Εθνικό Μετσόβιο Πολυτεχνείο

Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Λειτουργικά Συστήματα

Άσκηση 3:

Συγχρονισμός

ΟΜΑΔΑ: oslab100

Ασπρογέρακας Ιωάννης (03118942)

Κανατάς Άγγελος-Νικόλαος (03119169)

Σε αυτήν την εργαστηριακή άσκηση, θα χρησιμοποιήσουμε διαδεδομένους μηχανισμούς συγχρονισμού (*POSIX mutexes, semaphores, condition variables και gcc atomic operations*), για να επιλύσουμε 3 προβλήματα **συγχρονισμού** σε πολυνηματικές εφαρμογές βασισμένες στο πρότυπο *POSIX threads*:

- Συγχρονισμός δύο νημάτων για πρόσβαση σε κοινά δεδομένα (***κοινή μεταβλητή*** `val`), έτσι ώστε να μην δημιουργούνται ασυνέπειες (**1.1**).
- Συγχρονισμός νημάτων για παράλληλο υπολογισμό και σχεδίαση στο τερματικό του ***Mandelbrot set*** (**1.2**).
- Συγχρονισμός νημάτων με δεδομένους περιορισμούς για αυτά – ***virtual kindergarten*** (**1.3**).

Άσκηση 1.1 (Συγχρονισμός σε υπάρχοντα κώδικα)

Δίνεται το πρόγραμμα **simplesync.c**, το οποίο λειτουργεί ως εξής: Αφού αρχικοποιήσει μία μεταβλητή `val=0`, δημιουργεί δύο νήματα τα οποία εκτελούνται ταυτόχρονα: το πρώτο νήμα αυξάνει N φορές την τιμή της μεταβλητής `val` κατά 1, το δεύτερο τη μειώνει N φορές κατά 1. Αυτό που θέλουμε να επιτύχουμε είναι συγχρονισμός των δύο νημάτων $\mathbf{T}_{\text{increase}}$ και $\mathbf{T}_{\text{decrease}}$, έτσι ώστε η τελική τιμή της μεταβλητής `val` να είναι **0**.

Αρχικά, χρησιμοποιούμε το παρεχόμενο Makefile για να μεταγλωττίσουμε και να εκτελέσουμε το πρόγραμμα:

```
## Simple sync (two versions)
simplesync-mutex: simplesync-mutex.o
	$(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o

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 -c -o simplesync-atomic.o simplesync.c
```

```
oslaba100@os-node2:~/ex3/helpers$ ls
Makefile    mandel-lib.c  mandel.c       proc-common.h   rand-fork.c
kgarten.c   mandel-lib.h  proc-common.c  pthread-test.c  simplesync.c
oslaba100@os-node2:~/ex3/helpers$ make
gcc -Wall -O2 -pthread -c -o pthread-test.o pthread-test.c
gcc -Wall -O2 -pthread -o pthread-test pthread-test.o
gcc -Wall -O2 -pthread -DSYNC_MUTEX -c -o simplesync-mutex.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-mutex simplesync-mutex.o
gcc -Wall -O2 -pthread -DSYNC_ATOMIC -c -o simplesync-atomic.o simplesync.c
gcc -Wall -O2 -pthread -o simplesync-atomic simplesync-atomic.o
gcc -Wall -O2 -pthread -c -o kgarten.o kgarten.c
gcc -Wall -O2 -pthread -o kgarten kgarten.o
gcc -Wall -O2 -pthread -c -o mandel-lib.o mandel-lib.c
gcc -Wall -O2 -pthread -c -o mandel.o mandel.c
gcc -Wall -O2 -pthread -o mandel mandel-lib.o mandel.o
oslaba100@os-node2:~/ex3/helpers$ ls
Makefile    mandel        mandel.c       pthread-test    simplesync-atomic    simplesync.c
kgarten     mandel-lib.c  mandel.o       pthread-test.c  simplesync-atomic.o
kgarten.c   mandel-lib.h  proc-common.c  pthread-test.o  simplesync-mutex
kgarten.o   mandel-lib.o  proc-common.h  rand-fork.c     simplesync-mutex.o
oslaba100@os-node2:~/ex3/helpers$ ./simplesync-atomic
About to increase variable 1000000 times
About to decrease variable 1000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -835827.
oslaba100@os-node2:~/ex3/helpers$ ./simplesync-mutex
About to increase variable 1000000 times
About to decrease variable 1000000 times
```

```
#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
```

Παρατηρούμε αρχικά ότι δημιουργούνται δύο διαφορετικά εκτελέσιμα (**simplesync-atomic** και **simplesync-mutex**) από το ίδιο αρχείο πηγαίου κώδικα **simplesync.c**. Αυτό οφείλεται στο ότι ο **gcc** κάνει predefine τις macros SYNC_ATOMIC και SYNC_MUTEX μέσω του -Dmacro option για τον preprocessor. Έτσι, όπως βλέπουμε παραπάνω, ανάλογα με το ποια macro έχει γίνει predefined "θέτουμε" το USE_ATOMIC_OPS ίσο με 1 ή 0. Παρακάτω στον κώδικα, ανάλογα με την τιμή που έχει η macro USE_ATOMIC_OPS, ακολουθούμε και διαφορετική λύση για το παραπάνω πρόβλημα συγχρονισμού που πρέπει να επιλύσουμε (συγκεκριμένα, όπως θα δούμε παρακάτω, αν είναι ίση με 1 ακολουθούμε σχήμα συγχρονισμού με *gcc atomic operations*, ενώ αν είναι ίση με 0 ακολουθούμε σχήμα συγχρονισμού με *POSIX mutexes*).

Παρατηρούμε, επίσης, ότι όταν το τρέξουμε (είτε το simplesync-atomic, είτε το simplesync-mutex), δεν έχουμε το σωστό αποτέλεσμα αφού τα νήματα δεν συγχρονίζουν την εκτέλεσή τους. Αυτό σημαίνει ότι σε επίπεδο εντολών assembly που μεταφράζεται ο πηγαίος κώδικας, η αύξηση και μείωση μίας τιμής που βρίσκεται στην μνήμη αντιστοιχεί σε πολλαπλές εντολές, που αν δεν συγχρονιστούν και επομένως "μπλέξουν" μεταξύ τους, θα έχουμε ανεπιθύμητα αποτελέσματα (race conditions). Αξίζει να σημειώσουμε ότι στην πράξη ο μεταγλωττιστής μπορεί να αναδιατάξει εντολές, ο επεξεργαστής να εκτελέσει εντολές εκτός σειράς (Out of Order Execution), ενώ σε πολυεπεξεργαστικά συστήματα μοιραζόμενης μνήμης δεν είναι προφανές πότε θα φανούν οι αλλαγές που έκανε ένας επεξεργαστής σε μία θέση μνήμης στους υπόλοιπους. Επιπλέον, αν ο χρονοδρομολογητής του ΛΣ είναι διακοπτός (*preemptive scheduling*), τότε μία διεργασία-νήμα μπορεί να διακοπεί σε οποιοδήποτε σημείο της εκτέλεσής της.

Αυτό, λοιπόν, απαιτεί τον ορισμό **κρίσιμων τμημάτων (critical sections)** – τμήματα κώδικα στα οποία μπορεί να βρίσκεται το πολύ μία διεργασία-νήμα την φορά και επομένως επιτρέπουν την ατομική εκτέλεση τμημάτων κώδικα. Μπορούμε να επιτύχουμε την ατομική εκτέλεση τμημάτων κώδικα με διάφορους μηχανισμούς συγχρονισμού, όπως **ατομικές εντολές** (*atomic operations*), **περιστροφικά κλειδώματα** (*spinlocks*), **κλειδώματα αμοιβαίου αποκλεισμού** (*mutexes*), **σημαφόρους** (*semaphores*), καθώς και **ελεγκτές/παρακολουθητές** (*monitors*).

Επεκτείνουμε λοιπόν τον παραπάνω κώδικα, έτσι ώστε η εκτέλεση των δύο νημάτων να είναι συγχρονισμένη και επομένως η τιμή της μεταβλητής `val` στο τέλος της εκτέλεσης των δύο νημάτων $\mathbf{T_{increase}}$ και $\mathbf{T_{decrease}}$ να είναι ίση με **0**. Εδώ θα χρησιμοποιήσουμε σχήμα συγχρονισμού με **gcc atomic operations** (ατομικές λειτουργίες, όπως ορίζονται από το υλικό και εξάγονται στον προγραμματιστή μέσω ειδικών εντολών – *builtins* – του μεταγλωττιστή gcc) και σχήμα συγχρονισμού με **POSIX mutexes**. Τα εκτελέσιμα για τις δύο διαφορετικές λύσεις στο παραπάνω πρόβλημα συγχρονισμού είναι τα **simplesync-atomic** και **simplesync-mutex** αντίστοιχα, όπως αναφέραμε παραπάνω.

Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.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)
```

```c
#define N 1000000

#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; //mutex initialization

void *increase_fn(void *arg) //T_increase
{
        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 with gcc atomic
                        __sync_fetch_and_add(&ip, 1); //operations
                } else { //sync with mutex
                        ret=pthread_mutex_lock(&mutex); //acquire the lock (or
                        if(ret)                         //wait to acquire it)
                                perror_pthread(ret, "pthread_mutex_lock");
                        ++(*ip); //critical section
                        ret=pthread_mutex_unlock(&mutex); //release the lock
                        if(ret)
                                perror_pthread(ret, "pthread_mutex_unlock");
                }
        }
        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}

void *decrease_fn(void *arg) //T_decrease
{
        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 with gcc atomic
                        __sync_fetch_and_add(&ip, -1); //operations
                } else { //sync with mutex
                        ret=pthread_mutex_lock(&mutex); //acquire the lock (or
                        if(ret)                         //wait to acquire it)
                                perror_pthread(ret, "pthread_mutex_lock");
                        --(*ip); //critical section
                        ret=pthread_mutex_unlock(&mutex); //release the lock
                        if(ret)
                                perror_pthread(ret, "pthread_mutex_unlock");
                }
        }
        fprintf(stderr, "Done decreasing variable.\n");

        return NULL;
}
```

```c
int main(int argc, char *argv[])
{ /* main thread */
        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);

        ret=pthread_mutex_destroy(&mutex); //destroy the mutex object
        if(ret)
                perror_pthread(ret, "pthread_mutex_destroy");

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

        return ok;
}
```

Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Μεταγλωττίζοντας το και εκτελώντας τα δύο προγράμματα που προκύπτουν έχουμε:

```
oslaba100@os-node2:~/ex3/1.1$ ls
Makefile           simplesync-atomic.o  simplesync-mutex.o
simplesync-atomic  simplesync-mutex     simplesync.c
oslaba100@os-node2:~/ex3/1.1$ ./simplesync-atomic
About to increase variable 1000000 times
About to decrease variable 1000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.
oslaba100@os-node2:~/ex3/1.1$ ./simplesync-mutex
About to increase variable 1000000 times
About to decrease variable 1000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.
```

Παρατηρούμε, λοιπόν, την ορθή λειτουργία των προγραμμάτων και επομένως τον συγχρονισμό των δύο νημάτων $\mathbf{T}_{\text{increase}}$ και $\mathbf{T}_{\text{decrease}}$.

1, 2.

Παρακάτω, χρησιμοποιώντας την εντολή `time(1)`, φαίνεται ο χρόνος εκτέλεσης των εκτελέσιμων (για ***N=1000000***):

- **Χωρίς συγχρονισμό**

