**Λειτουργικά Συστήματα**

**Άσκηση 3: Συγχρονισμός**

**1.1 Συγχρονισμός σε υπάρχοντα κώδικα**

/\*

\* simplesync.c

\*

\* A simple synchronization exercise.

\*

\* Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>

\* Operating Systems course, ECE, NTUA

\*

\*/

#include <errno.h>

#include <stdio.h>

#include <stdlib.h>

#include <unistd.h>

#include <pthread.h>

/\*

\* POSIX thread functions do not return error numbers in errno,

\* but in the actual return value of the function call instead.

\* This macro helps with error reporting in this case.

\*/

#define perror\_pthread(ret, msg) \

do { errno = ret; perror(msg); } while (0)

#define N 10000000

/\* Dots indicate lines where you are free to insert code at will \*/

pthread\_mutex\_t new\_mutex;

#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

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) {

\_\_sync\_fetch\_and\_add(ip, 1);

} else {

pthread\_mutex\_lock(&new\_mutex);

/\* You cannot modify the following line \*/

++(\*ip);

pthread\_mutex\_unlock(&new\_mutex);

}

}

fprintf(stderr, "Done increasing variable.\n");

return NULL;

}

void \*decrease\_fn(void \*arg)

{

int i;

volatile int \*ip = arg;

fprintf(stderr, "About to decrease variable %d times\n", N);

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

if (USE\_ATOMIC\_OPS) {

\_\_sync\_sub\_and\_fetch(ip, 1);

} else {

pthread\_mutex\_lock(&new\_mutex);

/\* You cannot modify the following line \*/

--(\*ip);

pthread\_mutex\_unlock(&new\_mutex);

}

}

fprintf(stderr, "Done decreasing variable.\n");

return NULL;

}

int main(int argc, char \*argv[])

{

int val, ret, ok;

pthread\_t t1, t2;

/\*

\* Initial value

\*/

val = 0;

/\*

\* Create threads

\*/

ret = pthread\_create(&t1, NULL, increase\_fn, &val);

if (ret) {

perror\_pthread(ret, "pthread\_create");

exit(1);

}

ret = pthread\_create(&t2, NULL, decrease\_fn, &val);

if (ret) {

perror\_pthread(ret, "pthread\_create");

exit(1);

}

/\*

\* Wait for threads to terminate

\*/

ret = pthread\_join(t1, NULL);

if (ret)

perror\_pthread(ret, "pthread\_join");

ret = pthread\_join(t2, NULL);

if (ret)

perror\_pthread(ret, "pthread\_join");

/\*

\* Is everything OK?

\*/

ok = (val == 0);

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

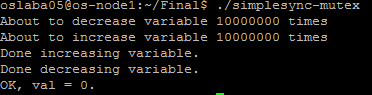
return ok;

}

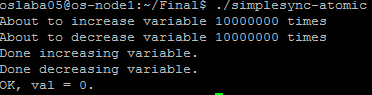
Χρησιμοποιήσαμε το παρεχόμενο Makefile για να μεταγλωττίσουμε και να τρέξουμε το πρόγραμμα. Παρατηρήσαμε ότι για κάθε αρχείο .c χρησιμοποιείται η παράμετρος -pthread άρα ο κώδικας χρησιμοποιεί νήματα (threads). Τέλος παρατηρήσαμε πως το αρχείο Makefile εκτελεί δύο μεταγλωττίσεις για το αρχείο simplesync.c και παράγει δύο εκτελέσιμα . Το ένα παράγεται με τη παράμετρο -DSYNC\_MUTEX και ένα με τη παράμετρο -DSYNC\_ATOMIC. Και τα δύο χρησιμοποιούν τον κώδικα simplesync.c για να πράξουν ένα object file το καθένα. Από τα object files φτιάχνονται τα εκτελέσιμα simplesync-atomic και simplesync-mutex.

Παρακάτω φαίνεται η έξοδος του εκτελέσιμου:

* simplesync-mutex



* simplesync-atomic



**Απαντήσεις**:

1. Παρατηρούμε ότι στη περίπτωση όπου έχουμε συγχρονισμό ο χρόνος που απαιτείται για να εκτελεστεί το πρόγραμμα είναι μεγαλύτερος από τη περίπτωση που δεν έχουμε συγχρονισμό. Αυτό οφείλεται στο γεγονός ότι χωρίς την ύπαρξη του συγχρονισμού πολλές πράξεις δεν υπολογίζονται και έτσι απαιτείται λιγότερος χρόνος για την εκτέλεση του προγράμματος. Σε αυτή τη περίπτωση όμως υπάρχει μεγάλη πιθανότητα το τελικό αποτέλεσμα να είναι λάθος.

