# EECS 440 System Design of a Search Engine

Winter 2021

Lecture 15: Various system design problems

Nicole Hamilton
https://web.eecs.umich.edu/~nham/
nham@umich.edu

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# details

1. All but one team got 150/150 on HW6. One team missed slightly on perf.
2. Four teams are at 180/180 on HW7. Four teams have not submitted anything yet.
3. Midterm grading is at 89%. We're almost done.
4. Tomorrow is a well-being break for TuTh classes and some classes are extending the break. Would you like your own well-being break on Wed? (OH only, no lecture?)
5. Would you like to do a session on negotiations with some breakouts to negotiate some agreements?
6. ADUE Joanna Millchick announced last night that in F22, most CoE classes will in person on campus and may not offer remote options.

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# Comparing hashing functions

Decided to compare various hashing functions.

1. Simple XOR
2. Fowler-Noll-Vo
3. Ethernet polynomial

```
uint32_t SimpleXORHash( const char *s )
   {
   uint32_t h = 0;
   if ( s )
      for ( const char *p = s;  *p;  p++ )
         h = ( h < 1 ) ^ ( uint32_t )*p;
   return h;
   }
```

# Comparing hashing functions

Decided to compare various hashing functions with the HashTable HW.

1. Simple XOR
2. Fowler-Noll-Vo
3. Ethernet polynomial

```
uint32_t FowlerNollVoHash( const char *s )
   {
   const uint64_t
      offsetBasis = 14695981039346656037ul,
      prime = 1099511628211ul;

   uint64_t h = offsetBasis;

   if ( s )
      {
      unsigned char c;
      for ( const char *p = s;
            c = ( unsigned char )*p++;   )
         h = ( h ^ ( uint64_t )c ) * prime;
      }
   return ( uint32_t )h;
   }
```

# Comparing hashing functions

Decided to compare various hashing functions.

1. Simple XOR
2. Fowler-Noll-Vo
3. Ethernet polynomial

```
uint32_t EthernetPolynomial( const char *s )
   {
   uint32_t h = 0;
   if ( s )
      {
      unsigned char c;
      for ( const char *p = s;
            c = ( unsigned char )*p++;  )
         h = UpdateCRC( h, c );
      }
   return h;
   }
```

# Test conditions

2.2MB sample HTML crudely parsed.

| | Words | Lines |
|---|---:|---:|
| Number of tokens | 215,994 | 62,001 |
| Average token length | 8.25 | 34.64 |
| Unique keys | 18,860 | 14,766 |
| Total characters (not counting terminating nulls) | 1,782,086 | 2,147,964 |

# Parsed as words

**Elapsed times in ticks**

| | Brand A | Brand B | Brand C |
|---|---|---|---|
| Build | 555,476,900 | 17,769,900 | 14,326,100 |
| Search | 536,284,200 | 17,299,000 | 14,737,200 |
| Optimize | 54,763,500 | 624,200 | 528,400 |
| Search | 525,788,400 | 17,194,100 | 13,730,000 |
| Top 10 | 343,800 | 627,800 | 604,200 |

# Parsed as lines

**Elapsed times in ticks**

| | Brand A | Brand B | Brand C |
|---|---|---|---|
| Build | 1,610,992,600 | 13,128,100 | 8,177,000 |
| Search | 1,590,755,300 | 14,002,200 | 9,401,200 |
| Optimize | 331,890,000 | 499,700 | 405,000 |
| Search | 1,585,298,600 | 13,881,800 | 9,313,200 |
| Top 10 | 241,600 | 546,200 | 502,600 |

# Parsed as words

| | Brand A | Brand B | Brand C |
|---|---|---|---|
| *Initially* | | | |
| Number of Buckets | 8,192 | 8,192 | 8,192 |
| Buckets in use | 96 | 7,433 | 7,347 |
| Number of Collisions | 18,764 | 11,427 | 11,513 |
| Longest collision length | 2,794 | 9 | 10 |
| *After optimization* | | | |
| Number of Buckets | 32,768 | 32,768 | 32,768 |
| Buckets in use | 96 | 14,337 | 14,280 |
| Number of Collisions | 18,764 | 4,523 | 4,580 |
| Longest collision length | 2,794 | 5 | 5 |