```
oslaba100@os-node1:~/ex3/helpers$ ls
Makefile   mandel        mandel.c       pthread-test    simplesync
kgarten    mandel-lib.c  mandel.o       pthread-test.c  simplesync.c
kgarten.c  mandel-lib.h  proc-common.c  pthread-test.o  simplesync.o
kgarten.o  mandel-lib.o  proc-common.h  rand-fork.c
oslaba100@os-node1:~/ex3/helpers$ time ./simplesync
About to increase variable 1000000 times
About to decrease variable 1000000 times
Done increasing variable.
Done decreasing variable.
NOT OK, val = -993587.

real    0m0.006s
user    0m0.008s
sys     0m0.000s
```

- **Συγχρονισμός με χρήση ατομικών εντολών (gcc atomic operations)**

```
oslaba100@os-node1:~/ex3/1.1$ ls
Makefile           simplesync-atomic.o  simplesync-mutex.o
simplesync-atomic  simplesync-mutex     simplesync.c
oslaba100@os-node1:~/ex3/1.1$ time ./simplesync-atomic
About to increase variable 1000000 times
About to decrease variable 1000000 times
Done increasing variable.
Done decreasing variable.
OK, val = 0.

real    0m0.012s
user    0m0.016s
sys     0m0.004s
```

- **Συγχρονισμός με χρήση κλειδωμάτων αμοιβαίου αποκλεισμού (POSIX mutexes)**

```
oslaba100@os-node1:~/ex3/1.1$ ls
Makefile           simplesync-atomic.o  simplesync-mutex.o
simplesync-atomic  simplesync-mutex     simplesync.c
oslaba100@os-node1:~/ex3/1.1$ time ./simplesync-mutex
About to increase variable 1000000 times
About to decrease variable 1000000 times
Done decreasing variable.
Done increasing variable.
OK, val = 0.

real    0m2.689s
user    0m2.636s
sys     0m2.736s
```

Παρατηρούμε, λοιπόν, ότι ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό είναι μεγαλύτερος σε σχέση με το χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό. Αυτό συμβαίνει, διότι στο αρχικό πρόγραμμα (χωρίς συγχρονισμό), δεν υπάρχει ο περιορισμός κάθε νήμα να εκτελεί ***ατομικά*** τον κώδικα στο κρίσιμο τμήμα που έχουμε ορίσει, και επομένως η εκτέλεση του κρίσιμου τμήματος από τα δύο νήματα είναι "παράλληλη". Όταν συγχρονίζουμε τα δύο νήματα (και με τους δύο μηχανισμούς συγχρονισμού που χρησιμοποιήσαμε παραπάνω), τότε στο κρίσιμο τμήμα βρίσκεται το πολύ ένα νήμα και επομένως η εκτέλεσή του από τα δύο νήματα είναι "σειριακή".

Συγκεκριμένα, όταν χρησιμοποιούμε σχήμα συγχρονισμού με **gcc atomic operations**, τότε ο συγχρονισμός υλοποιείται σε επίπεδο υλικού και εγγυάται την ***ατομική*** εκτέλεση των εντολών που βρίσκονται στο κρίσιμο τμήμα που έχουμε ορίσει (υπάρχει περιορισμένος αριθμός ατομικών εντολών και επομένως για πιο "δύσκολα" προβλήματα συγχρονισμού καταφεύγουμε στους άλλους μηχανισμούς που αναφέραμε παραπάνω, οι οποίοι "εσωτερικά" στην δομή τους χρησιμοποιούν και ατομικές εντολές). Σε σχέση με το σχήμα συγχρονισμού με **POSIX mutexes**, λοιπόν, ο χρόνος εκτέλεσης είναι μικρότερος. Αυτό συμβαίνει λόγω της πιο σύνθετης δομής των *POSIX mutexes* (περιέχουν "εσωτερικά" και ατομικές εντολές όπως αναφέραμε), που επομένως η χρήση τους αντιστοιχεί σε περισσότερες εντολές σε επίπεδο assembly, αλλά και κυρίως στο γεγονός ότι εμπλέκουν και το ΛΣ ώστε να *"βάζει σε αναμονή"* και να *"ξυπνάει"* τις διεργασίες-νήματα όταν αυτές θα πρέπει να περιμένουν για να αποκτήσουν το lock και όταν είναι ελεύθερο το lock για να το αποκτήσουν αντίστοιχα. Επομένως, η απαίτηση για αυτές τις υπηρεσίες από το ΛΣ προσθέτει χρόνο για **context switch** και **δρομολόγηση** των διεργασιών-νημάτων.

3, 4.

Παρακάτω, φαίνεται το τροποποιημένο Makefile καθώς και η διαδικασία μεταγλώττισης για να παράγουμε τους ενδιάμεσους κώδικες assembly για τις δύο λύσεις στο παραπάνω πρόβλημα συγχρονισμού:

```
CC = gcc
CFLAGS = -O2 -pthread

all: simplesync-mutex simplesync-atomic

simplesync-mutex: simplesync-mutex.o
	$(CC) $(CFLAGS) -o simplesync-mutex simplesync-mutex.o

simplesync-atomic: simplesync-atomic.o
	$(CC) $(CFLAGS) -o simplesync-atomic simplesync-atomic.o

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 -c -o simplesync-atomic.o simplesync.c

simplesync-mutex.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c

simplesync-atomic.s: simplesync.c
	$(CC) $(CFLAGS) -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c

clean:
	rm -f *.o *.s simplesync-mutex simplesync-atomic
```

```
oslaba100@os-node2:~/ex3/1.1$ ls
Makefile  simplesync.c
oslaba100@os-node2:~/ex3/1.1$ make simplesync-atomic.s
gcc -O2 -pthread -DSYNC_ATOMIC -S -g -o simplesync-atomic.s simplesync.c
oslaba100@os-node2:~/ex3/1.1$ make simplesync-mutex.s
gcc -O2 -pthread -DSYNC_MUTEX -S -g -o simplesync-mutex.s simplesync.c
```

Παρακάτω, λοιπόν, φαίνονται οι εντολές στις οποίες μεταφράζεται η χρήση ατομικών λειτουργιών του gcc:

- $\mathbf{T_{increase}}$

```
.L2:
        .loc 1 36 0
        lock addq       $1, 8(%rsp)
```

- $\mathbf{T_{decrease}}$

```
.L7:
        .loc 1 60 0
        lock subq       $1, 8(%rsp)
```

Για το **LOCK prefix (lock)**:

The LOCK # signal is asserted during execution of the instruction following the lock prefix. This signal can be used in a multiprocessor system to ensure exclusive use of shared memory while LOCK # is asserted.

Οι εντολές στις οποίες μεταφράζεται η χρήση POSIX mutexes φαίνονται παρακάτω:

```
.LHOTE14:
        .globl  mutex
        .bss
        .align 32
        .type   mutex, @object
        .size   mutex, 40
```

- $\mathbf{T_{increase}}$

```
.L4:
        .loc 1 38 0
        movl    $mutex, %edi
        call    pthread_mutex_lock
```

```
.L2:
        .loc 1 41 0
        movl    (%r12), %eax
        .loc 1 42 0
        movl    $mutex, %edi
        .loc 1 41 0
        addl    $1, %eax
        movl    %eax, (%r12)
        .loc 1 42 0
        call    pthread_mutex_unlock
```

- $\mathbf{T_{decrease}}$

```
.L19:
        .loc 1 62 0
        movl    $mutex, %edi
        call    pthread_mutex_lock
```

```
.L17:
        .loc 1 65 0
        movl    (%r12), %eax
        .loc 1 66 0
        movl    $mutex, %edi
        .loc 1 65 0
        subl    $1, %eax
        movl    %eax, (%r12)
        .loc 1 66 0
        call    pthread_mutex_unlock
```

Η δομή των **POSIX mutexes** καθώς και οι συναρτήσεις χειρισμού αυτών (**pthread_mutex_lock()**, **pthread_mutex_unlock()**) είναι αρκετά πολύπλοκες, ενώ ο πηγαίος κώδικάς τους βρίσκεται εδώ:

pthread_mutex_lock.c

pthread_mutex_unlock.c

Για να δούμε την assembly της **pthread_mutex_lock** κάνουμε τα παρακάτω:

Static linking

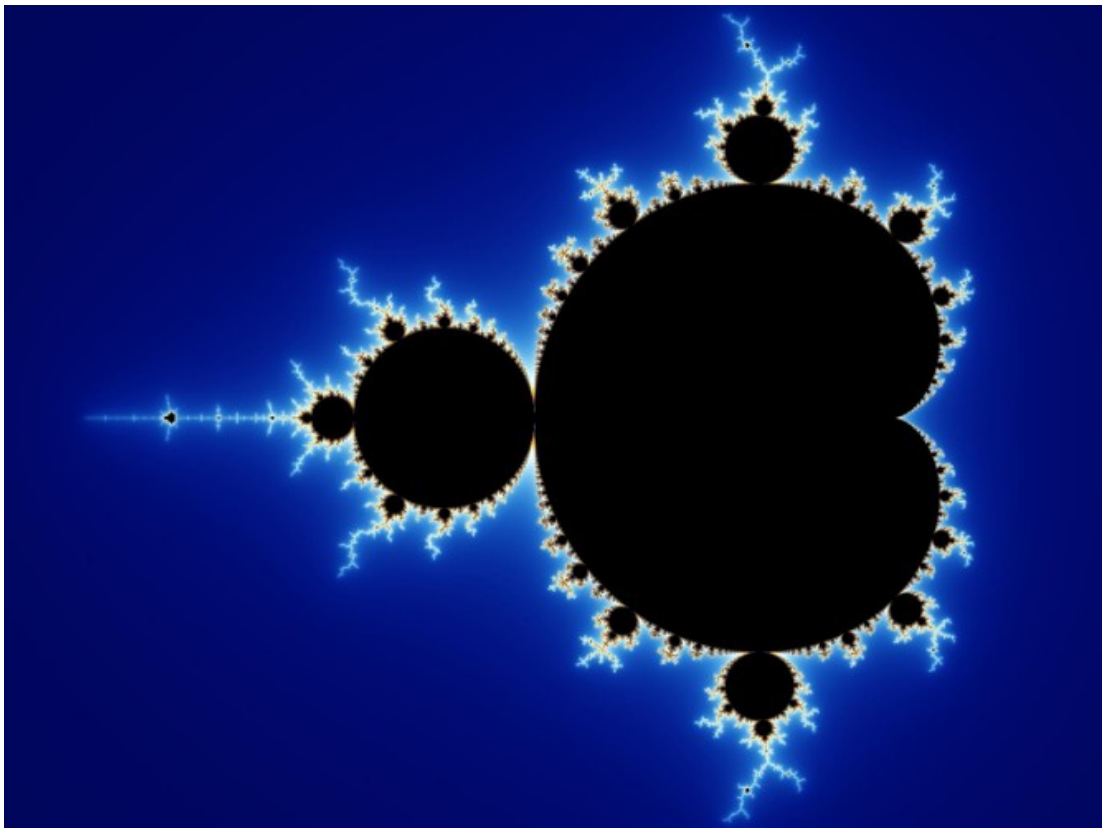
```
simplesync-mutex: simplesync-mutex.o
        $(CC) $(CFLAGS) -static -o simplesync-mutex simplesync-mutex.o
```

