Operating Systems

3rd LAB REPORT

Γεωργία Μπουσμπουκέα- el19059
Παναγιώτης Μιχελάκης- el19119

Oslaba59

1.

The purpose of this exercise is to understand the usefulness of POSIX mutexes and atomic operations via an example of calculations on a variable with the use of two threads.

Executing intiially the two files simplesync - atomic, simplesync - mutex we notice that the result is not the desired, that is 0. In fact, every time we execute them a different result occurs. This is due to the lack of synchronization between the two threads, which run in parallel. Specifically, as they manage the same data, the variable val, and there are race conditions, before the execution of a command from one thread is completed, it is interrupted by the other and the result is not correct.

The two executable files are due to compilation, as in Makefile we have different compilation commands with flags –D SYNC_MUTEX and -D SYNC_ATOMIC, which set the value of USE_ATOMIC_OPS in simplesync.c. This value determines whether mutexes or individual commands will be used. In the initial form of code this value does not play any role.

We fix this next:

In the case of POSIX mutexes (in the else of each function), that is for the executable simplesync - mutex, we define a mutex for the critical section of the two parts, that is the change of the value of the variable. So if the first thread adds, the second is blocked by the lock and waits. It will perform its function as soon as the first thread unlocks and then it will lock in its turn and so on.

In case of atomic variables, we use the built-in functions __sync_add_and_fetch, __sync_sub_and_fetch. They allow the contents of a variable to be changed atomically - that is, as a continuous unit. Specifically, the addition e.g. translates in assebly into more than one command and there is a risk that another thread will execute one command of its own, before they are completed in their entirety, which would alter the contents of the variable. By using atomic commands we ensure that this does not happen.

1)

We notice that the execution time is shorter in the original case, as then there was a greater degree of parallelism. With the extension of the code, in each case only one thread at a time was executed in the critical areas of the program.

2)

time ./simplesync-mutex

```
real    0m5.308s
user    0m6.372s
sys     0m3.648s
```

time ./simplesync-atomic

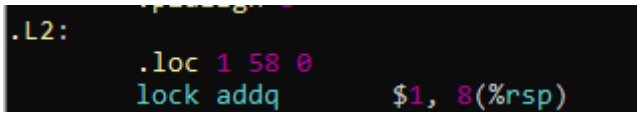
```
real    0m1.843s
user    0m3.620s
sys     0m0.012s
```

We observe that the execution time using atomic commands is shorter than the one using mutexes. This is because atomic commands are hardware commands, while mutexes are software tools and cause extra delay by calling locks-unlocks. Of course, hardware commands have other disadvantages, such as their complexity and limited number.

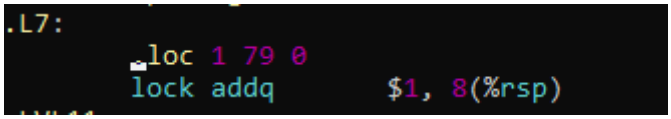
3)

By using the –g, –S flags in the gcc compilation command we have as output a .s file, which includes the assembly code of the .c program. The .loc identifier specifies the line of the .c file, to which the assembly command corresponds. Therefore we are able to find the assembly commands corresponding to the atomic commands. Specifically:

