Synchronization V

Dmitri Loguinov
Texas A&M University

March 1, 2016
**Homework #2**

- **Suggestion:** develop incrementally from hw #1
  - 2a: Introduce CC 2.0 batching (push/pop up to 10K rooms, send them in one message), but keep the rest
  - Confirm correctness; run benchmarks for report question 2
  - 2b: Replace D with bit hash table; confirm result matches 2a
  - 2c: Replace U with custom queue (single push/pop); confirm result matches 2a-2b
  - 2d: Introduce batch-mode push/pop; confirm result
  - 2e: Optimize synchronization; confirm result

- Make sure to print commas in large numbers
- Remember to benchmark on beefybox
Chapter 5: Roadmap

5.1 Concurrency
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer
Messages

• Messages are discrete chunks of information exchanged between processes
  - This form of IPC is often used between different hosts
• Three main types
  - Pipes (one-to-one)
  - Mailslots (one-to-many among hosts in the active directory domain)
  - Sockets (TCP/IP)

• In general form, message consists of fixed header and some payload
• Header may specify
  - Version and protocol #
  - Message length, type, various attributes
  - Status and error conditions
• Already studied enough in homework #1
Chapter 5: Roadmap

5.1 Concurrency
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer
Reader-Writer (RW)

- RW is another famous synchronization problem
- Assume a shared object that is accessed by M readers and K writers in parallel
- **Example**: suppose hw#1 restricted robot MOVE commands to only adjacent rooms
  - This requires construction of a global graph G as new edges are being discovered from the threads (writer portion)
  - To make a move, each thread has to plot a route to the new location along the shortest path in G (reader portion)
- Any number of readers may read concurrently
  - However, writers need **exclusive** access to the object (i.e., must mutex against all readers and other writers)
**Reader-Writer**

- **Q:** Based on your intuition, do readers or writers usually access the object more frequently?

- **First stab at the problem:**
  - **RW 1.0**
    ```cpp
    Reader::GoRead () {
      mutexRcount.Lock();
      // first reader blocks writers
      if (readerCount == 0)
        semaW.Wait();
      readerCount ++;
      mutexRcount.Unlock();

      // read object
      mutexRcount.Lock();
      readerCount--;
      // last reader unblocks writers
      if (readerCount == 0)
        semaW.Release();
      mutexRcount.Unlock();
    }
    ```

- **Infinite stream of readers, then what?**
  - Writers never get access

- **RW 1.0 gives readers priority and starves writers**
Reader-Writer

- Another policy is to let the OS load-balance the order in which readers and writers enter the critical section
  - RW 1.1

```c
Reader::GoRead () {
    semaWriterPending.Wait();
    semaWriterPending.Release();
    mutexRcount.Lock();
    // first reader blocks writers
    if (readerCount == 0)
        semaW.Wait();
    readerCount ++;
    mutexRcount.Unlock();
    // read object
    mutexRcount.Lock();
    readerCount--;
    // last reader unblocks writers
    if (readerCount == 0)
        semaW.Release();
    mutexRcount.Unlock();
}
```

```c
Writer::GoWrite () {
    semaWriterPending.Wait();
    semaW.Wait();
    // write object
    semaW.Release();
    semaWriterPending.Release();
}
```

- Serves readers/writers in FIFO order if kernel mutex is fair
- What if 100x more readers than writers?
Reader-Writer

- Final policy: writers have absolute priority
  - Given a pending writer, no reader may enter
  - RW 1.2

```cpp
Reader::GoRead () {
    semaWriterPending.Wait();
    semaWriterPending.Release();
    mutexRcount.Lock(); // first reader blocks writers
    if (readerCount++ == 0)
        semaW.Wait();
    mutexRcount.Unlock();
    // read object
    mutexRcount.Lock(); // last reader unblocks writers
    if (--readerCount == 0)
        semaW.Release();
    mutexRcount.Unlock();
}
```

```cpp
Writer::GoWrite () {
    mutexWcount.Lock();
    if (writerCount++ == 0)
        semaWriterPending.Wait();
    mutexWcount.Unlock();
    semaW.Wait(); // write object
    semaW.Release();
    mutexWcount.Lock();
    if (--writerCount == 0)
        semaWriterPending.Release();
    mutexWcount.Unlock();
}
```

