Λειτουργικά Συστήματα Υπολογιστών

3η Εργαστηριακή Αναφορά

Θοδωρής Παπαρρηγόπουλος el18040
Ορφέας Φιλιππόπουλος el18082

Άσκηση 1

Πηγαίος 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 mutex = PTHREAD_MUTEX_INITIALIZER;
//volatile int val;
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 */
			//++(*ip);
			__sync_add_and_fetch(ip,1);

			/* ... */
		} else {
			pthread_mutex_lock(&mutex);
			/* ... */
			/* You cannot modify the following line */
			++(*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) {
			/* ... */
			/* You can modify the following line */
//			--(*ip);
			__sync_sub_and_fetch(ip,1);
			/* ... */
		} else {
			pthread_mutex_lock(&mutex);
			/* ... */
			/* You cannot modify the following line */
			--(*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);
	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);

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

	return ok;
}
```

Ερωτήσεις

1)
Αρχικά χρονομετρώντας την εκτέλεση χωρίς το lock and unlock έχουμε:
oslaba33@os-node1:~/third_lab/sync$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
NOT OK, val = -1252840.

real 0m0.039s
user 0m0.072s
sys 0m0.000s

Βάζοντας τα lock and unlock (mutex) έχουμε:
oslaba33@os-node1:~/third_lab/sync$ 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 0m26.911s
user 0m26.732s
sys 0m27.072s

oslaba33@os-node1:~/third_lab/sync$ time ./simplesync-
simplesync-atomic simplesync-mutex
oslaba33@os-node1:~/third_lab/sync$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real 0m0.421s
user 0m0.836s
sys 0m0.000s

1) Χωρίς την χρήση συγχρονισμού του προγράμματος η εκτέλεση είναι πιο γρήγορη, ωστόσο δεν βγάζει σωστό αποτέλεσμα γεγονός το οποίο περιμένουμε (race conditions). Ως λύση χωρίς χρήση συγχρονισμού θα μπορούσαμε να έχουμε μια global volatile variable (που αποθηκεύεται απευθείας στη κύρια μνήμη) όπου όποτε άλλαζε στην μία διεργασία θα άλλαζε και στην άλλη.

2) Μεταξύ της χρήσης mutexes και atomic operations, φαίνεται φανερά ότι η χρήση των mutexes καθιστά πολύ πιο αργή την εκτέλεση του προγράμματος. Αυτό συμβαίνει καθώς τα mutexes εκτελούν system calls προκειμένου να μπουν σε αναμονή ενώ τα atomic operations τρέχουν σε επίπεδο hardware. Επιπλέον, τα mutexes τρέχουν atomic operations στην υλοποίηση τους.

3) Οι ατομικές εντολές είναι σε επίπεδο hardware οποτε μεταγράζονται σε μια εντολή σε assembly:
__sync_add_and_fetch(ip,1) -> lock addl $1,(%ebx)
__sync_sub_and_fetch(ip,1) -> lock addl $-1,(%ebx)

4) Οι POSIX mutexes είναι σε εντολές:
Για το κλείδωμα
mvl $mutex1,(%esp)
call pthread_mutex_lock

Για το ξεκλείδωμα
mvl $mutex1,(%esp)
call pthread_mutex_unlock

Άσκηση 2

Πηγαίος Κώδικας

```
/*
 * 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 <signal.h>
#include <semaphore.h>
#include <pthread.h>
#include <errno.h>
#include "mandel-lib.h"


#define MANDEL_MAX_ITERATION 100000
#define perror_pthread(ret, msg) \
        do { errno = ret; perror(msg); } while (0)



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

sem_t *mutex;

struct thread_info_struct {
        pthread_t tid;
        int *arr;
        int thrid;
        int thrcnt;
};

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_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;
}

void usage(char *argv0)
{
	fprintf(stderr, "Usage: %s thread_count array_size\n\n"
		"Exactly two argument required:\n"
		"    thread_count: The number of threads to create.\n"
		"    array_size: The size of the array to run with.\n",
		argv0);
	exit(1);
}


/*
 * 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);
	}
```

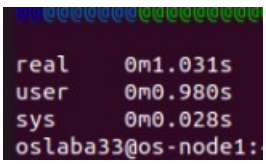
```
}

void *  compute_and_output_mandel_line(void *arg)
{
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        struct thread_info_struct *thr = arg;
//      int color_val[x_chars];
        int i;
        for(i = thr->thrid; i < y_chars; i += thr->thrcnt){
                compute_mandel_line(i, thr->arr);
                sem_wait(&mutex[i % thr->thrcnt]);
                output_mandel_line(1, thr->arr);
//              int color_val[x_chars];
                sem_post(&mutex[(i+1) % thr->thrcnt]);
        }
        return NULL;
}

void SIGINT_handler (int signum){
        reset_xterm_color(1);
        exit(1);
}

int main(int argc, char** argv)
{
//      int line;

        int thrcnt,i,ret;
      struct thread_info_struct *thr;
      signal(SIGINT, SIGINT_handler);
      if (argc != 2)
          usage(argv[0]);
      if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
          fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
          exit(1);
      }


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

        thr = safe_malloc(thrcnt*sizeof(*thr));


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

      mutex = safe_malloc(thrcnt*sizeof(sem_t));
```

```
        for (i=0; i < thrcnt; i++) {
                thr[i].thrid = i;
                thr[i].thrcnt = thrcnt;
                thr[i].arr = safe_malloc(x_chars*sizeof(int));
                if (i == 0)
                        sem_init(&mutex[i], 0, 1);
                else
                        sem_init(&mutex[i], 0, 0);
                ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
                if (ret) {
                        perror_pthread(ret, "pthread_create()");
                        exit(1);
                }
        }

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



//      for (line = 0; line < y_chars; line++) {
//              compute_and_output_mandel_line(1, line);
//      }

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

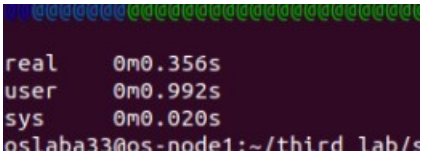
Ερωτήσεις

1) Χρειάζονται τόσοι σημαιοφόροι όσοι και ο αριθμός των threads.

2)
Είμαστε σε μηχάνημα με 3 πυρήνες για αυτό και θα βάλουμε 3 threads ως input στο παράλληλο πρόγραμμα.
Σειριακά απαιτείται χρόνος:



Παράλληλα απαιτείται χρόνος:



Παρατηρούμε αρκετή μείωση του χρόνου εκτέλεσης (η μείωση είναι αρκετά έντονη ακόμα και με το μάτι)

3)
Έχοντας υλοποιήσει από την αρχή το παράλληλο πρόγραμμα (δηλαδή ο υπολογισμός των γραμμών να γίνεται παράλληλα αλλά η τύπωση να γίνεται σειριακά) δεν έχουμε κάτι να σχολιάσουμε στο ερώτημα αυτό.

4) Αν πατήσουμε Ctrl-C ενώ το πρόγραμμα εκτελείται θα φύγουμε από την εκτέλεση του προγράμματος, ωστόσο θα αλλάξει το χρώμα του terminal. Ουσιαστικά δηλαδή δεν θα επαναφερθεί η αρχική κατάσταση διότι δεν πρόλαβε το πρόγραμμα να φτάσει στο τέλος του όπου και γίνεται η reset_xterm_color(1);. Για αυτό χρησιμοποιούμε το signal handler:

```
void SIGINT_handler (int signum){
        reset_xterm_color(1);
        exit(1);
}
```
και εκτελούμαι στη main: signal(SIGINT, SIGINT_handler);