__sync_add_and_fetch -> line 58:

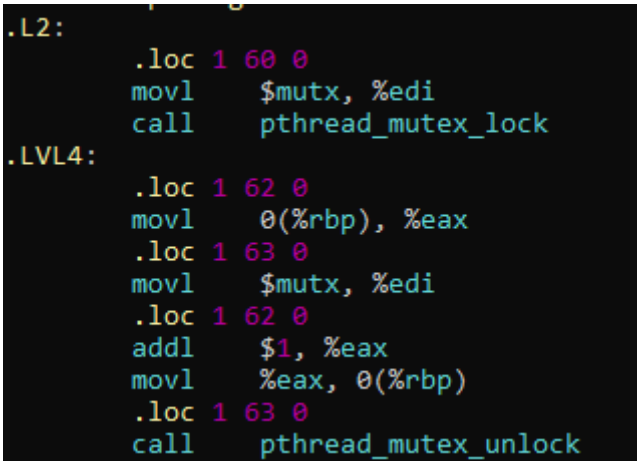


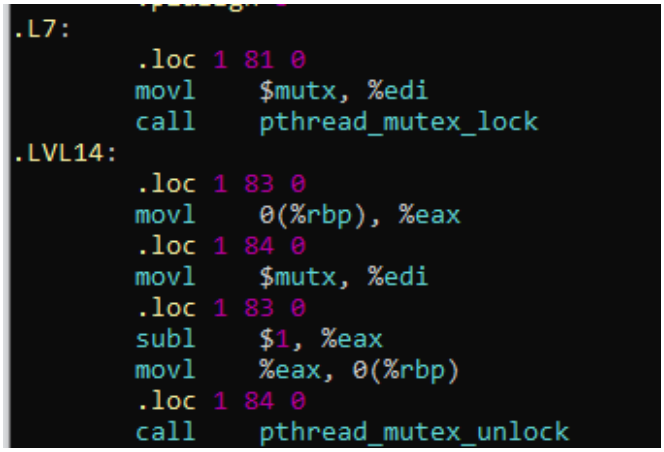
__sync_sub_and_fetch -> line 79:



4)

Similarly:




.L7:
.loc 1 81 0
movl $mutx, %edi
call pthread_mutex_lock
.LVL14:
.loc 1 83 0
movl 0(%rbp), %eax
.loc 1 84 0
movl $mutx, %edi
.loc 1 83 0
subl $1, %eax
movl %eax, 0(%rbp)
.loc 1 84 0
call pthread_mutex_unlock

Code:

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




/*
 * POSIX thread functions do not return error numbers in errno,
 * but in the actual return value of the function call instead.
 * This macro helps with error reporting in this case.
 */




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

#define N 10000000

/* Dots indicate lines where you are free to insert code at will */
/* ... */
#if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
# error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
#endif

#if defined(SYNC_ATOMIC)
 #define USE_ATOMIC_OPS 1
#else
 #define USE_ATOMIC_OPS 0
#endif


pthread_mutex_t mutx;

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_add_and_fetch(&ip, 1);
                } else {
                        pthread_mutex_lock(&mutx);
                        /* You cannot modify the following line */
                        ++(*ip);
                        pthread_mutex_unlock(&mutx);
                }
        }
        fprintf(stderr, "Done increasing variable.\n");

        return NULL;
}
```

```c
void *decrease_fn(void *arg)
{
        int i;
        volatile int *ip = arg;
        fprintf(stderr, "About to decrease variable %d times\n", N);
        for (i = 0; i < N; i++) {
                if (USE_ATOMIC_OPS) {
                        __sync_sub_and_fetch(&ip, -1);
                } else {
                        pthread_mutex_lock(&mutx);
                        /* You cannot modify the following line */
                        --(*ip);
                        pthread_mutex_unlock(&mutx);
                }
        }
        fprintf(stderr, "Done decreasing variable.\n");
        return NULL;
}


int main(int argc, char *argv[])
{
        int val, ret, ok;
        pthread_t t1, t2;
        pthread_mutex_init(&mutx, NULL);
        /*
         * 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");

        pthread_mutex_destroy(&mutx);
        /*
         * Is everything OK?
         */
        ok = (val == 0);
        printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
        return ok;
}
```

2.

The purpose of this exercise is to compute the Mandelbrot set using Escape Time Algorithm and then print its points in the terminal. The calculation is done via threads, the number of which, nthreads, is given as an input to the program.

The Mandelbrot set is computed and printed line by line an the i-th thread undertakes the computation of lines i, i+nthreads, i+2nthreads,... . The need for sychronisation is clear, as we want the lines to be printed in order (we don't really care about the order of computation). That is achieved using semaphores.

1)

We create nthreads semaphores, initiallising only the first to 1. The semaphore waits before the critical area (which is the printout) and in this way latter executions of this area by the same thread are prevented, until its turn comes. After the critical area, the next (in circular order) semaphore posts, allowing the next thread to enter the area and so on, until the necessary number of lines is printed.

2)

In the serial program, where only one thread exists:

`time ./mandel 1`

```
real    0m1.548s
user    0m1.448s
sys     0m0.052s
```

In the program where two threads share the load of computation, achieving parallelism:

`time ./mandel 2`

