ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

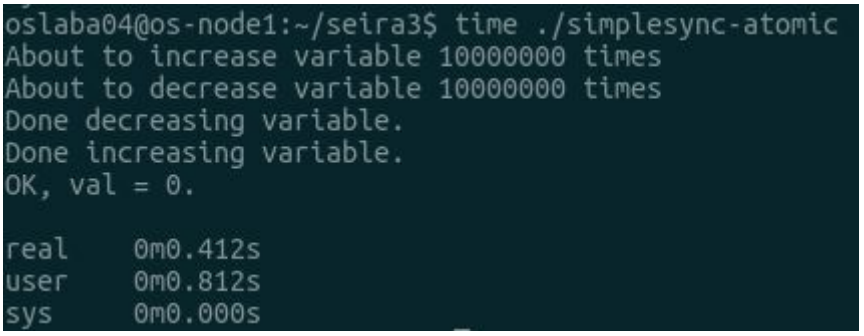
Εργαστήριο Λειτουργικών Συστημάτων

Αναφορά 3ης Εργαστηριακής Άσκησης

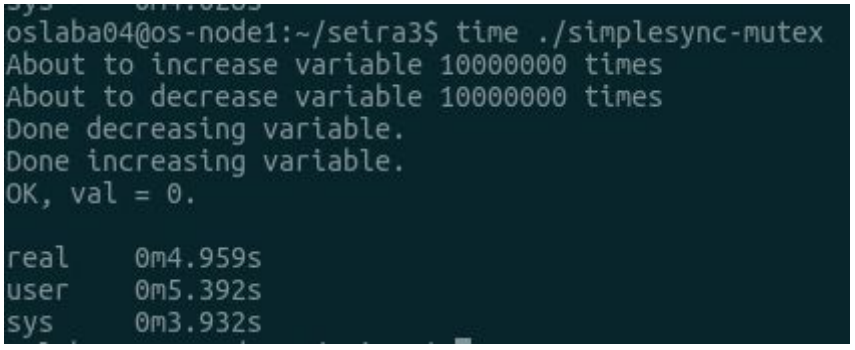
Ιάσων - Λάζαρος Παπαγεωργίου | Α.Μ: 03114034
Γρηγόρης Θανάσουλας | Α.Μ: 03114131

Άσκηση 1

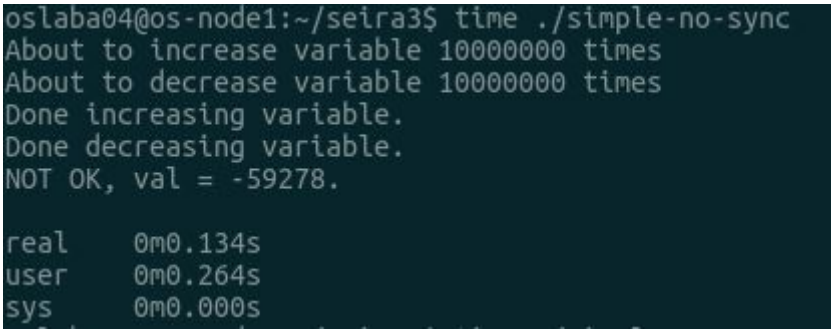
1. Ο χρόνος εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό είναι διακριτά μεγαλύτερος από το χρόνο εκτέλεσης του αρχικού προγράμματος. Αυτό συμβαίνει διότι ο συγχρονισμός, είτε με ατομικές εντολές είτε με mutexes, εμπεριέχει καθυστερήσεις (για την αντιμετώπιση race conditions) οι οποίες δεν υφίστανται όταν τα threads δεν περιέχουν καμία δομή συγχρονισμού.