2. Η μέθοδος ατομικών λειτουργιών είναι πιο γρήγορη από τη μέθοδο POSIX mutexes. Αυτό οφείλεται στο γεγονός ότι με POSIX mutexes οι διεργασίες αναστέλλουν την λειτουργίας τους όσο το κύριο τμήμα είναι κατειλημμένο. Οι ατομικές λειτουργίες όμως προσπαθούν να εκτελεστούν συνέχεια μέχρι να τα καταφέρουν .

3.

**1.2 Παράλληλος υπολογισμός του συνόλου Mandelbrot**

/\*

\* mandel.c

\*

\* A program to draw the Mandelbrot Set on a 256-color xterm.

\*

\*/

#include <errno.h>

#include <stdio.h>

#include <unistd.h>

#include <assert.h>

#include <string.h>

#include <math.h>

#include <stdlib.h>

#include <semaphore.h>

#include <pthread.h>

#include "mandel-lib.h"

#define MANDEL\_MAX\_ITERATION 100000

#define perror\_pthread(ret, msg) \

do { errno = ret; perror(msg); } while (0)

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compile-time parameters \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* Output at the terminal is is x\_chars wide by y\_chars long

\*/

int y\_chars = 50;

int x\_chars = 90;

sem\_t \*sem;

int nthreads;

/\*

\* 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.

\*/

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 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 \*thread\_start\_fn(void \*arg)

{

int line,\*x = arg;

for (line = \*x; line < y\_chars; line +=nthreads) {

int color\_val[x\_chars];

compute\_mandel\_line(line, color\_val);

/\*

\* 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

\*/

sem\_wait(&sem[\*x]);

output\_mandel\_line(1, color\_val);

/\* increments (unlocks) the semaphore pointed to by sem. \*/

if (\*x == nthreads-1) sem\_post(&sem[0]);

else sem\_post(&sem[\*x + 1]);

}

return NULL;

}

int main(int argc, char \*argv[])

{

int ret,i;

/\*

\* Parse the command line

\*/

if (argc != 2){

fprintf(stderr, "Usage: %s NTHREADS \n\n"

"Exactly one argument required:\n"

" NTHREADS: The number of threads.\n\n",

argv[0]);

exit(1);

}

if (safe\_atoi(argv[1], &nthreads) < 0 || nthreads <= 0) {

fprintf(stderr, "`%s' is not valid for `nthreads'\n", argv[1]);

exit(1);

}

/\*

\* Create threads

\*/

pthread\_t \*tid;

tid = malloc(nthreads\*sizeof(pthread\_t));

int \*id;

id = malloc(nthreads\*sizeof(int));

xstep = (xmax - xmin) / x\_chars;

ystep = (ymax - ymin) / y\_chars;

/\*

\* Create semaphores

\*/

sem = malloc(nthreads\*sizeof(sem\_t));

/\*

\* initializes the unnamed semaphore at the address pointed to by sem.

\* int sem\_init(sem\_t \*sem, int pshared, unsigned int value);

\* pshared = 0, then the semaphore is shared between the threads of a process

\* The value argument specifies the initial value for the semaphore.

\*/

sem\_init(&sem[0],0,1); // semaphore 0 : allows 1 thread inside

for(i = 1; i<nthreads; i++ )

sem\_init(&sem[i],0,0); // other semaphores allow 0 threads inside

/\* Spawn new thread \*/

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

id[i] = i ;

ret = pthread\_create(&tid[i], NULL,thread\_start\_fn,&id[i]);

if (ret) {

perror\_pthread(ret, "pthread\_create");

exit(1);

}

}

/\*

\* Wait for all threads to terminate

\*/

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

ret = pthread\_join(tid[i], NULL);

if (ret) {

perror\_pthread(ret, "pthread\_join");

exit(1);