# Parsed as lines

| | Brand A | Brand B | Brand C |
|---|---|---|---|
| *Initially* | | | |
| Number of Buckets | 8,192 | 8,192 | 8,192 |
| Buckets in use | 3 | 6,859 | 6,867 |
| Number of Collisions | 14,763 | 7,907 | 7,899 |
| Longest collision length | 14,763 | 8 | 7 |
| | | | |
| *After optimization* | | | |
| Number of Buckets | 32,768 | 32,768 | 32,768 |
| Buckets in use | 96 | 11,905 | 11,926 |
| Number of Collisions | 18,764 | 2,861 | 2,840 |
| Longest collision length | 2,794 | 4 | 5 |

## So, what was Brand A, Brand B and Brand C?

So, what was Brand A, Brand B and Brand C?

Brand A was the XOR.  It’s terrible.

Brand B was the Ethernet polynomial.  It’s best at avoiding collisions but the inner loop is slower because it requires a table lookup.

Brand C was FNV. It’s not quite as good at avoiding collisions but it’s fast.

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# Testing performance

```
34 C% time wc *.cpp > nul
0:00:00.02
35 C% wsl
tcsh 1% time wc *.cpp > /dev/null
0.000u 0.015s 0:00.02 50.0%     0+0k 0+0io 0pf+0w
tcsh 2% which time
time: shell built-in command.
tcsh 3%
```

You might try using the time command built into most shells …

# Problems with time command

1. Not very high resolution.
2. Mixes in process startup time.
3. Hard to measure performance of a compute-bound activity that’s supposed to run fast.

# Solution

You need help from the OS which should provide some kind of high resolution clock, likely in *ticks*.

To measure performance:

1. Snapshot the time at the start.
2. Run some number of iterations of the algorithm under test.
3. Snapshot the time at the end.
4. Calculate elapsed time and compare to a *benchmark*.

Here's how I tested your performance on the best subsequence homework.

```
int main( int argc, char **argv )
   {

   int64_t timingLoopCount = atol( argv[ 0 ] );
   double perfLimit = atof( argv[ 1 ] );

   // Find out how fast the submitted algorithm runs runs.

   int64_t knownGoodTicks =
         TestPerformance( KnownGoodBestSubsequence, timingLoopCount ),

      testTicks
         TestPerformance( MostPositiveSubsequence, timingLoopCount );

   bool perfFailure = testTicks > knownGoodTicks*perfLimit;

   cout << "Performance is " <<
      ( perfFailure ? "slow" : "okay" ) << endl;
   }
```

```
int64_t TestPerformance(
      int subsequence( const int a[ ], const int n, int &L, int &R ),
      int64_t timingLoopCount )
   {
   chrono::high_resolution_clock::time_point start =
      chrono::high_resolution_clock::now( );

   for ( int i = 0;  i < TestSetSize;  i++ )
      {
      int left, right;
      for ( int j = 0; j < timingLoopCount;  j++ )
         subsequence( TestSet[ i ].sequence, TestSet[ i ].n, left, right );
      }

   chrono::high_resolution_clock::time_point finish =
      chrono::high_resolution_clock::now( );
   chrono::high_resolution_clock::duration delta = finish - start;
   return delta.count( );
   }
```

Here's the timer class I gave you in HW 6 and 7.

```
#pragma once
#include <iostream>
#include <chrono>

class Timer
   {
   private:

      std::chrono::high_resolution_clock::time_point start, finish;

   public:

      void Start( )
         {
         start = std::chrono::high_resolution_clock::now( );
         }

      void Finish( )
         {
         finish = std::chrono::high_resolution_clock::now( );
         }
```

```
    unsigned long Elapsed( )
        {
        return ( finish - start ).count( );
        }

    void PrintElapsed( )
        {
        std::cout << "Elapsed time = " << Elapsed( ) <<
            " ticks" << std::endl << std::endl;
        }
};
```

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# Instrumenting memory usage

Sometimes, you would like to know about your own memory usage.

