ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ

ΗΛΙΑΣ ΝΤΟΝΤΟΡΟΣ
ΘΟΔΩΡΗΣ- ΙΩΑΝΝΗΣ ΚΙΤΣΗΣ

ΑΝΑΦΟΡΑ 3ης ΣΕΙΡΑΣ ΑΣΚΗΣΕΩΝ

ΑΣΚΗΣΗ 1Η

3.1.1 Οι χρόνοι εκτέλεσης των εκτελέσιμων που εκτελούν συγχρονισμό είναι μεγαλύτεροι από τον χρόνο εκτέλεσης του αρχικού προγράμματος χωρίς συγχρονισμό. Αυτό διότι κάθε στιγμή στο κρίσιμο σημείο υπάρχει μόνο ένα νήμα ενώ στο αρχικό πρόγραμμα δεν ξέρουμε κάθε φορά ποιο νήμα θα χρησιμοποιηθεί.

```
Simplesync-atomic:
real    0m0,051s
user    0m0,094s
sys     0m0,000s

Simplesync-mutex:
real    0m1,546s
user    0m1,827s
sys     0m1,218s
```

3.1.2 Η μέθοδος συγχρονισμού με atomic operations είναι φανερά πιο γρήγορη από αυτή με τα mutexes . Τα atomic operations είναι low level,χρησιμοποιεί δηλαδή εντολές του επεξεργαστή χωρίς κάποιο νήμα να μπει σε sleep. Στα mutexes,όμως, έχουμε locks και unlocks,δηλαδή sleep-wake του νήματος που περιμένει να μπει στο κρίσιμο σημείο, γι’αυτό είναι και πιο αργή μέθοδος.

3.1.3 Βάζοντας την παράμετρο -S στην μεταγλώττιση στα παραγόμενα αρχεία assembly βρίσκουμε για το thread της αύξησης :

```
1.  .LBE17:
2.    .loc 1 49 9 is_stmt 1 view .LVU17
3.    .loc 1 51 13 view .LVU18
4.    lock addq $1, (%rsp)
5.    .loc 1 47 24 view .LVU19
```

Και για το thread της μείωσης :

```
1.  .LBE25:
2.    .loc 1 73 9 is_stmt 1 view .LVU47
3.    .loc 1 75 13 view .LVU48
4.    lock subq $1, (%rsp)
5.    .loc 1 71 24 view .LVU49
```

3.1.4 Για το pthread_mutex_lock() έχουμε :

```
1.  .loc 1 55 0
2.          movl    $mutex1, %edi
3.          call    pthread_mutex_lock
```

Και για το pthread_mutex_unlock() :

```
1.  .loc 1 57 0
2.          movl    $mutex1, %edi
3.          call    pthread_mutex_unlock
```