Μετρήσεις χρόνου για εκτελέσιμο με ατομικές λειτουργίες



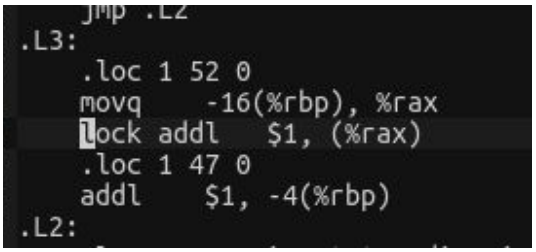
Μετρήσεις χρόνου για εκτελέσιμο με χρήση locks

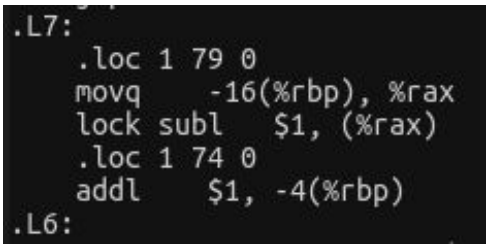


Μετρήσεις χρόνου για εκτελέσιμο χωρίς συγχρονισμό

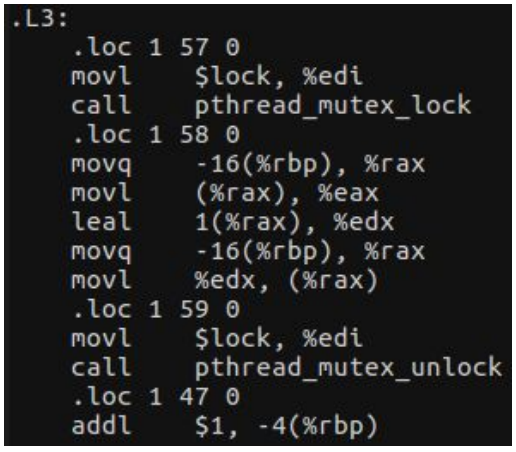
2. Όπως φαίνεται και από τις παραπάνω εκτελέσεις, η χρήση ατομικών λειτουργιών είναι γρηγορότερη. Αυτό συμβαίνει διότι οι ατομικές λειτουργίες υλοποιούνται κατευθείαν στο hardware, εν αντιθέσει με τα mutexes που περιλαμβάνουν την κλήση συναρτήσεων για την υλοποίηση (βλέπε παρακάτω call pthread_mutex_lock κτλ.).

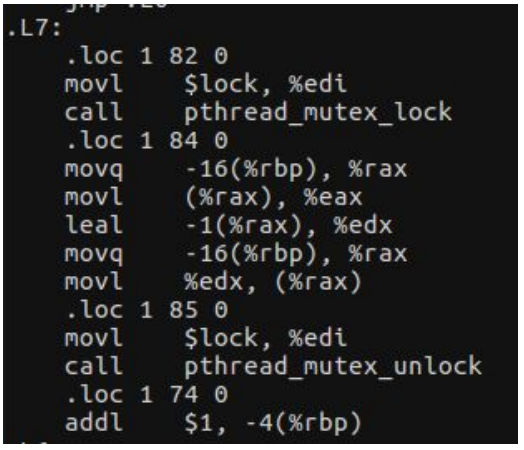
3. Εκτελώντας **gcc -g -S -DSYNC_ATOMIC -pthread simplesync.c -o assembly**, στα περιεχόμενα του αρχείου Assembly βρίσκουμε τις σχετικές εντολές στις οποίες μεταφράζονται τα atomic operations. Οι σχετικές εντολές προς τον επεξεργαστή είναι lock addl και lock subl. Προφανώς, η εντολή lock που προηγείται, δίνει οδηγία στον επεξεργαστή να εκτελέσει την εντολή που ακολουθεί ως atomic operation.





4. Εκτελώντας **gcc -g -S -DSYNC_MUTEX -pthread simplesync.c -o assembly_mutex**, στα περιεχόμενα του αρχείου assembly_mutex βρίσκουμε τις σχετικές εντολές στις οποίες μεταφράζονται τα mutexes. Βλέπουμε εδώ και τους λόγους για τους οποίους η χρήση των mutexes είναι πιο χρονοβόρα, καθώς για το lock και το unlock των mutexes της κάθε πράξης απαιτούνται δύο system calls κάθε φορά, ένα για το κλείδωμα και ένα για το ξεκλείδωμα.





