Locks

Arnav Gupta

November 15, 2024

Contents

1 Lock Taxonomy

Locks can be either spinning or blocking.

Spinning locks busy wait until an event occurs, so the task oscillates between ready and running due to time slicing (can also optionally yield).

Blocking locks do not busy wait but block until an event occurs, so some other mechanism must unblock the waiting task when the event happens.

2 Spin Lock

When a task is busy waiting, it usually loops until the critical section is unlocked or it is preempted and placed on the ready queue.

To increase uniprocessor efficiency, a task can:

- explicitly terminate its time slice
- move back to the ready state after only one event check fails

`yield` relinquishes the time-slice by rescheduling the running task back onto the ready queue. To increase multiprocessor efficiency, a task can yield after N event checks fail.

Adaptive Spin Lock: allow adjustment of spin duration

Most spin-lock implementations can cause starvation of 1+ tasks.

Spin lock is appropriate and necessary in situations where this is no other work to do.

2.1 Implementation

μC++ provides `uSpinLock` (non-yielding) and `uLock` (yielding)

```
class uSpinLock {
    public:
        uSpinLock(); // open
        void acquire();
        bool tryacquire(); // makes only 1 attempt to acquire lock
        void release();
}

class uLock {
    public:
        uLock( unsigned int value = 1 );
        void acquire();
        bool tryacquire(); // makes only 1 attempt to acquire lock
        void release();
}
```

Locks built from atomic hardware instructions. Locks start closed (0) or opened (1).

Starvation can occur in theory, but rare in practice.

Cannot assign to or copy a lock, must use as reference/pointer. Can be used for synchronization or mutual exclusion.

3 Blocking Locks

For blocking locks, the acquiring task is not the only one responsible for detecting an open lock:

- acquiring task makes 1 check for open lock and then blocks
- releasing task is responsible for detecting a blocked acquirer and transferring or releasing the lock

Blocking locks reduce busy waiting through **cooperation**.

All blocking locks have

- state to facilitate lock semantics
- list of blocked acquirers

3.1 Mutex Lock

Used solely to provide mutual exclusion:

- separates lock usage between synchronization and mutual exclusion
- permits optimizations and checks as the lock only provides one specialized function

Mutex locks can be **single acquisition** (acquirer cannot acquire again) or **multiple acquisition** (acquirer can acquire multiple times). Multiple acquisition locks are called **owner locks**.

Owner locks can handle looping/recursion and may require 1 release or as many as it requires.

3.1.1 Implementation

```
class MutexLock {
        bool avail;
        Task * owner;
        queue<Task> blocked;
        SpinLock lock;
    public:
        MutexLock() : avail { true }, owner{ nullptr } {}
        void acquire() {
            lock.acquire();
            while ( !avail && owner != currThread() ) {
                yieldNoSchedule( lock );
                lock.acquire();
            }
            avail = false;
            owner = currThread();
            lock.release();
        }

        void release() {
            lock.acquire();
            if ( owner != currThread() ) {
                // error check
            }
            owner = nullptr;
```

```
            if ( !blocked.empty() ) {
                // remove task from blocked list and make ready
            }
            avail = true;
            lock.release();
        }
}
```

`yieldNoSchedule` yields the processor time slice but does not reschedule thread to the ready queue.

This code allows for barging since `avail` and the lock reset.

For **barging avoidance**, must hold `avail` between releasing and unblocking task (protects against bounded overtaking):

```
class MutexLock {
        bool avail;
        Task * owner;
        queue<Task> blocked;
        SpinLock lock;
    public:
        MutexLock() : avail { true }, owner{ nullptr } {}
        void acquire() {
            lock.acquire();
            if ( !avail && owner != currThread() ) {
                yieldNoSchedule( lock );
                // do not reacquire lock since avail == false
            } else {
                avail = false;
                lock.release();
            }
            owner = currThread();
        }

        void release() {
            lock.acquire();
            owner = nullptr;
            if ( !blocked.empty() ) {
                // remove task from blocked list and make ready
            } else {
                avail = true;
```

```
            }
            lock.release();
        }
}
```

With these changes, bargers enter mutual exclusion protocol but block so the released task does not busy wait. This means mutual exclusion conceptually passed from releasing to unblocking tasks.