}

}

/\*

\* Destroy all semaphores

\*/

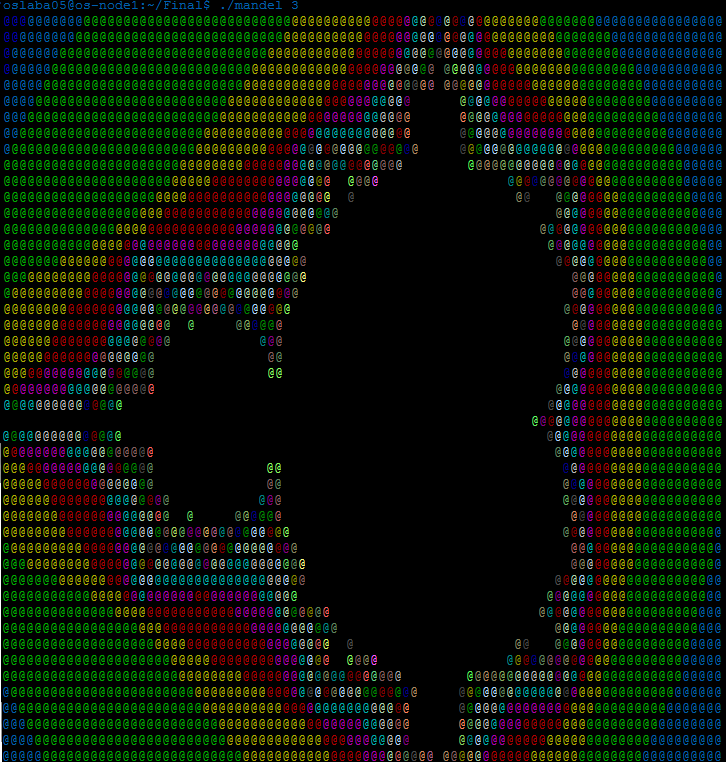
for(i = 0; i<nthreads; i++ )

sem\_destroy(&sem[i]);

return 0;

}

Παρακάτω φαίνεται η έξοδος του εκτελέσιμου



**Απαντήσεις**:

1. Για το σχήμα συγχρονισμού που υλοποιήσαμε χρειάστηκαν τόσοι σημαφόροι όσα είναι και τα νήματα, ώστε να μπορούμε να ελέγχουμε πότε θα έχει πρόσβαση το κάθε νήμα στο κρίσιμο τμήμα. Αυτό έχει νόημα καθώς τα νήματα μπορούν να εκτελούνται παράλληλα. Έτσι όταν ένα νήμα θέλει να χρησιμοποιήσει κάποιο κοινό πόρο μπαίνει στη διαδικασία να κλειδώσει μειώνοντας το σημαφόρο του και παίρνοντας έτσι τον έλεγχο. Με το τρόπο αυτό αποτρέπει σε άλλα νήματα να κάνουν την ίδια κίνηση εώς ότου αυτό να δώσει τον έλεγχο στον σημαφόρο του επόμενου στη σειρά νήματος.

2. Για την σειριακή εκτέλεση:



Για την παράλληλη εκτέλεση:



3. Από τα πιο πάνω αποτελέσματα συμπεραίνουμε πως η παράλληλη εκτέλεση απαιτεί λιγότερο χρόνο από τη σειριακή. Αυτό εξηγείται εύκολα αφού το κάθε νήμα δεν περιμένει να τελειώσει το προηγούμενό του για να ξεκινήσει.Το κρίσιμο τμήμα είναι μικρό επειδή αποτελείται μόνο από τη φάση εξόδου της κάθε γραμμής, αφού ο υπολογισμός της κάθε γραμμής δεν εξαρτάται από τον υπολογισμό της προηγούμενης γραμμής. Έτσι και γι' αυτό το λόγο υπάρχει μια επιτάχυνση στο χρόνο εκτέλεσης .

4. Όταν πατάμε Ctrl-C ενώ το πρόγραμμα εκτελείται, αυτό τερματίζει. Επομένως η αλλαγή των χρωμάτων των χαρακτήρων δεν επαναφέρεται και παραμένει το χρώμα που τέθηκε τελευταίο με αποτέλεσμα να παραμένει χρωματισμένο το terminal. Θα μπορούσαμε να αντιμετωπίσουμε το πρόβλημα τη συνάρτηση χειρισμού σήματος sighandler() στην οποία θα υπάρχουν οι εντολές reset\_xterm\_color(1) και η exit(1). Έτσι με το πάτημα του Ctrl-C, θα ενεργοποιήται η sighandler() και θα επαναφέρει το αρχικό χρώμα πριν τερματίσει το πρόγραμμα.

**1.3 Επίλυση προβλήματος συγχρονισμού**

/\*

\* kgarten.c

\*

\* A kindergarten simulator.

\* Bad things happen if teachers and children

\* are not synchronized properly.