Κώδικας Άσκησης 1

```
1  /*
2  * simplesync.c
3  *
4  * A simple synchronization exercise.
5  *
6  * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
7  * Operating Systems course, ECE, NTUA
8  *
```

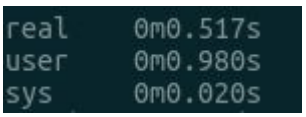
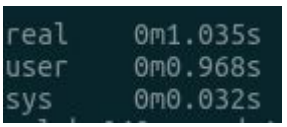
```c
 */

#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 lock;

void *increase_fn(void *arg)
{
    int i;
    volatile int *ip = arg;

    fprintf(stderr, "About to increase variable %d times\n", N);
    for (i = 0; i < N; i++) {
            if (USE_ATOMIC_OPS) {
            /* ... */
            /* You can modify the following line */
            //++(*ip);
            __sync_fetch_and_add (ip, 1);
            /* ... */
            } else {
            /* ... */
            /* You cannot modify the following line */
            pthread_mutex_lock(&lock);
            ++(*ip);
            pthread_mutex_unlock(&lock);
            /* ... */
            }
```

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

    return NULL;
}

void *decrease_fn(void *arg)
{
    int i;
    volatile int *ip = arg;

    fprintf(stderr, "About to decrease variable %d times\n", N);
    for (i = 0; i < N; i++) {
            if (USE_ATOMIC_OPS) {
            /* ... */
            /* You can modify the following line */
            //--(*ip);
            __sync_fetch_and_sub (ip, 1);
            } else {
            /* ... */
            pthread_mutex_lock(&lock);
            /* You cannot modify the following line */
            --(*ip);
            pthread_mutex_unlock(&lock);
            /* ... */
            }
    }
    fprintf(stderr, "Done decreasing variable.\n");

    return NULL;
}


int main(int argc, char *argv[])
{
    int val, ret, ok;
    pthread_t t1, t2;

    /*
    * Initial value
    */
    val = 0;

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

    /*
    * Create threads
    */
    ret = pthread_create(&t1, NULL, increase_fn, &val);
```

```
115     if (ret) {
116             perror_pthread(ret, "pthread_create");
117             exit(1);
118     }
119     ret = pthread_create(&t2, NULL, decrease_fn, &val);
120     if (ret) {
121             perror_pthread(ret, "pthread_create");
122             exit(1);
123     }
124
125     /*
126     * Wait for threads to terminate
127     */
128     ret = pthread_join(t1, NULL);
129     if (ret)
130             perror_pthread(ret, "pthread_join");
131     ret = pthread_join(t2, NULL);
132     if (ret)
133             perror_pthread(ret, "pthread_join");
134
135     /*
136     * Is everything OK?
137     */
138     ok = (val == 0);
139
140     printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
141
142     // Destroy lock
143     pthread_mutex_destroy(&lock);
144
145     return ok;
146 }
```

Άσκηση 2

1. Χρειαζόμαστε τόσους σημαφόρους, όσα είναι και τα threads μας.
2.

 Εκτέλεση παράλληλου με 2 threads	 Εκτέλεση σειριακού προγράμματος

3. Το παράλληλο πρόγραμμα εμφανίζει επιτάχυνση, διότι το κάθε thread υπολογίζει τις γραμμές που του αναλογούν σε διαφορετικό πυρήνα, με αποτέλεσμα οι υπολογισμοί να γίνονται παράλληλα και συνεπώς γρηγορότερα. Το κρίσιμο τμήμα είναι μόνο ο κώδικας που τυπώνει το αποτέλεσμα. Εάν είχαμε συμπεριλάβει στο κρίσιμο τμήμα τους υπολογισμούς, η χρήση των threads θα επιτάχυνε πολύ λιγότερο την διαδικασία, αφού αυτή θα πλησίαζε περισσότερο τη σειριακή εκτέλεση.

4. Όπως φαίνεται και στο παρακάτω σχήμα, αν πατήσουμε Ctrl + C κατά τη διάρκεια εκτέλεσης, το τερματικό παραμένει στο προηγούμενο χρώμα στο οποίο είχε γίνει set. Θα μπορούσαμε να τροποποιήσουμε τον κώδικα, ώστε όταν πιάνει το σήμα SIGINT και να κάνει reset το χρώμα του terminal όπως φαίνεται παρακάτω:

```
// Out of main
void int_handler(int signum) {
    printf("Caught SIGINT signal! Reseting terminal color and
exiting!\n");
    reset_xterm_color(1);
    exit(1);
}

// Inside main
struct sigaction action;
action.sa_handler = int_handler;
sigaction (SIGINT, &action, NULL);
```

Κώδικας Άσκησης 2

```
 1 /*
 2  * mandel.c
 3  *
 4  * A program to draw the Mandelbrot Set on a 256-color xterm.
 5  *
 6  */

 7
 8 #include <stdio.h>
 9 #include <unistd.h>
10 #include <assert.h>
11 #include <string.h>
12 #include <math.h>
13 #include <stdlib.h>
14 #include <pthread.h>
15 #include <semaphore.h>
16 #include <signal.h>
17
18 #include "mandel-lib.h"
19
20 #define MANDEL_MAX_ITERATION 100000
21
22 // Declare a pointer for semaphore in global context
23 sem_t* semaphore;
24 int N_THREADS;
25
26 /***************************
27  * Compile-time parameters *
28  ***************************/
29
30 /*
31  * Output at the terminal is is x_chars wide by y_chars long
32 */
```

```
33 int y_chars = 50;
34 int x_chars = 90;
35
36 /*
37  * The part of the complex plane to be drawn:
38  * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
39 */
40 double xmin = -1.8, xmax = 1.0;
41 double ymin = -1.0, ymax = 1.0;
42
43 /*
44  * Every character in the final output is
45  * xstep x ystep units wide on the complex plane.
46  */
47 double xstep;
48 double ystep;
49
50 /*
51  * This function computes a line of output
52  * as an array of x_char color values.
53  */
54 void compute_mandel_line(int line, int color_val[])
55 {
56     /*
57     * x and y traverse the complex plane.
58     */
59     double x, y;
60
61     int n;
62     int val;
63
64     /* Find out the y value corresponding to this line */
65     y = ymax - ystep * line;
66
67     /* and iterate for all points on this line */
68     for (x = xmin, n = 0; n < x_chars; x+= xstep, n++) {
69
70     /* Compute the point's color value */
71     val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
72     if (val > 255)
73             val = 255;
74
75     /* And store it in the color_val[] array */
76     val = xterm_color(val);
77     color_val[n] = val;
78     }
79 }
80
81 /*
82  * This function outputs an array of x_char color values
83  * to a 256-color xterm.
84  */
85 void output_mandel_line(int fd, int color_val[])
86 {
87     int i;
88
89     char point ='@';
90     char newline='\n';
91
92     for (i = 0; i < x_chars; i++) {
```

