```c
#include <pthread.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define RECURSIVE

/**
 * @note 3.1 a)
 * Gemaess der manpage von pthread_mutex_lock gilt: "If the mutex type is PTHREAD_MUTEX_DEFAULT,
 * attempting to recursively lock the mutex results in undefined behavior."
 * Undefiniertes Verhalten kann hier ein Programmabsturz oder Entstehen eines Zombie-Threads sein.
 *
 * 3.1 b) #define RECURSIVE added
 *
 * 3.1 c) Fuer 1 Thread sucht das gegebene Programm in einem Thread nach dem key 0, den es nirgends
 * gibt. tree_lock wird intern einen Zaehler verwenden, der fuer jedes lock/unlock
 * inkrementiert/dekrementiert wird. Ist der Zaehler 0, darf wieder ein anderer Thread locken.
 * Ich habe eine Ausgabe generieren lassen, wo die Einrueckung den aktuellen Zaehlerstand
 * visualisiert:

  Examining node with key = 3 and value = 1
  Examining left child
    Examining node with key = 2 and value = 2
    Exiting search_tree
  Examining right child
    Examining node with key = 1 and value = 3
    Exiting search_tree
  Exiting search_tree

 * 3.1 d) Wenn jeder Thread beim Schreiben/Lesen den gesamten Baum sperrt, wird der Baum praktisch
 * sequentiell abgearbeitet. Das waere aber nicht notwendig, wenn
 * bspw. n Threads nur lesend zugreifen wollen und kein Schreibvorgang stattfinden soll.
 * Vernachlaessigbar waere das, wenn der Baum nur wenige Elemente hat und gut balanciert ist oder
 * wenn es nur wenige Threads gibt, die auf ihn zugreifen muessen.
 */

typedef struct node_ {
    int key;
    int value;
    struct node_ *left, *right;
} node_t;

pthread_mutex_t tree_lock;

typedef struct {
    int me;
    int key;
    node_t *node;
} thread_arg_t;

int lock_count = 0;

void pad(const char *format, ...) {
    for (int i = 0; i < lock_count; ++i)
        printf("  ");

    va_list args;
    va_start(args, format);
    vprintf(format, args);
    va_end(args);
}

void lock() {
    pthread_mutex_lock(&tree_lock);
    ++lock_count;
}

void unlock() {
    pthread_mutex_unlock(&tree_lock);
    --lock_count;
}

int search_tree(node_t *node, int key) {
    lock();
    pad("Examining node with key = %d and value = %d\n", node->key, node->value);

    if (node->key == key) {
        /* solution is found here */
        printf("key = %d, value = %d\n", key, node->value);
        unlock();
        return 1;
```

```
84:     }
85:
86:     if (node->left != NULL) {
87:         pad("Examining left child\n");
88:         if (search_tree(node->left, key)) {
89:             unlock();
90:             return 1;
91:         }
92:     }
93:
94:     if (node->right != NULL) {
95:         pad("Examining right child\n");
96:         if (search_tree(node->right, key)) {
97:             unlock();
98:             return 1;
99:         }
100:    }
101:
102:    pad("Exiting search_tree\n");
103:
104:    unlock();
105:    return 0;
106: }
107:
108: void *search_thread_func(void *arg) {
109:    thread_arg_t *targ = (thread_arg_t *)arg;
110:
111:    if (!search_tree(targ->node, targ->key))
112:        printf("Thread %d: Search unsuccessfull\n", targ->me);
113:
114:    return NULL;
115: }
116:
117: int main(int argc, char *argv[]) {
118:    node_t A, B, C;
119:    pthread_mutexattr_t attr;
120:    char *omp_num_threads_as_char;
121:    int n_threads = 1;
122:    pthread_t *thread;
123:    thread_arg_t *arg;
124:    int i;
125:
126:    omp_num_threads_as_char = getenv("OMP_NUM_THREADS");
127:
128:    if (omp_num_threads_as_char != NULL)
129:        if (strlen(omp_num_threads_as_char) > 0)
130:            n_threads = atoi(omp_num_threads_as_char);
131:
132:    thread = (pthread_t *)malloc(n_threads * sizeof(pthread_t));
133:    arg = (thread_arg_t *)malloc(n_threads * sizeof(thread_arg_t));
134:
135:    pthread_mutexattr_init(&attr);
136:
137: #ifdef RECURSIVE
138:    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
139: #endif
140:
141:    pthread_mutex_init(&tree_lock, &attr);
142:
143:    A.key = 3;
144:    A.value = 1;
145:    B.key = 2;
146:    B.value = 2;
147:    C.key = 1;
148:    C.value = 3;
149:
150:    A.left = &B;
151:    A.right = &C;
152:    B.left = NULL;
153:    B.right = NULL;
154:    C.left = NULL;
155:    C.right = NULL;
156:
157:    printf("Starting %d threads...\n", n_threads);
158:
159:    for (i = 0; i < n_threads; i++) {
160:        arg[i].me = i;
161:        arg[i].key = i;
162:        arg[i].node = &A;
163:        pthread_create(&thread[i], NULL, search_thread_func, &arg[i]);
164:    }
165:
166:    printf("Joining threads...\n");
```

```
167:
168:     for (i = 0; i < n_threads; i++)
169:         pthread_join(thread[i], NULL);
170:
171:     pthread_mutex_destroy(&tree_lock);
172:     pthread_mutexattr_destroy(&attr);
173:
174:     free(thread);
175:     free(arg);
176:
177:     return EXIT_SUCCESS;
178: }
```

```c
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

/**
 * 3.2 Logs:
Using the version from the lecture
Started read thread with ID 1
Thread with ID 1 acquired read access
Started read thread with ID 2
Thread with ID 2 acquired read access
Thread with ID 2 acquired read access
Thread with ID 1 acquired read access
Thread with ID 2 acquired read access
Thread with ID 1 acquired read access
Thread with ID 2 acquired read access
Thread with ID 1 acquired read access
Thread with ID 2 acquired read access
Thread with ID 1 acquired read access
Write thread acquired write access

Wie zu erwarten kommt der Schreibthread hier erst zum Ende dran

Using the modified version
Started read thread with ID 1
Thread with ID 1 acquired read access
Started read thread with ID 2
Thread with ID 2 acquired read access
Thread with ID 2 acquired read access
Thread with ID 1 acquired read access
Thread with ID 2 acquired read access
Thread with ID 1 acquired read access
Thread with ID 1 acquired read access
Thread with ID 1 acquired read access
Write thread acquired write access
Thread with ID 2 acquired read access
Thread with ID 2 acquired read access

Da in der modifizierten Variante die Lesethreads haeufiger warten, kommt hier der Schreibthread
etwas frueher dran.
 */

typedef struct {
    bool modified;
    bool provocative;
} read_options_t;

typedef struct _rw_lock_t {
    pthread_mutex_t m;
    pthread_cond_t c;
    int num_r, num_w;
    int num_wr; // write requests
} rw_lock_t;

rw_lock_t lock;

int rw_lock_init(rw_lock_t *rwl) {
    rwl->num_r = 0;
    rwl->num_w = 0;
    rwl->num_wr = 0;
    pthread_mutex_init(&rwl->m, NULL);
    pthread_cond_init(&rwl->c, NULL);
    return 0;
}

int rw_lock_rlock(rw_lock_t *rwl) {
    pthread_mutex_lock(&rwl->m);

    while (rwl->num_w > 0)
        pthread_cond_wait(&rwl->c, &rwl->m);

    rwl->num_r++;

    pthread_mutex_unlock(&rwl->m);
    return 0;
}

int rw_lock_rlock_modified(rw_lock_t *rwl) {
    pthread_mutex_lock(&rwl->m);

    while (rwl->num_w > 0 || (rwl->num_wr > 0 && rwl->num_r > 0))
```

```
84:         pthread_cond_wait(&rwl->c, &rwl->m);
85:
86:     rwl->num_r++;
87:
88:     pthread_mutex_unlock(&rwl->m);
89:     return 0;
90: }
91:
92: int rw_lock_wlock(rw_lock_t *rwl) {
93:     pthread_mutex_lock(&rwl->m);
94:
95:     rwl->num_wr++;
96:     while (rwl->num_w > 0 || rwl->num_r > 0)
97:         pthread_cond_wait(&rwl->c, &rwl->m);
98:     rwl->num_wr--;
99:
100:    rwl->num_w = 1;
101:
102:    pthread_mutex_unlock(&rwl->m);
103:    return 0;
104: }
105:
106: int rw_lock_runlock(rw_lock_t *rwl) {
107:    pthread_mutex_lock(&rwl->m);
108:    rwl->num_r--;
109:
110:    if (rwl->num_r == 0)
