 Moritz Lunk, bt712482 | Yannik Schroeder, bt722248 | Justin Stoll, bt709742

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

/**
 * Output:
    Mutex: sum = 1048576
    Elapsed time: 9.607800ms
    Partial sums: sum = 1048576
    Elapsed time: 362.100000us
 * The mutex version is somewhat (~10x) slower because we have a total of 2**20 calls to
 * pthread_mutex_lock and pthread_mutex_unlock each and each thread has to wait for the other
 * making the calculation essentially sequential
 */

int sum = 0;
const int N = 1048576;
int *array;
int sum1 = 0, sum2 = 0;
pthread_mutex_t mutex;

void *thread1_mutex(void *arg) {
    for (int i = 0; i < N / 2; ++i) {
        pthread_mutex_lock(&mutex);
        sum += array[i];
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

void *thread2_mutex(void *arg) {
    for (int i = N / 2; i < N; ++i) {
        pthread_mutex_lock(&mutex);
        sum += array[i];
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

void *thread1_partial(void *arg) {
    for (int i = 0; i < N / 2; ++i)
        sum1 += array[i];

    return NULL;
}

void *thread2_partial(void *arg) {
    for (int i = N / 2; i < N; ++i)
        sum2 += array[i];

    return NULL;
}

void print_elapsed_time(struct timespec start, struct timespec end) {
    const double time_ns = (end.tv_sec - start.tv_sec) * 1e9 + (end.tv_nsec - start.tv_nsec);
    const char *time_units[] = {"ns", "us", "ms", "s"};

    int i = 0;
    double converted_time = time_ns;
    while (converted_time > 1e3 && i < (sizeof(time_units) / sizeof(time_units[0])) - 1) {
        converted_time /= 1e3;
        ++i;
    }

    printf("Elapsed time: %lf%s\n", converted_time, time_units[i]);
}

int main() {
    array = (int *)malloc(sizeof(int) * N);
    for (int i = 0; i < N; ++i)
        array[i] = 1;

    pthread_t threads[4];

    struct timespec start, end;
    clock_gettime(CLOCK_MONOTONIC, &start);

    pthread_create(&threads[0], NULL, thread1_mutex, NULL);
    pthread_create(&threads[1], NULL, thread2_mutex, NULL);

```

```
    for (int i = 0; i < 2; ++i)
        pthread_join(threads[i], NULL);

    clock_gettime(CLOCK_MONOTONIC, &end);

    printf("Mutex: sum = %d\n", sum);
    print_elapsed_time(start, end);

    clock_gettime(CLOCK_MONOTONIC, &start);

    pthread_create(&threads[2], NULL, thread1_partial, NULL);
    pthread_create(&threads[3], NULL, thread2_partial, NULL);

    for (int i = 2; i < 4; ++i)
        pthread_join(threads[i], NULL);

    clock_gettime(CLOCK_MONOTONIC, &end);

    printf("Partial sums: sum = %d\n", sum1 + sum2);
    print_elapsed_time(start, end);