- Works fine except first writer still must compete

OS chooses between one writer and M readers
Reader-Writer

• To ensure priority for the first writer, need to prevent readers from competing for semaWriterPending
  – RW 1.3

```cpp
Reader::GoRead () {
    mutexDontCompete.Lock();
    semaWriterPending.Wait();
    mutexRcount.Lock();
    // first reader blocks writers
    if (readerCount++ == 0)
        semaW.Wait();
    mutexRcount.Unlock();
    semaWriterPending.Release();
    mutexDontCompete.Unlock();
    // read object
    mutexRcount.Lock();
    // last reader unblocks writers
    if (--readerCount == 0)
        semaR.Wait();
    mutexRcount.Unlock();
}
```

```cpp
Writer::GoWrite () {
    mutexWcount.Lock();
    if (writerCount++ == 0)
        semaWriterPending.Wait();
    mutexWcount.Unlock();
    semaW.Wait();
    // write object
    semaW.Release();
    mutexWcount.Lock();
    if (--writerCount == 0)
        semaWriterPending.Release();
    mutexWcount.Unlock();
}
```

• Textbook solution
  – Works even if semaphore is unfair
• Elimination of one mutex and some rearrangement of locks is possible to make things simpler

- RW 1.4

```cpp
Reader::GoRead () {
    mutexRcount.Lock();
    semaWriterPending.Wait();
    semaWriterPending.Release();
    // pending writer gets unblocked here
    if (readerCount++ == 0)
        // first reader blocks writers
        semaW.Wait();
    mutexRcount.Unlock();

    // read object
    mutexRcount.Lock();
    // last reader unblocks writers
    if (--readerCount == 0)
        semaW.Release();
    mutexRcount.Unlock();
}

Writer::GoWrite () {
    mutexWcount.Lock();
    if (writerCount++ == 0)
        semaWriterPending.Wait();
    mutexWcount.Unlock();
    semaW.Wait();
    // write object
    semaW.Release();
    mutexWcount.Lock();
    if (--writerCount == 0)
        semaWriterPending.Release();
    mutexWcount.Unlock();
}
```

• Of course now readers can be starved…
Chapter 5: Roadmap

5.1 Concurrency
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer

Performance
Windows APIs

- GetCurrentProcess() and GetCurrentProcessId()
  - Return a handle and PID, respectively
- EnumProcesses(), OpenProcess()
  - Enumerates PIDs in the system, opens access to them
- TerminateProcess() kills another process by its handle
  - ExitProcess() voluntarily quits (similar to C-style exit())
- GetProcessTimes(): see hw2
  - Time spent on the CPU (both in kernel-mode and user-mode)
- Available resources
  - GlobalMemoryStatus(): physical RAM, virtual memory
  - GetActiveProcessorCount(): how many CPUs
- CPU utilization: see cpu.cpp in sample project
**Performance**

- **WaitForSingleObject**
  - Always makes a kernel-mode transition and is painfully slow
  - Mutex, Semaphore, Event all rely on it
- **Faster mutex is CRITICAL_SECTION (CS)**
  - It busy-spins for a fixed number of CPU cycles in user mode
  - If unsuccessful, gives up and calls kernel Mutex
- **While kernel objects can be used between processes, CS can only be used between threads within a process**
  - Internally it uses interlocked exchange to spin
- **Drawback to CS: must sleep-spin if waiting for other events to occur**

```cpp
CRITICAL_SECTION cs;
InitializeCriticalSection (&cs);
// mutex.Lock()
EnterCriticalSection (&cs);
// mutex.Unlock()
LeaveCriticalSection (&cs);
```
Performance

• Condition variables emulate Mesa monitor functionality
  – In performance, similar to CS (i.e., spins in user mode)
  – Secret (monitor) mutex is explicit pointer to some CS
• PC 3.0 that actually works in Windows

```
CONDITION_VARIABLE cv;
InitializeConditionVariable (&cv);
```

```
pcQueue::push (Item x) {
  EnterCriticalSection (&cs);
  while ( Q.isFull () )
    SleepConditionVariable (&cvNotFull, &cs, ...);
  Q.add (x);
  LeaveCriticalSection (&cs);
  WakeConditionVariable (&cvNotEmpty);
}
```

• Slim RW locks
  – AcquireSRWLockShared (reader)
  – AcquireSRWLockExclusive (writer)

pop() is similar
Example 1: compute $\pi$ in a Monte Carlo simulation

- Generate $N$ random points in 1x1 square and compute the fraction of them that falls into unit circle at the origin
- Probability to hit the red circle?

This probability is the visible area of the circle divided by the area of the square (i.e., 1)

- Quarter of a circle gives us $\pi/4$

```c
DWORD WINAPI ThreadPi (LONG *hitCircle) {
    for (int i=0; i < ITER; i++) {
        // uniform in [0,1]
        x = rand.Uniform(); y = rand.Uniform();
        if (x*x + y*y < 1)
            IncrementSync (hitCircle);
    }
}

main () {
    // run N ThreadPi() threads
    // wait to finish
    double pi =
        4*hitCircle/ITER/nThreads;
}
```
Performance