```
angelos@ubuntu-linux:~/Documents/source_code/c/os-lab/ex3/1.1$ ls
Makefile           simplesync-atomic.o  simplesync-mutex
simplesync-atomic  simplesync.c         simplesync-mutex.o
angelos@ubuntu-linux:~/Documents/source_code/c/os-lab/ex3/1.1$ gdb -q simplesync-mutex
Reading symbols from simplesync-mutex...
(gdb) disassemble pthread_mutex_lock
Dump of assembler code for function __pthread_mutex_lock:
   0x00000000004052a0 <+0>:     endbr64
   0x00000000004052a4 <+4>:     mov    0x10(%rdi),%eax
   0x00000000004052a7 <+7>:     mov    %eax,%edx
   0x00000000004052a9 <+9>:     and    $0x17f,%edx
   0x00000000004052af <+15>:    nop
   0x00000000004052b0 <+16>:    and    $0x7c,%eax
   0x00000000004052b3 <+19>:    jne    0x405330 <__pthread_mutex_lock+144>
   0x00000000004052b5 <+21>:    push   %rbx
   0x00000000004052b6 <+22>:    sub    $0x10,%rsp
   0x00000000004052ba <+26>:    test   %edx,%edx
   0x00000000004052bc <+28>:    jne    0x405338 <__pthread_mutex_lock+152>
   0x00000000004052be <+30>:    mov    0xda184(%rip),%ecx        # 0x4df448 <__pthread_force_elision>
   0x00000000004052c4 <+36>:    test   %ecx,%ecx
   0x00000000004052c6 <+38>:    jne    0x405300 <__pthread_mutex_lock+96>
   0x00000000004052c8 <+40>:    xor    %eax,%eax
   0x00000000004052ca <+42>:    mov    $0x1,%edx
   0x00000000004052cf <+47>:    lock cmpxchg %edx,(%rdi)
   0x00000000004052d3 <+51>:    jne    0x405370 <__pthread_mutex_lock+208>
   0x00000000004052d9 <+57>:    mov    0x8(%rdi),%edx
   0x00000000004052dc <+60>:    test   %edx,%edx
   0x00000000004052de <+62>:    jne    0x405425 <__pthread_mutex_lock+389>
   0x00000000004052e4 <+68>:    mov    %fs:0x2d0,%eax
   0x00000000004052ec <+76>:    mov    %eax,0x8(%rdi)
   0x00000000004052ef <+79>:    addl   $0x1,0xc(%rdi)
   0x00000000004052f3 <+83>:    nop
   0x00000000004052f4 <+84>:    xor    %eax,%eax
   0x00000000004052f6 <+86>:    add    $0x10,%rsp
   0x00000000004052fa <+90>:    pop    %rbx
```

atomic operation

.
.
.

Άσκηση 1.2 (Παράλληλος υπολογισμός του συνόλου Mandelbrot)



Το **σύνολο Mandelbrot** ορίζεται ως το σύνολο των σημείων **c** του μιγαδικού επιπέδου, για τα οποία η ακολουθία $z_{n+1}=z_n^2+c$, $z_0=0$ είναι φραγμένη Ένας απλός αλγόριθμος για την σχεδίασή του είναι ο εξής: Ξεκινάμε αντιστοιχίζοντας την επιφάνεια σχεδίασης σε μία περιοχή του μιγαδικού επιπέδου. Για κάθε pixel παίρνουμε τον αντίστοιχο μιγαδικό c και υπολογίζουμε επαναληπτικά την ακολουθία $z_{n+1}=z_n^2+c$, $z_0=0$, έως ότου $|z_n|>2$ (*escape condition*) ή το n ξεπεράσει μία προκαθορισμένη τιμή (*iteration limit*). Ο αριθμός των επαναλήψεων που απαιτήθηκαν αντιστοιχίζεται ως χρώμα του συγκεκριμένου pixel (***escape time algorithm***).

Δίνεται το πρόγραμμα **mandel.c**, το οποίο υπολογίζει και εξάγει στο τερματικό εικόνα του συνόλου Mandelbrot, με ένα μπλοκ χαρακτήρων διαστάσεων **`x_chars`** στηλών και **`y_chars`** γραμμών. Η εκτέλεση του προγράμματος είναι ***σειριακή***, αφού έχουμε μόνο ένα νήμα εκτέλεσης που υπολογίζει και εκτυπώνει το μπλοκ χαρακτήρων γραμμή προς γραμμή.

Εμείς θα επεκτείνουμε το παραπάνω πρόγραμμα, έτσι ώστε ο υπολογισμός να κατανέμεται σε **Ν νήματα εκτέλεσης** (*POSIX threads*). Έτσι, η κατανομή του υπολογιστικού φόρτου γίνεται ανά γραμμές: Για Ν νήματα, το *i*-οστό (με *i*=0, 1, 2, …, N-1) αναλαμβάνει τις γραμμές *i*, *i*+N, *i*+2N, … (ξεκινώντας από το πρώτο νήμα, ανάθεση γραμμών με κυκλική επαναφορά). Ωστόσο, αυτό απαιτεί **συγχρονισμό** των Ν νημάτων έτσι ώστε η σειρά εκτύπωσης των γραμμών να είναι η σωστή. Αυτό θα το επιτύχουμε με το σχήμα συγχρονισμού που φαίνεται παρακάτω στον κώδικά μας, χρησιμοποιώντας **σημαφόρους** (*POSIX semaphores*).

Ο **πηγαίος κώδικας (source code)** της άσκησης φαίνεται παρακάτω:

```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 <errno.h>

#include "../helpers/mandel-lib.h"
#include "../helpers/proc-safe.h" //contains safe_atoi() and safe_malloc()

#define MANDEL_MAX_ITERATION 100000

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

/*
 * 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);
        }
}
/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt is not
 * drawn in a funny colour if the user "terminates" the execution with Ctrl-C.
 */
void sigint_handler(int signum)
{
        reset_xterm_color(1);
        exit(1);
}

sem_t *sem; //globalize the address so it can be visible from all threads
int nrthreads;//globalize the thread_count so it can be visible from all threads

/*
 * Each thread computes and outputs the lines i, i+n, i+2*n, i+3*n, ..., where n
 * is the thread_count and i is the thread index. The computation is parallel
 * but the output must be in the right order. The serialization of the output is
 * achieved by the use of thread_count number of semaphores. The output of the
 * lines is the critical section, which means that only one thread must output
 * at a time. The first thread (i=0) has an 'unlocked' semaphore, so it
 * outputs its line, blocks its semaphore and then unlocks the next thread to
 * output (all the other threads have ‘locked’ semaphores). This circularly
 * leads to the right ordering of the output. Note that the computation is
 * parallel because it is not included in the critical section. If we add it to
 * the critical section the program execution is not parallel!
 */
```

```
void *compute_and_output_mandel_line(void *thread_index)
{
        int i, color_val[x_chars]; //color_val is not shared among threads

        for(i=(int)thread_index; i<y_chars; i+=nrthreads) {
                compute_mandel_line(i, color_val); //the computation is parallel
                if(sem_wait(&sem[(int)thread_index])<0) { //wait for my turn to
                                                          //output, then lock
                        perror("sem_wait");               //my sema till someone
                        exit(1);                          //unlocks me to output
                }                                         //again
                output_mandel_line(1, color_val); //critical section
                if(sem_post(&sem[((int)thread_index+1)%nrthreads])<0) {
                        perror("sem_post"); //unlock circularly the next thread
                        exit(1);            //to output
                }
        }

        return NULL;
}

int main(int argc, char **argv)
{ //main-thread
        int i, ret;

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

        /*
         * check if arguments are OK
         */
        if((argc!=2)||(safe_atoi(argv[1], &nrthreads)<0)||(nrthreads<=0)) {
                fprintf(stderr, "Usage: %s thread_count\n\n"
                        "Exactly one argument required:\n"
                        "       thread_count: The number of threads to create.\
n",
                        argv[0]);
                        exit(1);
        }

        /*
         * signal handling
         */
        struct sigaction sa;
        sa.sa_handler=sigint_handler;
        sa.sa_flags=0;
        sigemptyset(&sa.sa_mask);
        if(sigaction(SIGINT, &sa, NULL)<0) {
                perror("sigaction");
                exit(1);
        }

        sem=(sem_t*)safe_malloc(nrthreads*sizeof(sem_t)); //the number of
                                                          //semaphores we use
                                                          //is equal to the
                                                          //thread_count

        if(sem_init(&sem[0], 0, 1)<0) { //initialize first semaphore to value 1
                                        //(unlocked)
                perror("sem_init");     //(thread[0] outputs first the 0 line)
                exit(1);
        }
        for(i=1; i<nrthreads; i++) { //initialize the other semaphores to value
                                     //0 (locked)
```

```
                if(sem_init(&sem[i], 0, 0)<0) {
                        perror("sem_init");
                        exit(1);
                }
        }

        /*
         * create threads
         */
        pthread_t thread[nrthreads];
        for(i=0; i<nrthreads; i++) {
                ret=pthread_create(&thread[i], NULL,
compute_and_output_mandel_line, (void*)i);
                if(ret) {
                        perror_pthread(ret, "pthread_create");
                        exit(1);
                }
        }

        /*
         * wait for all threads to terminate
         */
        for(i=0; i<nrthreads; i++) {
                ret=pthread_join(thread[i], NULL);
                if(ret)
                        perror_pthread(ret, "pthread_join");
        }

        /*
         * destroy the semaphores
         */
        for(i=0; i<nrthreads; i++) {
                if(sem_destroy(&sem[i])<0) {
                        perror("sem_destroy");
                        exit(1);
                }
        }

        free(sem);

