Operating Systems 2016/17
Tutorial-Assignment 7

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

Question 7.1: Race Conditions

a. Explain the term *race condition* with this scenario: Two people try to access a bank account simultaneously. One person tries to deposit 100 Euros, while another wants to withdraw 50 Euros. These actions trigger two update operations in a central bank system. Both operations run in "parallel" on the same computer, each represented by a single thread executing the following code:

```
current = get_balance();
current += delta;
set_balance(current);
```

where `delta` is either 100 or -50 in our example.

Solution:
A race condition is a situation where the correctness of a number of operations depends on the order in which the operations are executed. Let us assume that the initial balance is 1000 Euros. It might happen that the first thread copies the current balance (i.e., 1000*) into the variable* `current` *and is then preempted by the scheduler. The second thread starts, also copies the current balance (*1000*), withdraws 50 Euros, and updates the balance to 950 Euros. When the first thread resumes, it will add the deposit of 100 Euros to* `current` *and write the result back, so that the final balance will be 1100 Euros, but should actually be only 1050 Euros.*

The problem here is that reading the old balance, updating it, and writing it back is not atomic. If it was, either thread 1 or thread 2 would finish its update before the other is allowed to modify the balance.

b. Determine the lower and upper bounds of the final value of the shared variable `tally` as printed in the following program:

```
const int N = 50;
int tally;


void total ()
{
    for( int i = 0; i < N; ++i )
        tally += 1;
}
```

```
int main ()
{
    tally = 0;

    #pragma omp parallel for
    for( int i = 0; i < 2; ++i )
        total();

    printf( "%d\n", tally );
    return 0;
}
```

Assume that threads can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction.

Solution:
On casual inspection, it appears that `tally` *will fall into the range of* $50 \leq$ `tally` ≤ 100 *since between 0 and 50 increments could go unrecorded due to lack of mutual exclusion. The basic argument contends that running these two threads concurrently should not derive results lower than the results produced by executing these threads sequentially.*

Consider the following interleaved sequence of the load, increment, and store operations:

(a) *Thread A loads the value of* `tally` *and increments it, but then loses the processor (it has already incremented its register to 1, but has not yet stored this value back into* `tally`*).*

(b) *Thread B loads the value of* `tally` *(still zero) and performs 49 complete increment operations, losing the processor just after it has stored the value 49 into the shared variable* `tally`*.*

(c) *Thread A regains control long enough to perform its first store operation (replacing the previous value of 49 in* `tally` *with 1) but after this it is immediately forced to relinquish the processor again.*

(d) *Thread B resumes long enough to load 1 (the current value of* `tally`*) into its register, but then it too is forced to give up the processor (note that this was B's final load operation).*

(e) *Thread A is rescheduled, but this time it is not interrupted and runs to completion, performing its remaining 49 load, increment, and store operations, which results in setting the value of* `tally` *to 50.*

(f) *Thread B is rescheduled with only one increment and store operation to perform before it terminates. It increments its register value to 2 and stores this value as the final value of the shared variable.*

Some thought will reveal that a value of lower than 2 cannot occur. Thus the proper and a bit astonishing range of values for the shared variable `tally` *is* $[2, 100]$.

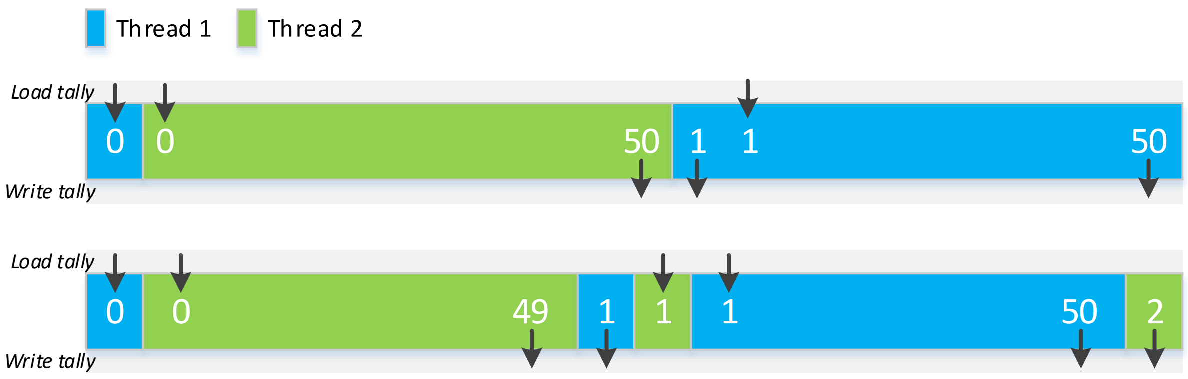

Figure 1: Execution timeline for the two threads in the worst case. The (wrong) intuitive solution at the top, the correct solution at the bottom.

c. Suppose that an arbitrary number $t > 2$ of parallel threads are performing the above procedure `total`. What (if any) influence does the value of t have on the range of the final values of `tally`?

