Operating Systems – sprint 2022
Tutorial-Assignment 8

Instructor: Hans P. Reiser

Question 8.1: 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).

b. What is the difference between a reentrant and a non-reeentrant mutex?

Solution:
A reentrant mutex (or recursive mutex) is a particular variant of a mutex that can be locked multiple times by the same process/thread without causing a deadlock. This means that a thread already holding the mutex may call the lock method another time (without blocking). A reentrant mutex tracks the number of times it has been locked by one thread. It becomes available to be locked by another thread only if equally many unlock operations have been performed.

As an example, the native Java mutex implementation (synchronized objects) are reentrant.

By default, a POSIX pthread mutex is not reentrant, but it is possible to create a mutex of type `PTHREAD_MUTEX_RECURSIVE`*, which is reentrant (see next question).*

c. The following code causes a deadlock. Explain why! How can this problem be solved (in a way such that all access to `value` is still protected by a mutex)

```
#include <pthread.h>

pthread_mutex_t lock;
int value = 10;

void initialize ()
{
    pthread_mutex_init(&lock, NULL);
}

void dosomethingelse() {
    pthread_mutex_lock(&lock);
    if(value==100) { value = 52; }
    pthread_mutex_unlock(&lock);
}

void dosomething() {
    pthread_mutex_lock(&lock);
    value = value + 10;
    dosomethingelse();
    value = value − 10;
    pthread_mutex_unlock(&lock);
}

int main() {
     initialize ();
    dosomething();
}
```

Solution:

```
#define _GNU_SOURCE
#include <pthread.h>

int value = 10;

#if USE_STATIC_INITIALIZATION
pthread_mutex_t lock = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;
void initialize () {}
#else
pthread_mutex_t lock;
void initialize ()
{
    pthread_mutexattr_t attr;
    pthread_mutexattr_init(&attr);
    pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
    pthread_mutex_init(&lock, &attr);
}
#endif

void dosomethingelse() {
    pthread_mutex_lock(&lock);
    if(value==100) { value = 52; }
    pthread_mutex_unlock(&lock);
}

void dosomething() {
    pthread_mutex_lock(&lock);
    value = value + 10;
    dosomethingelse();
    value = value - 10;
    pthread_mutex_unlock(&lock);
}

int main() {
     initialize ();
    dosomething();
}
```

d. Using a CPU register for a spinlock's variable would be much faster than the implementation with a variable in main memory. Why would such a spinlock not work?

Solution:
CPU registers are local to a single CPU core/ a single thread. Upon context swith, their content is also switched. However, the lock variable of a spinlock must be shared between threads, and thus it cannot be placed in a CPU register.

Question 8.2: 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
    }
}
```

b. Assume you have only a single producer and a single consumer. Can you simplify the code from part (a) in that case?

Solution:
If you have a single producer and a single consumer, then the counting sempaphore becomes a mutex. Only a sinlge thread will be allowed to enter the critical section below `sem_wait`*. In this case, the additional* `lock` *is not necessary.*

c. Assume you have modified the code such that only two semaphores are used, but the mutex lock has been removed from the solution of part (a). What can go wrong now, if you have a single consumer, but multiple producers?

Solution:

- *Two theads A and B may execute* `produce()`*.*
- *A and B will call* `sem_wait(&empty)`*. As the buffer size is larger than 1, they both can continue immediately.*
- *A and B might concurrently read the same* `index` *value and write their item to the same buffer position.*