);
        for (ist_thread = 0; ist_thread < NTHREADS; ist_thread++) {
                thr[ist_thread].thrid = ist_thread;
                thr[ist_thread].N = NTHREADS;
                /* Spawn new thread(s) */
                ret = pthread_create(&thr[ist_thread].tid, NULL, thread_compute,
&thr[ist_thread]);
                if (ret) {
                        perror( "pthread_create");
                        exit(1);
                }
        }
        signal(SIGINT, handle_interrupt); //when Ctrl-C is pressed
        //joining threads
        for(ist_thread = 0; ist_thread< NTHREADS; ist_thread++){
                ret = pthread_join(thr[ist_thread].tid,NULL);
                if (ret) {
                        perror("pthread_join");
                        exit(1);
                }
        }
        //destroying semaphores
        for(ist_thread=0; ist_thread<NTHREADS; ist_thread++){
                sem_destroy(&sem[ist_thread]);
        }
        free(sem);
        free(thr);
        reset_xterm_color(1);
```

```
        return 0;
}
```

ΕΡΩΤΗΣΕΙΣ

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

Το πλήθος των σημαφόρων εξαρτάται προφανώς από το πλήθος των νημάτων που θα χρησιμοποιήσουμε. Τα νήματα που χρειάζονται είναι το πολύ 50, δεδομένου ότι κάθε νήμα αναφέρεται σε μία γραμμή τουλάχιστον. Άρα, στην περίπτωση που δοθεί ως παράμετρος πλήθος νημάτων μικρότερο ή ίσο του 50, αυτό θα είναι και το πλήθος των σημαφόρων. Στην περίπτωση όμως που δοθεί ως παράμετρος πλήθος νημάτων μεγαλύτερο του 50, τότε κάνουμε την παραδοχή να δουλέψουμε με 50 νήματα και άρα και 50 σημαφόρους.

2. Πόσος χρόνος απαιτείται για την ολοκλήρωση του σειριακού και του παράλληλου προγράμματος με δύο νήματα υπολογισμού; Χρησιμοποιήστε την εντολή time(1) για να χρονομετρήσετε την εκτέλεση ενός προγράμματος, π.χ., time sleep 2. Για να έχει νόημα η μέτρηση, δοκιμάστε σε ένα μηχάνημα που διαθέτει επεξεργαστή δύο πυρήνων. Χρησιμοποιήστε την εντολή cat /proc/cpuinfo για να δείτε πόσους υπολογιστικούς πυρήνες διαθέτει κάποιο μηχάνημα.

Αρχικά, εκτελούμε την εντολή “cat /proc/cpuinfo” και βλέπουμε ότι το σύστημά μας διαθέτει 7 επεξεργαστές.

Κατόπιν, με την εντολή “time ./mandel” υπολογίζεται ο χρόνος εκτέλεσης του αρχικού προγράμματος (σειριακής εκτέλεσης):

```
real    0m1.022s
user    0m0.988s
sys     0m0.008s
```

Ομοίως, με την εντολή “time ./mandel2 2” υπολογίζεται ο χρόνος εκτέλεσης του προγράμματος παράλληλης εκτέλεσης με τη χρήση δύο νημάτων:

```
real    0m0.520s
user    0m0.972s
sys     0m0.028s
```

Όπως είναι αναμενόμενο, ο χρόνος εκτέλεσης του προγράμματος με την σειριακή εκτέλεση είναι μεγαλύτερος από το δεύτερο πρόγραμμα με τη χρήση δύο νημάτων, αφού γίνεται πλέον κάποια παραλληλοποίηση της διαδικασίας.

3. 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;

Παρατηρούμε ότι το πρόγραμμα με τον παράλληλο υπολογισμό του Mandelbrot με threads παρουσιάζει επιτάχυνση έναντι του σειριακού. Το κρίσιμο τμήμα του σχήματος συγχρονισμού που περιέχεται στο αρχείο mandel2.c περιέχει μόνο τη φάση της εξόδου κάθε γραμμής και όχι αυτή του υπολογισμού της, καθώς στην πρώτη απαιτείται σειριακή εμφάνιση των γραμμών για να είναι σωστό το αποτέλεσμα μας (κάτι το οποίο δεν απαιτείται στην φάση του υπολογισμού).

4. Τι συμβαίνει στο τερματικό αν πατήσετε Ctrl-C ενώ το πρόγραμμα εκτελείται; σε τι κατάσταση αφήνεται, όσον αφορά το χρώμα των γραμμάτων; Πώς θα μπορούσατε να επεκτείνετε το mandel.c σας ώστε να εξασφαλίσετε ότι ακόμη κι αν ο χρήστης πατήσει Ctrl-C, το τερματικό θα επαναφέρεται στην προηγούμενη κατάστασή του;

Όταν πατήσουμε τα πλήκτρα Ctrl+C στέλνεται στο πρόγραμμα μας ένα ***SIGINT***. Παρατηρούμε ότι αν πατήσουμε Ctrl+C κατά τη διάρκεια της εκτύπωσης του Mandelbrot τότε το τερματικό παραμένει στο χρώμα που είχε η τελευταία εκτύπωση πριν δοθεί το σήμα. Για να αποτρέψουμε αυτή τη συμπεριφορά θα πρέπει να εισάγουμε μία κλήση του signal handler η οποία θα υλοποιεί μία συνάρτηση που ουσιαστικά θα κάνει reset στην default τιμή του χρώματος του terminal. Αυτό μπορεί να γίνει με την εντολή ***signal(SIGINT ,handle_interrupt); ,*** όπου η handle_interrupt είναι η εξής:

```
void handle_interrupt(){
        reset_xterm_color(1);
        printf("\n");
        exit(1);
}
```