111:        pthread_cond_signal(&rwl->c);
112:
113:    pthread_mutex_unlock(&rwl->m);
114:    return 0;
115: }
116:
117: int rw_lock_wunlock(rw_lock_t *rwl) {
118:    pthread_mutex_lock(&rwl->m);
119:
120:    rwl->num_w = 0;
121:    pthread_cond_broadcast(&rwl->c);
122:
123:    pthread_mutex_unlock(&rwl->m);
124:    return 0;
125: }
126:
127: void *read_thread(void *arg) {
128:    pthread_t threadID = pthread_self();
129:    read_options_t read_options = *(read_options_t *)(arg);
130:    printf("Started read thread with ID %lu\n", (unsigned long)(threadID));
131:
132:    for (int i = 0; i < 5; ++i) {
133:        if (read_options.modified)
134:            rw_lock_rlock_modified(&lock);
135:        else
136:            rw_lock_rlock(&lock);
137:
138:        printf("Thread with ID %lu acquired read access\n", (unsigned long)threadID);
139:
140:        if (read_options.provocative && i == 0)
141:            sleep(1);
142:        else
143:            sleep(2);
144:
145:        rw_lock_runlock(&lock);
146:    }
147:
148:    return NULL;
149: }
150:
151: void *write_thread(void *) {
152:    sleep(5);
153:    rw_lock_wlock(&lock);
154:    printf("Write thread acquired write access\n");
155:    rw_lock_wunlock(&lock);
156:
157:    return NULL;
158: }
159:
160: int main(int argc, char **argv) {
161:    rw_lock_init(&lock);
162:    bool modified = false;
163:
164:    if (argc == 2 && strcmp(argv[1], "-modified") == 0)
165:        modified = true;
166:
```

```
167:     if (modified)
168:         printf("Using the modified version\n");
169:     else
170:         printf("Using the version from the lecture\n");
171: 
172:     pthread_t threads[3];
173:     read_options_t read_options[2];
174:     for (int i = 0; i < 2; ++i) {
175:         read_options[i].modified = modified;
176:         read_options[i].provocative = (i == 1);
177:     }
178: 
179:     for (int i = 0; i < 3; ++i) {
180:         int result;
181: 
182:         if (i == 0)
183:             result = pthread_create(&threads[i], NULL, read_thread, &read_options[i]);
184:         else if (i == 1)
185:             result = pthread_create(&threads[i], NULL, read_thread, &read_options[i]);
186:         else
187:             result = pthread_create(&threads[i], NULL, write_thread, NULL);
188: 
189:         if (result != 0) {
190:             fprintf(stderr, "Error creating thread\n");
191:             exit(1);
192:         }
193:     }
194: 
195:     for (int i = 0; i < 3; ++i) {
196:         int result = pthread_join(threads[i], NULL);
197: 
198:         if (result != 0) {
199:             fprintf(stderr, "Error joining thread\n");
200:             exit(1);
201:         }
202:     }
203: }
```

```c
#include <float.h>
#include <pthread.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

/**
 * 3.3 a)
 * Moeglich: Wenn die Ergebnisse der Iterationen nicht voneinander abhaengen und die
 * Ausfuehrungszeit nicht durch I/O begrenzt ist sondern durch die CPU
 *
 * Lohnenswert: Wenn fuer ein
 * langsames Programm der Profiler zeigt, dass die Schleife einen wesentlichen Anteil an der
 * Ausfuehrungszeit hat
 *
 * Bezogen auf die Schleifen im gegebenen Programm:
 *
     // Setting the initial values
     for (int i = 1; i < grid_size; i++)
         // ...
    Nicht parallelisierbar, da das Ergbnis der vorherigen Iteration benoetigt wird (T_k[i] = ...
 T_k[i-1] ...).

    // Time loop
    for (int k = 0; k < num_time_steps; k++)
    {
        // ...
    }
    Nicht parallelisierbar, da die Berechnung von T_kn auf T_k basiert (was dem T_kn der letzten
 Iteration entspricht).

    // System loop
    for (int i = 0; i < grid_size; i++)
    {
        // ...
    }
    Parallelisierbar, da T_k sich in der Schleife nicht aendert.