```
real    0m0.912s
user    0m1.452s
sys     0m0.060s
```

The real time in the 2-thread case is significantly less than the first case, while the time percieved by the user is almost the same.

3)

As mentioned, the critical area is only the printing of Mandelbrot lines and not the computation, as each line is independent from the others. In that way, the computation can be done in parallel, inducing acceleration. If the computation was also in the critical section, less functions could be done in parallel and the speed would be lower.

4)

Pressing Ctrl+C causes the cancelation of the executed program. That means that the reset_xterm_color(1) command is not executed and the terminal is left with the color of the last printed symbol. We could handle these behavior using a signal handler, which would execute this command whenever a SIGINT signal had been sent (Cntrl+C= SIGINT).

Code:

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

#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include <semaphore.h>
#include "mandel-lib.h"
#include <pthread.h>
#include <errno.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;

/*
 * 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;

struct thread_info_struct {
        pthread_t tid; /* POSIX thread id, as returned by the library */

        int *color_val; /* Pointer to array to manipulate */
        int thrid; /* Application-defined thread id */
        int thrcnt;
};

sem_t *semaphore;
```

```c
/*
 * This function computes a line of output
 * as an array of x_char color values.
 */
void compute_mandel_line(int line, int color_val[])
{
        /*
         * x and y traverse the complex plane.
         */
        double x, y;

        int n;
        int val;

        /* Find out the y value corresponding to this line */
        y = ymax - ystep * line;

        /* and iterate for all points on this line */
        for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {

                /* Compute the point's color value */
                val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
                if (val > 255)
                        val = 255;

                /* And store it in the color_val[] array */
                val = xterm_color(val);
                color_val[n] = val;
        }
}

/*
 * This function outputs an array of x_char color values
 * to a 256-color xterm.
 */
void output_mandel_line(int fd, int color_val[])
{
        int i;

        char point ='@';
        char newline='\n';

        for (i = 0; i < x_chars; i++) {
                /* Set the current color, then output the point */
                set_xterm_color(fd, color_val[i]);
                if (write(fd, &point, 1) != 1) {
                        perror("compute_and_output_mandel_line: write point");
                        exit(1);
                }
        }

        /* Now that the line is done, output a newline character */
        if (write(fd, &newline, 1) != 1) {
                perror("compute_and_output_mandel_line: write newline");
                exit(1);
        }
}
```

```c
void compute_and_output_mandel_line(int fd, int line, void *arg)
{

        struct thread_info_struct *thr=arg;
        /*
         * A temporary array, used to hold color values for the line being drawn
         */
        //int color_val[x_chars];


        compute_mandel_line(line, thr->color_val);// each thread computes its lines in parallel
        sem_wait(&semaphore[line%thr->thrcnt]);// lock the critical area since we
                                               // want only one thread to be writing in the console
        output_mandel_line(fd, thr->color_val);// print the corresponding line
        sem_post(&semaphore[(line+1)%thr->thrcnt]);// unlock the critical section for the next thread

}



void *help(void *arg){
        struct thread_info_struct *thr=arg;
        int i;
        for(i=thr->thrid; i<y_chars; i+=thr->thrcnt){// The thread i computes the lines
                                                     // i, i+thrcnt, i + 2*thrcnt, i +3*thrcnt,.....
                compute_and_output_mandel_line(1, i, thr);
        }
        return NULL;
}


void *safe_malloc(size_t size)
{
        void *p;

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

        return p;
}



int safe_atoi(char *s, int *val)
{
        long l;
        char *endp;

        l = strtol(s, &endp, 10);
        if (s != endp && *endp == '\0') {
                *val = l;
                return 0;
        } else
                return -1;
}
```

```c
int main(int argc, char **argv)
{
        int nthreads;
        if(argc !=2) {
                printf("Exactly one argument required:\n"
                       "thread_count: The number of threads to create.\n");
                exit(1);
        }
        if (safe_atoi(argv[1], &nthreads) < 0 || nthreads <= 0) {
                fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
                exit(1);
        }

        int i, ret;
        struct thread_info_struct *thr;
        thr=safe_malloc(nthreads*sizeof(*thr));
        semaphore=safe_malloc(nthreads*sizeof(sem_t));

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


        for(i=0; i<nthreads; i++){
                /*initialise per-thread structure */
                thr[i].thrid=i;
                thr[i].thrcnt= nthreads;
                thr[i].color_val=safe_malloc(x_chars*sizeof(int));
                /*spawn new thread*/
                if(i==0) sem_init(&semaphore[i], 0, 1);
                else sem_init(&semaphore[i], 0, 0);
                ret=pthread_create(&thr[i].tid, NULL, help, &thr[i]);
                if(ret){
                        perror_pthread(ret, "pthread_create");
                        exit(1);
                }
        }


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


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



        reset_xterm_color(1);

        return 0;
}
```

3.