How many objects do you have on the heap, total size, etc.

Problem is to intercept the new and delete operators.

Yes, you can do that.

```
// Toy program to investigate how memory is used by the STL
// string class.

#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

// Instrument new and delete by overriding the new and delete
// operators (yes, you can do that!) with a wrapper around malloc
// and free, adding a boundary tag to record the size of the memory
// request.

// total reflects the net total amount requested by the user and
// does not include any boundary-tagging overhead.

size_t total;
```

```
void *operator new( size_t size )
   {
   total += size;
   size_t *q = ( size_t * )malloc( size + sizeof( size_t ) ),
      *p = q + 1;
   *q = size;
   cout << size << " bytes allocated at " << p <<
      ", total = " << total << endl;
   return ( void *)p;
   }

void operator delete( void *p )
   {
   size_t *q = ( size_t * )p - 1;
   total -= *q;
   cout << *q << " bytes freed at " << p <<
      ", total = " << total << endl;
   free( ( void * )q );
   };
```

```
int main( )
{
   cout << "1.  starting total on the heap = " << total << endl;
   cout << "    putting a string on the stack" << endl << endl;

   string p;

   cout << endl << "2. sizeof an empty string on the stack = " <<
      sizeof( p ) << endl;
   cout << "   total on the heap = " << total << endl << endl;

   cout << "3.  about to new an empty string" << endl << endl;

   string *s = new string( );

   cout << endl << "4.  empty string has been allocated, total = " <<
      total << endl << endl;

   *s = "hello";

   cout << endl << "5.  string has been set it to hello, total = " <<
      total << endl << endl;
```

```
    *s += " world how are you i am fine";

    cout << endl <<  "6.  added 28 chars, total = " << total <<
       ", about to c_str" << endl << endl;

    cout << "cstr = " << s->c_str( ) << endl;

    cout << endl << "7.  about to delete" << endl << endl;

    delete s;

    cout << endl << "8.  string has been deleting, exiting total = " <<
       total << endl;
}
```

```
51 C% Debug\sizestring.exe | head -20
1.   starting total on the heap = 0
     putting a string on the stack

8 bytes allocated at 00A2FD34, total = 8

2. sizeof an empty string on the stack = 28
   total on the heap = 8

3.   about to new an empty string

28 bytes allocated at 00A259AC, total = 36
8 bytes allocated at 00A2FE84, total = 44

4.   empty string has been allocated, total = 44


5.   string has been set it to hello, total = 44

48 bytes allocated at 00A256A4, total = 92

52 C%
```

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# Failing gracefully

If you're writing a serious application you intend to ship to customers, you have to anticipate *it will fail* in their hands.

If it does, it should *fail gracefully* reporting what happened, freeing up any resources, leaving files in a consistent state, capturing debug information, etc.

# Exceptions

Two meanings:

1. *In C++*, it's something you throw, then catch higher up the call stack. You can't catch anything you didn't throw.
2. *To the operating system*, it's a hardware interrupt due to a null pointer deref (a seg fault), a floating point exception, ^C typed at the keyboard, etc.

```
#include <iostream>
#include <string>
using namespace std;

int main( int argc, char **argv )
   {
   string s;
   while ( cin >> s )
      {
      try
         {
         cout << "stoi( " << s << " ) = " << stoi( s ) << endl;
         }
      catch ( invalid_argument )
         {
         cout << "stoi( " << s << " ) is invalid" << endl;
         }
      catch ( out_of_range )
         {
         cout << "stoi( " << s << " ) is out of range" << endl;
         }

      }
   }
```

```
198 C% cl /EHsc stoi.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.11.25508.2 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

stoi.cpp
Microsoft (R) Incremental Linker Version 14.11.25508.2
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:stoi.exe
stoi.obj
199 C% stoi.exe
5
stoi( 5 ) = 5
hello
stoi( hello ) is invalid
124124124123412341234123412341234
stoi( 124124124123412341234123412341234 ) is out of range
^Z
```

