ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

3η Άσκηση: Συγχρονισμός

ΦΟΙΤΗΤΕΣ: ΚΥΡΙΑΚΟΥ ΔΗΜΗΤΡΗΣ 03117601
ΧΑΤΖΗΧΡΙΣΤΟΦΗ ΧΡΙΣΤΟΣ 03117711
ΟΜΑΔΑ: OSLABC16
ΕΞΑΜΗΝΟ: 6ο

1.1 Συγχρονισμός σε υπάρχοντα κώδικα

Παρατήρηση: Οι δύο διεργασίες δεν είναι συγχρονισμένες με αποτέλεσμα η τελική τιμή της μεταβλητής να είναι αρκετά διαφορετική από το 0 και κάθε φορά που εκτελούνται τα δύο προγράμματα να εκτυπώνουν διαφορετικό αποτέλεσμα.

Παρατήρηση: Από το αρχείο simplesync.c δημιουργούνται δύο εκτελέσιμα αρχεία. Το αρχείο simplesync-mutex και το αρχείο simplesync-atomic. Αυτό συμβαίνει διότι το Makefile δημιουργεί δύο object files την πρώτη φορά με πάραμετρο -DSYNC_MUTEX και τη δεύτερη φορά με παράμετρο -DSYNC_ATOMIC. Με αυτό τον τρόπο δημιουργούνται τα object files για τις δύο περιπτώσεις και τα αντίστοιχα εκτελέσιμα αρχεία.

Ενδεικτική έξοδος εκτέλεσης:

```
oslabc16@os-node1:~/Ex3$ ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
oslabc16@os-node1:~/Ex3$ ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

ΚΥΡΙΑΚΟΥ ΔΗΜΗΤΡΗΣ 03117601
ΧΑΤΖΗΧΡΙΣΤΟΦΗ ΧΡΙΣΤΟΣ 03117711

Ερωτήσεις

1. Με τη χρήση της εντολής time(1) καταμετρήθηκε ο χρόνος εκτέλεσης των εκτελέσιμων για όλες τις περιπτώσεις. Παρατηρείται ότι τα εκτελέσιμα που δεν απαιτούν συγχρονισμό είναι πολύ πιο γρήγορα από τα εκτελέσιμα χωρίς συγχρονισμό. Αυτό συμβαίνει διότι χωρίς συγχρονισμό εκτελούνται οι διεργασίες ταυτόχρονα χωρίς να σταματά η λειτουργία τους άρα τελειώνουν και πιο γρήγορα.

Χωρίς συγχρονισμό	Με POSIX mutexes	Με ατομικές λειτουργίες
real 0m0.039s user 0m0.072s sys 0m0.000s	real 0m3.541s user 0m3.788s sys 0m2.548s	real 0m0.412s user 0m0.808s sys 0m0.000s

2. Παρατηρείται ότι η χρήση ατομικών λειτουργιών είναι πολύ πιο γρήγορη από τη χρήση POSIX mutexes. Αυτό συμβαίνει διότι τα posix mutexes σταματούν τη λειτουργία της διεργασίας και εκτελούν διαδοχικούς ελέγχους μέχρι να της δοθεί άδεια να εισέλθει στο κρίσιμο τμήμα και να αποκλείσει με τη σειρά της την είσοδο των άλλων διεργασιών στο ΚΤ. Η πιο πάνω διαδικασία είναι πολύ χρονοβόρα. Από την άλλη οι ατομικές εντολές λειτουργούν ατομικά όμως δεν απαιτούν τη διαδικασία του κλειδώματος και ξεκλειδώματος με αποτέλεσμα να είναι πιο γρήγορες.

3. Όπως αναφέρεται στην ιστοσελίδα https://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Atomic-Builtins.html οι ατομικές λειτουργίες __sync_add_and_fetch(*ptr,value), __sync_sub_and_fetch(*ptr,value) αντιστοιχούν στις εντολές:

```
__sync_add_and_fetch(*ptr,value):
*ptr += value; return *ptr;
*ptr = ~*ptr & value; return
```