    Die Parallelisierung ist auch lohnenswert, da diese Schleife num_time_steps-Mal verwendet wird.

    // Computing statistics of the final temperature of the grid
    for (int i = 0; i < grid_size; i++)
    {
        // ...
    }
    Parallelsierbar, wenn das setzen von (z.B. globalen) T_max und T_min z.B. durch Mutexe
 geschuetzt wird. Aber nicht lohnenswert, da die Schleife nur einmal verwendet wird.

    for (int i = 0; i < grid_size; i++)
        // ...
    Parallelsierbar, wenn das setzen von T_average z.B. durch einen Mutex geschuetzt wird.
    Aber nicht lohnenswert, da die Schleife nur einmal verwendet wird.
 *
 * 3.3 c)
 * Speedups siehe 3_3_speedups.png. Er ist fast vernachlaessigbar: Das Programm kann sich nicht
 vollstaendig parallelisieren lassen (s. a)). Dazu kommt der Overhead fuer die Threadkommunikation
 (v. a. barrier_wait). AuÃ\237erdem laesst sich nicht beeinflussen, ob die Threads tatsaechlich auch auf
 mehreren Prozessoren vom Betriebssystem verteilt werden.
 *
 * 3.4 a)
 * Einzelne Bloecke im Cache haben oft eine GroeÃ\237e von etwa 64 Bytes und werden cache
 * lines genannt. Benachbarte Elemente eines Arrays, die von mehreren Threads genutzt werden,
 koennen
 * in derselben cache line liegen. Jeder Prozessor hat seine eigene, lokale Kopie des Caches.
 * Schreibt bspw. Thread 1 in die cache line auf seinem Prozessor, muss Thread 2 seine lokale Kopie
 * neu laden (um cache coherence zu erhalten), obwohl das gar nicht noetig waere. Das wird false
 sharing genannt.
 *
 * 3.4 b) Mithilfe von posix_memalign kann ermittelt werden, wie groÃ\237 eine cache line ist. Dann muss
 * jeder Thread seine Mutexes im Array mit leerem Speicher mindestens der GroeÃ\237e einer cache line
 * padden, bspw. indem zusaetzliche, ungenutzte mutexes in das Array gelegt werden. Dann liegt jeder
 * mutex auf seiner eigenen cache line, was das Neuladen verhindert.
 *
 * 3.4 c) Das Lock ist der Teil vom Mutex, der regelmaeÃ\237ig beschrieben werden muss und false sharing
 * verursachen kann. Liegt das Lock auf dem Heap, sind die zu schreibenden Daten haeufiger weit
 * auseinander (es sei denn malloc legt sie zufaellig direkt nebeneinander, was unwahrscheinlich
 * ist). Das wuerde false sharing verhindern. Liegt das Lock im Mutex, wird false sharing
 * wahrscheinlicher.
 */

pthread_barrier_t barrier;

```

```c
84: typedef struct {
85:     double *T_k;
86:     double *T_kn;
87:     double conductivity_constant;
88:     int start_index;
89:     int end_index;
90:     int grid_size;
91:     int num_time_steps;
92:     double delta_t;
93: } grid_t;
94:
95: int min(int a, int b) {
96:     if (a < b)
97:         return a;
98:
99:     return b;
100: }
101:
102: void calculate_segments(int N, int segment_count, int *segments) {
103:     int segment_width = N / segment_count;
104:     int remainder = N % segment_count;
105:
106:     for (int i = 0; i < segment_count; ++i)
107:         segments[i] = i * segment_width + min(i, remainder);
108: }
109:
110: void print_elapsed_time(struct timespec start, struct timespec end) {
111:     const double time_ns = (end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
112:     const char *time_units[] = {"ns", "us", "ms", "s"};
113:
114:     int i = 0;
115:     double converted_time = time_ns;
116:     while (converted_time > 1e3 && i < (sizeof(time_units) / sizeof(time_units[0])) - 1) {
117:         converted_time /= 1e3;
118:         ++i;
119:     }
120:
121:     printf("Elapsed time: %lf%s\n", converted_time, time_units[i]);
122: }
123:
124: void *calc_next_step(void *arg) {
125:     grid_t grid = *(grid_t *)arg;
126:
127:     for (int j = 0; j < grid.num_time_steps; ++j) {
128:         for (int i = grid.start_index; i < grid.end_index; i++) {
