

# ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ ΥΠΟΛΟΓΙΣΤΩΝ

## ΑΣΚΗΣΗ 3 Συγχρονισμός

Ομάδα oslab39

Δημήτριος Βασιλείου el19830

Ιωάννης Ρόκομος el19061

### ΑΣΚΗΣΗ 1.1:

Από το δοθέν Makefile παρατηρούμε πως ενώ έχουμε ένα c αρχείο, το **simplesync.c**, παράγονται δύο εκτελέσιμα, το **simplesync-atomic** και το **simplesync-mutex**. Αυτό συμβαίνει, διότι στον κώδικα του **simplesync.c** υπάρχουν δύο flags, το **SYNC\_ATOMIC** και το **SYNC\_MUTEX**. Επίσης μέσα στο Makefile έχουμε σαν targets δύο object αρχεία, το **simplesync-mutex.o** το οποίο στη εντολή μεταγλώττισης δέχεται σαν flag το **-DSYNC\_MUTEX**, και το **simplesync-atomic.o** το οποίο έχει flag **-DSYNC\_ATOMIC**. Μέσα στον κώδικα του **simplesync.c** ελέγχουμε το flag του target και με αυτόν τον τρόπο αποφασίζουμε ποια θα είναι η τιμή του flag **USE\_ATOMIC\_OPS**. Αν δοθεί σαν flag το **SYNC\_ATOMIC** τότε το **USE\_ATOMIC\_OPS** παίρνει τιμή 1 οπότε το εκτελέσιμο **simplesync-atomic** εκτελεί τον κώδικα μέσα στα block που αρχίζουν μέσα στα **if (USE\_ATOMIC\_OPS)**. Αντίστοιχα όταν το flag είναι **SYNC\_MUTEX** εκτελείται ο κώδικας μέσα στα **else**.

Ο πηγαίος κώδικας φαίνεται παρακάτω:

```
#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;

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

```

```

        /* ... */
        pthread_mutex_lock(&lock);
        /* You cannot modify the following line */
        --(*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;

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

    /*
     * Initial value
     */
    val = 0;

    /*
     * Create threads
     */
    ret = pthread_create(&t1, NULL, increase_fn, &val);
    if (ret) {
        perror_thread(ret, "pthread_create");
        exit(1);
    }
    ret = pthread_create(&t2, NULL, decrease_fn, &val);
    if (ret) {
        perror_thread(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);
pthread_mutex_destroy(&lock);
return ok;
}

```

### Απαντήσεις στις ερωτήσεις:

1). Το output της εκτέλεσης του **simplesync-atomic** είναι το ακόλουθο:

```

oslaba39@os-node1:~/Ask3/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    0m0.442s
user    0m0.836s
sys     0m0.000s
oslaba39@os-node1:~/Ask3/sync$

```

Το output της εκτέλεσης του **simplesync-mutex** είναι το ακόλουθο:

```

oslaba39@os-node1:~/Ask3/sync$ time ./simplesync-mutex
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m8.263s
user    0m8.820s
sys     0m7.676s

```

Το output της εκτέλεσης του **simplesync-atomic** χωρίς συγχρονισμό είναι το ακόλουθο:

```

oslaba39@os-node1:~/Ask3/sync$ time ./simplesync-atomic
About to increase variable 10000000 times
About to decrease variable 10000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = 1186682.

real    0m0.041s
user    0m0.072s
sys     0m0.000s

```

Παρατηρούμε ότι η υλοποίηση του **simplesync.c** χωρίς συγχρονισμό εκτελείται πολύ γρηγορότερα από τις υλοποιήσεις με mutexes και atomic operations, διότι δεν δαπανάται χρόνος για συγχρονισμό μεταξύ των νημάτων.

2). Από την έξοδο των εκτελέσιμων παρατηρούμε ότι γρηγορότερη μέθοδος συγχρονισμού είναι η χρήση ατομικών λειτουργιών (atomic operations). Αυτό συμβαίνει, διότι στην υλοποίηση με mutexes, δαπανάται χρόνος για την είσοδο στο κρίσιμο τμήμα καθώς ένα νήμα πρέπει να ελέγξει αν το mutex είναι ελεύθερο ή κλειδωμένο. Αντιθέτως η υλοποίηση με ατομικές λειτουργίες είναι γρηγορότερη διότι το mutex αντικαθίσταται από μία assembly εντολή.

