**ΛΕΙΤΟΥΡΓΙΚΑ ΣΥΣΤΗΜΑΤΑ**

ΗΛΙΑΣ ΝΤΟΝΤΟΡΟΣ

ΘΟΔΩΡΗΣ- ΙΩΑΝΝΗΣ ΚΙΤΣΗΣ

ΑΝΑΦΟΡΑ 3ης ΣΕΙΡΑΣ ΑΣΚΗΣΕΩΝ

ΑΣΚΗΣΗ 1Η

* + 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

* + 1. 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.

* + 1. 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

* + 1. 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. \*/
11. #include <errno.h>
12. #include <stdio.h>
13. #include <stdlib.h>
14. #include <unistd.h>
15. #include <pthread.h>
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)
25. #define N 10000000
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
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;
40. void \*increase\_fn(void \*arg)
41. {
42. int i;
43. volatile int \*ip = arg;
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");
61. return NULL;
62. }
64. void \*decrease\_fn(void \*arg)
65. {
66. int i;
67. volatile int \*ip = arg;
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);
80. --(\*ip);
81. pthread\_mutex\_unlock(&mutex1);
82. }
83. }
84. fprintf(stderr, "Done decreasing variable.\n");
86. return NULL;
87. }
89. int main(int argc, char \*argv[])
90. {
91. int val, ret, ok;
92. pthread\_t t1, t2;
94. val = 0;
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. }
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");
122. /\*
123. \* Is everything OK?
124. \*/
125. ok = (val == 0);
127. printf("%sOK, val = %d.\n", ok ? "" : "NOT ", val);
129. return ok;
130. }

ΑΣΚΗΣΗ 2Η

* + 1. Ο αριθμός των σημαφόρων που χρειάζεται είναι όσα είναι τα threads που χρησιμοποιούμε.

* + 1. 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” .
    2. 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.

* + 1. 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. \*/
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"
17. #define MANDEL\_MAX\_ITERATION 100000
19. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*
20. \* Compile-time parameters \*
21. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
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;
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;
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;
44. struct thread\_stats\_struct
45. {
46. pthread\_t tid;
47. int \*color\_val;
48. int thrid;
49. int thrcnt;
50. };
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. }
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. }
76. return p;
77. }
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;
90. int n;
91. int val;
93. /\* Find out the y value corresponding to this line \*/
94. y = ymax - ystep \* line;
96. /\* and iterate for all points on this line \*/
97. for (x = xmin, n = 0; n < x\_chars; x += xstep, n++)
98. {
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;
105. /\* And store it in the color\_val[] array \*/
106. val = xterm\_color(val);
107. color\_val[n] = val;
108. }
109. }
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;
119. char point = '$';
120. char newline = '\n';
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. }
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. }
141. void \*compute\_and\_output\_mandel\_line(void \*arg)
142. {
143. struct thread\_stats\_struct \*thread = arg;
144. int i;
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. }
154. return 0;
155. }
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 required:\n"
163. " thread\_count: The number of threads to create.\n",
164. argv[0]);
165. }
167. int line;
168. int i, ret, thrcnt;
169. struct thread\_stats\_struct \*thread;
171. xstep = (xmax - xmin) / x\_chars;
172. ystep = (ymax - ymin) / y\_chars;
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. }
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\_mandel\_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. }

ΑΚΣΗΣΗ 3Η

* + 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.
    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. \*/
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>
27. /\*
28. \* POSIX thread functions do not return error numbers in errno,
29. \* but in the actual return value of the function call instead.
30. \* This macro helps with error reporting in this case.
31. \*/
32. #define perror\_pthread(ret, msg) \
33. do \
34. { \
35. errno = ret; \
36. perror(msg); \
37. } while (0)
39. /\* A virtual kindergarten \*/
40. struct kgarten\_struct
41. {
43. /\*
44. \* Here you may define any mutexes / condition variables / other variables
45. \* you may need.
46. \*/
48. pthread\_cond\_t cond;
50. /\*
51. \* You may NOT modify anything in the structure below this
52. \* point.
53. \*/
55. int vt;
56. int vc;
57. int ratio;
59. pthread\_mutex\_t mutex;
60. };
62. /\*
63. \* A (distinct) instance of this structure
64. \* is passed to each thread
65. \*/
66. struct thread\_info\_struct
67. {
68. pthread\_t tid; /\* POSIX thread id, as returned by the library \*/
70. struct kgarten\_struct \*kg;
71. int is\_child; /\* Nonzero if this thread simulates children, zero otherwise \*/
73. int thrid; /\* Application-defined thread id \*/
74. int thrcnt;
75. unsigned int rseed;
76. };
78. int safe\_atoi(char \*s, int \*val)
79. {
80. long l;
81. char \*endp;
83. l = strtol(s, &endp, 10);
84. if (s != endp && \*endp == '\0')
85. {
86. \*val = l;
87. return 0;
88. }
89. else
90. return -1;
91. }
93. void \*safe\_malloc(size\_t size)
94. {
95. void \*p;
97. if ((p = malloc(size)) == NULL)
98. {
99. fprintf(stderr, "Out of memory, failed to allocate %zd bytes\n",
100. size);
101. exit(1);
102. }
104. return p;
105. }
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. }
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];
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"};
141. thing = rand() % 4;
142. sex = rand() % 2;
144. namecnt = sex ? sizeof(boys) / sizeof(boys[0]) : sizeof(girls) / sizeof(girls[0]);
145. nameidx = rand() % namecnt;
146. name = sex ? boys[nameidx] : girls[nameidx];
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);
154. /\* Output everything in a single atomic call \*/
155. printf("%s", buf);
156. }
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;
166. c = kg->vc;
167. t = kg->vt;
168. r = kg->ratio;
170. fprintf(stderr, "Thread %d: Teachers: %d, Children: %d\n",
171. thr->thrid, t, c);
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. }
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. {
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);
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. }
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. }
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. }
239. pthread\_mutex\_lock(&thr->kg->mutex);
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);
248. --(thr->kg->vt);
249. pthread\_mutex\_unlock(&thr->kg->mutex);
250. }
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;
262. fprintf(stderr, "Thread %d of %d. START.\n", thr->thrid, thr->thrcnt);
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);
273. fprintf(stderr, "Thread %d [%s]: Entered.\n", thr->thrid, nstr);
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);
284. usleep(rand\_r(&thr->rseed) % 1000000);
286. fprintf(stderr, "Thread %d [%s]: Exiting.\n", thr->thrid, nstr);
287. /\* CRITICAL SECTION END \*/
289. if (thr->is\_child)
290. child\_exit(thr);
291. else
292. teacher\_exit(thr);
294. fprintf(stderr, "Thread %d [%s]: Exited.\n", thr->thrid, nstr);
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);
300. pthread\_mutex\_lock(&thr->kg->mutex);
301. verify(thr);
302. pthread\_mutex\_unlock(&thr->kg->mutex);
303. }
305. fprintf(stderr, "Thread %d of %d. END.\n", thr->thrid, thr->thrcnt);
307. return NULL;
308. }
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;
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. }
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. }
338. /\*
339. \* Initialize kindergarten and random number generator
340. \*/
341. srand(time(NULL));
343. kg = safe\_malloc(sizeof(\*kg));
344. kg->vt = kg->vc = 0;
345. kg->ratio = ratio;
347. ret = pthread\_mutex\_init(&kg->mutex, NULL);
348. if (ret)
349. {
350. perror\_pthread(ret, "pthread\_mutex\_init");
351. exit(1);
352. }
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));
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();
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. }
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. }
396. printf("OK.\n");
398. return 0;
399. }