Solution:
The range of final values of the shared variable `tally` *is* $[2, 50 \cdot t]$, *since it is possible for all other threads to be initially scheduled and run to completion in step (e) before thread B would finally destroy their work by finishing last.*

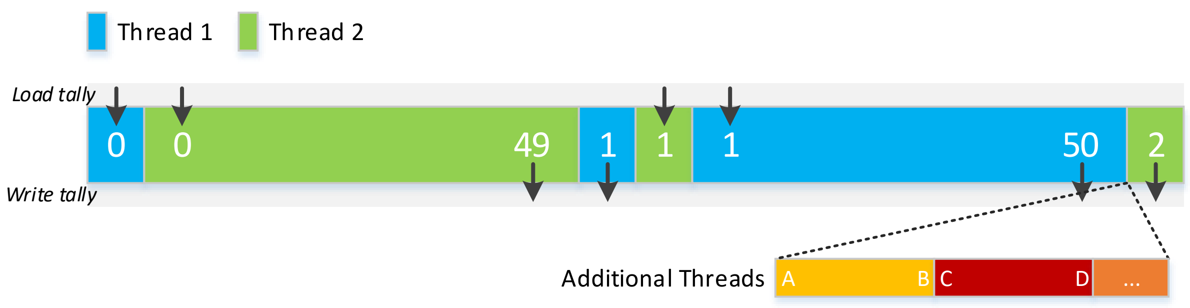

Figure 2: Execution timeline for $t > 2$ threads in the worst case.

d. Now suppose userlevel threads (i.e., the many-to-one model) were used. Would this change make a difference to the output?

Solution:
With (cooperatively scheduled) userlevel threads, no hardware parallelism is leveraged. Hence, only one of the userlevel threads runs at any point in time. As long as the active thread does not voluntarily yield between reading and writing `tally`, *each thread will increment the variable without being preempted, so that* `tally` *will have the correct value when printed.*

e. Finally consider a modified `total` routine:

```
void total ()
{
    for( int i = 0; i < N; ++i )
    {
        tally += 1;
        sched_yield();
    }
}
```

What will be printed in the one-to-one model, when a voluntary yield is added?

Solution:
With the one-to-one model, `sched_yield` *only causes a plethora of additional context switches, but does not guarantee that no thread is interrupted* between *reading and writing* `tally`. *As a consequence, any value between 2 and 100 may be printed.*

Question 7.2: Critical Sections

a. Explain the terms *critical section*, *entry section*, *exit section*, and *remainder section*.

Solution:
A critical section is a code region in which a thread accesses some common data such as a shared variable. As soon as one thread is executing inside a critical section, no other thread is allowed to execute the same critical section.

A critical section is enclosed by an entry section and an exit section. In the entry section, a thread must request permission to enter the critical section. In the exit section, it may signal completion of the critical section, so to allow other threads to enter.

The remaining code after the exit section is called remainder section.

b. Enumerate and explain the requirements for a valid synchronization solution.

Solution:

Exlusiveness: *No two concurrent activities can enter a critical section (CS) that is protected by a synchronization primitive.*

Progress: *Threads in the remainder section do not prevent threads in front of the CS (waiting at the synchronization primitive) from entering the CS.*

Bounded Waiting: *Bounded Waiting ensures that a thread waiting in front of a CS will eventually have the chance to enter the CS, that is, it ensures some "fairness" among threads that compete for the CS.*

c. Recap the banking example from the previous question. How could the race condition be avoided?