3). Εκτελούμε την ακόλουθη εντολή για να παράξουμε assembly κώδικα για το simplesync.c με χρήση ατομικών λειτουργιών: **gcc -S -g -DSYNC\_ATOMIC simplesync.c**

→ Για το νήμα που αυξάνει το N έχουμε τον παρακάτω κώδικα assembly:

```
.LC0:
.string "About to increase variable %d times\n"
.LC1:
.string "Done increasing variable.\n"
.text
.globl increase_fn
.type increase_fn, @function
increase_fn:
.LFB2:
.file 1 "simplesync.c"
.loc 1 42 0
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movq %rdi, -24(%rbp)
.loc 1 44 0
movq -24(%rbp), %rax
movq %rax, -16(%rbp)
.loc 1 46 0
movq stderr(%rip), %rax
movl $10000000, %edx
movl $.LC0, %esi
movq %rax, %rdi
movl $0, %eax
call fprintf
.loc 1 47 0
movl $0, -4(%rbp)
jmp .L2
```

→ Για το νήμα που μειώνει το N έχουμε τον παρακάτω κώδικα assembly:

```
.LC2:
.string "About to decrease variable %d times\n"
.LC3:
.string "Done decreasing variable.\n"
.text
.globl decrease_fn
.type decrease_fn, @function
decrease_fn:
.LFB3:
.loc 1 69 0
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $32, %rsp
movq %rdi, -24(%rbp)
.loc 1 71 0
movq -24(%rbp), %rax
movq %rax, -16(%rbp)
.loc 1 73 0
movq stderr(%rip), %rax
movl $10000000, %edx
movl $.LC2, %esi
movq %rax, %rdi
movl $0, %eax
call fprintf
.loc 1 74 0
movl $0, -4(%rbp)
jmp .L6
```

4). Εκτελούμε την ακόλουθη εντολή για να παράξουμε assembly κώδικα για το **simplesync.c** με χρήση ατομικών λειτουργιών: **gcc -S -g -DSYNC\_MUTEX simplesync.c**

→ Για το νήμα που αυξάνει το N έχουμε τον παρακάτω κώδικα assembly:

```
.L3:  
    .loc 1 56 0  
    movl    $lock, %edi  
    call    pthread_mutex_lock
```

→ Για το νήμα που μειώνει το N έχουμε τον παρακάτω κώδικα assembly:

```
.L7:  
    .loc 1 83 0  
    movl    $lock, %edi  
    call    pthread_mutex_lock
```

### ΑΣΚΗΣΗ 1.3:

Ο πηγαίος κώδικας φαίνεται παρακάτω:

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

#include "mandel-lib.h"

#define MANDEL_MAX_ITERATION 100000

/*
 * 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)

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

/*
 * 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;
```

```

sem_t *mutex;

/*-----Everything for threads-----*/

/*
 * 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 *color_val; /* Pointer to array to manipulate. Each
thread manipulates a line and gives each character a color */

    int thrid; /* Application-defined thread id */
    int thrcnt; /* Number of total threads*/
};

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 *safe_malloc(size_t size)
{
    void *p;

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

    return p;
}

/
*-----

```



```

-----*/

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"
        "\n",
        argv0);
    exit(1);
}

/*
 * 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)
{
    int i;
    struct thread_info_struct *thr = arg;
    int *color_val = thr->color_val;

    /* thread i manipulates lines i, i + n, i + 2n ....*/
    for (i = thr->thrid; i < y_chars; i += thr->thrcnt) {
        compute_mandel_line(i, color_val);
        sem_wait(&mutex[i % thr->thrcnt]);
        output_mandel_line(1, color_val);
        sem_post(&mutex[(i + 1) % thr->thrcnt]);
    }

    return NULL;
}

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

```

```

/*
 * Parse the command line. User gives number of threads
 */
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;

/* Create array of thrcnt threads */
thr = safe_malloc(thrcnt * sizeof(*thr));

/* Create array of mutexes of length thrcnt */
mutex = malloc(thrcnt * sizeof(*mutex));

/*
 * draw the Mandelbrot Set, one line at a time.
 * Output is sent to file descriptor '1', i.e., standard output.
 */
sem_init(&mutex[0], 0, 1);
for (i = 1; i < thrcnt; i++) {
    sem_init(&mutex[i], 0, 0);
}

for (i = 0; i < thrcnt; i++) {
    /* Initialize per-thread structure */
    thr[i].thrid = i;
    thr[i].thrcnt = thrcnt;
    thr[i].color_val = safe_malloc(x_chars * sizeof(int));
    /* 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 < thrcnt; i++) {

```