Ο κώδικας για την πρώτη άσκηση :

```
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.   *
9.   */
10.
11. #include <errno.h>
12. #include <stdio.h>
13. #include <stdlib.h>
14. #include <unistd.h>
15. #include <pthread.h>
16.
17. /*
18.  * POSIX thread functions do not return error numbers in errno,
19.  * but in the actual return value of the function call instead.
20.  * This macro helps with error reporting in this case.
21.  */
22. #define perror_pthread(ret, msg) \
23.     do { errno = ret; perror(msg); } while (0)
24.
25. #define N 10000000
26.
27. /* Dots indicate lines where you are free to insert code at will */
28. /* ... */
29. #if defined(SYNC_ATOMIC) ^ defined(SYNC_MUTEX) == 0
30. # error You must #define exactly one of SYNC_ATOMIC or SYNC_MUTEX.
31. #endif
32.
33. #if defined(SYNC_ATOMIC)
34. # define USE_ATOMIC_OPS 1
```

```
35. #else
36. # define USE_ATOMIC_OPS 0
37. #endif
38. pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
39.
40. void *increase_fn(void *arg)
41. {
42.     int i;
43.     volatile int *ip = arg;
44.
45.     fprintf(stderr, "About to increase variable %d times\n", N);
46.     for (i = 0; i < N; i++)
47.     {
48.         if (USE_ATOMIC_OPS)
49.         {
50.             __sync_fetch_and_add(&ip, 1);
51.         }
52.         else
53.         {
54.             pthread_mutex_lock(&mutex1);
55.             ++(*ip);
56.             pthread_mutex_unlock(&mutex1);
57.         }
58.     }
59.     fprintf(stderr, "Done increasing variable.\n");
60.
61.     return NULL;
62. }
63.
64. void *decrease_fn(void *arg)
65. {
66.     int i;
67.     volatile int *ip = arg;
68.
69.     fprintf(stderr, "About to decrease variable %d times\n", N);
70.     for (i = 0; i < N; i++)
71.     {
72.         if (USE_ATOMIC_OPS)
73.         {
74.             __sync_fetch_and_add(&ip, -1);
75.         }
76.         else
77.         {
78.             pthread_mutex_lock(&mutex1);
79.
80.             --(*ip);
81.             pthread_mutex_unlock(&mutex1);
82.         }
83.     }
84.     fprintf(stderr, "Done decreasing variable.\n");
85.
86.     return NULL;
87. }
88.
89. int main(int argc, char *argv[])
90. {
91.     int val, ret, ok;
92.     pthread_t t1, t2;
93.
94.     val = 0;
95.
96.     /*
97.      * Create threads
98.      */
99.     ret = pthread_create(&t1, NULL, increase_fn, &val);
100.      if (ret)
101.      {
102.          perror_pthread(ret, "pthread_create");
103.          exit(1);
104.      }
```

```
105.      ret = pthread_create(&t2, NULL, decrease_fn, &val);
106.      if (ret)
107.      {
108.          perror_pthread(ret, "pthread_create");
109.          exit(1);
110.      }
111.
112.      /*
113.       * Wait for threads to terminate
114.       */
115.      ret = pthread_join(t1, NULL);
116.      if (ret)
117.          perror_pthread(ret, "pthread_join");
118.      ret = pthread_join(t2, NULL);
119.      if (ret)
120.          perror_pthread(ret, "pthread_join");
121.
122.      /*
123.       * Is everything OK?
124.       */
125.      ok = (val == 0);
126.
127.      printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
128.
129.      return ok;
130.  }
```

ΑΣΚΗΣΗ 2Η

3.2.1 Ο αριθμός των σημαφόρων που χρειάζεται είναι όσα είναι τα threads που χρησιμοποιούμε.

3.2.2 Παρατηρώ πως συγκριτικά με το σειριακό πρόγραμμα ο χρόνος στο παράλληλο πρόγραμμα με 2 νήματα έχει περίπου μισό πραγματικό χρόνο και περίπου μισό χρόνο συστήματος, ενώ ο χρόνος που αντιλαμβάνεται ο χρήστης είναι σχεδόν ίδιος. Με την εκτέλεση της εντολής cat /proc/cpuinfo προκύπτει → “cpu cores : 8” .

3.2.3 Το πρόγραμμα μας εμφανίζει επιτάχυνση καθώς το κρίσιμο κομμάτι είναι η εκτύπωση των διαφορετικών γραμμών οπότε ο υπολογισμός όταν μοιράζεται σε διαφορετικά threads γίνεται πιο γρηγορά. Δηλαδή αν κάθε νήμα υπολόγιζε και εκτύπωνε την γραμμή που έπρεπε και μετά συνέχιζε το επόμενο νήμα τότε δεν θα είχαμε βελτίωση στον χρόνο εκτέλεσης του προγράμματος.

3.2.4 Αν πατήσουμε Ctrl-C πριν ολοκληρωθεί το προγραμμα τότε δεν έχουν γίνει reset τα χρώματα και το τερματικό εμφανίζει ότι γράφουμε με χρώμα. Για να το λύσουμε αυτό θα χρησιμοποιήσουμε signal handler και όταν λάβουμε σήμα διακοπής από τον χρήστη

θα φροντίσουμε πρώτα να κάνουμε reset color και μετά να διακόψουμε το πρόγραμμα μας.