Solution:
Race conditions can be avoided if it is ensured that "critical" operations are performed atomically. To ensure atomicity, a synchronization primitive can be used, for example a lock, or a semaphore. A synchronization primitive ensures that at most one thread can be inside a critical section. Using a synchronization primitive `L`, we could fix the code from above:

```
lock(L);
current = get_balance();
current += delta;
set_balance(current);
unlock(L);
```

Now, even if one thread is preempted while updating the balance, the other cannot enter this critical code section, because it will have to wait (either actively or passively) at the `lock(L)` instruction until the other thread performs the `unlock(L)` operation. Keep in mind that we have just passed on the problem to the lock function!

Question 7.3: Synchronization Primitives

a. Distinguish the various types of synchronization objects and summarize their respective operations' semantics: spinlocks, counting semaphores, binary semaphores, and mutex objects.

Solution:

Spinlock `l`: `lock(l)`/`unlock(l)`
Uses busy waiting and atomic instructions (such as `test-and-set`) to ensure mutual exclusion. Wastes CPU time, thus only recommended for short critical sections. Inefficient on single-processor systems.

(Counting) Semaphore `sem`: `wait(sem)`/`signal(sem)`
Each call to `wait` decrements the counter of the semaphore. If the counter falls below 0, the thread/process executing `wait` is blocked and appended to the semaphore's queue. A call to `signal` increments the counter of the semaphore. If it is still less or equal to zero, a thread/process is removed from the queue and unblocked. The counter is not directly accessible for the users of a semaphore.

Counting semaphores can be used to implement bounded buffers (signaling and synchronization) with the counter initially set to the number of items (and/or free slots) in the buffer; they can also be used to implement mutual exclusion (see mutex).

No signals are lost, but `wait(sem)`/`signal(sem)` *operations must be paired correctly.*

Binary Semaphore sem: `wait(sem)`/`signal(sem)`
Mutex (Lock) m: `lock(m)`/`unlock(m)`

A counting semaphore whose counter can only take the values 0 or 1. Note that the semaphore might additionally need to store whether its queue is empty.

Calls to `signal(sem)` *wake up a thread from* `sem`*'s queue (if any) — multiple calls to* `signal(sem)` *without calls to* `wait(sem)` *or threads already waiting in* `sem`*'s queue cause signal losses.*

Condition Variable cond: `wait(cond)`/`signal(cond)`

Always used in conjunction with a mutex. Allows a thread to acquire a lock, check for a certain condition and go to sleep if the condition is not met. The lock is automatically released, when the thread is blocked and reacquired, when the thread is unblocked.

Consider the implementation of the worker pool in assignment 4. The work queue needs to be protected by a mutex, otherwise it can be corrupted by parallel access from the producers and workers.

Without a condition variable, a producer would acquire the lock, submit work, release the lock, and finally wake up a worker.

A worker on the other side, would acquire the lock and check the work queue for new work. If no work is available it would 1) releases the lock (so new work can be submitted), and 2) go to sleep (block). If the operations 1 and 2 are not performed atomically (as with a condition variable), the producer could send the wakeup signal inbetween the operations 1 and 2. Since the worker is still running, the wakeup has no effect. However, the next step for the worker is to go to sleep (although the queue contains work).

Question 7.4: Producer-Consumer Problem

a. Solve the producer-consumer problem for the following buffer using a single pthread mutex and two semaphores:

```
#define BUFFER_SIZE 10
int buffer[BUFFER_SIZE];
int index = 0; // Current element in buffer
```

Solution:

```
pthread_mutex_t lock;
sem_t fill, empty;

void initialize()
{
    pthread_mutex_init(&lock, NULL);
    sem_init(&fill, 0, 0);              // Initialize to 0
    sem_init(&empty, 0, BUFFER_SIZE);   // Initialize to buffer size
}

void* producer_thread_main(void* arg)
{
    while (1) {
        int item = produce(); // Produce a new item

        // Wait for empty slot and "reserve" it atomically
        sem_wait(&empty);
        // Only one thread may modify the buffer at a time
        pthread_mutex_lock(&lock);
        buffer[index++] = item;
        pthread_mutex_unlock(&lock);

        // Signal consumer threads that an item is ready
        sem_post(&fill);
    }
}

void* consumer_thread_main(void* arg)
{
    while (1) {
        // Wait for an item in the buffer and claim it for this consumer
        sem_wait(&fill);

        // Only one thread may modify the buffer at a time
        pthread_mutex_lock(&lock);
        int item = buffer[--index];
        pthread_mutex_unlock(&lock);

        // Signal producer threads that an buffer slot is empty again
        sem_post(&empty);

        consume(item); // Do something useful with the item
    }
}
```