Chapter 11.6: Thread Synchronization

When there are multiple threads of control sharing the same memory, there needs to be a consistent view.

Consistency problems won’t exist if

* The variables each thread use is exclusive from each other.
* A variable is read-only but used by multiple threads

Only when there is a potential of one thread modifying another thread’s variable synchronization of the threads are needed.

When a thread modifies a variable, the other threads can potentially see inconsistencies when reading the value of that variable.

* Interleaving memory cycles: if thread A reads a variable and takes two write cycles, and thread B reads the variable in between the two write cycles.
* Solution: using a lock, allows only one thread to access the variable at a time. Thread B will be unable to read the variable until thread A releases the lock.

Increment Operation:

1. Read the memory location into a register
2. Increment the value in the register
3. Write the new value back into the register

Inconsistency occurs if two threads increments the value at almost the same time without synchronizing with each other.

If data is sequentially consistent, then no additional synchronization is needed.

* Sequentially consistent = when multiple threads can’t observe inconsistencies in the data

In a sequentially consistent environment, modifications to the data can be described as a sequential step of operations taken by running threads

11.6.1 Mutexes

A mutex is a lock that is set before accessing a shared resource and then released when done. Any other thread will be blocked while mutex is set until it is released.

If multiple threads are blocked when mutex is released, then the first thread to run will be able to set the new lock. The others will continue to be locked and have to wait for their turn.

The above describes a mutual exclusion mechanism. It only works if threads are designed to follow the same data-access rules.

Mutex variable:

Pthread\_mutex\_t data type.

Initialized by setting it to PTHREAD\_MUTEX\_INITIALIZER (static allocation) or calling pthread\_mutex\_init, which will use pthread\_mutex\_destroy to free memory.

Default attributes of mutex:

int pthread\_mutex\_init(pthread\_mutex\_t \*restrict *mutex*, const pthread\_mutexattr\_t \*restrict *attr*);

Set *attr* to NULL.

Lock and unlock:

int pthread\_mutex\_lock(pthread\_mutex\_t \**mutex*);

int pthread\_mutex\_trylock(pthread\_mutex\_t \**mutex*);

int pthread\_mutex\_unlock(pthread\_mutex\_t \**mutex*);

trylock is used to lock the mutex conditionally when a thread can’t afford to block.

If both trylock and lock are called at the same time, trylock will execute without blocking. Otherwise, the trylock will fail and return EBUSY without locking the mutex.

11.6.2 Deadlock Avoidance

Deadlock in a thread occurs when the thread tries to lock a mutex twice. Or, if two threads try to lock each other mutexes while already holding mutex and block, a deadlock happens because each thread needs a resource held be the other thread, and hence neither can proceed.

Avoiding deadlocks:

Controlling the order of mutexes locking.

Example: two mutexes, A and B, that needs to be locked at the same time. All threads must lock one before the other, otherwise if a thread locks in the opposite order from the other, a deadlock will occur.

Other approaches:

Release locks and try again later.

Use pthread\_mutex\_trylock to avoid deadlocking.

If already holding locks and pthread\_mutex\_trylock is successful, then proceed. If not, release the locks already there, clean up and try again later.

11.6.3 pthread\_mutex\_timedlock Function

The function allows binding the time a thread uses to block when a mutex it is trying to acquire is already locked. Similar to pthread\_mutex\_lock function but with the added feature of a timeout value. When this value is reached the function will return error code ETIMEDOUT without locking the mutex.

int pthread\_mutex\_timedlock(pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict tsptr);

Timeout value = absolute time (block until time X. NOT block for X amount of seconds)

Data type = timespec structure, describes time in terms of seconds and nanoseconds.

11.6.4 Reader-Writer Locks

Similar to mutexes but allows a higher degree of parallelism.

Three possible states:

1. Locked in read mode
2. Locked in write mode
3. Unlocked

A reader-writer lock in write mode can only be held by one thread at a time

A reader-writer lock in read mode can be held by multiple thread at the same time.

In write-mode:

All threads attempting to lock the reader-writer lock blocks until the reader-writer lock is unlocked.

In read-mode:

All threads attempting to lock the reader-writer lock are given access, but any threads attempting to lock the reader-writer lock in write mode will block until all the other threads have released their read locks.

Reader-writer locks are used more often for situation in which data structures are read more than they are modified.

Reader-writer locks are also called shared-exclusive locks.

Initializer and destroyer:

Int pthread\_rwlock\_init(pthread\_rwlock\_t \*restrict *rwlock*, const pthread \_rwlockattr\_t \*restrict *attr*);

* *attr* = null if initialized with default attributes

Int pthread\_rwlock\_destroy(pthread\_rwlock\_t \**rwlock*);

* freeing the memory

PTHREAD\_RWLOCK\_INITIALIZER = initializing statically

Lock and unlocking:

int pthread\_rwlock\_rdlock(pthread\_rwlock\_t \**rwlock*);

* lock a reader-writer lock in read mode

int pthread\_rwlock\_wrlock(pthread\_rwlock\_t \**rwlock*);

* lock a reader-writer lock in write mode

int pthread\_rwlock\_unlock(pthread\_rwlock\_t \**rwlock*);

* to unlock a reader-writer lock

Depending on implementation, check the return value of the rdlock function, because there can be a limit on the number of times a reader-writer lock can be locked in shared mode.