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

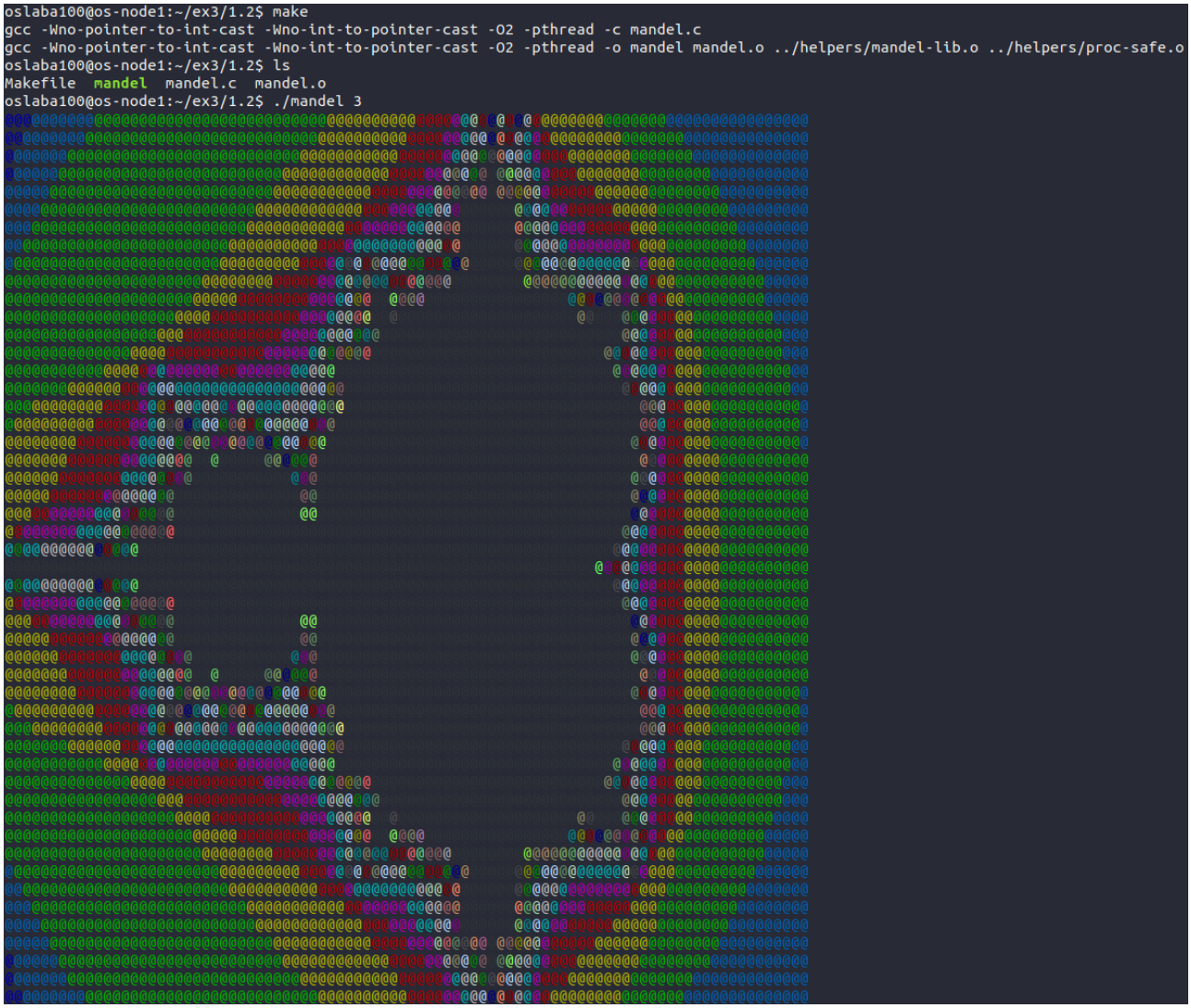
Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
CC = gcc
CFLAGS = -Wno-pointer-to-int-cast -Wno-int-to-pointer-cast -O2 -pthread

.PHONY: mandel clean

mandel: mandel.o ../helpers/mandel-lib.o ../helpers/proc-safe.o
        $(CC) $(CFLAGS) -o mandel mandel.o ../helpers/mandel-lib.o ../helpers/proc-safe.o

mandel.o: mandel.c ../helpers/mandel-lib.h ../helpers/proc-safe.h
        $(CC) $(CFLAGS) -c mandel.c

clean:
        rm -f mandel.o mandel
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **mandel** καθώς και η έξοδος εκτέλεσής του είναι η εξής:

```
oslaba100@os-node1:~/ex3/1.2$ make
gcc -Wno-pointer-to-int-cast -Wno-int-to-pointer-cast -O2 -pthread -c mandel.c
gcc -Wno-pointer-to-int-cast -Wno-int-to-pointer-cast -O2 -pthread -o mandel mandel.o ../helpers/mandel-lib.o ../helpers/proc-safe.o
oslaba100@os-node1:~/ex3/1.2$ ls
Makefile  mandel  mandel.c  mandel.o
oslaba100@os-node1:~/ex3/1.2$ ./mandel 3
```



1, 3.

Παρακάτω, θα αναλύσουμε το σχήμα συγχρονισμού που υλοποιούμε, απαντώντας ταυτόχρονα και στις ερωτήσεις.

Όπως αναφέραμε και παραπάνω, επεκτείναμε το δοθέν πρόγραμμα, έτσι ώστε ο υπολογισμός να κατανέμεται σε **N νήματα εκτέλεσης**. Η κατανομή του υπολογιστικού φόρτου γίνεται ανά γραμμές: Για N νήματα, το *i*-οστό (με *i*=0, 1, 2, …, N-1) αναλαμβάνει τις γραμμές *i*, *i*+N, *i*+2N, … (ξεκινώντας από το πρώτο νήμα, ανάθεση γραμμών με κυκλική επαναφορά). Η εκτύπωση των γραμμών απαιτεί τον συγχρονισμό των νημάτων, έτσι ώστε αυτές να εμφανίζονται με την σωστή σειρά. Επομένως, ορίζουμε το **output των γραμμών ως κρίσιμο τμήμα**, όπως φαίνεται και παραπάνω στον κώδικά μας (το πολύ ένα thread μπορεί να είναι στο κρίσιμο τμήμα μία χρονική στιγμή, οπότε η εκτύπωση των γραμμών γίνεται ατομικά). Ο υπολογισμός των γραμμών δεν βρίσκεται μέσα στο κρίσιμο τμήμα και επομένως είναι ***παράλληλος***, όπως και θέλαμε να

επιτύχουμε. Για τον συγχρονισμό χρησιμοποιούμε τόσους σημαφόρους όσα και τα νήματα εκτέλεσης που έχουμε στο πρόγραμμά μας.

Το σχήμα συγχρονισμού που έχουμε υλοποιήσει έχει ως εξής: Αρχικά, δημιουργούμε τόσους σημαφόρους όσα και τα νήματα εκτέλεσης και τους αρχικοποιούμε όλους στην τιμή 0 (**"locked"**), εκτός από τον σημαφόρο που αντιστοιχεί στο *thread '0'* που τον αρχικοποιούμε στην τιμή 1 (**"unlocked"**). Αυτό το κάνουμε διότι θέλουμε πρώτο στο κρίσιμο τμήμα να μπει το *thread '0'* (οπότε και να κάνει output την γραμμή **'0'**) και έπειτα να ξεκλειδώσει τον σημαφόρο του επόμενου κυκλικά thread (*thread '1'*) που κάνει output την γραμμή **'1'**. Συγκεκριμένα, κάθε thread αφού υπολογίσει την γραμμή που του αντιστοιχεί, περιμένει μέχρι το προηγούμενο κυκλικά thread να το "ξεκλειδώσει", *****έπειτα "κλειδώνει" τον εαυτό του, μπαίνει στο κρίσιμο τμήμα οπότε και κάνει output την γραμμή που του αντιστοιχεί και "ξεκλειδώνει" κυκλικά το επόμενο thread κ.ο.κ.. Loop-άροντας σε κάθε thread έτσι ώστε να εκτυπώσει όλες τις γραμμές που του αντιστοιχούν (*i*, *i*+N, *i*+2N, … και *i*<`y_chars`) με το παραπάνω σχήμα συγχρονισμού επιτυγχάνεται η σωστή σειρά εμφάνισης των γραμμών του μπλοκ χαρακτήρων του Mandelbrot set στο τερματικό, ενώ ο υπολογισμός παραμένει παράλληλος. Αν προσθέταμε τον υπολογισμό στο κρίσιμο τμήμα τότε δεν θα κερδίζαμε τίποτα σε χρόνο εκτέλεσης σε σχέση με το δοθέν πρόγραμμα, αφού ο υπολογισμός θα ήταν ***σειριακός*** (ίσως να είχαμε και μεγαλύτερο χρόνο εκτέλεσης, λόγω των νημάτων και του μηχανισμού συγχρονισμού). Παρακάτω, θα δούμε και πρακτικά, τις διαφορές στον χρόνο εκτέλεσης μεταξύ της σειριακής και της παράλληλης υλοποίησης.

*****: Από τα man-pages για την **sem_wait(3)** έχουμε:

`sem_wait``() decrements (locks) the semaphore pointed to by` *`sem`*`. If the semaphore's value is greater than zero, then the decrement proceeds, and the function returns, immediately. If the semaphore currently has the value zero, then the call blocks until either it becomes possible to perform the decrement (i.e., the semaphore value rises above zero), or a signal handler interrupts the call.`

2.

Τώρα, θα συγκρίνουμε τους **χρόνους εκτέλεσης** του σειριακού και του παράλληλου προγράμματος. Τις μετρήσεις τις κάνουμε στο δικό μας μηχάνημα με τα specs που φαίνονται παρακάτω:

```
angelos@ubuntu-linux:~/Documents/source_code/c/os-lab/ex3/1.2$ cat /proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 113
model name      : AMD Ryzen 5 3600X 6-Core Processor
stepping        : 0
microcode       : 0x8701012
cpu MHz         : 2200.455
cache size      : 512 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
```

- **Σειριακός υπολογισμός (1 νήμα εκτέλεσης)**:

```
time ./mandel
real    0m0.305s
user    0m0.297s
sys     0m0.004s
```

≈×2

- **Παράλληλος υπολογισμός (2 νήματα εκτέλεσης)**:

```
time ./mandel 2
real    0m0.165s
user    0m0.295s
sys     0m0.024s
```

- **Παράλληλος υπολογισμός (6 νήματα εκτέλεσης)**:

```
time ./mandel 6
real    0m0.063s
user    0m0.335s
sys     0m0.008s
```

- **Παράλληλος υπολογισμός (10 νήματα εκτέλεσης)**:

```
time ./mandel 10
real    0m0.044s
user    0m0.354s
sys     0m0.012s
```

- **Παράλληλος υπολογισμός (10000 νήματα εκτέλεσης)**:

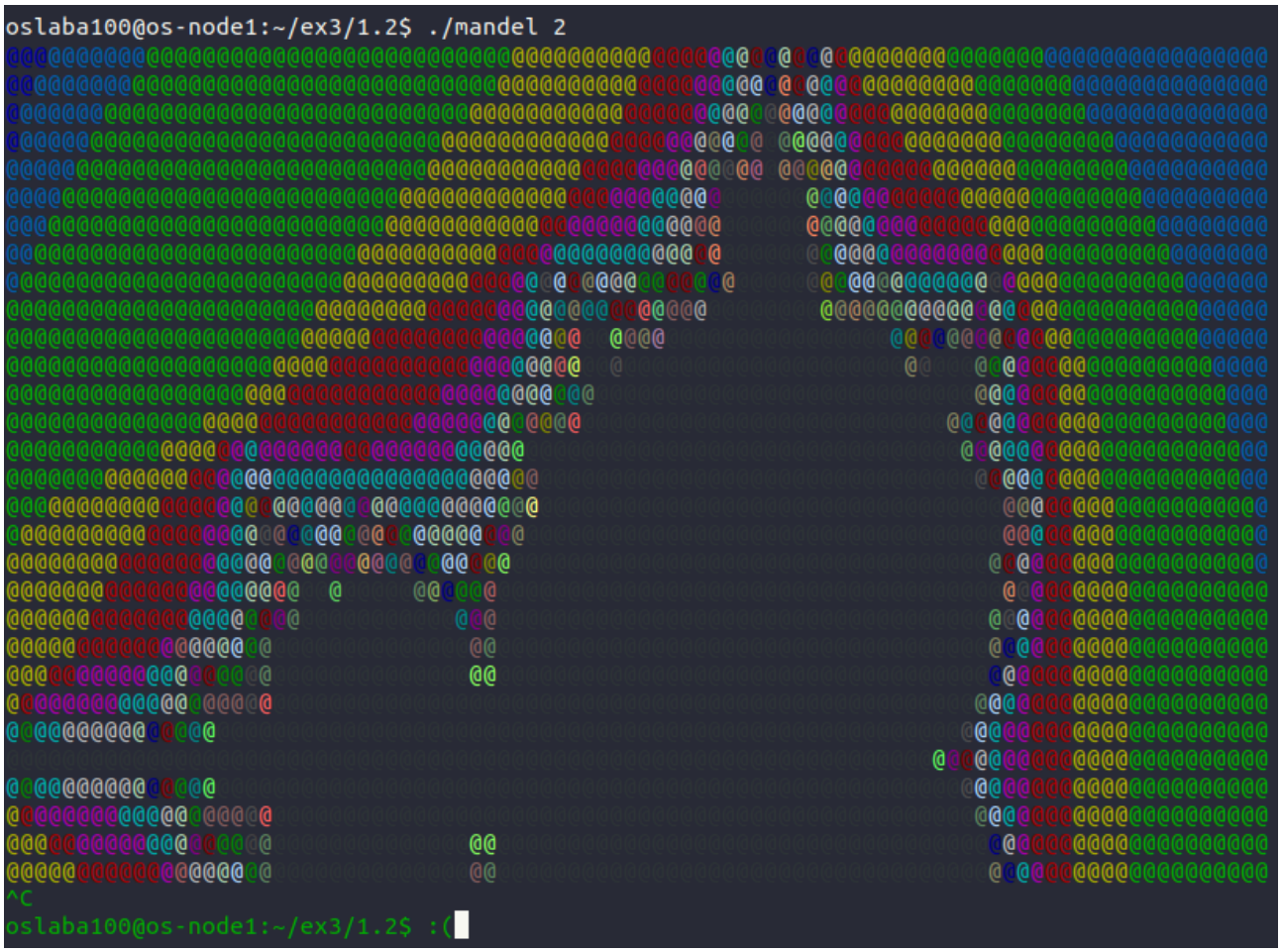
```
time ./mandel 10000
real    0m0.176s
user    0m0.353s
sys     0m0.246s
```

Παρατηρούμε, λοιπόν, τα εξής:

- Ο χρόνος εκτέλεσης του σειριακού υπολογισμού είναι **σχεδόν διπλάσιος** σε σχέση με τον χρόνο εκτέλεσης του παράλληλου υπολογισμού με 2 νήματα.
- Αυξάνοντας τον αριθμό των νημάτων εκτέλεσης N μέχρι κάποιο σημείο (N=10), παρατηρούμε κάποια μετρήσιμη διαφορά στον χρόνο εκτέλεσης, ενώ από εκεί και πέρα (**N>10**) η διαφορά είναι αμελητέα (***βλέπε νόμο του Amdahl***).
- Αυξάνοντας κατά πολύ τον αριθμό των νημάτων **(>5000)** παρατηρείται ολοένα και μεγαλύτερη αύξηση στον χρόνο εκτέλεσης. Αυτό είναι προφανές και οφείλεται στο γεγονός ότι γενικά δεν είναι αμελητέο το "κόστος" δημιουργίας των νημάτων, ενώ το main-thread αναμένει για τον τερματισμό όλων των νημάτων με την *pthread_join()*. Επιπλέον, αξίζει να σημειώσουμε ότι το μπλοκ χαρακτήρων που εκτυπώνουμε εδώ αποτελείται μόνο από **50** γραμμές.

4.

Παραπάνω, στον κώδικά μας, έχουμε ορίσει έναν ***signal handler*** για το σήμα **SIGINT (Ctrl-C)**. Χωρίς αυτόν, ας δούμε τι επίδραση έχει στο τερματικό αν πατήσουμε Ctrl-C (στείλουμε το σήμα SIGINT) κατά την διάρκεια εκτέλεσης του προγράμματός μας:

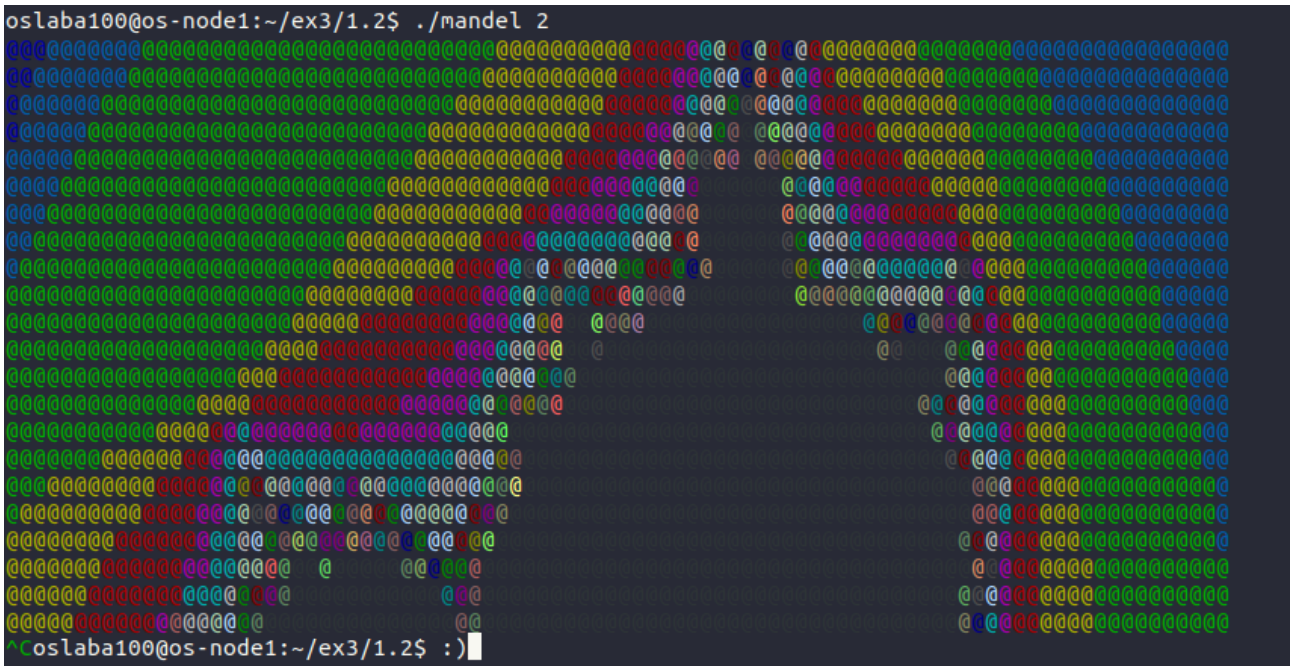


Παρατηρούμε λοιπόν ότι το χρώμα των γραμμάτων στο τερματικό έχει μείνει ίδιο με το χρώμα του τελευταίου χαρακτήρα που τυπώθηκε (βλέπε τις συναρτήσεις **set_xterm_color()** και **output_mandel_line()**). Αν το πρόγραμμα εκτελεστεί χωρίς διακοπή (δεν πατήσουμε Ctrl-C), τότε το χρώμα των γραμμάτων επανέρχεται στο default με την συνάρτηση **reset_xterm_color()** που καλείται πριν το main-thread κάνει return. Επομένως, αυτό που κάναμε παραπάνω στον κώδικά μας ήταν να ορίσουμε έναν signal handler ως εξής:

```
/*
 * signal handling
 */
struct sigaction sa;
sa.sa_handler=sigint_handler;
sa.sa_flags=0;
sigemptyset(&sa.sa_mask);
if(sigaction(SIGINT, &sa, NULL)<0) {
        perror("sigaction");
        exit(1);
}
```

```c
/*
 * Catch SIGINT (Ctrl-C) with the sigint_handler to ensure the prompt is not
 * drawn in a funny colour if the user "terminates" the execution with Ctrl-C.
 */
void sigint_handler(int signum)
{
        reset_xterm_color(1);
        exit(1);
}
```

Επομένως, αν κάνουμε το ίδιο πείραμα σταματώντας πρόωρα την εκτέλεση του προγράμματός μας με Ctrl-C, μετά την προσθήκη του ***signal handling*** έχουμε:



Παρατηρούμε λοιπόν ότι έχει γίνει reset του χρώματος των γραμμάτων στο τερματικό στο default.

(Αξίζει να σημειώσουμε ότι στο μηχάνημά μας δεν παρουσιάζεται αυτό το πρόβλημα στο **gnome-terminal** παρά μόνο στο **xterm**).

Άσκηση 1.3 (Επίλυση προβλήματος συγχρονισμού)

Σε αυτήν την άσκηση, θα προσομοιώσουμε την λειτουργία ενός **νηπιαγωγείου**. Σε ένα νηπιαγωγείο (*"kindergarten"*), βρίσκονται παιδιά και δάσκαλοι. Οι κανονισμοί επιβάλλουν να είναι πάντα παρών ένας δάσκαλος ανά **r** παιδιά (αναλογία παιδιών/δασκάλων το πολύ **r**:1), ώστε να εξασφαλίζεται η σωστή επίβλεψη των παιδιών. Επομένως, θα πρέπει να ισχύει κάθε στιγμή: $c \leq t \cdot r$, όπου **c** ο αριθμός των παιδιών στο νηπιαγωγείο, **t** ο αριθμός των δασκάλων στο νηπιαγωγείο και **r**:1 η μέγιστη επιτρεπόμενη αναλογία παιδιών/δασκάλων.

Αρχικά το νηπιαγωγείο είναι άδειο. Στην συνέχεια, παιδιά και δάσκαλοι μπαίνουν στο νηπιαγωγείο, κάθονται μέσα για λίγο, βγαίνουν, κάθονται για λίγο έξω μέχρι να ξαναμπούν, μπαίνουν κ.ο.κ..