```
 93      /* Set the current color, then output the point */
 94      set_xterm_color(fd, color_val[i]);
 95      if (write(fd, &point, 1) != 1) {
 96              perror("compute_and_output_mandel_line: write point");
 97              exit(1);
 98      }
 99      }
100
101     /* Now that the line is done, output a newline character */
102     if (write(fd, &newline, 1) != 1) {
103     perror("compute_and_output_mandel_line: write newline");
104     exit(1);
105     }
106 }
107
108 void *safe_malloc(size_t size)
109 {
110     void *p;
111
112     if ((p = malloc(size)) == NULL) {
113     fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
114             size);
115     exit(1);
116     }
117
118     return p;
119 }
120
121 void compute_and_output_mandel_line(int fd, int line)
122 {
123     /*
124     * A temporary array, used to hold color values for the line being drawn
125     */
126     int color_val[x_chars];
127
128     compute_mandel_line(line, color_val);
129     // Wait for the corresponding sempahore, then output line
130     sem_wait(&semaphore[line % N_THREADS]);
131     output_mandel_line(fd, color_val);
132     // Increase sempahore of next line
133     sem_post(&semaphore[(line + 1) % N_THREADS]);
134 }
135
136
137
138 void *thread_function(void* line_arg) {
139     int* line_ptr = (int*)line_arg;
140     // Compute and  lines line, line  + N_THREADS, line + 2*NTHREADS, etc
141     int line;
142     for (line = *line_ptr; line < y_chars; line += N_THREADS) {
143     compute_and_output_mandel_line(1, line);
144     }
145     pthread_exit(0);
146 }
147
148 void int_handler(int signum) {
149     printf("Caught SIGINT signal! Reseting terminal color and exiting!\n");
150     reset_xterm_color(1);
151     exit(1);
152 }
```

```
153
154 int main(int argc, char* argv[])
155 {
156     // Check if sufficient arguments have been provided
157     if (argc != 2) {
158     printf("Usage: ./mandel <N_THREADS>\n");
159     exit(1);
160     }
161
162     N_THREADS = atoi(argv[1]);
163     printf("Running for N_THREADS = %d\n", N_THREADS);
164
165     struct sigaction action;
166     action.sa_handler = int_handler;
167     sigaction (SIGINT, &action, NULL);
168
169     /* Allocate memory space for semaphores, initialize
170     * them to 0, except for the semaphore[0] wich should
171     * be initialized to value 1
172     */
173     semaphore = safe_malloc(N_THREADS * sizeof(sem_t));
174     sem_init(&semaphore[0], 0, 1);
175     int i = 0;
176     for (i = 1; i < N_THREADS; i++) {
177     sem_init(&semaphore[i], 0, 0);
178     }
179
180     int line;
181
182     xstep = (xmax - xmin) / x_chars;
183     ystep = (ymax - ymin) / y_chars;
184
185     // Allocate space for threads type
186     pthread_t *threads = safe_malloc(N_THREADS * sizeof(pthread_t));
187     int* args = safe_malloc(N_THREADS * sizeof(int));
188
189
190     /*
191     * draw the Mandelbrot Set, one line at a time.
192     * Output is sent to file descriptor '1', i.e., standard output.
193     */
194
195     // Create N_THREADS
196     for (line = 0; line < N_THREADS; line++) {
197     args[line] = line;
198     pthread_create(&threads[line], NULL, thread_function, &args[line]);
199     }
200
201     for (line = 0; line < N_THREADS; line++) {
202     pthread_join(threads[line], NULL);
203     }
204
205     reset_xterm_color(1);
206     return 0;
207 }
```

Άσκηση 3

1. Η υλοποίηση μας δε δίνει κάποια προτεραιότητα στην teachers_exit έναντι της child_enter. Επομένως, θα εκτελεστεί κανονικά η child_enter και θα μπει ένα παιδί, παρότι κάποιος δάσκαλος επιθυμούσε να βγει από το νηπιαγωγείο.
2. Ναι, υπάρχουν καταστάσεις συναγωνισμού στο κώδικα του testing. Τέτοια κατάσταση είναι αυτή που απεικονίζεται παρακάτω, όπου γίνεται προσπάθεια κλειδώματος του mutex ώστε να γίνει verify το thread. Το κλείδωμα αυτό είναι απαραίτητο ώστε να αποτραπεί η μεταβολή των στοιχείων του thread απο άλλο νήμα, όσο γίνεται το verify.

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

Επιπλέον, μία ακόμη προφανής κατάσταση συναγωνισμού, προκύπτει όταν ένα παιδί φεύγει, οπότε ειδοποιεί τόσο τους δασκάλους που θέλουν να κάνουν exit αλλά και παιδιά που έχουν ζητήσει να κάνουν enter. Εδώ οι δάσκαλοι και τα παιδιά συναγωνίζονται μεταξύ τους για το ποιος θα κάνει lock το mutex και θα να επιτύχει το στόχο -με την προυπόθεση να τηρούνται οι περιορισμοί- όποιο οριστεί πρώτο από τον χρονοδρομολογητή.

Παρόμοια συμβαίνει όταν πολλοί δάσκαλοι θέλουν να εξέλθουν και ειδοποιηθούν να εξέλθει ένας, οπότε και συναγωνίζονται μεταξύ τους για το ποιος θα εξέλθει, ή όταν πολλά παιδιά επιθυμούν να εισέλθουν και ενημερώνεται να εισέλθει κάποιο.

Κώδικας Άσκησης 3