Conditioner versions of locking:

int pthread\_rwlock\_tryrdlock(pthread\_rwlock\_t \**rwlock*);

int pthread\_rwlock\_trywrlock(pthread\_rwlock\_t \**rwlock*);

if a lock can be acquired, the functions return 0, otherwise it’ll return EBUSY.

Can be used to avoid deadlocks.

11.6.5 Reader-Writer Locking with Timeouts

The reader-writer lock version of a timeout function.

int pthread\_rwlock\_timedrdlock(pthread\_rwlock\_t \*restrict *rwlock*, const struct timespec \*restrict *tsptr*);

int pthread\_rwlock\_timedwrlock(pthread\_rwlock\_t \*restrict *rwlock*, const struct timespec \*restrict *tsptr*);

Same as their “untimed” counterparts.

The *tsptr* argument points to a timespec structure, it specifies the time at which the thread should stop blocking.

If lock cannot be acquired, then ETIMEDOUT error will be returned.

11.6.6 Condition Variables

Another synchronization mechanism available to threads. Provides a place for threads to rendezvous. Allows threads to wait in a race-free way for arbitrary conditions to occur when used with mutexes. Condition protected by a mutex, because a thread can only modify the condition if the mutex is locked.

Data type: pthread\_cond\_t

Initializer: PTHREAD\_COND\_INITIALIZER (static allocation) or pthread\_cond\_init(pthread cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr);

Destroyer: pthread\_cond\_destroy(pthread\_cond\_t \*cond);

Int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

* function to wait for a condition is true, returns 0 if ok otherwise error
* the mutex passed protects the condition. It is passed locked, placing thread on the waiting list for the condition to be true and for mutex to be unlocked
* when function returns, mutex is locked again.

int pthread\_cond\_timewait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex, const struct timespec \*restrict tsptr);

* the same as the wait function with the addition of the timeout(tsptr).
* If timeout expires without the condition occurring, pthread\_cond\_timedwaitwill reacquire the mutex and return the error ETIMEDOUT.
* If successful, a thread needs to reevaluate the condition

Int pthread\_cond\_signal(pthread\_cond\_t \*cond);

* Notifies threads that a condition has been satisfied by waking up at least one thread waiting on a condition

Int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

* Notifies by waking up all threads waking on a condition

11.6.7: Spin Locks

It is basically a mutex, but instead of blocking a process by sleeping, it is blocked by busy waiting (spinning) until lock can be acquired. Commonly used in situations where locks are held for short periods of times and threads don’t want to incur the cost of being descheduled.

Low-level primitives to implement other types of locks. Cons is that spin locks can waste CPU resources: while a thread is spinning and waiting for a lock to become available, the CPU can’t do anything else.

Int pthread\_spin\_init(pthread\_spinlock\_t \*lock, int pshared);

Int pthread\_spin\_destroy(pthread\_spinlock\_t \*lock);

If Thread Process-Shared Synchronization option supported, then pshared matters.

Pshared represents the process-shared attribute, which indicates how the spin lock will be acquired.

* PTHREAD\_PROCESS\_SHARED = spin lock acquired by threads that have access to the lock’s underlying memory
* PTHREAD\_PROCESS\_PRIVATE = spin lock accessed only from threads within the process that initialized it.

Int pthread\_spin\_lock(pthread\_spinlock\_t \*lock);

* Spin until the lock is qcquired
* If a spin lock is currently unlocked, then this function can lock it without spinning
* If the thread already has the lock locked, then the results are undefined
* Potentially fail with EDEADLK error or spin indefinitely.

Int pthread\_spin\_trylock(pthread\_spinlock\_t \*lock);

* Return EBUSY error if the lock can’t be acquired immediately

Int pthread\_spin\_unlock(pthread\_spinlock\_t \*lock);

* Unlocks no matter which locking function is called
* If an unlocked lock is unlocked, then the results are undetermined.

11.6.8 Barriers

Another form of synchronization mechanism that can be used to coordinate multiple threads working in parallel. It allows each thread to wait until all cooperating threads have reache the same point, and then continues execution from there.

One form of barrier, pthread\_join, allows one thread to wait until another thread exits.

Barrier objects allows an arbitrary number of threads to wait until all the threads have completed processing, but the threads don’t have to exit.

Int pthread\_barrier\_init(pthread\_barrier\_t \*restrict barrier, const pthread\_barrierattr\_t \*restrict attr, unsigned int count);

* Initializer
* count argument specifies the number of threads that must reach the barrier before all of the threads will be allowed to continue
* when barrier count is reached and the threads are unblocked, the barrier can be used again
* attr specifies the attributes of the barrier object

int pthread\_barrier\_destroy(pthread\_barrier\_t \*barrier);

* deinitalizer

int pthread\_barrier\_wait(pthread\_barrier\_t \*barrier);

* used to indicate that a thread is done with its work and is ready to wait for all the other threads to catch up.
* The thread calling this function is put to sleep if the barrier count is not reached.
* If count is satisfied, then the calling function is the last function, and all threads are awakened.
* To one arbitrary thread, pthread\_barrier\_wait returns a value PTHREAD\_BARRIER\_SERIAL\_THREAD, remaining threads see return value 0