Δίνεται το πρόγραμμα **kgarten.c**, όπου τα παιδιά και οι δάσκαλοι υλοποιούνται χρησιμοποιώντας δύο διαφορετικούς τύπους νημάτων $\mathbf{T_C}$ και $\mathbf{T_T}$ αντίστοιχα (*POSIX threads*) και εκτελούν την παραπάνω λειτουργία. Το νηπιαγωγείο περιγράφεται από στιγμιότυπο της δομής

kgarten_struct, το οποίο είναι κοινό για όλα τα νήματα (βλ. τον πηγαίο κώδικα για περισσότερες λεπτομέρειες). Ας παρατηρήσουμε τι συμβαίνει αν τρέξουμε το παραπάνω πρόγραμμα, π.χ. για **n=10, c=7, r=2**, όπου **n** ο αριθμός των νημάτων, **c** ο αριθμός των παιδιών που αλληλεπιδρούν με το νηπιαγωγείο, **n-c=t**=3 ο αριθμός των δασκάλων που αλληλεπιδρούν με το νηπιαγωγείο και **r**:1 η μέγιστη επιτρεπόμενη αναλογία παιδιών/δασκάλων:

```
oslaba100@orion:~/ex3/helpers$ ./kgarten 10 7 2
Thread 4 of 10. START.
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
THREAD 2: CHILD ENTER
Thread 2 [Child]: Entered.
            Thread 2: Teachers: 0, Children: 1
*** Thread 2: Oh no! Little Nick put his finger in the wall outlet and got electrocuted!
*** Why were there only 0 teachers for 1 children?!
```

Παρατηρούμε λοιπόν ότι στο δοθέν πρόγραμμα τα νήματα δεν συγχρονίζονται σωστά (**τα παιδιά και οι δάσκαλοι μπαίνουν και φεύγουν όποτε θέλουν χωρίς να "ρωτάνε" και να ¨ενημερώνουν"**), <u>οπότε και δεν υπάρχει η σωστή επίβλεψη των παιδιών</u> (η συνάρτηση **verify()** ελέγχει αν ικανοποιείται η συνθήκη σωστής επίβλεψης $c \leq t \cdot r$ και αν όχι μας ενημερώνει ότι κάτι κακό έχει συμβεί σε κάποιο από τα παιδιά).

Επομένως, εμείς θα επεκτείνουμε το παραπάνω πρόγραμμα, έτσι ώστε να συγχρονίζονται σωστά τα νήματα και κάθε στιγμή να εξασφαλίζεται η σωστή επίβλεψη των παιδιών. Αυτό θα το επιτύχουμε με το παρακάτω **σχήμα συγχρονισμού**:

- Για να μπει ένα παιδί στο νηπιαγωγείο (**child_enter()**) θα πρέπει να δει αν υπάρχουν επαρκείς δάσκαλοι για την επίβλεψη των παιδιών μετά την είσοδο του. Επομένως, περιμένει για να μπει έως ότου "ειδοποιηθεί" και ισχύει $\boldsymbol{c < t \cdot r}$.
- Για να φύγει ένας δάσκαλος από το νηπιαγωγείο (**teacher_exit()**) θα πρέπει να δει αν υπάρχουν επαρκείς δάσκαλοι για τα παιδιά που βρίσκονται μέσα στο νηπιαγωγείο μετά την έξοδο του. Επομένως, περιμένει για να βγει έως ότου "ειδοποιηθεί" και ισχύει $\boldsymbol{c \leq (t-1) \cdot r}$.
- Ένα παιδί μπορεί να βγει από το νηπιαγωγείο (**child_exit()**) όποτε θέλει. Όταν το κάνει, "ειδοποιεί¨ παιδιά και δασκάλους που περιμένουν να μπουν και να βγουν αντίστοιχα και αν ισχύει η συνθήκη τους το κάνουν, αλλιώς περιμένουν εκ νέου.
- Ένας δάσκαλος μπορεί να μπει στο νηπιαγωγείο (**teacher_enter()**) όποτε θέλει. Όταν το κάνει, "ειδοποιεί" παιδιά και δασκάλους που περιμένουν να μπουν και να βγουν αντίστοιχα και αν ισχύει η συνθήκη τους το κάνουν, αλλιώς περιμένουν εκ νέου.

Οι παραπάνω λειτουργίες θα πρέπει να γίνονται **ατομικά** για να αποφύγουμε τυχόν race conditions και άλλα προβλήματα.

Υλοποιούμε το παραπάνω σχήμα συγχρονισμού με ***POSIX mutexes***, ***condition variables***. Οι λεπτομέρειες υλοποίησης φαίνονται παρακάτω στον κώδικά μας με επεξηγηματικά σχόλια.

Ο **πηγαίος κώδικας (source code)** για την άσκηση φαίνεται παρακάτω:

```
/*
 * A kindergarten simulator.
 * Bad things happen if teachers and children
 * are not synchronized properly.
 */
```

```c
#include <time.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#include "../helpers/proc-safe.h" //contains safe_atoi and safe_malloc

/*
 * 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)

/* A virtual kindergarten */
struct kgarten_struct {      //we need only one condition variable if we
        pthread_cond_t cond; //don't care about prioritizing children
                             //entering or teachers exiting
        int vt; //number of teachers in the kindergarten
        int vc; //number of children in the kindergarten
        int ratio; //given ratio that must be respected or bad things happen

        pthread_mutex_t mutex; //for critical sections and
};                             //association with the cond var

/*
 * 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; /* Every thread interacts with the
kindergarten */
        int is_child;  /* Nonzero if this thread simulates children, zero
otherwise */

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

void usage(char *argv0)
{
        fprintf(stderr, "Usage: %s thread_count child_threads c_t_ratio\n\n"
                "Exactly three arguments 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)
{
        int ret;
        if (!thr->is_child) {
                fprintf(stderr, "Internal error: %s called for a Teacher
thread.\n",
                        __func__);
                exit(1);
        }

        fprintf(stderr, "THREAD %d: CHILD WAITS TO ENTER\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        while(!((thr->kg->vc)<(thr->kg->vt)*(thr->kg->ratio))) { //i can enter
                                                                 //if c<t*r
                ret=pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
                if(ret) {
                        perror_pthread(ret, "pthread_cond_wait");
                        exit(1);
                }
        }
        ++(thr->kg->vc);
        pthread_mutex_unlock(&thr->kg->mutex);
}
```

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

        fprintf(stderr, "THREAD %d: CHILD WAITS TO EXIT\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        --(thr->kg->vc);
        ret=pthread_cond_broadcast(&thr->kg->cond); //notify children to enter
                                                    //and teachers to exit
        if(ret) {                                   //without prioritizing (one
                                                    //cond var)!
                perror_pthread(ret, "pthread_cond_broadcast");
                exit(1);
        }
        pthread_mutex_unlock(&thr->kg->mutex);
}

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

        fprintf(stderr, "THREAD %d: TEACHER WAITS TO ENTER\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        ++(thr->kg->vt);                            //notify children to enter
                                                    //and teachers to exit
        ret=pthread_cond_broadcast(&thr->kg->cond); //without prioritizing (one
                                                    //cond var)!
        if(ret) {
                perror_pthread(ret, "pthread_cond_broadcast");
                exit(1);
        }
        pthread_mutex_unlock(&thr->kg->mutex);
}

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

        fprintf(stderr, "THREAD %d: TEACHER WAITS TO EXIT\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        while(!((thr->kg->vc)<=(thr->kg->vt-1)*(thr->kg->ratio))) {
                                            //i can exit if c<=(t-1)*r
```

```
                ret=pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
                if(ret) {
                        perror_pthread(ret, "pthread_cond_wait");
                        exit(1);
                }
        }
        --(thr->kg->vt);
        pthread_mutex_unlock(&thr->kg->mutex);
}

/*
 * 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) { //it must always be c<=t*r or bad things happen!
                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(;;) { //user terminates kindergarten simulation with Ctrl-C
                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);

                /* Verify */
                pthread_mutex_lock(&thr->kg->mutex);
                verify(thr); //critical section!      //verify kindergarten
                                                      //state at "random"
                pthread_mutex_unlock(&thr→kg→mutex);  //moments (depends
                                                      //partially on the OS
                                                      //scheduler)
                /* Just stay in the kindergarten for a while */
                //usleep(rand_r(&thr->rseed)%1000000/(thr->is_child ? 10000 :
1));
```

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

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

                /* Verify */
                pthread_mutex_lock(&thr->kg->mutex); //verify kindergarten state
                                                     //at "random" moments
                verify(thr); //critical section!     //(depends partially on the
                                                     //OS scheduler)
                pthread_mutex_unlock(&thr->kg->mutex);
        }

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

        return NULL;
}

/*
 * The output order of the diagnostic messages is "random"
 * and depends partially on the scheduler.
 */

int main(int argc, char *argv[])
{ //main-thread
        int i, ret, thrcnt, chldcnt, ratio;
        struct thread_info_struct *thr;
        struct kgarten_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)); //seed
```

```
kg=safe_malloc(sizeof(*kg));
kg->vt=kg->vc=0; //at the beginning of the simulation the kindergarten
                 //is empty
kg->ratio=ratio;

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

ret=pthread_cond_init(&kg->cond, NULL); //cond var initialization
if(ret) {
        perror_pthread(ret, "pthread_cond_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); //first create all the children
                                     //threads
        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<thrcnt; i++) {
        ret=pthread_join(thr[i].tid, NULL);
        if(ret) {
                perror_pthread(ret, "pthread_join");
                exit(1);
        }
}

/*
 * destroy the mutex and cond var objects
 */
ret=pthread_mutex_destroy(&kg->mutex);
if(ret) {
        perror_pthread(ret, "pthread_mutex_destroy");
        exit(1);
}
ret=pthread_cond_destroy(&kg->cond);
if(ret) {
        perror_pthread(ret, "pthread_cond_destroy");
        exit(1);
}
```

```
        free(thr);
        free(kg);

        printf("OK.\n");

        return 0;
}
```

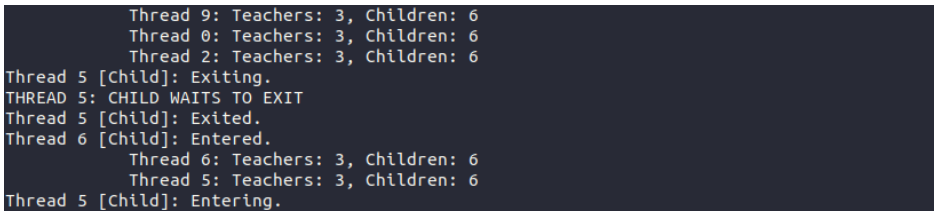
Η λειτουργία του παραπάνω κώδικα φαίνεται από τα επεξηγηματικά σχόλια που περιέχει.