For **barging prevention**, must hold `lock` between releasing and unblocking task (protects against unbounded overtaking):

```
class MutexLock {
        bool avail;
        Task * owner;
        queue<Task> blocked;
        SpinLock lock;
    public:
        MutexLock() : avail { true }, owner{ nullptr } {}
        void acquire() {
            lock.acquire();
            if ( !avail && owner != currThread() ) {
                yieldNoSchedule( lock );
                // do not reacquire lock
            } else {
                avail = false;
            }
            owner = currThread();
            lock.release();
        }

        void release() {
            lock.acquire();
            owner = nullptr;
            if ( !blocked.empty() ) {
                // remove task from blocked list and make ready
                // do not release lock
            } else {
                avail = true;
                lock.release();
            }
```

```
        }
}
```

The critical section is not bracketed by the spin lock when the lock is passed, so this works.

An alternative would be to leave the lock owner at the front of the blocked list to act as availability and owner variable. If the critical section is acquired, the blocked list must have a node on it to check if it is in use.

3.1.2 uOwnerLock

```
class uOwnerLock {
    public:
        uOwnerLock();
        uBaseTask * owner(); // nullptr if no owner, else owner address
        unsigned int times(); // number of times lock has been acquired by owner
        void acquire();
        bool tryacquire();
        void release();
}
```

Must release as many times as acquired.

Can use `_Finally` or RAII to ensure lock is always released. However, this cannot be used for barging prevention.

3.1.3 Stream Locks

Can use `osacquire` for output streams and `isacquire` for input streams to ensure predictable streams.

3.2 Synchronization Lock

Used solely to block tasks waiting for synchronization.

Only state is a list of blocked tasks, so:

- acquiring task always blocks (no state to make it conditional)
- release is lost when no waiting task (no state to remember it)

Uses wait for acquire and signal for release.

3.2.1 Implementation

Needs mutual exclusion for safe implementation.

Location of mutual exclusion classifies synchronization lock:

- **external lock**: use an external lock to protect task list
- **internal lock**: use an internal lock to protect state

With external locking:

```
class SyncLock {
        Task * list;
    public:
        SyncLock() : list{ nullptr } {}
        void acquire( MutexLock & m ) {
            // add self to task list
            yieldNoSchedule( m );
            // possibly reacquire mutex lock
        }
        void release() {
            if ( list != nullptr ) {
                // remove task from blocked list and make it ready
            }
        }
}
```

Uses external task to avoid lost release and need mutual exclusion to protect task list (and possible external state). The releasing task detects a blocked task and performs necessary cooperation.

To use with barging avoidance:

```
m.acquire();
if ( !s.empty() ) {
    s.release();
} else {
    occupied = false;
}
m.release();
```

To use with barging prevention:

```
m.acquire();
```

```
if ( !s.empty() ) {
    s.release();
} else {
    occupied = false;
    m.release();
}
```

With internal locking:

```
class SyncLock {
        Task * list;
        SpinLock lock;
    public:
        SyncLock() : list{ nullptr } {}
        void acquire( MutexLock & m ) {
            lock.acquire();
            // add self to task list
            m.release();
            // can be interrupted here, but fine since spin lock
            yieldNoSchedule( lock );
            m.acquire(); // possibly reacquire mutex lock
        }
        void release() {
            lock.acquire();
            if ( list != nullptr ) {
                // remove task from blocked list and make it ready
            }
            lock.release();
        }
}
```

This still takes an external lock for barging avoidance/prevention.

3.2.2 uCondLock

```
class uCondLock {
    public:
        uCondLock();
        void wait( uOwnerLock & lock );
        bool signal(); // unblocks in FIFO order
        bool broadcast(); // unblocks all waiting tasks
```

```
        bool empty(); // are blocked tasks on the queue?
}
```

`wait` atomically blocks the calling task and releases the argument owner lock, and reacquires it before returning.

`signal` and `broadcast` do nothing for an empty conditional and return false, otherwise return true.

3.2.3 Programming Pattern

Must provide external mutual exclusion and protect against lost signal (release).

Should surround conditional lock with mutex lock.

3.3 Barrier

Coordinates a group of tasks performing a concurrent operation surrounded by sequential operations. Meant for synchronization, not mutual exclusion.