• Six-core AMD Phenom II X6, 2.8 GHz
• Two modes of operation
  – No affinity set (threads run on the next available core)
  – Each thread is permanently bound to one of the 6 cores
• Total k threads
• The basic kernel Mutex
  – $\pi \approx 3.13$
  – CPU $\approx 16\%$
  – Requires 2 kernel-mode switches per increment
  – Runs almost twice as slow with 20K threads

```
IncrementSync (LONG *hitCircle) {
    WaitForSingleObject (mutex, INFINITE);
    (*hitCircle) ++;
    ReleaseMutex (mutex);
}
```

<table>
<thead>
<tr>
<th></th>
<th>$k = 60$</th>
<th>$k = 20K$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all cores</td>
<td>same core</td>
</tr>
<tr>
<td></td>
<td>384K/s</td>
<td>447K/s</td>
</tr>
</tbody>
</table>

```c
SetThreadAffinityMask (GetCurrentThread(),
1 << (threadID % nCPUs));
```
• AtomicSwap
  - \( \pi \approx 3.1405 \)
  - CPU = 100% (locks up the computer)
  - Unable to start more than 7K threads since the CPU is constantly busy

• AtomicSwap and yield
  - When cannot obtain mutex, yield to other threads if they are ready to run
  - \( \pi \approx 3.1412 \)
  - CPU = 100%, but computer much more responsive

```
LONG taken = 0;   // shared flag
IncrementSync (LONG *hitCircle) {
    while (InterlockedExchange (&taken, 1) == 1)
    {
        (*hitCircle) ++;
        taken = 0;
    }
}
```

<table>
<thead>
<tr>
<th>k = 60</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cores</td>
<td>same core</td>
</tr>
<tr>
<td>448K/s</td>
<td>485K/s</td>
</tr>
</tbody>
</table>

```
LONG taken = 0;   // shared flag
IncrementSync (LONG *hitCircle) {
    while (InterlockedExchange (&taken, 1) == 1)
    {
        SwitchToThread();
        (*hitCircle) ++;
        taken = 0;
    }
}
```

<table>
<thead>
<tr>
<th>k = 60</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cores</td>
<td>same core</td>
</tr>
<tr>
<td>6.8M/s</td>
<td>6.8M/s</td>
</tr>
</tbody>
</table>
Performance

- CRITICAL_SECTION
  - $\pi \approx 3.1417$
  - CPU = 36%
- Interlocked increment
  - $\pi \approx 3.1416$
  - CPU = 100%
  - Fastest approach so far
- No sync (dumb approach)
  - CPU = 100%
  - Some of the concurrent updates are lost due to cache sync problems

```
CRITICAL_SECTION cs;
IncrementSync (LONG *hitCircle) {
  EnterCriticalSection (&cs);
  (*hitCircle) ++;
  LeaveCriticalSection(&cs);
}
```

<table>
<thead>
<tr>
<th>k = 60</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cores</td>
<td>same core</td>
</tr>
<tr>
<td>6.9M/s</td>
<td>15.9M/s</td>
</tr>
</tbody>
</table>

```
IncrementSync (LONG *hitCircle) {
  InterLockedIncrement (hitCircle);
}
```

<table>
<thead>
<tr>
<th>k = 60</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cores</td>
<td>same core</td>
</tr>
<tr>
<td>19.4M/s</td>
<td>19.2M/s</td>
</tr>
</tbody>
</table>

```
IncrementSync (LONG *hitCircle) {
  (*hitCircle)++; 
}
```

<table>
<thead>
<tr>
<th>k = 60</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>all cores</td>
<td>same core</td>
</tr>
<tr>
<td>25.5M/s</td>
<td>19.9M/s</td>
</tr>
</tbody>
</table>

\[ \pi \approx 1.21 \] \[ \pi \approx 1.03 \] \[ \pi \approx 0.96 \] \[ \pi \approx 1.33 \]
Performance

• No sync (correct approach)
  – \( \pi \approx 3.1415 \)
  – 202M/s, 100% CPU, bottlenecked by rand.Uniform()

• Lessons
  – Kernel mutex is slow, should be avoided
  – CRITICAL_SECTION is the best general mutex
  – Interlocked operations are best for 1-line critical sections
  – Affinity mask makes a big difference in some cases

• If you can write code only using local variables and synchronize rarely, it can be 1000x faster than kernel mutex and 10x faster than Interlocked

DWORD WINAPI ThreadPi (LONG *hitCircle) {
  LONG counter = 0;
  for (int i=0; i < ITER; i++) {
    // uniform in [0,1]
    x = rand.Uniform(); y = rand.Uniform();
    if (x*x + y*y < 1)
      counter ++;
  }
  InterlockedAdd (hitCircle, counter);
}
Performance