\*

\*

\* Author:

\* Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>

\*

\* Additional Authors:

\* Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>

\* Anastassios Nanos <ananos@cslab.ece.ntua.gr>

\* Operating Systems course, ECE, NTUA

\*

\*/

#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\_pthread(ret, msg) \

do { errno = ret; perror(msg); } while (0)

/\* A virtual kindergarten \*/

struct kgarten\_struct {

/\*

\* Here you may define any mutexes / condition variables / other variables

\* you may need.

\*/

pthread\_cond\_t cond;

/\*

\* 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 %zd 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);

while((((thr->kg->vc)/(thr->kg->ratio))>=(thr->kg->vt))||((thr->kg->vt)==0))

//#childs / ratio >= #teachers OR #teachers == 0

pthread\_cond\_wait(&thr->kg->cond,&thr->kg->mutex);

//atomically unlocks the mutex and waits for the condition variable cond to be signalled.

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

++(thr->kg->vc);

pthread\_mutex\_unlock(&thr->kg->mutex);

}

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);

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

//restarts all the threads that are waiting on the condition variable cond.

--(thr->kg->vc);

pthread\_cond\_broadcast(&thr->kg->cond);

//--(thr->kg->vc);

pthread\_mutex\_unlock(&thr->kg->mutex);

}

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);

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

//restarts all the threads that are waiting on the condition variable cond.

++(thr->kg->vt);

pthread\_cond\_broadcast(&thr->kg->cond);

//++(thr->kg->vt);

pthread\_mutex\_unlock(&thr->kg->mutex);

}

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);

while((((thr->kg->vt)-1)\*(thr->kg->ratio))<=(thr->kg->vc))

//(#teachers-1) \* ratio <= #childs

pthread\_cond\_wait(&thr->kg->cond,&thr->kg->mutex);

// atomically unlocks the mutex and waits for the condition variable cond to be signalled.

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

--(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) {

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 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));

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);

}

ret = pthread\_cond\_init(&kg->cond, NULL);

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);

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);

}

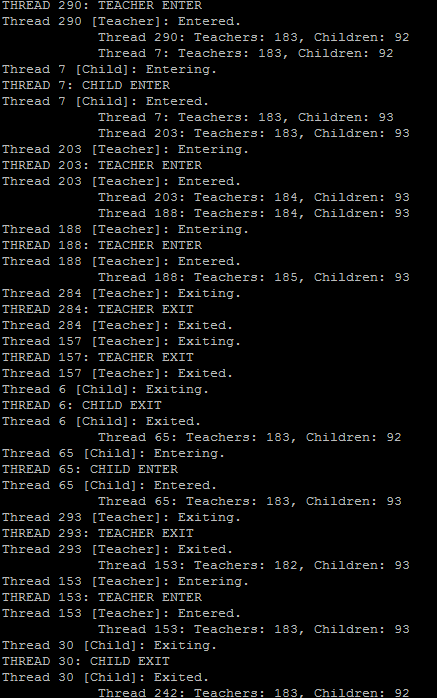
}

printf("OK.\n");

return 0;

}

Παρακάτω φαίνεται η έξοδος του εκτελέσιμου (συνεχίζεται επ' αόριστον)



**Απαντήσεις**:

1. Όταν ένας δάσκαλος επιχειρεί να φύγει ελέγχει αν το επιτρέπει η αναλογία παιδιών που περιμένουν να μπουν και υπολειπόμενων δασκάλων που είναι μέσα. Το ίδιο συμβαίνει και όταν ένα παιδί επιχειρεί να μπει . Τόσο τα παιδιά που φεύγουν όσο και οι δάσκαλοι που μπαίνουν στέλνουν σήμα στους δασκάλους και στα παιδιά που περιμένουν να βγουν και να μπουν αντίστοιχα, να συνεχίσουν. Ο πρώτος που θα πάρει το κλειδί ελέγχει αν το επιτρέπει η συνθήκη του να συνεχίσει αλλιώς περιμένει (wait) και στέλνει το κλειδί σε άλλους που περιμένουν.

2. Κάθε φορά που περισσότερα από ένα νήματα ζητούν το κλειδί για να έχουν έλεγχο στο κρίσιμο τμήμα , παρατηρούμε καταστάσεις συναγωνισμού στον κώδικα. Επίσης, καταστάσεις συναγωνισμού παρατηρούνται κάθε φορά που περισσότερα από ένα νήματα ζητούν το κλειδί στις συναρτήσεις εισόδου και εξόδου των παιδιών ή των δασκάλων.