```
__sync_sub_and_fetch(*ptr,value):
*ptr -= value; return *ptr;
*ptr = ~*ptr & value; return
```

Στη συνέχεια παράχθηκε ο κώδικας σε assembly με την εντολή gcc -S -g -DSYNC_ATOMIC simplesync.c και οι αντίστοιχες εντολές ήταν:

```
__sync_add_and_fetch(*ptr,value):
.loc 1 51 0
movq    -16(%rbp), %rax
lock addl   $1, (%rax)
```

```
__sync_sub_and_fetch(*ptr,value):
.loc 1 72 0
movq    -16(%rbp), %rax
lock subl   $1, (%rax)
```

4. Παράχθηκε ο κώδικας σε assembly με την εντολή gcc -S -g -DSYNC_MUTEX simplesync.c. Οι εντολές pthread_mutex_lock(&mutex) και pthread_mutex_unlock(&mutex) μεταφράστηκαν ως εξής:

```
.loc 1 53 0
movl    $mutex, %edi
call    pthread mutex lock
```

```
.loc 1 55 0
movl    $mutex, %edi
call    pthread_mutex_unlock
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

Ερωτήσεις

1. Κάθε νήμα χρειάζεται το δικό του σημαφόρο άρα υπάρχουν NTHREADS σημαφόροι.
2. Με την εντολή `cat /proc/cpuinfo` εξακριβώθηκε ότι το σύστημα διαθέτει 8 πυρήνες.

Σειριακή εκτέλεση προγράμματος	Παράλληλη εκτέλεση προγράμματος με δύο νήματα
real 0m1.023s user 0m0.992s sys 0m0.004s	real 0m0.520s user 0m0.968s sys 0m0.032s

3. Για τη μελέτη της επιτάχυνσης εκτελέστηκε το πρόγραμμα με διαφορετικό αριθμό νημάτων και τα αποτελέσματα ήταν τα ακόλουθα:

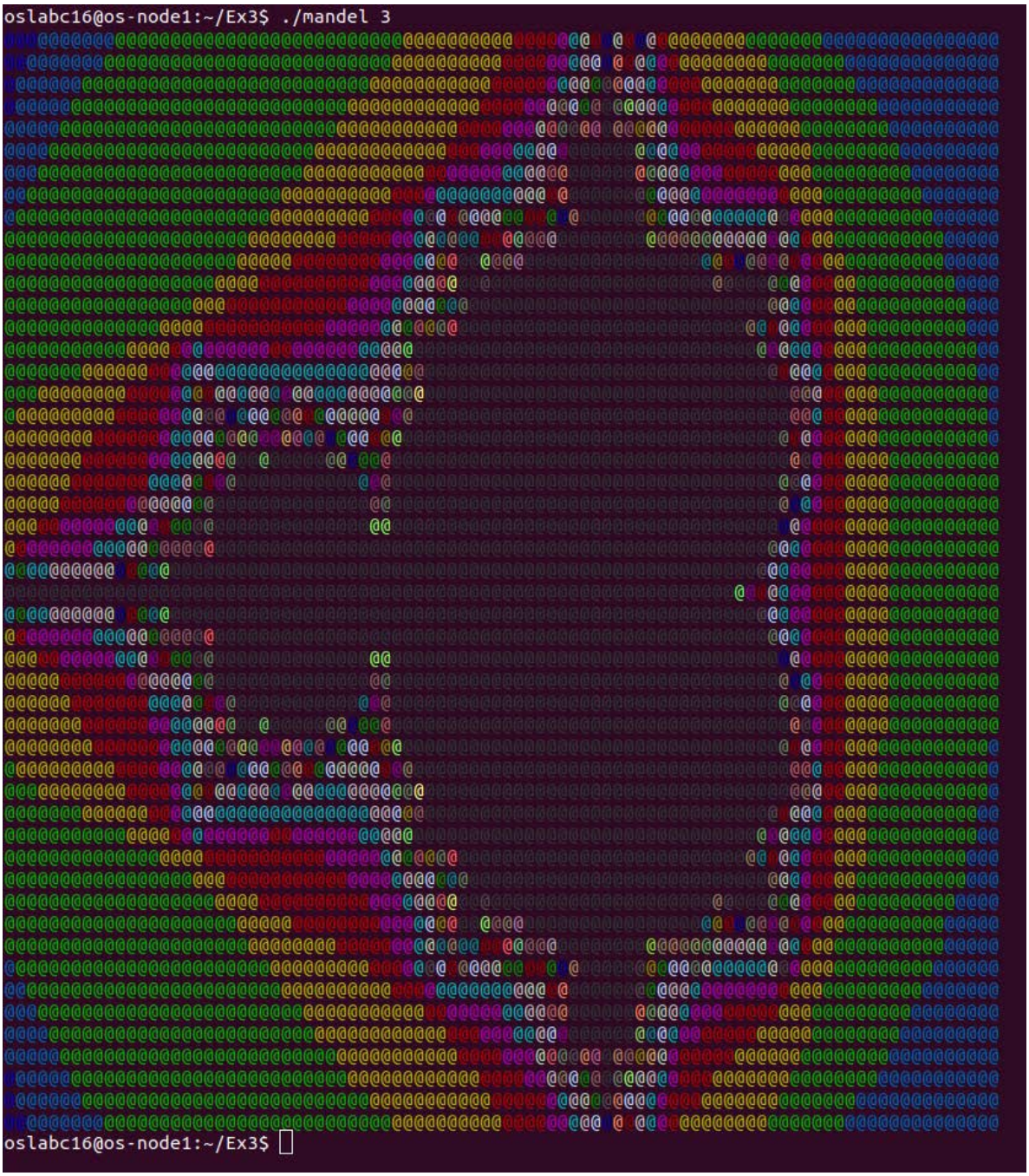
Με 4 νήματα	*Με 8 νήματα*	*Με 16 νήματα*	*Με 32 νήματα*	*Με 50 νήματα*
real 0m0.265s user 0m0.992s sys 0m0.008s	real 0m0.158s user 0m0.976s sys 0m0.032s	real 0m0.158s user 0m0.972s sys 0m0.024s	real 0m0.152s user 0m0.988s sys 0m0.012s	real 0m0.145s user 0m0.992s sys 0m0.008s