    free(array);
}
```

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

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

bool is_blocked = false;
int thread_count = 0;
int winner = 0;
int three_counts[] = {0, 0, 0};

void *thread(void *arg) {
    pthread_mutex_lock(&mutex);
    int ID = ++thread_count;
    pthread_mutex_unlock(&mutex);

    // seed the rng with a different seed for every thread
    unsigned int seed = time(NULL) + ID;
    srand(seed);

    int number;
    while (winner == 0) {
        number = rand() % 6 + 1;

        if (number == 3) {
            ++three_counts[ID - 1];

            pthread_mutex_lock(&mutex);
            if (three_counts[ID - 1] == 3 && winner == 0) {
                winner = ID;
                pthread_cond_signal(&cond);
                pthread_mutex_unlock(&mutex);
                break;
            }
            pthread_mutex_unlock(&mutex);
        } else
            three_counts[ID - 1] = 0;

        if (number != 1 && number != 6)
            continue;

        pthread_mutex_lock(&mutex);
        if (winner != 0) {
            pthread_mutex_unlock(&mutex);
            break;
        }

        if (number == 1 && !is_blocked) {
            is_blocked = true;
            pthread_cond_wait(&cond, &mutex);
            is_blocked = false;

        } else if ((number == 1 || number == 6) && is_blocked)
            pthread_cond_signal(&cond);

        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

int main() {
    srand(time(NULL));
    pthread_t threads[3];

    for (int i = 0; i < 3; ++i) {
        int result = pthread_create(&threads[i], NULL, thread, NULL);
        if (result != 0) {
            fprintf(stderr, "Error creating thread\n");
            exit(1);
        }
    }

    for (int i = 0; i < 3; ++i) {
        int result = pthread_join(threads[i], NULL);
        if (result != 0) {
            fprintf(stderr, "Error joinging thread\n");
            exit(1);
        }
    }
```

```
84:
85:     printf("Winner: Thread %d\n", winner);
86:
87:     return 0;
88: }
```

```c
/**
 * @note 2.3 a)
 * Vgl. Vorlesung Betriebssysteme, Coffmann (1971):
 * - Wechselseitiger Ausschluss (mutual exclusion): Jedes involvierte Betriebsmittel ist entweder
 * exklusiv belegt oder frei
 * - Zusaetzliche Belegung (Hold-and-wait): Die Prozesse haben bereits Betriebsmittel belegt, wollen
 * zusaetzliche Betriebsmittel belegen und warten darauf, dass sie frei werden.
 * - Keine vorzeitige Rueckgabe (No preemption): Bereits belegte Betriebsmittel koennen den
 * Prozessen nicht einfach wieder entzogen werden
 * - Gegenseitiges Warten (Circulat wait): Es existiert ein Zyklus von zwei oder mehr Prozessen, bei
 * denen jeweils einer die Betriebsmittel vom naechsten belegen will, die dieser belegt hat
 */

/*
    b)
    Das modifizierte Programm kommt in einem Deadlock,
    weil (angenommen der forward locker (FL) ist zuerst an der yield flag) der Programmablauf
   folgendermassen ist: FL lockt mutex[0] erreicht die yieldflag und wartet auf ein Signal BL lockt
   mutex[2] erreicht die yieldflag, weckt FL auf und wartet auf ein Signal FL lockt mutex[1]
   erreicht die yieldflag, weckt BL auf und wartet auf ein Signal BL versucht mutex[1] zu locken,
   der ist aber bereits von FL gesperrt, gibt also alle Mutexe frei, erreicht die yieldflag, weckt
   FL und wartet auf ein Signal FL lock mutex[2] erreicht die yieldflag, weckt BL auf und wartet auf
   ein Signal BL versucht mutex[2] zu locken, scheitert, und wartet darauf, dass mutex[2] frei wird
        => Beide Threads warten aufeinander
        => Deadlock

    Verwendet man statt pthread_mutex_lock pthread_mutex_unlock fuer mutex[0] in FL und mutex[2] in
   BL, dann laeuft das Programm bis entweder FL oder BL fertig sind. Der andere Thread bleibt dann
   haengen, weil er kein Signal mehr bekommen kann.
*/

#include <errno.h> // Fuer EBUSY
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> // Fuer sleep()

pthread_mutex_t mutex[3] = {PTHREAD_MUTEX_INITIALIZER, PTHREAD_MUTEX_INITIALIZER,
                            PTHREAD_MUTEX_INITIALIZER};
pthread_cond_t yield_cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t yield_mutex = PTHREAD_MUTEX_INITIALIZER;

int backoff = 1;     // == 1: mit Backoff-Strategie
int yield_flag = 0; // > 0: Verwende sched_yield, <= 0: sleep

void *lock_forward(void *arg);
void *lock_backward(void *arg);

int main(int argc, char *argv[]) {
    pthread_t f, b;

    if (argc > 1) {
        backoff = atoi(argv[1]);
    }
    if (argc > 2) {
        yield_flag = atoi(argv[2]);
    }

    pthread_create(&f, NULL, lock_forward, NULL);
    pthread_create(&b, NULL, lock_backward, NULL);

    pthread_exit(NULL); // Die beiden anderen Threads laufen weiter
}

void wait() {
    pthread_mutex_lock(&yield_mutex);
    pthread_cond_signal(&yield_cond);
    pthread_cond_wait(&yield_cond, &yield_mutex);
    pthread_mutex_unlock(&yield_mutex);
}

void *lock_forward(void *arg) {
    int iterate, i, status;

    for (iterate = 0; iterate < 10; iterate++) {
        for (i = 0; i < 3; i++) {
            if (i == 0 || !backoff) {
                status = pthread_mutex_lock(&mutex[i]);
            } else {
                status = pthread_mutex_trylock(&mutex[i]);
            }

```

```
84:             if (status == EBUSY) {
85:                 for (--i; i >= 0; i--) {
86:                     pthread_mutex_unlock(&mutex[i]);
87:                 }
88:             } else {
89:                 printf("forward locker got mutex %d\n", i);
90:             }
91:
92:             if (yield_flag) {
93:                 if (yield_flag > 0) {
94:                     wait();
95:                 } else {
96:                     sleep(1);
97:                 }
98:             }
99:         }
100:
101:        for (i = 2; i >= 0; i--) {
102:            pthread_mutex_unlock(&mutex[i]);
103:        }
104:    }
105:
106:    return NULL;
107: }
108:
109: void *lock_backward(void *arg) {
110:    int iterate, i, status;
111:
112:    for (iterate = 0; iterate < 10; iterate++) {
113:        for (i = 2; i >= 0; i--) {
114:            if (i == 2 || !backoff) {
115:                status = pthread_mutex_lock(&mutex[i]);
116:            } else {
117:                status = pthread_mutex_trylock(&mutex[i]);
118:            }
119:
120:            if (status == EBUSY) {
121:                for (++i; i < 3; i++) {
122:                    pthread_mutex_unlock(&mutex[i]);
123:                }
124:            } else {
125:                printf("backward locker got mutex %d\n", i);
126:            }
127:
128:            if (yield_flag) {
129:                if (yield_flag > 0) {
130:                    wait();
131:                } else {
132:                    sleep(1);
133:                }
134:            }
135:        }
136:
137:        for (i = 0; i < 3; i++) {
138:            pthread_mutex_unlock(&mutex[i]);
139:        }
140:    }
141:
142:    return NULL;
143: }
```