```
 1 /*
 2  * kgarten.c
 3  *
 4  * A kindergarten simulator.
 5  * Bad things happen if teachers and children
 6  * are not synchronized properly.
 7  *
 8  *
 9  * Author:
10  * Vangelis Koukis <vkoukis@cslab.ece.ntua.gr>
11  *
12  * Additional Authors:
13  * Stefanos Gerangelos <sgerag@cslab.ece.ntua.gr>
14  * Anastassios Nanos <ananos@cslab.ece.ntua.gr>
15  * Operating Systems course, ECE, NTUA
16  *
17  */
18
19 #include <time.h>
20 #include <errno.h>
```

```c
#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 enough_teachers;

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

    pthread_mutex_t mutex;
};

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

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

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

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

```
 74     char *endp;
 75
 76     l = strtol(s, &endp, 10);
 77     if (s != endp && *endp == '\0') {
 78             *val = l;
 79             return 0;
 80     } else
 81             return -1;
 82 }
 83
 84 void *safe_malloc(size_t size)
 85 {
 86     void *p;
 87
 88     if ((p = malloc(size)) == NULL) {
 89             fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
 90             size);
 91             exit(1);
 92     }
 93
 94     return p;
 95 }
 96
 97 void usage(char *argv0)
 98 {
 99     fprintf(stderr, "Usage: %s thread_count child_threads c_t_ratio\n\n"
100             "Exactly two argument required:\n"
101             "       thread_count: Total number of threads to create.\n"
102             "       child_threads: The number of threads simulating children.\n"
103             "       c_t_ratio: The allowed ratio of children to teachers.\n\n",
104             argv0);
105     exit(1);
106 }
107
108 void bad_thing(int thrid, int children, int teachers)
109 {
110     int thing, sex;
111     int namecnt, nameidx;
112     char *name, *p;
113     char buf[1024];
114
115     char *things[] = {
116             "Little %s put %s finger in the wall outlet and got electrocuted!",
117             "Little %s fell off the slide and broke %s head!",
118             "Little %s was playing with matches and lit %s hair on fire!",
119             "Little %s drank a bottle of acid with %s lunch!",
120             "Little %s caught %s hand in the paper shredder!",
121             "Little %s wrestled with a stray dog and it bit %s finger off!"
122     };
123     char *boys[] = {
124             "George", "John", "Nick", "Jim", "Constantine",
125             "Chris", "Peter", "Paul", "Steve", "Billy", "Mike",
126             "Vangelis", "Antony"
```

```
127     };
128     char *girls[] = {
129             "Maria", "Irene", "Christina", "Helena", "Georgia", "Olga",
130             "Sophie", "Joanna", "Zoe", "Catherine", "Marina", "Stella",
131             "Vicky", "Jenny"
132     };
133
134     thing = rand() % 4;
135     sex = rand() % 2;
136
137     namecnt = sex ? sizeof(boys)/sizeof(boys[0]) : sizeof(girls)/sizeof(girls[0]);
138     nameidx = rand() % namecnt;
139     name = sex ? boys[nameidx] : girls[nameidx];
140
141     p = buf;
142     p += sprintf(p, "*** Thread %d: Oh no! ", thrid);
143     p += sprintf(p, things[thing], name, sex ? "his" : "her");
144     p += sprintf(p, "\n*** Why were there only %d teachers for %d children?!\n",
145             teachers, children);
146
147     /* Output everything in a single atomic call */
148     printf("%s", buf);
149 }
150
151 void child_enter(struct thread_info_struct *thr)
152 {
153     if (!thr->is_child) {
