Practice: Klingon Problem

The Klingons are attacking. The Federation vessels can escape through the wormhole, but sensors indicate that the wormhole is unstable. The ships' captains plan to create a subspace distortion to prevent the wormhole from closing on them while they are in it. To do this, they will enter the wormhole three at a time while emitting a tachyon pulse through their main deflector dishes. Each ship must lower its shields before initiating its tachyon pulse, and once a ship starts emitting tachyons it can have no contact with the other ships.

Implement a synchronization scheme to allow the Federation to retreat through the wormhole in an orderly fashion. Your scheme should have the property that no ship lowers its shields until just before it enters the wormhole, no ship enters the wormhole until two others are ready to go in with it, and all ships in each group of three enter the wormhole before any of the ships in the next group.

```
long num = 0; next=3;     int going_in = 0;  // solution assumes no overflow
void EnterWormhole ( )
{   int mynum;
    wormholeMutex.acquire ( );
    num = num + 1;
    mynum =  num;
    while ( mynum > next) nextGroup.wait(&wormholeMutex);

    while (num < next){ three.wait (&wormholeMutex);}
    else {three.broadcast();}

    going_in = going_in +1;
    if (going_in == 3) {next = next + 3; going_in = 0;
          nextGroup.broadcast();}
    wormholeMutex.release ( );
}
```

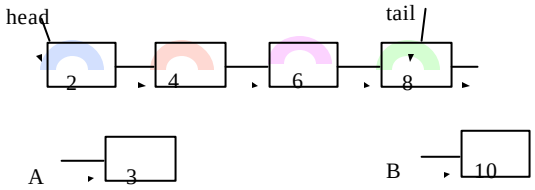
```
long num = 0; next=3;     // solution assumes no overflow
int going_in = 0;
Boolean deferred0 = false; deferred1 = false;

void EnterWormhole ( )
{   int mynum;
    wormholeMutex.acquire ( );
    num = num + 1;
    mynum =  num;
    while ( mynum > next) nextGroup.wait(&wormholeMutex);
    while (num < next){ three.wait (&wormholeMutex);}
    else deferred0 = true;
    going_in = going_in +1;
    if (going_in == 3) {next = next + 3; going_in = 0; deferred1 = true;}
    wormholeMutex.release ( );

     if (deferred0) {deferred0 = false; three.broadcast();}
     if (deferred1) { deferred1 = false; nextGroup.broadcast();}
}
```

Practice: Fine grain locking

Multiple threads inserting and deleting in a linked list



Solution Trick: Lock-Coupling

```
void InsertList (int key)
{
      firstlock.Acquire();
      if (key <= first–>data) {//insert as first, special case
                                 firstlock.Release(); return;}
      else{ ptr = first;
      ptr–>lock.Acquire(); firstlock.Release(); next=ptr–>next;
      while (next != null & key > next->data)    // Next isn't locked - why OK??
            {next–>lock.Acquire(); ptr–>lock.Release(); ptr=next;
            next=ptr–>next;}
       if (next == null) {//append after ptr; unlock ptr;}
       else if (key <= next–>data)
          {//insert between ptr and next; unlock ptr;}
}
```

Practice: Bridge Problem

Synchronize traffic over a narrow light-duty bridge on a public highway. Traffic may only cross the bridge in one direction at a time, and if there are ever more than 3 vehicles on the bridge at one time, it will collapse under their weight. Each car is to be represented by one thread, which executes the procedure OneVehicle in order to cross the bridge:

```
OneVehicle(int direc) //direc is either 0 or 1;
                      //giving the direction in which the car is to cross
{
ArriveBridge (direc);
CrossBridge (direc);
ExitBridge (direc);
}
```

Ultimate Problem

Simulate an ultimate frisbee game where each player is a thread. Start simple (e.g. where both the disc and the players move randomly around the field) and elaborate.

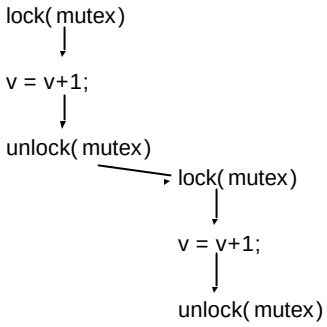
Eraser
Savage et al.

- Dynamic data race detection tool
- Checks that each shared memory access follows a consistent *locking discipline*
- *Data race* – when 2 concurrent threads access a shared variable and at least one is a write and the threads use no explicit synchronization to prevent simultaneous access.

Lamport's Happened-before

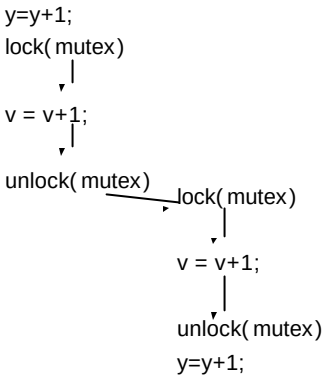
Previous work

- If 2 threads access a shared variable and the accesses are not ordered by *happens-before* then potential race.
- Depends on scheduler



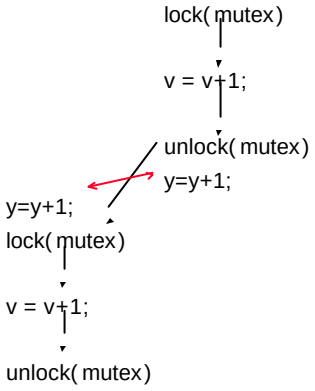
Previous work

- If 2 threads access a shared variable and the accesses are not ordered by *happens-before* then potential race.
- Depends on scheduler



Previous work

- If 2 threads access a shared variable and the accesses are not ordered by *happens-before* then potential race.
- Depends on scheduler



Lockset Algorithm

- C(v) – candidate locks for v
- locks-held(t) – set of locks held by thread t
- Lock refinement

for each v, init C(v) to set of all locks

On each access to v by thread t:

C(v) = C(v) ∩ locks-held(t)

If C(v) = {} issue warning

Example

	locks-held	C(v)
	{}	{mu1, mu2}
lock(mu1)	{mu1}	
v=v+1		{mu1}
unlock(mu1)	{}	
lock(mu2)	{mu2}	
v=v+1		{}
unlock(mu2)	{}	

More Sophistication

- Initialization without locks
- Read-shared data
- Read-write locking
- False Alarms still possible
- Don't start until see a second thread
- Report only after it becomes write shared
- Change algorithm to reflect lock types
  - On read of v by t:
    - $C(v) = C(v) \cap$ locks-held(t)
  - On write of v by t:
    - $C(v) = C(v) \cap$ write-locks-held(t)

Implementation

- Binary rewriting used
  - Add instrumentation to call Eraser runtime
  - Each load and store updates C(v)
  - Each Acquire and Release call updates locks-held(t)
  - Calls to storage allocator initializes C(v)
- Storage explosion handled by table lookup and use of indexes to represent sets
  - Shadow word holds index number

Common False Alarms - Annotations

- Memory reuse
- Private locks
- Benign races

```
if (some_condition) {
    LOCK m DO
        if (some_condition)
        {stuff}
    END
}
```

- EraseReuse – resets shadow word to virgin state
- Lock annotations
- EraserIgnoreOn()
  EraserIgnoreOff()

Core Loop of Lock-Coupling

```
// ptr->lock.Acquire(); has been done before loop

while (next != null & key > next->data)
      {next->lock.Acquire();
       ptr->lock.Release();
       ptr=next;
       next=ptr->next;
}
```