Ο κώδικας για την δεύτερη άσκηση :

```
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 <semaphore.h>
15. #include "mandel-lib.h"
16.
17. #define MANDEL_MAX_ITERATION 100000
18.
19. /***************************
20.  * Compile-time parameters *
21.  ***************************/
22.
23. /*
24.  * Output at the terminal is is x_chars wide by y_chars long
25.  */
26. int y_chars = 50;
27. int x_chars = 90;
28.
29. /*
30.  * The part of the complex plane to be drawn:
31.  * upper left corner is (xmin, ymax), lower right corner is (xmax, ymin)
32.  */
33. double xmin = -1.8, xmax = 1.0;
34. double ymin = -1.0, ymax = 1.0;
35.
36. /*
37.  * Every character in the final output is
38.  * xstep x ystep units wide on the complex plane.
39.  */
40. double xstep;
41. double ystep;
42. sem_t *mutex;
43.
44. struct thread_stats_struct
45. {
46.   pthread_t tid;
47.   int *color_val;
48.   int thrid;
49.   int thrcnt;
50. };
51.
52. int my_atoi(char *s, int *val)
53. {
54.   long l;
55.   char *endp;
56.   l = strtol(s, &endp, 10);
57.   if (s != endp && *endp == '\0')
58.   {
59.             *val = l;
60.             return 0;
```

```
61.   }
62.   else
63.             return -1;
64. }
65.
66. void *my_malloc(size_t size)
67. {
68.   void *p;
69.   if ((p = malloc(size)) == NULL)
70.   {
71.             fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
72.                               size);
73.             exit(1);
74.   }
75.
76.   return p;
77. }
78.
79. /*
80.  * This function computes a line of output
81.  * as an array of x_char color values.
82.  */
83. void compute_mandel_line(int line, int color_val[])
84. {
85.   /*
86.    * x and y traverse the complex plane.
87.    */
88.   double x, y;
89.
90.   int n;
91.   int val;
92.
93.   /* Find out the y value corresponding to this line */
94.   y = ymax - ystep * line;
95.
96.   /* and iterate for all points on this line */
97.   for (x = xmin, n = 0; n < x_chars; x += xstep, n++)
98.   {
99.
100.                    /* Compute the point's color value */
101.                    val = mandel_iterations_at_point(x, y, MANDEL_MAX_ITERATION);
102.                    if (val > 255)
103.                              val = 255;
104.
105.                    /* And store it in the color_val[] array */
106.                    val = xterm_color(val);
107.                    color_val[n] = val;
108.            }
109.  }
110.
111.  /*
112.   * This function outputs an array of x_char color values
113.   * to a 256-color xterm.
114.   */
115.  void output_mandel_line(int fd, int color_val[])
116.  {
117.            int i;
118.
119.            char point = '$';
120.            char newline = '\n';
121.
122.            for (i = 0; i < x_chars; i++)
123.            {
124.                      /* Set the current color, then output the point */
125.                      set_xterm_color(fd, color_val[i]);
126.                      if (write(fd, &point, 1) != 1)
127.                      {
128.                                perror("compute_and_output_mandel_line: write point");
129.                                exit(1);
130.                      }
```

```
131.            }
132.
133.            /* Now that the line is done, output a newline character */
134.            if (write(fd, &newline, 1) != 1)
135.            {
136.                    perror("compute_and_output_mandel_line: write newline");
137.                    exit(1);
138.            }
139.    }
140.
141.    void *compute_and_output_mandel_line(void *arg)
142.    {
143.            struct thread_stats_struct *thread = arg;
144.            int i;
145.
146.            for (i = thread->thrid; i < y_chars; i += thread->thrcnt)
147.            {
148.                    compute_mandel_line(i, thread->color_val);
149.                    sem_wait(&mutex[i % thread->thrcnt]);
150.                    output_mandel_line(1, thread->color_val);
151.                    sem_post(&mutex[(i + 1) % thread->thrcnt]);
152.            }
153.
154.            return 0;
155.    }
156.
157.    int main(int argc, char *argv[])
158.    {
159.            if (argc != 2)
160.            {
161.                    fprintf(stderr, "Usage: %s thread_count array_size\n\n"
162.                                                            "Exactly one argument re-
    quired:\n"
163.                                                            "    thread_count: The num-
    ber of threads to create.\n",
164.                                            argv[0]);
165.            }
166.
167.            int line;
168.            int i, ret, thrcnt;
169.            struct thread_stats_struct *thread;
170.
171.            xstep = (xmax - xmin) / x_chars;
172.            ystep = (ymax - ymin) / y_chars;
173.
174.            if (my_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0)
175.            {
176.                    fprintf(stderr, "`%s' is not valid for `thread_count'\n", argv[1]);
177.                    exit(1);
178.            }
179.
180.            thread = my_malloc(thrcnt * sizeof(*thread));
181.            mutex = my_malloc(thrcnt * sizeof(sem_t));
182.            for (i = 0; i < thrcnt; i++)
183.            {
184.                    /* Initialize per-thread structure */
185.                    thread[i].thrid = i;
186.                    thread[i].thrcnt = thrcnt;
187.                    thread[i].color_val = my_malloc(x_chars * sizeof(int));
188.                    (i == 0) ? sem_init(&mutex[i], 0, 1) : sem_init(&mutex[i], 0, 0);
189.                    /* Spawn new thread */
190.                    ret = pthread_create(&thread[i].tid, NULL, compute_and_output_man-
    del_line, &thread[i]);
191.            }
192.            for (i = 0; i < thrcnt; i++)
193.            {
194.                    ret = pthread_join(thread[i].tid, NULL);
195.                    if (ret)
196.                    {
197.                            //                  perror_pthread(ret, "pthread_join");
```

```
198.                                exit(1);
199.                        }
200.            }
201.            sem_destroy(&mutex);
202.            reset_xterm_color(1);
203.            return 0;
204.  }
205.
```

