Operating Systems 2016/17
Assignment 7

Prof. Dr. Frank Bellosa
Dipl.-Inform. Marc Rittinghaus

Submission Deadline: Monday, December 12th, 2017 – 23:59

A new assignment will be published every week, right after the last one was due. It must be completed before its submission deadline.

The assignments must be filled out online in ILIAS. Handwritten solutions are no longer accepted. You will find the online version for each assignment in your tutorial's directory. **P-Questions** are programming assignments. Download the provided template from ILIAS. Do not fiddle with the compiler flags. Submission instructions can be found on the first assignment.

In this assignment you will get familiar with synchronization.

T-Question 7.1: Synchronization

a. What are the three requirements for a valid solution of the critical-section problem? Give a short explanation for each. **2 T-pt**

b. Can spinlocks be implemented entirely in user-mode? Explain your answer. **1 T-pt**

c. Using a CPU register for a spinlock's lock variable would be much faster than the implementation with a variable in memory. Why would such a spinlock not work? **1 T-pt**

d. What is the idea behind Linux's futexes? **1 T-pt**

e. The `CRITICAL_SECTION` synchronization object in Windows works similarly to futexes in Linux. However, the documentation states that on single-processor systems, the spinlock is ignored. Why did the Microsoft developers choose this design? **1 T-pt**
http://msdn.microsoft.com/en-us/library/windows/desktop/ms682530%28v=vs.85%29.aspx

T-Question 7.2: Ring Buffer

Consider the following solution to synchronize the access to a shared ring buffer with *multiple* producers and a *single* consumer thread.

```
1  #define BUFFER_SIZE 10
2  int ringbuffer[BUFFER_SIZE]; // Buffer with 10 elements
3  int index_fill  = 0;         // Index to next filled buffer element
4  int index_empty = 0;         // Index to next empty buffer element
5
6  sem_t fill, empty;           // Semaphores to synchronize access
7
8  void initialize() {
9      // Initialize semaphores to all elements free
10     sem_init(&fill, 0, 0);            // Initialize to 0
11     sem_init(&empty, 0, BUFFER_SIZE); // Initialize to buffer size
12 }
```

```
13 void* producer_thread_main(void* arg) {
14     while (1) {
15         int item = produce();
16
17         // Wait for empty slot and
18         // "reserve" it atomically
19         sem_wait(&empty);
20
21         ringbuffer[index_empty] = item;
22         index_empty = (index_empty + 1)
23             % BUFFER_SIZE;
24
25         // Signal consumer thread
26         // that an item is ready
27         sem_post(&fill);
28     }
29 }
```

```
30 void* consumer_thread_main(void* arg) {
31     while (1) {
32         // Wait for an item in the buffer
33         // and claim it
34         sem_wait(&fill);
35
36         int item = ringbuffer[index_fill];
37         index_fill = (index_fill + 1)
38             % BUFFER_SIZE;
39
40         // Signal producer threads that
41         // an buffer slot is empty again
42         sem_post(&empty);
43
44         consume(item);
45     }
46 }
```

a. Give an execution sequence that causes an error. **2 T-pt**

b. What general code changes are necessary to prevent the error? You do not need to provide the actual code, but give line numbers to specify where changes are necessary. **2 T-pt**

P-Question 7.1: Barrier

Download the template **p1** for this assignment from ILIAS. You may only modify and upload the file `barrier.c`.

A barrier synchronization allows multiple threads to wait until all threads have reached a particular point of execution before any thread continues. Such a synchronization mechanism is useful in phased computations, in which threads executing the same code in parallel must all complete one phase before moving on to the next one.

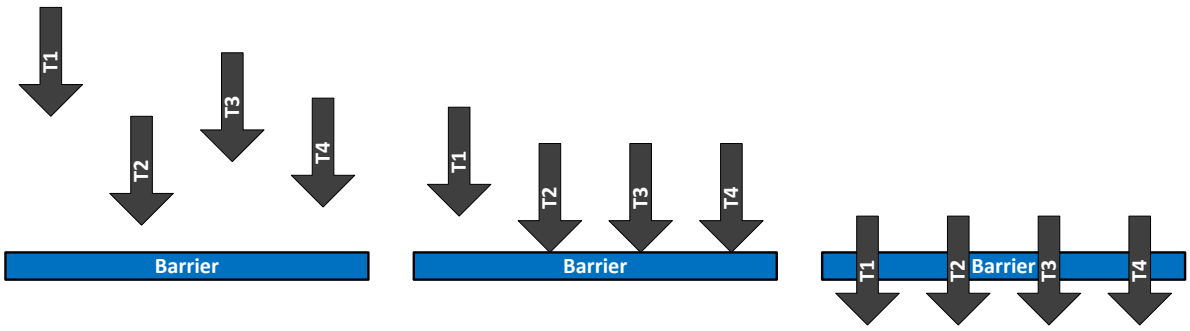

Abbildung 1: Barrier Synchronization

In this question you will write your own barrier synchronization object.