Παρακάτω φαίνεται και το **Makefile** που έχουμε φτιάξει:

```
CC = gcc
CFLAGS = -Wall -O2 -pthread

.PHONY: kgarten cp-kgarten clean

kgarten: kgarten.o ../helpers/proc-safe.o
        $(CC) $(CFLAGS) -o kgarten kgarten.o ../helpers/proc-safe.o

kgarten.o: kgarten.c ../helpers/proc-safe.h
        $(CC) $(CFLAGS) -c kgarten.c

cp-kgarten: cp-kgarten.o ../helpers/proc-safe.o #more children priority
        $(CC) $(CFLAGS) -o cp-kgarten cp-kgarten.o ../helpers/proc-safe.o

cp-kgarten.o: cp-kgarten.c ../helpers/proc-safe.h
        $(CC) $(CFLAGS) -c cp-kgarten.c

clean:
        rm -f kgarten.o cp-kgarten.o kgarten cp-kgarten
```

Η διαδικασία μεταγλώττισης και σύνδεσης ώστε να δημιουργήσουμε το εκτελέσιμο **kgarten** καθώς και η έξοδος εκτέλεσής του ενδεικτικά για **n=10, c=7, r=2** είναι η εξής:

```
oslaba100@os-node1:~/ex3/1.3$ ls
Makefile  cp-kgarten.c  kgarten.c
oslaba100@os-node1:~/ex3/1.3$ make
gcc -Wall -O2 -pthread -c kgarten.c
gcc -Wall -O2 -pthread -o kgarten kgarten.o ../helpers/proc-safe.o
oslaba100@os-node1:~/ex3/1.3$ ls
Makefile  cp-kgarten.c  kgarten  kgarten.c  kgarten.o
oslaba100@os-node1:~/ex3/1.3$ ./kgarten 10 7 2
Thread 0 of 10. START.
Thread 0 [Child]: Entering.
Thread 2 of 10. START.
Thread 2 [Child]: Entering.
Thread 1 of 10. START.
Thread 1 [Child]: Entering.
THREAD 1: CHILD WAITS TO ENTER
THREAD 0: CHILD WAITS TO ENTER
Thread 3 of 10. START.
Thread 3 [Child]: Entering.
THREAD 3: CHILD WAITS TO ENTER
Thread 4 of 10. START.
Thread 5 of 10. START.
Thread 5 [Child]: Entering.
THREAD 5: CHILD WAITS TO ENTER
THREAD 2: CHILD WAITS TO ENTER
Thread 4 [Child]: Entering.
Thread 8 of 10. START.
Thread 8 [Teacher]: Entering.
THREAD 8: TEACHER WAITS TO ENTER
THREAD 4: CHILD WAITS TO ENTER
Thread 7 of 10. START.
Thread 7 [Teacher]: Entering.
THREAD 7: TEACHER WAITS TO ENTER
Thread 9 of 10. START.
Thread 3 [Child]: Entered.
Thread 8 [Teacher]: Entered.
Thread 4 [Child]: Entered.
           Thread 8: Teachers: 2, Children: 4
Thread 7 [Teacher]: Entered.
Thread 9 [Teacher]: Entering.
Thread 5 [Child]: Entered.
Thread 6 of 10. START.
Thread 6 [Child]: Entering.
THREAD 6: CHILD WAITS TO ENTER
           Thread 3: Teachers: 2, Children: 4
THREAD 9: TEACHER WAITS TO ENTER
Thread 1 [Child]: Entered.
           Thread 1: Teachers: 3, Children: 4
Thread 9 [Teacher]: Entered.
           Thread 7: Teachers: 3, Children: 5
Thread 0 [Child]: Entered.
           Thread 5: Teachers: 3, Children: 5
           Thread 4: Teachers: 3, Children: 5
Thread 2 [Child]: Entered.
```



Αν το νηπιαγωγείο λειτουργεί σωστά (υπάρχει πάντα κατάλληλη επίβλεψη των παιδιών), τότε το πρόγραμμα και επομένως η προσομοίωση είναι συνεχούς λειτουργίας (μπορεί να τερματιστεί από τον χρήστη π.χ. με Ctrl-C). Παρατηρούμε, επομένως, ότι αφού το πρόγραμμά μας δεν τερματίζει ποτέ, έχουμε επιτύχει κατάλληλο συγχρονισμό των νημάτων και επομένως κατάλληλη επίβλεψη των παιδιών σύμφωνα με τον κανονισμό του νηπιαγωγείου κάθε στιγμή.

Αξίζει να σημειώσουμε ότι η σειρά εμφάνισης των μηνυμάτων είναι "τυχαία" και εξαρτάται μερικώς από τον scheduler του ΛΣ. Επιπλέον, η συνάρτηση **verify()** καλείται ατομικά από κάθε thread (την έχουμε ορίσει ως ***κρίσιμο τμήμα***, όπως φαίνεται παραπάνω στον κώδικά μας, για να αποφευχθούν race conditions έτσι ώστε να βλέπουμε με "παγωμένο" το νηπιαγωγείο τα πιο updated δεδομένα), σε "τυχαίες" χρονικές στιγμές που εξαρτώνται από τον scheduler του ΛΣ και τις διάρκειες παραμονής των παιδιών και των δασκάλων μέσα και έξω από το νηπιαγωγείο.

1.

Όπως φαίνεται παραπάνω στον κώδικά μας, αλλά και στην περιγραφή που δώσαμε για το σχήμα συγχρονισμού που υλοποιούμε, τα παιδιά και οι δάσκαλοι περιμένουν να μπουν και να βγουν αντίστοιχα από το νηπιαγωγείο μέχρις ότου κάποιο άλλο παιδί ή δάσκαλος τους "ειδοποιήσει" και αν ισχύει η συνθήκη στην οποία περιμένουν τότε μπαίνουν ή βγαίνουν αντίστοιχα. Η αναμονή των νημάτων επιτυγχάνεται με χρήση της **pthread_cond_wait()** που εκτελείται μέσα σε ένα while loop μέχρις ότου να ισχύσει η συνθήκη στην οποία περιμένουν. Συγκεκριμένα, αφού αποκτήσουν το **mutex** και μπουν μέσα στο κρίσιμο τμήμα, "μπλοκάρουν" με την *pthread_cond_wait()* μέχρι να "ειδοποιηθούν" και κάνουν release το mutex*. Έπειτα, ο scheduler επιλέγει ένα άλλο νήμα ώστε να αποκτήσει το mutex. Όταν τελικά "ειδοποιηθούν" με **pthread_cond_signal()** ή **pthread_cond_broadcast()** από ένα άλλο νήμα (παιδί ή δάσκαλος), τότε όταν ξανα-αποκτήσουν το mutex "τσεκάρουν" την συνθήκη τους και αν δεν ισχύει "μπλοκάρουν" εκ νέου μέσω του while loop ξανα-εκτελώντας *pthread_cond_wait()*. Όταν τελικά "ειδοποιηθούν" και ισχύει και η συνθήκη τους τότε βγαίνουν από το while loop και ανάλογα με το αν είναι είναι παιδιά ή δάσκαλοι μπαίνουν στο νηπιαγωγείο ή βγαίνουν από το νηπιαγωγείο αντίστοιχα. Τα παιδιά μπορούν να φύγουν από το νηπιαγωγείο όποτε θέλουν, ενώ οι δάσκαλοι μπορούν να μπουν στο νηπιαγωγείο όποτε θέλουν. Όταν το κάνουν "ειδοποιούν" στο δικό μας σχήμα συγχρονισμού μέσω της **pthread_cond_broadcast()** ΟΛΑ** τα παιδιά και τους δασκάλους που περιμένουν, με τον τρόπο που αναφέραμε παραπάνω.

Στο σχήμα συγχρονισμού μας έχουμε χρησιμοποιήσει **μία μεταβλητή συνθήκης** και επομένως μπορούμε να διαπιστώσουμε ότι δεν δίνεται κάποια προτεραιότητα στο να "ειδοποιούνται" παιδιά ή δάσκαλοι πρώτα, αφού η σειρά με την οποία θα "ξυπνήσουν" τα νήματα εξαρτάται από τον

scheduler του ΛΣ (εδώ λόγω του **Round-robin scheduling** και του γεγονότος ότι δημιουργούμε όλα τα νήματα-παιδιά πρώτα από τα νήματα-δασκάλους δημιουργείται κάποια προτεραιότητα ώστε να "ειδοποιούνται" πρώτα τα παιδιά – οι τυχαίες διάρκειες παραμονής μέσα και έξω από το νηπιαγωγείο προφανώς και επηρεάζουν τα όσα αναφέρουμε εδώ). Παρακάτω **(3.3)** υλοποιούμε σχήμα συγχρονισμού που επιτρέπει πιο έντονα να μπαίνουν στο νηπιαγωγείο παιδιά που φτάνουν με μεγαλύτερη πιθανότητα από δασκάλους που περιμένουν για να φύγουν.

*: Από τα man-pages για την **pthread_cond_wait(3p)** έχουμε:

```
These functions atomically release mutex and cause the calling thread to block
on the condition variable cond; atomically here means ``atomically with respect
to access by another thread to the mutex and then the condition variable''. That
is, if another thread is able to acquire the mutex after the about-to-block
thread has released it, then a subsequent call to pthread_cond_broadcast() or
pthread_cond_signal() in that thread shall behave as if it were issued after the
about-to-block thread has blocked.
```

: Από τα man-pages για τις **pthread_cond_broadcast(3p) και **pthread_cond_signal(3p)** έχουμε:

```
The pthread_cond_broadcast() function shall unblock all threads
currently blocked on the specified condition variable cond.

The pthread_cond_signal() function shall unblock at least one of
he threads that are blocked on the specified condition variable
cond (if any threads are blocked on cond).

If more than one thread is blocked on a condition variable, the scheduling
policy shall determine the order in which threads are
unblocked.
```

2.

Παραπάνω στον κώδικά μας, έχουμε εξαλείψει όλες τις καταστάσεις συναγωνισμού (*races*) που θα προέκυπταν, μέσω του ορισμού κρίσιμων τμημάτων και εξασφάλισης **ατομικής εκτέλεσης** αυτών με χρήση ***POSIX mutexes***. Έστω για παράδειγμα ότι δεν εξασφαλίζαμε την ατομική εκτέλεση τμημάτων κώδικα όπου δύο νήματα έχουν πρόσβαση σε κοινή μεταβλητή. Συγκεκριμένα, έστω ότι ένα νήμα-δάσκαλος μπαίνει στο νηπιαγωγείο και ταυτόχρονα ένα νήμα-δάσκαλος βγαίνει από το νηπιαγωγείο. Τότε, εφόσον δεν υπάρχει κατάλληλος συγχρονισμός των νημάτων στην πρόσβαση της κοινής μεταβλητής θα δημιουργηθούν ασυνέπειες (βλ. ***Άσκηση 1.1***). Έστω ακόμα ένα παράδειγμα όπου δεν έχουμε εξασφαλίσει την ατομική εκτέλεση της συνάρτησης **verify()**, οπότε και πιθανώς μπορεί να δημιουργηθούν ασυνέπειες στο "διάβασμα" των "σωστών" τιμών των μεταβλητών και επομένως στην αποτίμηση της συνθήκης σωστής επίβλεψης των παιδιών.

Προαιρετικές ερωτήσεις

3.3.

Όπως αναφέραμε και παραπάνω, εδώ θα υλοποιήσουμε σχήμα συγχρονισμού που επιτρέπει πιο έντονα να μπαίνουν στο νηπιαγωγείο παιδιά που φτάνουν με μεγαλύτερη πιθανότητα από δασκάλους που περιμένουν για να φύγουν. Αυτό το κάνουμε χρησιμοποιώντας τώρα δύο

μεταβλητές συνθήκης **condc** και **condt**. Τα νήματα-παιδιά που περιμένουν για να μπουν "μπλοκάρουν" στην μεταβλητή συνθήκης condc, ενώ τα νήματα-δάσκαλοι που περιμένουν για να φύγουν "μπλοκάρουν" στην μεταβλητή συνθήκης condt. Όταν παιδιά φεύγουν από το νηπιαγωγείο "ειδοποιούν" **πρώτα** τα παιδιά που περιμένουν, κάνοντας **pthread_cond_broadcast()** με το condition variable ***condc*** και έπειτα τους δασκάλους που περιμένουν, κάνοντας **pthread_cond_broadcast()** με το condition variable ***condt***. Αυτό δίνει κάποια προτεραιότητα στο να "ξυπνήσει" το ΛΣ τα νήματα-παιδιά πρώτα έναντι των νημάτων-δασκάλων. Επιπλέον, όταν δάσκαλοι μπαίνουν στο νηπιαγωγείο "ειδοποιούν" **μόνο** τα παιδιά που περιμένουν, κάνοντας **pthread_cond_broadcast()** με το condition variable ***condc***.

