Karlsruhe Institute of Technology

Operating Systems 2016/17
Solutions for Assignment 7

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

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**

Solution:

Mutual Exclusion *At most one thread can be in the critical section at any time*

Progress *No thread running outside of the critical section may block another thread from getting in*

- *If no thread is in the critical section, a thread trying to enter will eventually get in*
- *If no thread can enter the critical section → do not have have progress*

Bounded Waiting *Once a thread starts trying to enter the critical section, there is a bound on the number of times other threads get in*

- *You cannot make assumptions concerning relative speeds of threads*
- *Do not have bounded waiting if thread A waits to enter critical section while B repeatedly leaves and re-enters the critical section infinitely*

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

Solution:
Yes, spinlocks can be built entirely in user-mode. To implement a spinlock we only need a simple lock variable (e.g., an `int`) and an atomic `test-and-set` instruction provided by the hardware. As atomic instructions are not privileged they can be used in user-mode.

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**

Solution:
Registers are thread local as their contents is replaced on thread switches. However, the lock variable of a spinlock must be shared between threads and thus cannot be placed in a register.

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

Solution:
Futexes combine the advantages of (user-mode) spinlocks (no kernel entry necessary) and mutexes (no busy waiting). Before a thread blocks on the mutex and thus needs to enter the kernel, it first spins a certain time in user-mode, trying to acquire the spinlock. This way, the futex tries to avoid the costly blocking wait in the kernel.

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

Solution:
The idea behined futexes is that while a thread is still waiting on the spinlock, the thread holding the futex makes progress (i.e., runs on a different CPU) and thus may leave the critical section before the waiting thread performs a blocking wait.

The single-processor system, however, provides no hardware parallelism and only one of the threads (the one holding the lock or the one spinning) may run at a time. The spinning thread will therefore always run into the blocking wait, wasting all CPU cycles during the spinning phase.

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.

```c
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 }
```

```c
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 }
```

```c
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**

Solution:
As we have multiple producer threads, we potentially have multiple threads that are concurrently add a new item to the buffer:

(a) Thread A and B execute `produce()` *(line 15).*

(b) Thread A and B acquire a free slot in the buffer by entering through the `empty` *semaphore (line 19).*

(c) Thread A and B concurrently write to `buffer[index_empty]`*, using the same index and thus overwriting each other's items (line 21).*

On the consumer side, we do not have the same problem, because only a single consumer exists.

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**

Solution:
We have to introduce a mutex that synchronizes the access to the buffer on the producer side to prevent a race condition between the multiple producer threads. We thus add a lock-operation for the new mutex in line 20 and an unlock operation in line 24. If necessary the mutex can be initialized in `initialize()` *(e.g., line 9).*

Total: 10 T-pt