```

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

    reset_xterm_color(1);
    return 0;
}

```

Παραθέτουμε την έξοδο εκτέλεσης του προγράμματος για διάφορες τιμές του N(αριθμός νημάτων):

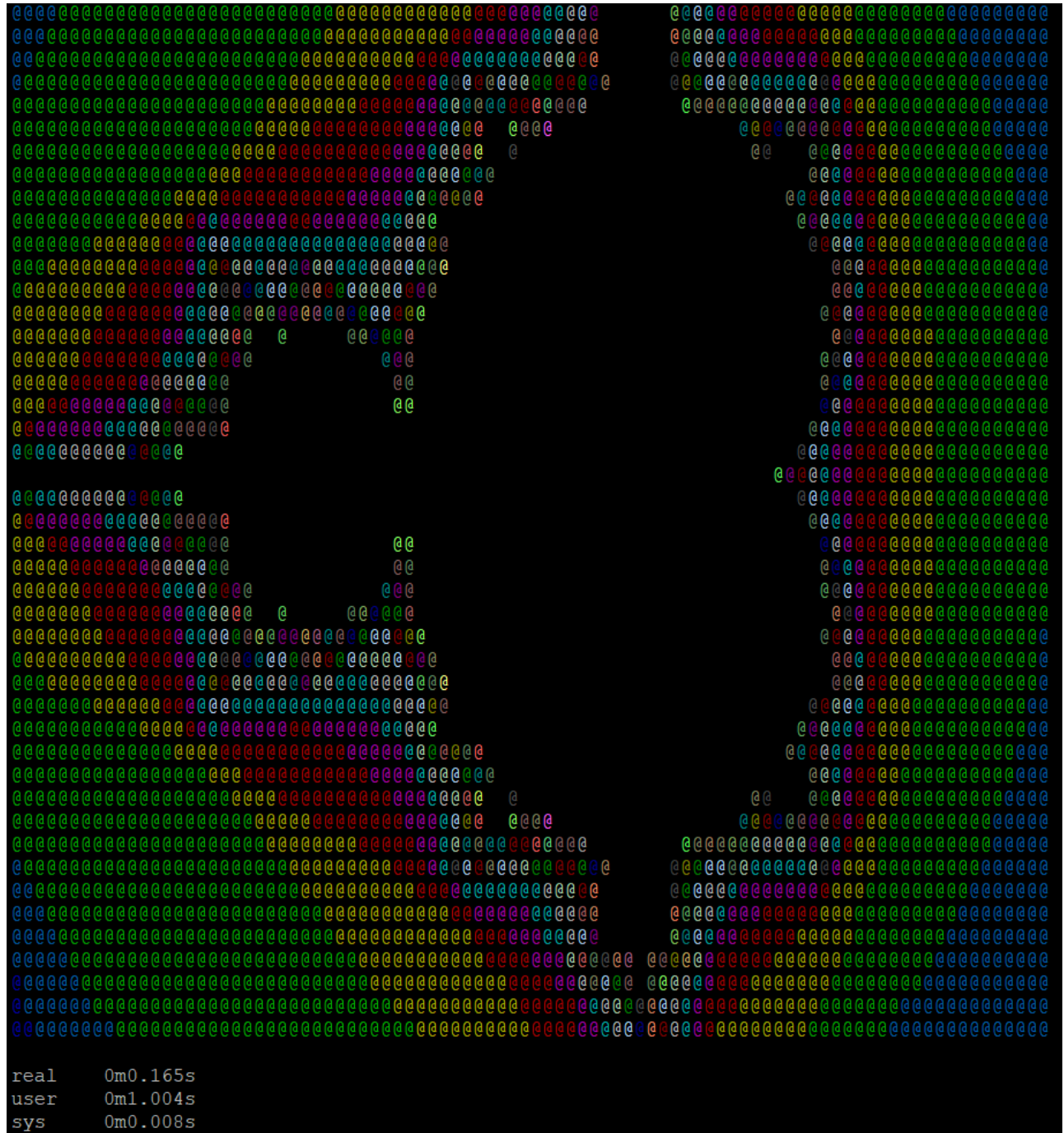
- Για N = 1:

```

real    0m1.033s
user    0m0.984s
sys      0m0.024s

```

- $\Gamma\alpha N = 10$ :



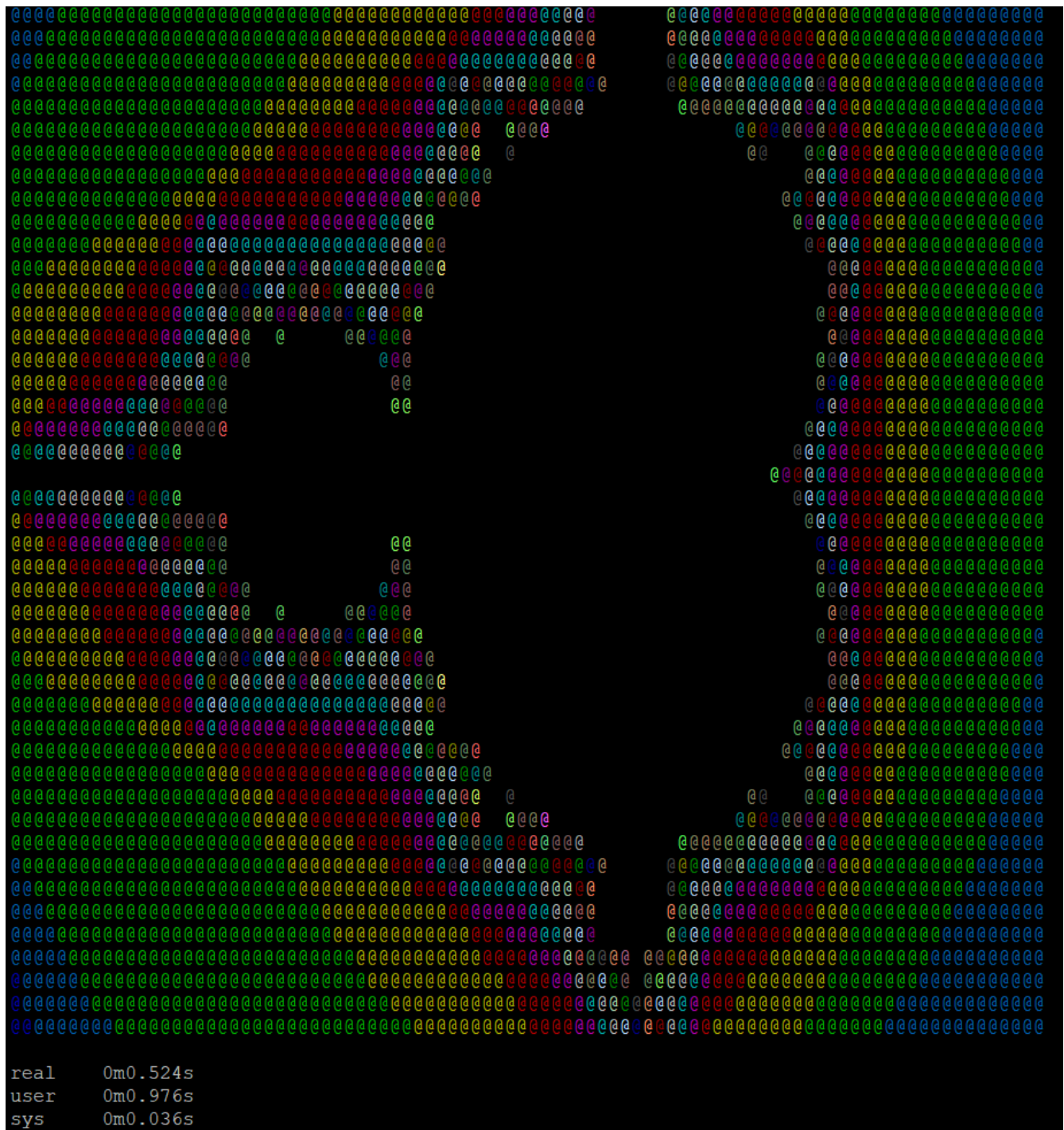
- $\Gamma\alpha N = 30$ :