ΑΚΣΗΣΗ 3Η

3.2.1 Στην ουσία το ερώτημα είναι : ποιος "νικάει" ? Τα νέα παιδιά που θέλουν να εισέλθουν ή ο ένας καθηγητής που περιμένει να φύγει. Λόγω του ότι ο καθηγητής είναι ένα νήμα ενώ τα παιδιά πολλά προκύπτει πιθανοτικά ότι τις περισσότερες φορές εισέρχονται τα παιδιά.

3.2.2 Υπάρχουν καταστάσεις συναγωνισμού (races) και συγκεκριμένα όταν φεύγει ένα παιδί και ο συνολικός αριθμός των παιδιών επιτρέπει σε έναν καθηγητή να φύγει . Το race condition δημιουργείται ανάμεσα σε κάποιο παιδί που θέλει να εισέλθει και σε καθηγητή που θέλει να φύγει.

Ο κώδικας για την τρίτη άσκηση :

```
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>
21. #include <stdio.h>
22. #include <stdlib.h>
23. #include <unistd.h>
24. #include <pthread.h>
25. #include <semaphore.h>
26.
27. /*
```

```
 * 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)
```

```
98.   {
99.             fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
100.                                       size);
101.                     exit(1);
102.            }
103.
104.            return p;
105.    }
106.
107.    void usage(char *argv0)
108.    {
109.            fprintf(stderr, "Usage: %s thread_count child_threads c_t_ratio\n\n"
110.                                                    "Exactly two argument required:\n"
111.                                                    "    thread_count: Total number of
    threads to create.\n"
112.                                                    "    child_threads: The number of
    threads simulating children.\n"
113.                                                    "    c_t_ratio: The allowed ratio of
    children to teachers.\n\n",
114.                                argv0);
115.            exit(1);
116.    }
117.
118.    void bad_thing(int thrid, int children, int teachers)
119.    {
120.            int thing, sex;
121.            int namecnt, nameidx;
122.            char *name, *p;
123.            char buf[1024];
124.
125.            char *things[] = {
126.                    "Little %s put %s finger in the wall outlet and got electrocuted!",
127.                    "Little %s fell off the slide and broke %s head!",
128.                    "Little %s was playing with matches and lit %s hair on fire!",
129.                    "Little %s drank a bottle of acid with %s lunch!",
130.                    "Little %s caught %s hand in the paper shredder!",
131.                    "Little %s wrestled with a stray dog and it bit %s finger off!"};
132.            char *boys[] = {
133.                    "George", "John", "Nick", "Jim", "Constantine",
134.                    "Chris", "Peter", "Paul", "Steve", "Billy", "Mike",
135.                    "Vangelis", "Antony"};
136.            char *girls[] = {
137.                    "Maria", "Irene", "Christina", "Helena", "Georgia", "Olga",
138.                    "Sophie", "Joanna", "Zoe", "Catherine", "Marina", "Stella",
139.                    "Vicky", "Jenny"};
140.
141.            thing = rand() % 4;
142.            sex = rand() % 2;
143.
144.            namecnt = sex ? sizeof(boys) / sizeof(boys[0]) : sizeof(girls) /
    sizeof(girls[0]);
145.            nameidx = rand() % namecnt;
146.            name = sex ? boys[nameidx] : girls[nameidx];
147.
148.            p = buf;
149.            p += sprintf(p, "* Thread %d: Oh no! ", thrid);
150.            p += sprintf(p, things[thing], name, sex ? "his" : "her");
151.            p += sprintf(p, "\n*** Why were there only %d teachers for %d children?!\n",
152.                                            teachers, children);
153.
154.            /* Output everything in a single atomic call */
155.            printf("%s", buf);
156.    }
157.
158.    /*
159.     * Verify the state of the kindergarten.
160.     */
161.    void verify(struct thread_info_struct *thr)
162.    {
163.            struct kgarten_struct *kg = thr->kg;
```

```
164.            int t, c, r;
165.
166.            c = kg->vc;
167.            t = kg->vt;
168.            r = kg->ratio;
169.
170.            fprintf(stderr, "Thread %d: Teachers: %d, Children: %d\n",
171.                                    thr->thrid, t, c);
172.
173.            if (c > t * r)
174.            {