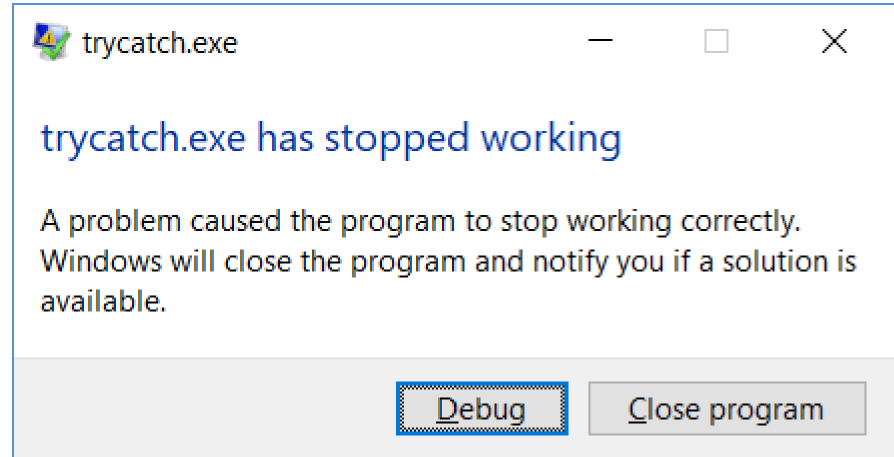
```cpp
// trycatch.cpp

// Cannot catch a null pointer deref with C++ try/catch.

// Nicole Hamilton nham@umich.edu

#include <iostream>
#include <cassert>
#include <stdexcept>
using namespace std;

int main( void )
   {
   int *p = 0;
   try
      {
      *p = 5;
      }
   catch ( ... )
      {
      cout << "Null pointer encountered" << endl;
      }
   }
```



```
// throw.cpp

// You could protect a possible null pointer deref
// with a test and a throw, but it's not realistic to do
// to all your derefs.

// Nicole Hamilton nham@umich.edu

#include <iostream>
using namespace std;
```

You could try to protect every pointer refer with a test and a throw, but that’s just not realistic.

```
int main( void )
    {
    int *p = 0;
    struct NullPointerError
        {
        int **BadPointer;
        NullPointerError( int **p )
            {
            BadPointer = p;
            }
        };

    try
        {
        if ( !p )
            throw NullPointerError( &p );
        *p = 5;
        }
    catch ( NullPointerError &e )
        {
        cout << "Null pointer at " << e.BadPointer <<" encountered" << endl;
        }
    }
```

```
217 C% cl /EHsc throw.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.11.25508.2 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

throw.cpp
Microsoft (R) Incremental Linker Version 14.11.25508.2
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:throw.exe
throw.obj
218 C% throw
Null pointer at 00000033E5CFFE40 encountered
219 C%
```

You really need OS support.

This is the one area where Windows is better than Linux.

Windows added there own keywords, `__try` and `__except`, to the C++ language.

```cpp
// tryexcept.cpp

// Simple Windows __try/__except example, intercepting
// a null pointer dereference using Microsoft try-except
// extension to C and C++.  Compile with /EHsc.

// Choices for the __except filter expression value are:
//
// EXCEPTION_CONTINUE_EXECUTION = -1
// EXCEPTION_CONTINUE_SEARCH    =  0
// EXCEPTION_EXECUTE_HANDLER    =  1

// Nicole Hamilton  nham@umich.edu

#include <windows.h>
#include <iostream>
#include <cassert>
#include <stdexcept>
using namespace std;
```

```
int main( void )
   {
   int *p = nullptr;

   __try
      {
      *p = 5;
      }

   __except ( EXCEPTION_EXECUTE_HANDLER ) // Intercept anything
      {
      cout << "Null pointer dereference encountered" << endl;
      }
   }
```

```
175 C% cl /EHsc tryexcept.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.11.25508.2 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

tryexcept.cpp
Microsoft (R) Incremental Linker Version 14.11.25508.2
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:tryexcept.exe
tryexcept.obj
176 C% tryexcept.exe
Null pointer dereference encountered
177 C%
```