```
real    0m1.033s
user    0m0.972s
sys     0m0.036s
```

Παραθέτουμε επίσης το output του mandel το οποίο χρησιμοποιεί πολυνηματισμό και συγχρονισμό. Χρησιμοποιούμε για την εκτέλεση 2 νήματα:




Παρατηρούμε ότι η υλοποίηση ακόμα και με χρήση 2 μόνο threads βελτιώνει σημαντικά την απόδοση καθώς ο χρόνος εκτέλεσης υποδιπλασιάζεται.



3). Από τα outputs για διάφορους αριθμούς νημάτων ( $N = 1, 10, 30$ ) που παραθέσαμε στο ερώτημα 1 βλέπουμε πως υπάρχει εμφανής επιτάχυνση του χρόνου εκτέλεσης. Αυτό συμβαίνει, διότι όλα τα νήματα υπολογίζουν ταυτόχρονα τις γραμμές που πρέπει να τυπώσουν, συνεπώς η αύξηση του αριθμού των νημάτων συνεπάγεται αύξηση στον χρόνο εκτέλεσης. Στο κρίσιμο τμήμα βρίσκεται μόνο η κλήση στην `output_mandel_line` η οποία τυπώνει μία γραμμή στην οθόνη.

4). Παρατηρούμε ότι πατώντας `Ctrl-C`, η εκτέλεση του προγράμματος διακόπτεται και το χρώμα των εντολών στον φλοιό γίνεται ίδιο με το χρώμα του τελευταίου χαρακτήρα που τυπώθηκε. Παραθέτουμε την έξοδο του προγράμματος πατώντας `Ctrl-C` κατά τη διάρκεια εκτέλεσης:



Για να λύσουμε αυτό το πρόβλημα και το τερματικό να ανακτήσει το default χρώμα, μπορούμε να χρησιμοποιήσουμε έναν `signal handler` ο οποίος θα εκτελεί την εντολή `reset`, έτσι ώστε όταν λαμβάνεται σήμα `Ctrl-C` να πραγματοποιείται η επιθυμητή ενέργεια.

### ΑΣΚΗΣΗ 1.3

Ο πηγαίος κώδικας φαίνεται παρακάτω:

```
#include <time.h>
#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_thread(ret, msg) \
    do { errno = ret; perror(msg); } while (0)

/* A virtual kindergarten */
struct kgarden_struct {

    /*
     * Here you may define any mutexes / condition variables /
other variables
     * you may need.
     */

    pthread_cond_t check_c;
    pthread_cond_t check_t;

    pthread_mutex_t mutex_children;
    pthread_mutex_t mutex_teachers;

    /*
     * You may NOT modify anything in the structure below this
     * point.
     */
    int vt;
    int vc;
    int ratio;

    pthread_mutex_t mutex;
};

/*
 * 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
*/

    struct kgarten_struct *kg;
    int is_child; /* Nonzero if this thread simulates children,
zero otherwise */

    int thrid; /* Application-defined thread id */
    int thrcnt;
    unsigned int rseed;
};

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 *safe_malloc(size_t size)
{
    void *p;

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

    return p;
}

void usage(char *argv0)
{
    fprintf(stderr, "Usage: %s thread_count child_threads
c_t_ratio\n\n"
        "Exactly two argument required:\n"
        "    thread_count: Total number of threads to
create.\n")

```

```

        "    child_threads: The number of threads simulating
children.\n"
        "    c_t_ratio: The allowed ratio of children to
teachers.\n\n",
        argv0);
    exit(1);
}

void bad_thing(int thrid, int children, int teachers)
{
    int thing, sex;
    int namecnt, nameidx;
    char *name, *p;
    char buf[1024];

    char *things[] = {
        "Little %s put %s finger in the wall outlet and got
electrocuted!",
        "Little %s fell off the slide and broke %s head!",
        "Little %s was playing with matches and lit %s hair
on fire!",
        "Little %s drank a bottle of acid with %s lunch!",
        "Little %s caught %s hand in the paper shredder!",
        "Little %s wrestled with a stray dog and it bit %s
finger off!"
    };
    char *boys[] = {
        "George", "John", "Nick", "Jim", "Constantine",
        "Chris", "Peter", "Paul", "Steve", "Billy", "Mike",
        "Vangelis", "Antony"
    };
    char *girls[] = {
        "Maria", "Irene", "Christina", "Helena", "Georgia",
        "Olga",
        "Sophie", "Joanna", "Zoe", "Catherine", "Marina",
        "Stella",
        "Vicky", "Jenny"
    };

    thing = rand() % 4;
    sex = rand() % 2;

    namecnt = sex ? sizeof(boys)/sizeof(boys[0]) :
sizeof(girls)/sizeof(girls[0]);
    nameidx = rand() % namecnt;
    name = sex ? boys[nameidx] : girls[nameidx];

    p = buf;
    p += sprintf(p, "*** Thread %d: Oh no! ", thrid);

```

```

        p += sprintf(p, things[thing], name, sex ? "his" : "her");
        p += sprintf(p, "\n*** Why were there only %d teachers for %d
children?!\n",
                    teachers, children);