175.                    bad_thing(thr->thrid, c, t);
176.                    exit(1);
177.            }
178.    }
179.    void child_enter(struct thread_info_struct *thr)
180.    {
181.            if (!thr->is_child)
182.            {
183.                    fprintf(stderr, "Internal error: %s called for a Teacher
    thread.\n",
184.                                            __func__);
185.                    exit(1);
186.            }
187.            pthread_mutex_lock(&thr->kg->mutex);
188.            while (((thr->kg->vt) * thr->kg->ratio) < (thr->kg->vc + 1))
189.            {
190.                    fprintf(stderr, "it's not safe to enter\n");
191.                    pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
192.            }
193.
194.            fprintf(stderr, "THREAD %d: CHILD ENTER\n", thr->thrid);
195.            ++(thr->kg->vc);
196.            pthread_mutex_unlock(&thr->kg->mutex);
197.    }
198.    void child_exit(struct thread_info_struct *thr)
199.    {
200.
201.            if (!thr->is_child)
202.            {
203.                    fprintf(stderr, "Internal error: %s called for a Teacher
    thread.\n",
204.                                            __func__);
205.                    exit(1);
206.            }
207.            fprintf(stderr, "THREAD %d: CHILD EXIT\n", thr->thrid);
208.
209.            pthread_mutex_lock(&thr->kg->mutex);
210.            --(thr->kg->vc);
211.            pthread_cond_broadcast(&thr->kg->cond);
212.            pthread_mutex_unlock(&thr->kg->mutex);
213.    }
214.
215.    void teacher_enter(struct thread_info_struct *thr)
216.    {
217.            if (thr->is_child)
218.            {
219.                    fprintf(stderr, "Internal error: %s called for a Child thread.\n",
220.                                            __func__);
221.                    exit(1);
222.            }
223.
224.            pthread_mutex_lock(&thr->kg->mutex);
225.            ++(thr->kg->vt);
226.            fprintf(stderr, "THREAD %d: TEACHER ENTER\n", thr->thrid);
227.            pthread_cond_broadcast(&thr->kg->cond);
228.            pthread_mutex_unlock(&thr->kg->mutex);
229.    }
230.    void teacher_exit(struct thread_info_struct *thr)
231.    {
```

```
232.            if (thr->is_child)
233.            {
234.                     fprintf(stderr, "Internal error: %s called for a Child thread.\n",
235.                                       __func__);
236.                     exit(1);
237.            }
238.
239.            pthread_mutex_lock(&thr->kg->mutex);
240.
241.            while (((thr->kg->vt) - 1) * thr->kg->ratio < thr->kg->vc)
242.            {
243.                     fprintf(stderr, "teacher we need you.\n");
244.                     pthread_cond_wait(&thr->kg->cond, &thr->kg->mutex);
245.            }
246.            fprintf(stderr, "THREAD %d: TEACHER EXIT\n", thr->thrid);
247.
248.            --(thr->kg->vt);
249.            pthread_mutex_unlock(&thr->kg->mutex);
250.  }
251.
252.  /*
253.   * A single thread.
254.   * It simulates either a teacher, or a child.
255.   */
256.  void *thread_start_fn(void *arg)
257.  {
258.            /* We know arg points to an instance of thread_info_struct */
259.            struct thread_info_struct *thr = arg;
260.            char *nstr;
261.
262.            fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);
263.
264.            nstr = thr->is_child ? "Child" : "Teacher";
265.            for (;;)
266.            {
267.                     fprintf(stderr, "Thread %d [%s]: Entering.\n", thr->thrid, nstr);
268.                     if (thr->is_child)
269.                              child_enter(thr);
270.                     else
271.                              teacher_enter(thr);
272.
273.                     fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);
274.
275.                     /*
276.                      * We're inside the critical section,
277.                      * just sleep for a while.
278.                      */
279.                     /* usleep(rand_r(&thr->rseed) % 1000000 / (thr->is_child ? 10000 :
    1)); */
280.                     pthread_mutex_lock(&thr->kg->mutex);
281.                     verify(thr);