129:             double dTdt_i =
130:                 grid.conductivity_constant * (-2 * grid.T_k[i] + grid.T_k[i != 0 ? i - 1 : 1] +
131:                                               grid.T_k[i != grid.grid_size - 1 ? i + 1 : i - 1]);
132:             grid.T_kn[i] = grid.T_k[i] + grid.delta_t * dTdt_i;
133:         }
134:
135:         double *temp = grid.T_kn;
136:         grid.T_kn = grid.T_k;
137:         grid.T_k = temp;
138:
139:         pthread_barrier_wait(&barrier);
140:     }
141:
142:     return NULL;
143: }
144:
145: void init_grid(grid_t *grid) {
146:     grid->grid_size = (30 * 1024 * 1024);
147:     grid->delta_t = 0.02;
148:     grid->conductivity_constant = 0.1;
149:     grid->num_time_steps = 3000;
150:
151:     // Current temperature
152:     grid->T_k = malloc(sizeof(double) * grid->grid_size);
153:     // Next temperature
154:     grid->T_kn = malloc(sizeof(double) * grid->grid_size);
155:
156:     // Setting the initial values
157:     grid->T_k[0] = 1. / 2.;
158:     for (int i = 1; i < grid->grid_size; i++)
159:         grid->T_k[i] = 3.59 * grid->T_k[i - 1] * (1 - grid->T_k[i - 1]);
160: }
161:
162: grid_t *create_thread_grids(grid_t base_grid, int number_of_threads) {
163:     int *segments = (int *)malloc(sizeof(int) * number_of_threads);
164:     calculate_segments(base_grid.grid_size, number_of_threads, segments);
165:
166:     grid_t *grids = (grid_t *)malloc(number_of_threads * sizeof(grid_t));
```

```
167:
168:     for (int i = 0; i < number_of_threads; ++i) {
169:         grids[i] = base_grid;
170:         grids[i].start_index = segments[i];
171:         grids[i].end_index = (i == number_of_threads - 1) ? base_grid.grid_size : segments[i + 1];
172:     }
173:
174:     free(segments);
175:
176:     return grids;
177: }
178:
179: grid_t run_simulation(int number_of_threads) {
180:     grid_t base_grid;
181:     init_grid(&base_grid);
182:     grid_t *grids = create_thread_grids(base_grid, number_of_threads);
183:
184:     pthread_t *threads = (pthread_t *)malloc(number_of_threads * sizeof(pthread_t));
185:
186:     struct timespec start, end;
187:     clock_gettime(CLOCK_MONOTONIC, &start);
188:
189:     for (int i = 0; i < number_of_threads; ++i)
190:         pthread_create(&threads[i], NULL, calc_next_step, &grids[i]);
191:
192:     for (int i = 0; i < number_of_threads; ++i)
193:         pthread_join(threads[i], NULL);
194:
195:     clock_gettime(CLOCK_MONOTONIC, &end);
196:
197:     print_elapsed_time(start, end);
198:
199:     free(threads);
200:     free(grids);
201:
202:     return base_grid;
203: }
204:
205: int main(int argc, char **argv) {
206:     if (argc != 2) {
207:         printf("Usage: %s <number of threads>\n", argv[0]);
208:         return 0;
209:     }
210:
211:     int number_of_threads = atoi(argv[1]);
212:     pthread_barrier_init(&barrier, NULL, number_of_threads);
213:
214:     grid_t base_grid = run_simulation(number_of_threads);
215:
216:     // Computing statistics of the final temperature of the grid
217:
218:     double T_max = DBL_MIN;
219:     double T_min = DBL_MAX;
220:     double T_average = 0;
221:
222:     for (int i = 0; i < base_grid.grid_size; i++) {
223:         T_max = T_max > base_grid.T_k[i] ? T_max : base_grid.T_k[i];
224:         T_min = T_min < base_grid.T_k[i] ? T_min : base_grid.T_k[i];
225:     }
226:
227:     for (int i = 0; i < base_grid.grid_size; i++)
228:         T_average += base_grid.T_k[i];
229:
230:     T_average = T_average / base_grid.grid_size;
231:
232:     printf("T_max: %f, T_min: %f, T_average: %f", T_max, T_min, T_average);
233:
234:     free(base_grid.T_k);
235:     free(base_grid.T_kn);
236:
237:     return 0;
238: }
```