Οι αλλαγές στον **πηγαίο κώδικα (source code)** φαίνονται παρακάτω:

```
struct kgarten_struct {
        pthread_cond_t condc; //now we use 2 condition variables because we want
        pthread_cond_t condt; //to prioritize children entering over teachers
                              //exiting

        int vt; //number of teachers in the kindergarten
        int vc; //number of children in the kindergarten
        int ratio; //given ratio that must be respected or bad things happen

        pthread_mutex_t mutex; //for critical sections and
};                              //association with the cond vars

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

        fprintf(stderr, "THREAD %d: CHILD WAITS TO ENTER\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        while(!((thr->kg->vc)<(thr->kg->vt)*(thr->kg->ratio))) { //i can enter
                                                                 //if c<t*r
                ret=pthread_cond_wait(&thr->kg->condc, &thr->kg->mutex);
                if(ret) {
                        perror_pthread(ret, "pthread_cond_wait");
                        exit(1);
                }
        }
        ++(thr->kg->vc);
        pthread_mutex_unlock(&thr->kg->mutex);
}

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

        fprintf(stderr, "THREAD %d: CHILD WAITS TO EXIT\n", thr->thrid);
```

```
        pthread_mutex_lock(&thr->kg->mutex);
        --(thr->kg->vc);
        ret=pthread_cond_broadcast(&thr->kg->condc); //prioritize children
                                                     //entering over teachers
        if(ret) {                                    //exiting!
                perror_pthread(ret, "pthread_cond_broadcast");
                exit(1);
        }
        ret=pthread_cond_broadcast(&thr->kg->condt);
        if(ret) {
                perror_pthread(ret, "pthread_cond_broadcast");
                exit(1);
        }
        pthread_mutex_unlock(&thr->kg->mutex);
}

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

        fprintf(stderr, "THREAD %d: TEACHER WAITS TO ENTER\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        ++(thr->kg->vt);
        ret=pthread_cond_broadcast(&thr->kg->condc); //notify only children to
                                                     //enter!
        if(ret) {
                perror_pthread(ret, "pthread_cond_broadcast");
                exit(1);
        }
        pthread_mutex_unlock(&thr->kg->mutex);
}

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

        fprintf(stderr, "THREAD %d: TEACHER WAITS TO EXIT\n", thr->thrid);

        pthread_mutex_lock(&thr->kg->mutex);
        while(!((thr->kg->vc)<=(thr->kg->vt-1)*(thr->kg->ratio))) {
                                //i can exit if c<=(t-1)*r
                ret=pthread_cond_wait(&thr->kg->condt, &thr->kg->mutex);
                if(ret) {
                        perror_pthread(ret, "pthread_cond_wait");
                        exit(1);
                }
        }
        --(thr->kg->vt);
        pthread_mutex_unlock(&thr->kg->mutex);
}
```

Τρέχουμε το παραπάνω πρόγραμμα **cp-kgarten**, αφού πρώτα το μεταγλωττίσουμε και κάνουμε το linking, ενδεικτικά για **n=20, c=10, r=2**:

```
.
.
.
Thread 15 [Teacher]: Entered.
            Thread 0: Teachers: 10, Children: 10
            Thread 10: Teachers: 10, Children: 10
            Thread 14: Teachers: 10, Children: 10
            Thread 17: Teachers: 10, Children: 10
            Thread 15: Teachers: 10, Children: 10
Thread 11 [Teacher]: Exiting.
THREAD 11: TEACHER WAITS TO EXIT
Thread 11 [Teacher]: Exited.
Thread 6 [Child]: Exiting.
THREAD 6: CHILD WAITS TO EXIT
Thread 6 [Child]: Exited.
            Thread 11: Teachers: 9, Children: 9
Thread 11 [Teacher]: Entering.
THREAD 11: TEACHER WAITS TO ENTER
Thread 11 [Teacher]: Entered.
            Thread 11: Teachers: 10, Children: 9
            Thread 6: Teachers: 10, Children: 9
Thread 6 [Child]: Entering.
THREAD 6: CHILD WAITS TO ENTER
Thread 6 [Child]: Entered.
            Thread 6: Teachers: 10, Children: 10
Thread 11 [Teacher]: Exiting.
THREAD 11: TEACHER WAITS TO EXIT
Thread 11 [Teacher]: Exited.
            Thread 11: Teachers: 9, Children: 10
Thread 11 [Teacher]: Entering.
THREAD 11: TEACHER WAITS TO ENTER
Thread 11 [Teacher]: Entered.
            Thread 11: Teachers: 10, Children: 10
.
.
.
```

Παρατηρούμε και εδώ σωστή λειτουργία του προγράμματος όπως και πριν (δεν τερματίζει), ενώ είναι εμφανής και η έντονη προτεραιότητα που δίνεται στα παιδιά που περιμένουν να μπουν στο νηπιαγωγείο έναντι των δασκάλων που περιμένουν για να βγουν από αυτό.

3.2.

Εκτελώντας το πρόγραμμα που προκύπτει από το αρχείο **rand-fork-v1.c** έχουμε:

```
oslaba100@os-node1:~/ex3/extra$ ls
rand-fork  rand-fork-v1.c  rand-fork-v2.c  rand-fork-v3.c
oslaba100@os-node1:~/ex3/extra$ ./rand-fork
334555972
334555972
334555972
334555972
334555972
334555972
334555972
334555972
334555972
334555972
```

Κοιτώντας τον πηγαίο κώδικα, αυτό που προσπαθεί να κάνει είναι να τυπώσει "τυχαίους" αριθμούς δημιουργώντας processes μέσω fork(). Με την **rand()** παράγεται μία ψευδοτυχαία ακολουθία αριθμών δεδομένου ενός σπόρου-seed (δίνεται με την **srand()**). Για δεδομένο σπόρο η ακολουθία τυχαίων αριθμών είναι γνωστή και παράγεται ντετερμινιστικά (**ψευδοτυχαία**). Επομένως, το παραπάνω πρόγραμμα αποτυγχάνει, αφού όλες οι διεργασίες παιδιά έχουν τον ίδιο σπόρο-ίδιο state του RNG και επομένως καλώντας την rand() θα παράγουν τον ίδιο ψευδοτυχαίο αριθμό.

Αυτό που θέλουμε να κάνουμε, λοιπόν, είναι να τροποποιήσουμε το παραπάνω πρόγραμμα έτσι ώστε οι διεργασίες παιδιά να παράγουν "τυχαίους" αριθμούς διαφορετικούς μεταξύ τους.

Μία τροποποίηση του παραπάνω προγράμματος φαίνεται παρακάτω (**rand-fork-v2.c**):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, const char *argv[])
{
        int count, i;
        pid_t pid;

        count = (argc > 1) ? atol(argv[1]) : 10;

        for (i=0; i<count; i++){

                /* do fork */
                if ((pid = fork()) < 0){
                        perror("fork");
                        exit(1);
                }

                /* father continues */
                if (pid != 0)
                        continue;

                /* child: run process */
                srand(time(NULL));      //seed inside the children processes
                printf("%d\n", rand()); //is the same because time() changes
                exit(0);                //once per second (if we fork >10000
                                        //processes we will see 'groups' of
                                        //pseudo-random numbers)
        }

        /* wait for children */
        for (i=0; i<count; i++){
                wait(NULL);
        }

        return 0;
}
```

Εκτελώντας το παραπάνω πρόγραμμα με όρισμα ενδεικτικά 20 έχουμε:

```
oslaba100@os-node1:~/ex3/extra$ ls
rand-fork  rand-fork-v1.c  rand-fork-v2.c  rand-fork-v3.c
oslaba100@os-node1:~/ex3/extra$ ./rand-fork 20
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
1117118677
```

Αυτό που κάναμε ήταν να κάνουμε seed μέσα στα child processes. Ο τρόπος που κάναμε seed είναι ο εξής: **srand(time(NULL))**. Παρατηρούμε ότι πάλι δεν παράγονται "τυχαίοι" αριθμοί διαφορετικοί μεταξύ τους. Αυτό συμβαίνει διότι η **time()** αλλάζει τιμή κάθε sec ενώ οι διεργασίες δημιουργούνται σε λιγότερο χρόνο κάτι που επομένως οδηγεί στο να έχουν τον ίδιο σπόρο και επομένως να παράγουν τον ίδιο ψευδοτυχαίο αριθμό. Αν αυξήσουμε κατά πολύ τον αριθμό των διεργασιών που δημιουργούμε (>10000) τότε θα δούμε ότι "ανά ομάδες" τα processes παράγουν ψευδοτυχαίους αριθμούς λόγω του ότι ο χρόνος δημιουργίας τους ξεπερνάει το 1 sec.

Παρακάτω, λοιπόν, τροποποιούμε ξανά το πρόγραμμά μας ως εξής (**rand-fork-v3.c**):

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, const char *argv[])
{
        int count, i;
        pid_t pid;

        count = (argc > 1) ? atol(argv[1]) : 10;

        for (i=0; i<count; i++){

                /* do fork */
                if ((pid = fork()) < 0){
                        perror("fork");
                        exit(1);
                }

                /* father continues */
                if (pid != 0)
                        continue;

                /* child: run process */
                srand(time(NULL)+getpid()); //the solution to have "random"
                                            //numbers generated by each child
```

```
                printf("%d\n", rand());      //process is to seed by adding
                exit(0);                     //getpid(). This ensures different
        }                                    //pseudo-random sequence generated
        /* wait for children */              //by each child process because
        for (i=0; i<count; i++){             //they have different PIDs (PIDs
                                             //aren't typically re-used that
                wait(NULL);                  //quickly).
        }

        return 0;
}
```

Εκτελώντας το παραπάνω πρόγραμμα ενδεικτικά με όρισμα 10 έχουμε:

```
oslaba100@os-node1:~/ex3/extra$ ls
rand-fork  rand-fork-v1.c  rand-fork-v2.c  rand-fork-v3.c
oslaba100@os-node1:~/ex3/extra$ ./rand-fork 10
1974944369
1680835964
296793914
2136290945
756949796
467477429
1226810196
1993359601
616088981
313269789
```

Παρατηρούμε λοιπόν ότι το πρόγραμμά μας λειτουργεί σωστά τώρα, όπου και παράγονται "τυχαίοι" αριθμοί όπως θέλαμε. Αυτό συμβαίνει καθώς έχουμε τροποποιήσει τον τρόπο που κάνουμε seed ως εξής: **srand(time(NULL)+getpid())**, όπου κάθε διεργασία παιδί τώρα έχει διαφορετικό σπόρο καθώς μεταξύ τους οι διεργασίες έχουν διαφορετικό PID (θεωρούμε ότι τα PIDs δεν ξαναχρησιμοποιούνται τόσο γρήγορα).