282.                     pthread_mutex_unlock(&thr->kg->mutex);
283.
284.                     usleep(rand_r(&thr->rseed) % 1000000);
285.
286.                     fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);
287.                     /* CRITICAL SECTION END */
288.
289.                     if (thr->is_child)
290.                              child_exit(thr);
291.                     else
292.                              teacher_exit(thr);
293.
294.                     fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);
295.
296.                     /* Sleep for a while before re-entering */
297.                     /* usleep(rand_r(&thr->rseed) % 100000 * (thr->is_child ? 100 :
    1)); */
298.                     usleep(rand_r(&thr->rseed) % 100000);
299.
```

```
300.                    pthread_mutex_lock(&thr->kg->mutex);
301.                    verify(thr);
302.                    pthread_mutex_unlock(&thr->kg->mutex);
303.            }
304.
305.            fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr->thrcnt);
306.
307.            return NULL;
308.    }
309.
310.    int main(int argc, char *argv[])
311.    {
312.            int i, ret, thrcnt, chldcnt, ratio;
313.            struct thread_info_struct *thr;
314.            struct kgarten_struct *kg;
315.
316.            /*
317.             * Parse the command line
318.             */
319.            if (argc != 4)
320.                    usage(argv[0]);
321.            if (safe_atoi(argv[1], &thrcnt) < 0 || thrcnt <= 0)
322.            {
323.                    fprintf(stderr, "%s' is not valid for thread_count'\n", argv[1]);
324.                    exit(1);
325.            }
326.
327.            if (safe_atoi(argv[2], &chldcnt) < 0 || chldcnt < 0 || chldcnt > thrcnt)
328.            {
329.                    fprintf(stderr, "%s' is not valid for child_threads'\n", argv[2]);
330.                    exit(1);
331.            }
332.            if (safe_atoi(argv[3], &ratio) < 0 || ratio < 1)
333.            {
334.                    fprintf(stderr, "%s' is not valid for c_t_ratio'\n", argv[3]);
335.                    exit(1);
336.            }
337.
338.            /*
339.             * Initialize kindergarten and random number generator
340.             */
341.            srand(time(NULL));
342.
343.            kg = safe_malloc(sizeof(*kg));
344.            kg->vt = kg->vc = 0;
345.            kg->ratio = ratio;
346.
347.            ret = pthread_mutex_init(&kg->mutex, NULL);
348.            if (ret)
349.            {
350.                    perror_pthread(ret, "pthread_mutex_init");
351.                    exit(1);
352.            }
353.
354.            ret = pthread_cond_init(&kg->cond, NULL);
355.            if (ret)
356.            {
357.                    perror_pthread(ret, "pthread_cond_init");
358.                    exit(1);
359.            }
360.            /*
361.             * Create threads
362.             */
363.            thr = safe_malloc(thrcnt * sizeof(*thr));
364.
365.            for (i = 0; i < thrcnt; i++)
366.            {
367.                    /* Initialize per-thread structure */
368.                    thr[i].kg = kg;
369.                    thr[i].thrid = i;
```

```
370.                    thr[i].thrcnt = thrcnt;
371.                    thr[i].is_child = (i < chldcnt);
372.                    thr[i].rseed = rand();
373.
374.                    /* Spawn new thread */
375.                    ret = pthread_create(&thr[i].tid, NULL, thread_start_fn, &thr[i]);
376.                    if (ret)
377.                    {
378.                              perror_pthread(ret, "pthread_create");
379.                              exit(1);
380.                    }
381.           }
382.
383.           /*
384.            * Wait for all threads to terminate
385.            */
386.           for (i = 0; i < thrcnt; i++)
387.           {
388.                    ret = pthread_join(thr[i].tid, NULL);
389.                    if (ret)
390.                    {
391.                              perror_pthread(ret, "pthread_join");
392.                              exit(1);
393.                    }
394.           }
395.
396.           printf("OK.\n");
397.
398.           return 0;
399.  }
```