Two kinds of barriers:

- threads == group size
- threads > group size

Barrier retains state about the events it manages (num tasks blocked on the barrier). Most barriers use internal locking.

Barrier blocks each task at call to `block` until all tasks have called `block`. Last task to call `block` does not block, instead it releases all other tasks.

Must specify in advance number of `block` operations before tasks released.

Barriers are commonly used for synchronized one-shot and for synchronized start and end in a cycle.

Using a barrier is cheaper than creating and deleting tasks for each computation.

3.3.1 Fetch Increment Barrier

A spinning, T == G barrier can be implemented with the fetch-increment instruction and a flag that waiters wait on.

3.3.2 uBarrier

μC++ barrier is a blocking, T > G, barging-prevention coroutine, where `main` can be resumed by the last task arriving at the barrier:

```
#include <uBarrier.h>
_Cormonitor uBarrier {
    protected:
        void main() { for ( ;; ) suspend(); }
        virtual void last() { resume(); } // called by last task to barier
    public:
        uBarrier( unsigned int total );
        unsigned int total() const; // # of tasks synchronizing
        unsigned int waiters() const; // # of waiting tasks
        void reset( unsigned int total ); // reset # of tasks synchronizing
        virtual void block(); //wait for Nth thread
}
```

`uBarrier` has implicit mutual exclusion so no barging.

Can build a barrier by inheriting from `uBarrier`, redefining `last`, `block`, and possibly `main`.

Coroutine barrier can be reused many times.

3.4 Binary Semaphore

Blocking equivalent to a yielding spin-lock, providing synchronization and mutual exclusion.

Acquire is `P` (waits if counter is 0, then decrements). Release is `V` (increases counter and unblocks waiting task if present).

Binary semaphore has only 2 states, open and closed.

3.4.1 Implementation

Has a:

- blocking task list
- `avail` $\rightarrow$ if event has occurred (state)
- spin lock to protect state

```
class BinSem {
    queue<Task> blocked; // blocked tasks
    bool avail; // resource available
    SpinLock lock;
public:
    BinSem( bool start = true ) : avail( start ) {}
    void P() {
        lock.acquire(); // prevention barging
        if ( !avail ) {
            // add self to blocked list
            yieldNoSchedule( lock );
            // do not reacquire lock
        }
        avail = false;
        lock.release();
    }
    void V() {
        lock.acquire();
        if ( !blocked.empty() ) {
            // remove task from blocked list and make ready
            // do not release lock
        } else {
            avail = true; // conditional reset
            lock.release(); // no race
        }
    }
};
```

Higher cost for synchronization if external lock already acquired.

3.5 Counting Semaphore

Allow a multi-valued semaphore, which allows for critical sections allowing N simultaneous tasks.

Done by augmenting `V` to allow increasing the counter an arbitrary amount.

3.5.1 Implementation

Change availability into a counter, set to some maximum on creation.

```
class CntSem {
```

```
    queue<Task> blocked; // blocked tasks
    int cnt; // resource being used
    SpinLock lock;
public:
    CntSem( int start = 1 ) : cnt( start ) {}
    void P() {
        lock.acquire(); // prevention barging
        cnt -= 1;
        if ( cnt < 0 ) {
            // add self to blocked list
            yieldNoSchedule( lock );
            // do not reacquire lock
        }
        lock.release();
    }
    void V() {
        lock.acquire();
        cnt += 1;
        if ( cnt <= 0 ) {
            // remove task from blocked list and make ready
            // do not release lock
        } else {
            lock.release(); // no race
        }
    }
};
```

To use a semaphore:

- for synchronization, semaphore at $0 \rightarrow$ waiting for an event
- for mutual exclusion, semaphore at $N \rightarrow$ controls a critical section

μC++ has a counting semaphore, which subsumes a binary semaphore

```
#include <uSemaphore.h>
class uSemaphore {
public:
    uSemaphore( unsigned int count = 1 );
    void P();
    bool TryP();
    void V( unsigned int times = 1 );
```

```
    int counter() const;
    bool empty() const; // threads blocked?
};
```

`P` decrements the counter, if the counter is ≥ 0, the calling task continues, else it blocks.