Προκύπτει ότι το πρόγραμμα εμφανίζει γραμμική επιτάχυνση όσο ο αριθμός των νημάτων είναι μικρότερος ή ίσος του αριθμού των πυρήνων. Για παράδειγμα ο συνολικός χρόνος εκτέλεσης με 4 νήματα είναι διπλάσιος από το χρόνο εκτέλεσης με 8 νήματα και μισός από το χρόνο εκτέλεσης με 2 νήματα. Όταν ο αριθμός των νημάτων ξεπερνά το 8 τότε οι χρόνοι εκτέλεσης είναι παραπλήσιοι. Αυτό συμβαίνει διότι το σύστημα δεν μπορεί να υποστηρίξει εκτέλεση περισσότερων νημάτων ταυτόχρονα από ότι ο αριθμός των πυρήνων του.

Το κρίσιμο τμήμα του προγράμματος αποτελείται από τη συνάρτηση output_mandel_line που είναι υπεύθυνη για την εκτύπωση κάθε γραμμής. Η συνάρτηση compute_mandel_line δεν περιλαμβάνεται στο ΚΤ διότι μπορεί να υπολογιστεί για κάθε γραμμή ανεξάρτητα από τις άλλες γραμμές. Μετά από δοκιμές που έγιναν παρατηρήθηκε ότι αν η συνάρτηση compute_mandel_line προστεθεί στο ΚΤ το τελικό αποτέλεσμα είναι μεν το ίδιο αλλά ο χρόνος εκτέλεσης είναι πολύ μεγαλύτερος. Από την άλλη όταν αφαιρέθηκε η συνάρτηση output_mandel_line από το ΚΤ, το τελικό αποτέλεσμα ήταν μια ασυνάρτητη εικόνα.