Windows can also give you additional information about the exception.

```
// tryexceptfilter.cpp

// A second Windows try/except example, intercepting
// a null pointer dereference using Microsoft try-except
// extension to C and C++ with an exception filter.
//
// Must be compiled with Microsoft Visual C++ /EHsc options.

// Nicole Hamilton  nham@umich.edu

#include <windows.h>
#include <iostream>
#include <cassert>
#include <stdexcept>
using namespace std;

int ExceptionFilter( DWORD exceptionCode,
      EXCEPTION_POINTERS *exceptionPointers )
   {
   cerr << "Exception code = " << hex << exceptionCode << endl;
   return EXCEPTION_EXECUTE_HANDLER;
   }
```

```
int main( void )
   {
   // In this example, we'll filter the exception, using
   // GetExceptionCode and GetExceptionInformation to
   // retrieve information about what happened.
   //
   // GetExceptionCode and GetExceptioninformation can
   // only be called from within the __except expression,
   // not from a subroutine.

   int *p = nullptr;

   __try
      {
      *p = 5;
      }
   __except ( ExceptionFilter( GetExceptionCode( ),
         GetExceptionInformation( )  ) )
      {
      cout << "Null pointer dereference encountered" << endl;
      }
   }
```

```
157 C% cl /EHsc tryexceptfilter.cpp
Microsoft (R) C/C++ Optimizing Compiler Version 19.11.25508.2 for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

tryexceptfilter.cpp
Microsoft (R) Incremental Linker Version 14.11.25508.2
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:tryexceptfilter.exe
tryexceptfilter.obj
158 C% tryexceptfilter.exe
Exception code = c0000005
Null pointer dereference encountered
```

Linux uses more complex “signal handler” model.

If invoked, the signal handler “long jumps” back to earlier place in your code, unwinding the stack.

But note!  Destructors are not run on anything popped off the stack by long jump.

```cpp
// LinuxSignalHandler.cpp

// Simple demonstration of how to use the Linux sigaction system call
// along with setjmp and longjmp to intercept a segfault (or any other
// exception.)

// Nicole Hamilton  nham@umich.edu

#include <unistd.h>
#include <setjmp.h>
#include <string.h>
#include <iostream>
using namespace std;

bool errorEncountered = false;
jmp_buf longjmpBuffer;
```

```
void SignalHandler( int signal, siginfo_t *info, void *context )
   {
   cerr << "Signal " << signal << " encountered." << endl;

   // Indicate that an error occurred so we don't go down
   // the same path again when we return.

   errorEncountered = true;

   // longjmp back to just before we entered the block
   // failed, returning the signal number.

   longjmp( longjmpBuffer, signal );
   }
```

```
int main( int argc, char **argv )
   {
   // Create a sigaction structure describing how we'd
   // like to intercept signals.

   struct sigaction action;
   memset( &action, 0, sizeof( action ) );
   action.sa_sigaction = SignalHandler;
   action.sa_flags = SA_SIGINFO;

   // Install the signal handler for seg faults.  (We have
   // do this for each signal we'd like to intercept.)

   int sigactionResult = sigaction( SIGSEGV, &action, nullptr );

   // Before entering suspect code, capture a convenient
   // spot to return to.  If we encounter an error, this
   // is where we'll pop out again.  (Think Groundhog Day
   // or Butterfly Effect.)
   //
   // The first time we setjmp, it returns 0.
   // But if we pop back out because of a longjmp,
   // it'll be with the value passed in the longjmp.

   int setjmpResult = setjmp( longjmpBuffer );
```

```
if ( setjmpResult == 0 )
   {
   // Trying for the first time.
   cout << "About to do the nullptr deref." << endl;
   int *p = nullptr;
   *p = 5;
   cout << "Never reached." << endl;
   }
else
   cout << "Failed, error = " << setjmpResult << endl;
}
```

```
181 C% g++ LinuxSignalHandler.cpp -o LinuxSignalHandler
182 C% LinuxSignalHandler.exe
About to do the nullptr deref.
Signal 11 encountered.
Failed, error = 11
183 C%
```