`TryP` returns true if the semaphore is acquired and false otherwise (never blocks).

`V` wakes up the task blocked for the longest time and increments the counter, can occur N times.

4 Lock Programming

4.1 Precedence Graph

Binary semaphore with `COBEGIN` are as powerful as `START` and `WAIT`.

Analyze which data and code depend on each other and display dependencies graphically in a **precedence graph**.

4.2 Buffering

Tasks communicate unidirectionally through a queue, with the producer adding items to the back of the queue and the consumer removing items from the front of the queue.

4.2.1 Unbounded Buffer

Two tasks communicate through a queue of unbounded length.

Since tasks work at different speeds, the producer may get ahead of the consumer. Producer never has to wait (infinite length) but the consumer may have to wait for the producer to add.

This can be solved with a counting semaphore controlling access to the shared queue.

4.2.2 Bounded Buffer

Two tasks communicate through a queue of bounded length.

Producer has to wait if buffer full, consumer has to wait if buffer empty.

This can be solved with an additional counting semaphore to keep track of if the buffer is empty.

4.3 Lock Techniques

Split binary semaphore: collection of semaphores where at most 1 of the collection has the value 1

- used when different kinds of tasks have to block separately
- cannot differentiate tasks blocked on the same semaphore

Split binary semaphores can be used for **baton passing**:

- there is exactly one conceptual baton
- nobody moves in the entry/exit code unless they have the baton
- once the baton is release, cannot read/write variables in entry/exit

Mutex/condition lock cannot perform baton passing to prevent barging if the signaled task must implicitly reacquire the mutex lock before continuing, since the signaler must release the mutex lock. This causes a race between the signalled and calling tasks, resulting in barging.

4.4 Readers and Writers Problem

Multiple tasks share the resource, with some reading and some writing. Readers can have simultaneous access, but writer access must be serialized.

A split-binary semaphore can be used to segregate arrivers, readers, and writers, using baton-passing to help understand complexity.

4.4.1 Solution 1

Problem:

- reader only checks for writer in resource, never writers waiting to use it
- this causes readers to barge ahead of writers who already waited
- with enough readers, writers could starve

4.4.2 Solution 2

Giving writers priority should work since usually less writers, but this causes writers to barge and potentially reader starvation.

4.4.3 Solution 3

Use alternation like Dekker's solution, by selecting from the opposite kind on exit.

With this, arriving readers cannot barge ahead of waiting writers and unblocking writers cannot barge ahead of a waiting reader, so this gives alternation for simultaneous waiting.

4.4.4 Solution 4

Problem: temporal barging, since the `last` flag and data could be stale or fresh.

Better to service readers and writers in **temporal order**, like FIFO but still allowing multiple concurrent readers.

This can be done by having readers and writers wait on the same semaphore, but then no differentiation between type of waiting task.

This can be fixed with a shadow queue that retains the kind of waiting task on the semaphore.

4.4.5 Solution 5

Can also cheat on cooperation by allowing for 2 checks for write instead of 1, using the reader/writer bench and writer chair.

On exit, if the chair is empty, unblock the task at the front of the reader/writer semaphore, but this can cause a reader to unblock a writer. The writer then waits in the chair, which is always checked first on exit.

4.4.6 Solution 6

There is still a temporal problem when tasks move from one blocking list to another. Fixing this required atomic block and release, using magic like turning off time-slicing.

An alternative fix is ticketing, where each reader/writer takes a ticket before putting the baton down. Before passing the baton, the serving counter is

incremented and then all blocked tasks are awoken. Each task checks their ticket, with only one proceeding (others reblock). Starvation is not an issue since the waiting queue has bounded length.

Another alternative is to use a list of **private semaphores**, one for each waiting task, with a list node being added before releasing the entry lock, and then blocking on the private semaphore. When the baton is passed, the private semaphore at the head of the queue is `V`'d, allowing it to continue.

4.4.7 Solution 7

Use split binary semaphores and baton passing, with tasks waiting in temporal order on the entry semaphore and only one writer waiting on the writer chair until readers leave the resource.

With this solution, waiting writer blocks holding baton to force other arriving tasks to wait on entry.

This solution is harder to reason about and does not generalize for other kinds of complex synchronization and mutual exclusion.