154             fprintf(stderr, "Internal error: %s called for a Teacher thread.\n",
155             __func__);
156             exit(1);
157     }
158
159     fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);
160
161     /*
162     * While Loop for spurious wakeups
163     *
https://stackoverflow.com/questions/8594591/why-does-pthread-cond-wait-have-spurious-wak
eups
164     */
165
166     pthread_mutex_lock(&thr->kg->mutex);
167     while (thr->kg->vc >= thr->kg->vt * thr->kg->ratio) {
168             pthread_cond_wait(&thr->kg->enough_teachers, &thr->kg->mutex);
169     }
170     ++(thr->kg->vc);
171     pthread_mutex_unlock(&thr->kg->mutex);
172 }
173
174 void child_exit(struct thread_info_struct *thr)
175 {
176
177     if (!thr->is_child) {
```

```
178             fprintf(stderr, "Internal error: %s called for a Teacher thread.\n",
179             __func__);
180             exit(1);
181     }
182
183     fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);
184
185     pthread_mutex_lock(&thr->kg->mutex);
186     --(thr->kg->vc);
187     pthread_cond_broadcast(&thr->kg->enough_teachers);
188     pthread_mutex_unlock(&thr->kg->mutex);
189 }
190
191 void teacher_enter(struct thread_info_struct *thr)
192 {
193     if (thr->is_child) {
194             fprintf(stderr, "Internal error: %s called for a Child thread.\n",
195             __func__);
196             exit(1);
197     }
198
199     fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);
200
201     pthread_mutex_lock(&thr->kg->mutex);
202     ++(thr->kg->vt);
203     pthread_cond_broadcast(&thr->kg->enough_teachers);
204     pthread_mutex_unlock(&thr->kg->mutex);
205 }
206
207 void teacher_exit(struct thread_info_struct *thr)
208 {
209     if (thr->is_child) {
210             fprintf(stderr, "Internal error: %s called for a Child thread.\n",
211             __func__);
212             exit(1);
213     }
214
215     fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);
216
217     pthread_mutex_lock(&thr->kg->mutex);
218     while (thr->kg->vc > (thr->kg->vt - 1) * thr->kg->ratio) {
219             pthread_cond_wait(&thr->kg->enough_teachers, &thr->kg->mutex);
220     }
221     --(thr->kg->vt);
222     pthread_mutex_unlock(&thr->kg->mutex);
223 }
224
225 /*
226  * Verify the state of the kindergarten.
227  */
228 void verify(struct thread_info_struct *thr)
229 {
230             struct kgarten_struct *kg = thr->kg;
```

```
231             int t, c, r;
232
233             c = kg->vc;
234             t = kg->vt;
235             r = kg->ratio;
236
237             fprintf(stderr, "            Thread %d: Teachers: %d, Children: %d\n",
238                     thr->thrid, t, c);
239
240             if (c > t * r) {
241                     bad_thing(thr->thrid, c, t);
242                     exit(1);
243             }
244 }
245
246
247 /*
248  * A single thread.
249  * It simulates either a teacher, or a child.
250  */
251 void *thread_start_fn(void *arg)
252 {
253     /* We know arg points to an instance of thread_info_struct */
254     struct thread_info_struct *thr = arg;
255     char *nstr;
256
257     fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);
258
259     nstr = thr->is_child ? "Child" : "Teacher";
260     for (;;) {
261             fprintf(stderr, "Thread %d [%s]: Entering.\n", thr->thrid, nstr);
262             if (thr->is_child)
263             child_enter(thr);
264             else
265             teacher_enter(thr);
266
267             fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);