# Agenda

1. Course details.
2. Some hashing results.
3. Testing performance.
4. Instrumenting memory usage.
5. Exception handling.
6. The shared reader problem.

# Basic problem

1. You have object in memory you would like to share between threads.
2. It doesn’t change very often but it gets read a lot.
3. Lots of threads can share access that object if they’re all just reading it.
4. But if any thread wants to write to it, it must lock out all the other threads while it does that.

# Multiple reader / single writer problem

I’m going to ask you to try solving it.

It shows up on whiteboard interviews.

1. We’ll assume some simple mutex and signaling facilities mapped to the OS.
2. I’ve give you a chance to invent your own solution. (Sorry, no autograder.)
3. I’ll show you some known solutions.

```
// Assume some basic Mutex and Signal mechanisms provided
// by the OS that we might wrapper as follows.

class Mutex
   {
   private:
      // ... a handle from the OS
   public:
      // Simple mutual exclusion.
      Take( );
      Release( );
      Mutex( );
      ~Mutex( );
   };
```

```cpp
class Signal
  {
  private:
     // ... a handle from the OS
  public:
     // Simple signaling mechanism that can be set (true)
     // reset (false).

     // Wait for the signal to be set.
     // If it's not set, you sleep until it is.
     // Waiting for signal to be set does not reset it.
     // Other waiting threads will also wake up so long
     // as the signal remains set.

     void Wait( );

     // Set or reset it.
     void Set( );
     void Reset( );

     Signal( bool initialState = false );
     ~Signal( );
  };
```

```
// Here is the multiple reader / single writer interface
// to be built.

class SharedReader
   {
   public:
      void ReadLock( ) = 0;
      void ReleaseReadLock( ) = 0;
      void WriteLock( ) = 0;
      void ReleaseWriteLock( ) = 0;
   };
```

```cpp
class Mutex
   {
   public:
      Take( );
      Release( );
      Mutex( );
      ~Mutex( );
   };
```

```cpp
class SharedReader
   {
   public:
      void ReadLock( ) = 0;
      void ReleaseReadLock( ) = 0;
      void WriteLock( ) = 0;
      void ReleaseWriteLock( ) = 0;
   };
```

```cpp
class Signal
   {
   public:
      void Wait( );
      void Set( );
      void Reset( );
      Signal( bool initialState = false );
      ~Signal( );
   };
```

```
class Mutex
   {
   public:
      Take( );
      Release( );
      Mutex( );
      ~Mutex( );
   };
```

```
class SharedReader
   {
   public:
      void ReadLock( ) = 0;
      void ReleaseReadLock( ) = 0;
      void WriteLock( ) = 0;
      void ReleaseWriteLock( ) = 0;
   };
```

```
class Signal
   {
   public:
      void Wait( );
      void Set( );
      void Reset( );
      Signal( bool initialState = false );
      ~Signal( );
   };
```