        /* Output everything in a single atomic call */
        printf("%s", buf);
    }

void child_enter(struct thread_info_struct *thr)
{
    if (!thr->is_child) {
        fprintf(stderr, "Internal error: %s called for a
Teacher thread.\n",
                __func__);
        exit(1);
    }

    pthread_mutex_lock(&thr->kg->mutex_children);
    while ((float)thr->kg->vt < ((float)(thr->kg->vc + 1) /
(float)thr->kg->ratio)) {
        pthread_cond_wait(&thr->kg->check_c, &thr->kg-
>mutex_children);
    }
    fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);

    //pthread_mutex_lock(&thr->kg->mutex);
    ++(thr->kg->vc);
    //pthread_mutex_unlock(&thr->kg->mutex);
    pthread_mutex_unlock(&thr->kg->mutex_children);
}

void child_exit(struct thread_info_struct *thr)
{
    if (!thr->is_child) {
        fprintf(stderr, "Internal error: %s called for a
Teacher thread.\n",
                __func__);
        exit(1);
    }

    pthread_mutex_lock(&thr->kg->mutex_children);

    fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);
    //pthread_mutex_lock(&thr->kg->mutex);
    --(thr->kg->vc);
    //pthread_mutex_unlock(&thr->kg->mutex);
    if ((thr->kg->vt - 1) * thr->kg->ratio >= thr->kg->vc) {

```

```

        pthread_cond_broadcast(&thr->kg->check_t);
    }
    pthread_mutex_unlock(&thr->kg->mutex_children);
}

void teacher_enter(struct thread_info_struct *thr)
{
    if (thr->is_child) {
        fprintf(stderr, "Internal error: %s called for a
Child thread.\n",
                __func__);
        exit(1);
    }
    pthread_mutex_lock(&thr->kg->mutex_teachers);
    fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);

    //pthread_mutex_lock(&thr->kg->mutex);
    ++(thr->kg->vt);
    //pthread_mutex_unlock(&thr->kg->mutex);
    if ((float)thr->kg->vt >= ((float)(thr->kg->vc + 1) /
(float)thr->kg->ratio)) {
        pthread_cond_broadcast(&thr->kg->check_c);
    }
    pthread_mutex_unlock(&thr->kg->mutex_teachers);
}

void teacher_exit(struct thread_info_struct *thr)
{
    if (thr->is_child) {
        fprintf(stderr, "Internal error: %s called for a
Child thread.\n",
                __func__);
        exit(1);
    }

    pthread_mutex_lock(&thr->kg->mutex_teachers);
    while ((thr->kg->vt - 1) * thr->kg->ratio < thr->kg->vc) {
        pthread_cond_wait(&thr->kg->check_t, &thr->kg-
>mutex_teachers);
    }
    fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);
    //pthread_mutex_lock(&thr->kg->mutex);
    --(thr->kg->vt);
    //pthread_mutex_unlock(&thr->kg->mutex);
    pthread_mutex_unlock(&thr->kg->mutex_teachers);
}

```

```

/*
 * Verify the state of the kindergarten.
 */
void verify(struct thread_info_struct *thr)
{
    struct kgarten_struct *kg = thr->kg;
    int t, c, r;

    c = kg->vc;
    t = kg->vt;
    r = kg->ratio;

    fprintf(stderr, "                Thread %d: Teachers: %d,
Children: %d\n",
            thr->thrid, t, c);

    if (c > t * r) {
        bad_thing(thr->thrid, c, t);
        exit(1);
    }
}

/*
 * A single thread.
 * It simulates either a teacher, or a child.
 */
void *thread_start_fn(void *arg)
{
    /* We know arg points to an instance of thread_info_struct */
    struct thread_info_struct *thr = arg;
    char *nstr;

    fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr-
>thrcnt);

    nstr = thr->is_child ? "Child" : "Teacher";
    for (;;) {
        fprintf(stderr, "Thread %d [%s]: Entering.\n", thr-
>thrid, nstr);
        if (thr->is_child)
            child_enter(thr);
        else
            teacher_enter(thr);

        fprintf(stderr, "Thread %d [%s]: Entered.\n", thr-
>thrid, nstr);