The purpose of this exercise is to stimulate a kindergarten with N people, C of which are children and the rest teachers and there is a specific ratio R=children/teachers. We make sure not to exceed this ratio using conditional variables. N, C, R as given as inputs to the program.

Each entity in the kindergarten (child or teacher) is represented by a thread and for each thread we create an object from the thread_info_struct, so as to keep track of its information. The state of the kindergarten is represented by a struct and every thread has access to its data (vt-> #of teachers, vc-># fo children and ratio). Each entity can enter or exit the kindergarten at any time (represented by sleep for random time), as long as the condition of the ratio is satisfied.

So we have four functions child_enter, child_exit, teacher_enter and teacher_exit which modify the number of teachers, vt, or children, vc, in the kindergarten accordingly. Of course, this is done with the use of mutexes, as these variables can be changed form every thread and we want only one thread to act on the variable at a time (critical sections). However, whenever a child wants to enter or a teacher wants to exit, we must check whether it can do so as to not violate the desirable ratio. Therefore we use a conditional variable, which causes the thread to wait until it's safe to change the corresponding value. This is achieved by broadcasting the conditional variable whenever a teacher enters or a child exits, so then the thread checks again if it's safe to proceed.

(The conditional variable unlocks the mutex until it receives a signal from a thread which is broadcasting and then locks again)

Whenever a teacher or a child enters we check if the condition of ratio is satisfied (it is always in our implementation), then sleep for random amound of time, then exit, then sleep again and finally recheck the condition. This process is repeated for every thread in an infinite loop.

1)

In our implementation if a teacher waits until it's safe to leave, that does not prevent new children from entering, if the ratio is not violated (e.g. vc=7, vt=4, ratio=2 teacher can not leave but a new child can enter), since the conditional variable has unlocked the mutex.

2)

In our implementation a problem may occur if two or more threads are waiting for a child to exit or a teacher to enter and when that happens, they may all get at the same time the broadcasted conditional variable and consider it's safe to proceed, while in reality only one of them should.

Code:

```
 *
 * 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 condChec;
        pthread_mutex_t mutex;
        /*
         * You may NOT modify anything in the structure below this
         * point.
         */
        int vt; // The number of teachers that are currently inside the kidnergarten
        int vc; // The number of children that are currently inside the kindergarten
        int ratio;

};

/*
 * 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;
};
```

```c
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);
}
```

```c
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);
        }

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

        pthread_mutex_lock(&thr->kg->mutex);
        while( (thr->kg->vc) + 1 > (thr->kg->vt) * (thr->kg->ratio)){
                pthread_cond_wait(&thr->kg->condCheck, &thr->kg->mutex);
        }
        ++(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);
        }

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

        pthread_mutex_lock(&thr->kg->mutex);
        --(thr->kg->vc);
        pthread_mutex_unlock(&thr->kg->mutex);
        pthread_cond_broadcast(&thr->kg->condCheck);//================================
}

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);
        }

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

        pthread_mutex_lock(&thr->kg->mutex);
        ++(thr->kg->vt);
        pthread_mutex_unlock(&thr->kg->mutex);
        pthread_cond_broadcast(&thr->kg->condCheck);//================================
}

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);
        }

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

        pthread_mutex_lock(&thr->kg->mutex);
        while( (thr->kg->vc) > (thr->kg->vt - 1) * (thr->kg->ratio) ){
                pthread_cond_wait(&thr->kg->condCheck,&thr->kg->mutex);
        }
        --(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);
        }
}
```

```c
/*
 * 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;
}
```

```c
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);
        pthread_cond_init(&kg->condCheck,NULL);//====================================
        if (ret) {
                perror_pthread(ret, "pthread_mutex_init");
                exit(1);
        }
```

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

        for (i = 0; i < thrcnt; i++) {
                /* Initialize per-thread structure */
                thr[i].kg = kg;// kg is a pointer to the virtual kidnergarten.
                                //Thus, all the generated threads will have the same kg pointer
                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;
}
-- INSERT --                                                                    31
```