268
269             /*
270             * We're inside the critical section,
271             * just sleep for a while.
272             */
273             /* usleep(rand_r(&thr->rseed) % 1000000 / (thr->is_child ? 10000 : 1)); */
274             pthread_mutex_lock(&thr->kg->mutex);
275             verify(thr);
276             pthread_mutex_unlock(&thr->kg->mutex);
277
278             usleep(rand_r(&thr->rseed) % 1000000);
279
280             fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);
281             /* CRITICAL SECTION END */
282
283             if (thr->is_child)
```

```
284             child_exit(thr);
285             else
286             teacher_exit(thr);
287
288             fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);
289
290             /* Sleep for a while before re-entering */
291             /* usleep(rand_r(&thr->rseed) % 100000 * (thr->is_child ? 100 : 1)); */

292             usleep(rand_r(&thr->rseed) % 100000);
293
294             pthread_mutex_lock(&thr->kg->mutex);
295             verify(thr);
296             pthread_mutex_unlock(&thr->kg->mutex);
297     }
298
299     fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr->thrcnt);
300
301     return NULL;
302 }
303
304
305 int main(int argc, char *argv[])
306 {
307     int i, ret, thrcnt, chldcnt, ratio;
308     struct thread_info_struct *thr;
309     struct kgarten_struct *kg;
310
311     /*
312     * Parse the command line
313     */
314     if (argc != 4)
315             usage(argv[0]);
316     if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0) {
317             fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
318             exit(1);
319     }
320     if (safe_atoi(argv[2], &chldcnt) < 0 || chldcnt < 0 || chldcnt > thrcnt) {
321             fprintf(stderr, "`%s' is not valid for `child_threads'\n", argv[2]);
322             exit(1);
323     }
324     if (safe_atoi(argv[3], &ratio) < 0 || ratio < 1) {
325             fprintf(stderr, "`%s' is not valid for `c_t_ratio'\n", argv[3]);
326             exit(1);
327     }
328
329
330     /*
331     * Initialize kindergarten and random number generator
332     */
333     srand(time(NULL));
334
335     kg = safe_malloc(sizeof(*kg));
```

```
336     kg->vt = kg->vc = 0;
337     kg->ratio = ratio;
338
339     ret = pthread_mutex_init(&kg->mutex, NULL);
340     if (ret) {
341             perror_pthread(ret, "pthread_mutex_init");
342             exit(1);
343     }
344
345     ret = pthread_cond_init(&kg->enough_teachers, NULL);
346     /* ... */
347
348     /*
349     * Create threads
350     */
351     thr = safe_malloc(thrcnt * sizeof(*thr));
352
353     for (i = 0; i < thrcnt; i++) {
354             /* Initialize per-thread structure */
355             thr[i].kg = kg;
356             thr[i].thrid = i;
357             thr[i].thrcnt = thrcnt;
358             thr[i].is_child = (i < chldcnt);
359             thr[i].rseed = rand();
360
361             /* Spawn new thread */
362             ret = pthread_create(&thr[i].tid, NULL, thread_start_fn, &thr[i]);
363             if (ret) {
364             perror_pthread(ret, "pthread_create");
365             exit(1);
366             }
367     }
368
369     /*
370     * Wait for all threads to terminate
371     */
372     for (i = 0; i < thrcnt; i++) {
373             ret = pthread_join(thr[i].tid, NULL);
374             if (ret) {
375             perror_pthread(ret, "pthread_join");
376             exit(1);
377             }
378     }
379
380     printf("OK.\n");
381
382     return 0;
383 }
```