4. Όταν το πρόγραμμα τερματίζεται βίαια, το χρώμα των γραμμάτων διατηρείται στο τελευταίο χρώμα που εκτυπώθηκε. Ένας εύκολος τρόπος αντιμετώπισης θα ήταν η εκτέλεση της εντολής reset_xterm_color(1) μετά από κάθε εκτύπωση γραμμής

Ενδεικτική έξοδος εκτέλεσης:



Παράρτημα: Κώδικας της άσκησης

1.1 Συγχρονισμές σε υπάρχοντα κώδικα

```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>

/*
 * 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;


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) {
                        //++(*ip);
                        __sync_add_and_fetch( ip, 1);
                } else {
                        /* ... */
                        pthread_mutex_lock(&mutex);
                        ++(*ip);
                        pthread_mutex_unlock(&mutex);
                }
        }
        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}
```

```c
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) {
                        //--(*ip);
                        __sync_sub_and_fetch( 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);
        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;
}
```

1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot

```c
/*
 * mandel.c
 *
 * A program to draw the Mandelbrot Set on a 256-color xterm using multiple t
 *
 */

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

//creating a semaphore and delcaring variable for the number of threads
sem_t *semaphore;
int NTHREADS;

//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;

//A (distinct) instance of this structure is passed to each thread
struct thread_info_struct {
        pthread_t tid; /* POSIX thread id, as returned by the library */
        int thrid; /* Application-defined thread id */
};

//ensures that input is valid and converts it to integer
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;
}

//ensures that there is enough space and allocates memory
void *safe_malloc(size_t size)
{
        void *p;

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

        return p;
}
```

```c
/*
 * 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
        int color_val[x_chars];
        //indicates the lines that are creted by the current thread
        int j;
        //makes a copy of the argument arg
        struct thread_info_struct *th = arg;
        //current contains the id of the current thread
        int current = th->thrid;
        //next contains the id of the thread that comes after the current
        int next = (current + 1 ) % NTHREADS;

        /*      ith thread computes the ith and all the k*NTHREADS+ith lines
                where K=1,2,3,... and k*NTHREADS+i < y_chars
                output_mandel_line is the critical section of the code
                and must be executed by one thread at a time.
                In each iteration the program computes the values for the line,
                then ensures that only the current thread is active
                and enters the critical section, prints the line
                and exits the critical section
        */
        for(j = th->thrid; j < y_chars; j += NTHREADS){
                compute_mandel_line(j, color_val);
                sem_wait(&semaphore[current]);
                output_mandel_line(1, color_val);
                sem_post(&semaphore[next]);
        }
        return NULL;
}
```

```
int main(int argc, char *argv[])
{
        int ret,i;
        struct thread_info_struct *thr;

        //checking input, if given number of threads is  valid, it is saved in the variable NTHREAD
        if(argc == 2){
                safe_atoi(argv[1], &NTHREADS);
                if(NTHREADS>y_chars){
                        printf("Usage error: Number of threads cannot be larger than y_chars\n");
                        exit(1);
                }
        }
        else{
                printf("Usage error: Exactly 1 argument required\n");
                exit(1);
        }

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

        //checks if there is enough space to allocate for threads and semaphores via safe_malloc
        thr = safe_malloc(NTHREADS*sizeof(sem_t));
        semaphore = safe_malloc(NTHREADS*sizeof(sem_t));

    //initialize semaphores, 1st semaphores is unlocked and set to 1
        //and all the others are locked and set to 0
    for(i=1; i<NTHREADS; i++){
        sem_init(&semaphore[i], 0, 0);
    }
        sem_init(&semaphore[0], 0, 1);

        //creates NTHREADS threads
        for (i = 0; i < NTHREADS; i++) {
                /* Initialize per-thread structure */
                thr[i].thrid = i;
                /* Spawn new thread */
                ret = pthread_create(&thr[i].tid, NULL, compute_and_output_mandel_line, &thr[i]);
                if (ret) {
                        perror_pthread(ret, "pthread_create");
                        exit(1);
                }
        }

        // Wait for all threads to terminate
        for (i = 0; i < NTHREADS; i++) {
                ret = pthread_join(thr[i].tid, NULL);
                if (ret) {
                        perror_pthread(ret, "pthread_join");
                        exit(1);
                }
      }

  reset_xterm_color(1);
 return 0;
```