a. Define necessary fields in the `_ThreadBarrier` structure for a barrier, which uses a pthread *condition variable* to synchronize threads and which waits for a specified number of threads to enter the barrier. The number of threads should be stored in the barrier object. **2 P-pt**

b. Implement the barrier allocation and free functions. **2 P-pt**

createBarrier() The function should allocate and initialize a new `_ThreadBarrier` structure on the heap and return it to the caller. The number of threads that the barrier should synchronize is supplied via the `threads` parameter. The method should return `NULL` on any failure and fail if the `threads` parameter does not contain a meaningful value for a barrier synchronization object!

deleteBarrier() The function should release a `_ThreadBarrier` object, previously allocated with `createBarrier()`.

```
ThreadBarrier *createBarrier(int threads);
void deleteBarrier(ThreadBarrier *barrier);
```

c. Implement the `enterBarrier()` function that performs the actual barrier synchronization. Your implementation should satisfy the following requirements: **2 P-pt**

- Uses the conditional variable in the supplied `_ThreadBarrier` to synchronize access to the barrier structure's fields and put threads to sleep.
- Wakes up all threads if the configured number of threads have tried to enter.
- Resets the barrier to be ready for re-use when appropriate.

```
void enterBarrier(ThreadBarrier *barrier);
```

P-Question 7.2: Ticket Spinlock

Download the template **p2** for this assignment from ILIAS. You may only modify and upload the file `tslock.c`.

Multiple threads are waiting on a spinlock. When using a regular spinlock, the thread that enters the spinlock is selected in-deterministically, based on the operating system's scheduling. The thread that is currently running and trying to acquire the lock gets it (as soon as it is released). This policy can lead to some threads waiting longer on the spinlock than others. For example, it may be the last thread that tried to enter the lock that gets it, although it waited for the shortest time.

A ticket-based spinlock brings fairness by assigning each thread that tries to enter the spinlock a ticket and granting access in ticket order. Consequently, no thread can cut in line.

In this question you will write your own ticket spinlock that uses simple integer values as tickets.

a. The template already defines the `_TicketSpinlock` structure for the ticket spinlock. Look at the comments and implement the `tslock_init()` function by setting appropriate start values for a given ticket spinlock. **1 P-pt**

```
void tslock_init(TicketSpinlock *tslock);
```

b. Implement the `tslock_lock()`/`tslock_unlock()` function pair that performs the actual lock acquisition, spinning and lock release. The functions should fulfill the following requirements: **3 P-pt**

- Assign a ticket to a thread trying to acquire a lock by atomically reading and incrementing the `ticketCounter` field. Use GCC inline assembler to express the atomic operation with the `lock xaddl <register>, <memory>` instruction. The `xaddl` instruction adds the value supplied in the register to the 32-bit integer variable at the given memory location and writes the *previous* value of the variable in the register. The `lock` prefix instructs the CPU to perform the operation atomically, that is, synchronized with other CPUs in the system. **Do not use GCC intrinsic functions for atomic operations!**
- Let the current thread spin until its ticket is set in the `currentTicket` field.
- Use `sched_yield()` to quickly release the processor during spinning.
- Increment `currentTicket` when appropriate to move to the next ticket.

Hints: Take a look at the solution of Question 3.2 (Assignment 3) on how to generally use GCC inline assembler. Note that you will have to modify the parameters for the inline assembler block to reflect the storage locations (register/memory) of variables, define output variables and give the compiler hints to what state (e.g., memory) your assembler code changes. The solution for this question will require only the single atomic instruction, no other assembler instructions are needed.

```
void tslock_lock(TicketSpinlock *tslock);
void tslock_unlock(TicketSpinlock *tslock);
```

Total:
10 T-pt
10 P-pt