        /*

```

```

        * We're inside the critical section,
        * just sleep for a while.
        */
        /* usleep(rand_r(&thr->rseed) % 1000000 / (thr-
>is_child ? 10000 : 1)); */
        pthread_mutex_lock(&thr->kg->mutex);
        verify(thr);
        pthread_mutex_unlock(&thr->kg->mutex);

        usleep(rand_r(&thr->rseed) % 1000000);

        fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr-
>thrid, nstr);

        /* CRITICAL SECTION END */

        if (thr->is_child)
            child_exit(thr);
        else
            teacher_exit(thr);

        fprintf(stderr, "Thread %d [%s]: Exited.\n", thr-
>thrid, nstr);

        /* Sleep for a while before re-entering */
        /* usleep(rand_r(&thr->rseed) % 100000 * (thr-
>is_child ? 100 : 1)); */
        usleep(rand_r(&thr->rseed) % 100000);

        pthread_mutex_lock(&thr->kg->mutex);
        verify(thr);
        pthread_mutex_unlock(&thr->kg->mutex);
    }

    fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr-
>thrcnt);

    return NULL;
}

int main(int argc, char *argv[])
{
    int i, ret, thrcnt, chldcnt, ratio;
    struct thread_info_struct *thr;
    struct kgarden_struct *kg;

    /*
     * Parse the command line
     */

```



```

    if (argc != 4)
        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);
    }
    if (safe_atoi(argv[2], &chldcnt) < 0 || chldcnt < 0 ||
chldcnt > thrcnt) {
        fprintf(stderr, "`%s' is not valid for
`child_threads'\n", argv[2]);
        exit(1);
    }
    if (safe_atoi(argv[3], &ratio) < 0 || ratio < 1) {
        fprintf(stderr, "`%s' is not valid for `c_t_ratio'\
n", argv[3]);
        exit(1);
    }

    /*
     * Initialize kindergarten and random number generator
     */
    srand(time(NULL));

    kg = safe_malloc(sizeof(*kg));
    kg->vt = kg->vc = 0;
    kg->ratio = ratio;

    ret = pthread_mutex_init(&kg->mutex, NULL);
    if (ret) {
        perror_pthread(ret, "pthread_mutex_init");
        exit(1);
    }

    /* ... */

    /*
     * Create threads
     */
    thr = safe_malloc(thrcnt * sizeof(*thr));

    for (i = 0; i < thrcnt; i++) {
        /* Initialize per-thread structure */
        thr[i].kg = kg;
        thr[i].thrid = i;
        thr[i].thrcnt = thrcnt;
        thr[i].is_child = (i < chldcnt);
        thr[i].rseed = rand();
    }

```

```

        /* Spawn new thread */
        ret = pthread_create(&thr[i].tid, NULL,
thread_start_fn, &thr[i]);
        if (ret) {
            perror_pthread(ret, "pthread_create");
            exit(1);
        }
    }

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

    printf("OK.\n");

    return 0;
}

```

Ένα στιγμιότυπο του output εκτέλεσης του προγράμματος για  $N = 9$ ,  $C = 7$ ,  $R = 2$  είναι το ακόλουθο:

```

Thread 1 [Child]: Entering.
    Thread 6: Teachers: 1, Children: 2
Thread 6 [Child]: Entering.
    Thread 7: Teachers: 1, Children: 2
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER ENTER
Thread 7 [Teacher]: Entered.
    Thread 7: Teachers: 2, Children: 2
THREAD 0: CHILD ENTER
Thread 0 [Child]: Entered.
    Thread 0: Teachers: 2, Children: 3
THREAD 5: CHILD ENTER
Thread 5 [Child]: Entered.
    Thread 5: Teachers: 2, Children: 4
Thread 4 [Child]: Exiting.
THREAD 4: CHILD EXIT
Thread 4 [Child]: Exited.
Thread 3 [Child]: Exiting.
THREAD 3: CHILD EXIT
Thread 3 [Child]: Exited.
THREAD 8: TEACHER EXIT
Thread 8 [Teacher]: Exited.
    Thread 4: Teachers: 1, Children: 2
Thread 4 [Child]: Entering.
Thread 4 [Child]: Entering.
    Thread 8: Teachers: 1, Children: 2
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER ENTER
THREAD 2: CHILD ENTER
Thread 2 [Child]: Entered.
    Thread 2: Teachers: 2, Children: 3
THREAD 1: CHILD ENTER
Thread 1 [Child]: Entered.
    Thread 1: Teachers: 2, Children: 4
Thread 8 [Teacher]: Entered.
    Thread 8: Teachers: 2, Children: 4
    Thread 3: Teachers: 2, Children: 4
Thread 3 [Child]: Entering.

```

### Απαντήσεις στις ερωτήσεις:

1). Ένας δάσκαλος ο οποίος θέλει να φύγει περιμένει στο condition variable **check\_t** μέχρι ο αριθμός των παιδιών στο νηπιαγωγείο να του επιτρέψει την έξοδο, εκτελώντας την συνάρτηση **pthread\_cond\_wait**. Τα νέα παιδιά που περιμένουν να μπουν στο νηπιαγωγείο περιμένουν στο condition variable **check\_c** μέχρι ο αριθμός δασκάλων στο νηπιαγωγείο να επιτρέψει την έξοδο κάποιου παιδιού. Η εκτέλεση της

**pthread\_cond\_wait(&thr→kg→check\_c, &thr→kg→mutex\_children)**

από κάθε παιδί ξεκλειδώνει το **mutex\_children** ώστε να περιμένουν στο **check\_c** πολλά παιδιά. Για να μπορέσει να μπει κάποιο παιδί, πρέπει να μπει κάποιος δάσκαλος στο νηπιαγωγείο ο οποίος θα εκτελέσει τη συνάρτηση **pthread\_cond\_broadcast(&thr→kg→check\_c)** ξεμπλοκάρωντας το condition variable **check\_c** έτσι ώστε να μπορέσει να μπει ένα παιδί στο νηπιαγωγείο. Ο δάσκαλος που περιμένει να μπει, θα μπει τελικά στο νηπιαγωγείο όταν λάβει **broadcast** στο **check\_c** από κάποιο παιδί που αποφασίσει να βγει ή από κάποιον δάσκαλο που θα μπει.

2). Το πρόγραμμα **kgarten.c** χρησιμοποιεί ένα mutex και για τις 4 συναρτήσεις **child\_enter**, **child\_exit**, **teacher\_enter**, **teacher\_exit**. Με αυτόν τον τρόπο όταν κάποιο thread εισέλθει στο κρίσιμο τμήμα και αφού αυξήσει τον αριθμό των καθηγητών ή μαθητών ανάλογα με το αν αναπαριστά καθηγητή ή μαθητή αντίστοιχα, όταν το απελευθερώσει τότε δεν ξέρουμε αν θα μπει ή αν θα βγει καθηγητής ή μαθητής στη συνέχεια με αποτέλεσμα την πιθανότητα να μην ισχύει η επιθυμητή αναλογία μεταξύ δασκάλων και μαθητών. Για παράδειγμα έστω ότι ένας μαθητής εισέρχεται στο νηπιαγωγείο, δεν έχει μπει κανένας δάσκαλος και έχουμε αναλογία 1 δάσκαλος για 3 μαθητές. Τότε, μπορεί τα επόμενα τρία threads να αναπαριστούν μαθητές που μπαίνουν στο νηπιαγωγείο, συνεπώς τώρα έχουμε 4 νέους μαθητές χωρίς κανένα επιπλέον δάσκαλο, οπότε το πρόγραμμά μας σταματάει, διότι δεν ισχύει η απαιτούμενη αναλογία.

Ωστόσο η δική μας υλοποίηση λύνει αυτά τα προβλήματα με την χρήση δύο mutexes και δύο condition variables όπως φαίνεται από τον κώδικα.

Ένα άλλο πρόβλημα ανακύπτει με την χρήση της συνάρτησης **verify** από κάποιο νήμα. Όταν για παράδειγμα ένα νήμα καλεί την συνάρτηση **verify** για να τυπώσει την τρέχουσα κατάσταση του νηπιαγωγείου, υπάρχει πιθανότητα ενδιάμεσα να παρεμβληθεί η είσοδος ή έξοδος κάποιου παιδιού ή δασκάλου. Συνεπώς το αποτέλεσμα της **verify** δεν θα είναι το σωστό. Το πρόβλημα αυτό παραμένει τόσο στην αρχική υλοποίηση του **kgarten.c** όσο και στην δική μας υλοποίηση.