- **Example 2**: unbounded producer-consumer
  - Producer batch = 1
    - \(Q\text{.size()} \leq 1\)
  - Producer batch = 10
    - \(Q\text{.size()} \to \infty\)
  - PC 1.1
    - Busy spins to enter
    - CPU is high, mostly spent in the kernel
    - Worst method in our comparison

```c
int batch; // PC 1.1
while (true) {
  while (true) {
    WaitForSingleObject(mutex, INFINITE);
    if (Q.size() > 0) {
      x = Q.pop();
      break;
    }
    ReleaseMutex(mutex);
  }
  ReleaseMutex(mutex);
  // do some work
  WaitForSingleObject(mutex, INFINITE);
  for (int i=0; i < batch; i++)
    Q.add(i+x);
  ReleaseMutex(mutex);
}
```

<table>
<thead>
<tr>
<th>k = 600</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch=1</td>
<td>batch=10</td>
</tr>
<tr>
<td>660/sec</td>
<td>187K/sec</td>
</tr>
</tbody>
</table>
Performance

• PC 1.2 sleeps on semaphore
  - CPU = 20%
• PC 1.4 releases semaphore in bulk
  - Speed-up by 40% over PC 1.2 with batch=10
  - CPU = 20%

<table>
<thead>
<tr>
<th></th>
<th>PC 1.2</th>
<th></th>
<th>PC 1.4 (hw1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>k = 600</td>
<td>k = 20K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>batch=1</td>
<td>batch=10</td>
<td>batch=1</td>
<td>batch=10</td>
</tr>
<tr>
<td>275K/s</td>
<td>130K/s</td>
<td>223K/s</td>
<td>112K</td>
</tr>
</tbody>
</table>

```c
int batch; // PC 1.2
while (true) {
    WaitForSingleObject(sema, INFINITE);
    WaitForSingleObject(mutex, INFINITE);
    x = Q.pop ();
    ReleaseMutex (mutex);

    WaitForSingleObject(mutex, INFINITE);
    for (int i=0; i < batch; i++) {
        Q.add (i+x);
        ReleaseSemaphore(sema, 1, NULL);
    }
    ReleaseMutex (mutex);
}
```

```c
int batch; // PC 1.4
while (true) {
    WaitForSingleObject(sema, INFINITE);
    WaitForSingleObject(mutex, INFINITE);
    x = Q.pop ();
    ReleaseMutex (mutex);

    WaitForSingleObject(mutex, INFINITE);
    for (int i=0; i < batch; i++) {
        Q.add (i+x);
        ReleaseSemaphore(sema, batch, NULL);
    }
    ReleaseMutex (mutex);
}
```
**Performance**

- **PC 2.1**
  - Adds `WaitAll`
  - CPU = 100%
  - Horrible performance
  - PC 3.2-3.3 similar

- **Back to 1.4**
  - Over 450% faster than 1.4 for `batch=10`
  - CPU = 100%

```c
HANDLE arr[] = {sema, mutex}; // PC 2.1
while (true) {
    WaitForMultipleObjects(2, arr, true, INFINITE);
    x = Q.pop ();
    ReleaseMutex (mutex);

    WaitForSingleObject(mutex, INFINITE);
    for (int i=0; i < batch; i++)
        Q.add (i+x);
    ReleaseMutex (mutex); ReleaseSemaphore(sema,batch,NULL);
}
```

```c
int batch; // PC 1.4 with CS
while (true) {
    WaitForSingleObject(sema, INFINITE);
    EnterCriticalSection (&cs);
    x = Q.pop ();
    LeaveCriticalSection (&cs);

    EnterCriticalSection (&cs);
    for (int i=0; i < batch; i++)
        Q.add (i+x);
    LeaveCriticalSection (&cs);
    ReleaseSemaphore(sema,batch,NULL);
}
```

<table>
<thead>
<tr>
<th></th>
<th>k = 600</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>batch=1</code></td>
<td>27K/s</td>
<td>worse</td>
</tr>
<tr>
<td><code>batch=10</code></td>
<td>27K/s</td>
<td>worse</td>
</tr>
</tbody>
</table>

**PC 1.4 w/CS**

<table>
<thead>
<tr>
<th></th>
<th>k = 600</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>batch=1</code></td>
<td>361K/s</td>
<td>280K/s</td>
</tr>
<tr>
<td><code>batch=10</code></td>
<td>850K/s</td>
<td>1.1M/s</td>
</tr>
</tbody>
</table>
Wrap-up

- PC 3.0
  - CPU = 100%
  - Breaks down when Q is persistently small
- PC 3.1
  - Uses kernel events, runs at 450K/s
- PC 3.4
  - CPU = 30%

<table>
<thead>
<tr>
<th>k = 600</th>
<th>k = 20K</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch=1</td>
<td>batch=10</td>
</tr>
<tr>
<td>205K/s</td>
<td>5.9M/s</td>
</tr>
</tbody>
</table>

PC 3.0 (hw2)