الجامعة اللبنانية
كلية التكنولوجيا

CCNE Department
Lebanese University

Real-Time Programming

Lecture 6-part2
Models of synchronization

Rola El Osta

Definition

- The R-W problem is another classic problem for which design of synchronization and concurrency mechanisms can be It is a situation where a data structure can be read and modified simultaneously by concurrent threads.
- There is a data area that is shared among a number of processes.
- Any number of readers may simultaneously write to the data area.
- Only one Writer is allowed access the critical area at any moment in time.
- If a writer is writing to the data area, no reader may read it. When no Writer is active any number of Readers can access the critical area.
- If there is at least one reader reading the data area, no writer may write to it.
- Readers only read and writers only write
- A process that reads and writes to a data area must be considered a writer
- To allow concurrent threads mutually exclusive access to some critical data structure, a mutually exclusive object, or mutex, is used.
- As an implementation mutex is a special case of a more generic synchronisation concept – semaphore. A simple image of a semaphore is a positive number which allows increment by one and decrement by one operations. If a decrement operation is invoked by a thread on a semaphore whose value is zero, the thread blocks until another thread increments the semaphore.

- we denote a semaphore by Semaphore(*n*), where *n* is an initializing number.
- Decrement and increment operations we denote as Wait/P() and Signal/V() correspondingly.
- In this notation a mutex is Semaphore(1), so we use only semaphore concept from now on.

REMARK:

Dear Students, please consider that the lock_writer mutex is to protect the access to the resource!

1st solution: writers wait when a reader reads

```
ressource_type *ressource;
int nb_read = 0;
mutex nb_reader = 1
mutex lock_writer = 1;

reader() {
        P(nb_reader) // a single reader changes the number at a time
        nb_read = nb_read + 1;
        if(nb_read == 1) // only the first lock the access to writers
        P(lock_writer);
        V(nb_reader)
        // no waiting for other readers
        reading(ressource); // lasts a random time
        P(nb_reader)
        nb_read = nb_read - 1;
        if(nb_read == 0)
        V(lock writer);
        V(nb_reader);
}

writer() {
        P(lock_writer); // competition with the first reader
        writing(ressource);
        V(lock_writer); // release the first reader
}
```

2nd solution: Readers have Priority

```
resource_type *resource;
int nb_read = 0;
mutex nb_reader= 1
mutex lock_writer = 1;
mutex wrt_wait = 1;

reader() {
        P(nb_reader) // a single reader changes the number at a time
        nb_read = nb_read + 1;
        if(nb_read == 1) // only the first lock the access to writers
        P(lock_writer);
        V(nb_reader)
        // no waiting for other readers
        reading(ressource); // lasts a random time
        P(nb_reader)
        nb_read = nb_read - 1;
        if(nb_read == 0)
        V(lock_writer);
        V(nb_reader);
}

writer() {
        P(wrt_wait); // the other writers remain stuck here
        P(lock_writer); // competition with the first reader
        writing(ressource);
        V(lock_writer); // release the first reader
        V(wrt_wait); // a reader has passed before giving way to the next
        writer
}
```

The end of a writer releases the lock_writer mutex thus releasing a possible reader before releasing the semaphore wrt_wait.

3rd solution: **Writers have Priority**

```
resource type *resource;
int nb_read = 0;
int nb_wrt = 0;
mutex nb_reader = 1
mutex nb_writer = 1
mutex lock_reader = 1;
mutex lock_writer = 1;
mutex rdr_wait = 1;

writer() {
        P(nb_writer);
        nb_writer = nb_writer + 1;
        if(nb_writer == 1)
        P(lock_reader); // stop the next readers
        V(nb_writer);
        P(lock_writer); // can create a queue of writers
        writing(resource);
        V(lock_writer);
        P(nb_writer);
        nb_writer = nb_writer - 1;
        if(nb_writer == 0)
        V(lock_reader);
        V(nb_writer); // it's the last writer who unblocks the reader (s)
}
```

```
reader() {
	// filter the readers one by one
	// competition between a reader and writers
	P(rdr_wait); // prevents other readers from passing
	P(lock_reader); // the lock is obtained when the last writer release it
	P(nb_reader);
	nb_lect = nb_lect + 1;
	if(nb_lect == 1)
		P(lock_writer); // competition with the writers (the first wins)
	V(nb_reader);
	V(lock_reader);
	// a writer can obtain the lock while the other readers wait (rdr_wait)
	V(rdr_wait);
	reading(resource);
	P(nb_reader);
	nb_reader = nb_reader - 1;
	if(nb_reader == 0)
		V(lock_writer);
	//the last release the writers
	V(nb_reader);
}
```

4th solution: equal chances-Fair Solution for the R-W Problem

```
resource_type *resource;
int nb_read = 0;
mutex lock_reader= 1
mutex lock_writer = 1;
mutex fifo = 1;

writer() {
        P(fifo); // queue of readers and writers
        P(lock_writer);
        V(fifo);
        writing(resource);
        V(lock_writer);
}


reader() {
        P(fifo); // queue of readers and writers
        P(lock_reader);
        nb_read = nb_read + 1;
        if(nb_read == 1)
                P(lock_writer);//competition with the writers
        V(lock_reader);
        V(fifo);
        reading(resource);
        P(lock_reader);
        nb_read = nb_read - 1;
        if(nb_read == 0)
                V(lock_writer);
        V(lock_reader);
}
```

Rwlock Lock (Readers-Writer Lock)

- A **readers–writer** (**RW**) or **shared-exclusive** lock is a synchronization primitive that solves one of the readers–writers problems. An RW lock allows concurrent access for read-only operations, while write operations require exclusive access. This means that multiple threads can read the data in parallel but an exclusive lock is needed for writing or modifying data. When a writer is writing the data, all other writers or readers will be blocked until the writer is finished writing. A common use might be to control access to a data structure in memory that cannot be updated atomically and is invalid (and should not be read by another thread) until the update is complete.
- Readers–writer locks are usually constructed on top of mutexes and condition variables, or on top of semaphores.

The following functions are used to initialize or destroy, lock or unlock, or try to lock a read-write lock.

Routines that Manipulate Read-Write Locks

Operation	Destination Discussion
Initialize a read-write lock	"pthread_rwlock_init(3THR)"
Read lock on read-write lock	"pthread_rwlock_rdlock(3THR)"
Read lock with a nonblocking read-write lock	"pthread_rwlock_tryrdlock(3THR)"
Write lock on read-write lock	"pthread_rwlock_wrlock(3THR)"
Write lock with a nonblocking read-write lock	"pthread_rwlock_trywrlock(3THR)"
Unlock a read-write lock	"pthread_rwlock_unlock(3THR)"
Destroy a read-write lock	"pthread_rwlock_destroy(3THR)"

- To initialize/ destroy the lock:

int pthread_rwlock_init(pthread_rwlock_t * rwl, const pthread_rwlockattr_t * attr);

int pthread_rwlock_destroy(pthread_rwlock_t* rwl);

- To read lock on read-write lock :

int pthread_rwlock_rdlock(pthread_rwlock_t* rwl);

int pthread_rwlock_tryrdlock(pthread_rwlock_t* rwl);

- rdlock : blocks the thread waiting for the lock
- tryrdlock : returns an error if the lock can not be acquired immediately
- To acquire the write lock :

int pthread_rwlock_wrlock(pthread_rwlock_t* rwl);

int pthread_rwlock_trywrlock(pthread_rwlock_t* rwl);

- To release the lock:

int pthread_rwlock_unlock(pthread_rwlock_t* rwl);

Example:

Thread_lock_rwlock.c