Stop here. I want to try to solve the shared reader problem here in class.

```
// This is the lock I created for my C shell in 1987,
// rewritten in modern C++ with OS abstractions.

// Assumes contention is low enough that writers are
// not starved (which was true in my C shell.)

// Requires readers to take and immediately release a
// single lock to acquire or free the resource.
// Writers simply take the lock to acquire and release
// to free.
```

```
class HamiltonLock : SharedReader
   {
   private:
      // Mutex to control access to the number of readers.
      Mutex m;
      // Signal when then number of readers goes to zero.
      Signal zeroReaders;
      // Current number of readers
      int readers;

   public:
      void ReadLock( )
         {
         m.Take( );
         readers++;
         zeroReaders.Reset( );
         m.Release( );
         }

      void ReleaseReadLock( )
         {
         m.Take( );
         if (--readers == 0)
            zeroReaders.Set( );
         m.Release( );
         }
```

```
void WriteLock( )
   {
   while ( true )
      {
      m.Take( );
      if ( readers == 0 )
         break;
      m.Release( );
      zeroReaders.Wait( );
      }
   }

ReleaseWriteLock( )
   {
   m.Release( );
   }
```

```
    ReleaseWriteLock( )
       {
       m.Release( );
       }

    HamiltonLock( ) : readers( 0 ), zeroReaders( true )
       {
       }

    ~HamiltonLock( )
       {
       }
};
```

```
// This is a variant on my lock created by Jordan Zimmerman
// in 1999 that mitigates writer starvation by preventing
// new readers from entering if there is a writer waiting.

// Readers take and release two locks to acquire.
// Writers take two locks and release one to acquire.
```

```
class ZimmermanLock : SharedReadInterface
   {
   private:
      // Mutex to control access to the number of readers.
      Mutex m;

      // Mutex added to prevent new readers from entering
      // when writers are waiting.
      Mutex readlock;

      // Signal when then number of readers goes to zero.
      Signal zeroReaders;

      // Current number of readers
      int readers;
```

```
public:
   void ReadLock( )
      {
      readlock.Take( );
      m.Take( );
      readers++;
      zeroReaders.Reset( );
      m.Release( );
      readlock.Release( );
      }

   void ReleaseReadLock( )
      {
      m.Take( );
      if ( --readers == 0 )
         zeroReaders.Set( );
      m.Release( );
      }
```

```
void WriteLock( )
    {
    readlock.Take( );
    while ( true )
        {
        m.Take( );
        if ( readers == 0 )
            break;
        m.Release( );
        zeroReaders.Wait( );
        }
    readlock.Release( );
    }

void ReleaseWriteLock( )
    {
    m.Release( );
    }
```

```
    ZimmermanLock( ) : readers( 0 ), zeroReaders( true )
        {
        }
    ~ZimmermanLock( )
        {
        }
};
```

```
// The 1997 GNU shared reader lock using cthreads
// and conditions rewritten as a modern C++ class.

// This lock refuses new readers if writers are waiting
// but requires both readers and writers to take two locks
// and immediately release one to acquire.

class GNULock : SharedReader
   {
   private:
      Mutex master;
      Condition wakeup;
      int readers,
         writersWaiting,
         readersWaiting;
```

```
public:
   void ReadLock( )
      {
      master.Take( );
      if ( reader == -1 || writersWaiting )
         {
         readersWaiting++;
         do
            ConditionWait( wakeup, master );
         while (; readers == -1 || writersWaiting )
         }
      reader++;
      master.Release( );
      }
```

```
void WriteLock( )
   {
   master.Take( );
   if ( readers )
      {
      writersWaiting++;
      do
         {
         // Condition wait, atomically giving up the
         // mutex and waiting for wakeup signal, then
         // atomically taking the mutex.
         ConditionWait( wakeup, master );
         }
      while ( readers == -1 || writersWaiting );
      readersWaiting--;
      }
   readers++;
   master.Release( );
   }
```

```
void ReleaseReadLock( )
   {
   master.Take( );
   readers--;
   if ( readersWaiting || writersWaiting )
      // Broadcast the wakeup signal.
      ConditionBroadcast( wakeup );
   master.Release( );
   }

void ReleaseWriteLock( )
   {
   master.Take( );
   readers = 0;
   if ( readersWaiting || writersWaiting )
      // Broadcast the wakeup signal.
      ConditionBroadcast( wakeup );
   master.Release( );
   }
```

```
GNULock( ) : readers( 0 ), readersWaiting( 0 ),
      writersWaiting( 0
   {
   }

~GNULock( );
};
```