CSCE 313-200
Introduction to Computer Systems
Spring 2024

Synchronization VII

Dmitri Loguinov
Texas A&M University

February 28, 2024

Back to Semaphores

- Version 3.0 with auto events / binary semaphores
  - PC 3.1

```
// all events are AUTO (binary semaphore)
pcQueue::push (Item x) {
    mutex.Lock();
    while ( Q.isFull() )
          mutex.Unlock();
          eventNotFull.Wait();
          mutex.Lock();
    Q.add (x);
    if ( !Q.isFull() )
          eventNotFull.Signal();
    eventNotEmpty.Signal();
    mutex.Unlock();
}
```

```
// all events are AUTO (binary semaphore)
Item pcQueue::pop () {
    mutex.Lock();
    while ( Q.isEmpty() )
          mutex.Unlock();
          eventNotEmpty.Wait();
          mutex.Lock();
    x = Q.remove();
    if ( !Q.isEmpty() )
          eventNotEmpty.Signal();
    eventNotFull.Signal();
    mutex.Unlock(); return x;
}
```

- Increments past max, stolen wake-ups are possible
- What if events were manual in the above?
  - Major performance hit: all threads wake up and busy spin on their while loops

Back to Semaphores

- If WaitAll is available, work “theft” can be avoided
  - PC 3.2

```
// all events are AUTO (binary semaphore)
pcQueue::push (Item x) {
    WaitAll (eventNotFull, mutex);
    Q.add (x);
    if ( !Q.isFull () )
          eventNotFull.Signal();
    eventNotEmpty.Signal();
    mutex.Unlock();
}
```

```
// both events are AUTO (binary semaphore)
Item pcQueue::pop () {
    WaitAll (eventNotEmpty, mutex);
    x = Q.remove ();
    if ( !Q.isEmpty() )
          eventNotEmpty.Signal();
    eventNotFull.Signal();
    mutex.Unlock(); return x;
}
```

- Now the same with manual-reset events
  - PC 3.3

```
// all events are MANUAL
pcQueue::push (Item x) {
    WaitAll (eventNotFull, mutex);
    Q.add (x);
    if ( Q.isFull () )
          eventNotFull.Reset();
    eventNotEmpty.Signal();
    mutex.Unlock();
}
```

```
// both events are MANUAL
Item pcQueue::pop () {
    WaitAll (eventNotEmpty, mutex);
    x = Q.remove ();
    if ( Q.isEmpty() )
          eventNotEmpty.Reset();
    eventNotFull.Signal();
    mutex.Unlock(); return x;
}
```

Back to Semaphores

- One more version to consider:
  - PC 3.4

```
pcQueue::push (Item x) {
    mutex.Lock();
    while ( Q.isFull() )
          mutex.Unlock();
          Sleep(DELAY);
          mutex.Lock();

    Q.add (x);

    mutex.Unlock();
}
```

```
Item Queue::pop () {
    mutex.Lock();
    while ( Q.isEmpty() )
          mutex.Unlock();
          Sleep(DELAY);
          mutex.Lock();

    x = Q.pop ();

    mutex.Unlock();
    return x;
}
```

- Probably the simplest approach
  - Arguably inefficient due to sleep-looping
  - May cause starvation for certain threads

Summary

All methods need at least a mutex, but additionally:

- PC 2.0 requires a counting semaphore
  - Ideal textbook solution since it's elegant and simple
  - Does not handle bursty push/pop
- PC 2.1 similar to 2.0, but further requires WaitAll
  - Even more elegant, but same drawbacks as 2.0
  - Does not work with eventQuit
- PC 3.0 requires monitors and condition variables
  - Possible in C++, but not optimal speed
- PC 3.1 requires just a binary semaphore
  - Allows stolen wake-ups, but can handle bursty data easily

Summary (Cont)

- PC 3.2 requires binary semaphore and WaitAll
  - Handles bursty data well, but more elegant than 3.1 and prevents stolen wake-ups
  - Signals unnecessarily if queue is rarely full or empty
- PC 3.3 requires manual events and WaitAll
  - Similar to 3.2, but less signaling when there is work to do
- PC 3.4 requires nothing beyond a mutex
  - Most flexible as threads can perform useful checks (e.g., the quit flag) while being awake, supports batch push/pop
  - Sleep-spinning is seemingly bad, or … is it?
- Ultimately, performance is what really matters
  - We'll consider a few benchmarks next time

Private Heaps

- Memory heaps
  - Normal new/delete ops go to the process heap
  - Internal mutex, slow delete
- Private heap doesn't need to mutex
  - Benchmark with 12 threads on a 6-core system

```
#define ITER   1e7
DWORD __stdcall HeapThread (...) {
    DWORD **arr = new (DWORD *) [ITER];
    for (int i=0; i < ITER; i++)
          arr[i] = new DWORD [1];

    for (int i=0; i < ITER; i++)
          delete arr[i];
}
```

3.3M/s

```
DWORD __stdcall HeapThread (...) {
   HANDLE heap = HeapCreate
          (HEAP_NO_SERIALIZE,
          4 * 1024 * sizeof(DWORD), 0);

   DWORD **arr = new (DWORD *) [ITER];
   for (int i = 0; i < ITER; i++)
          arr[i] = (DWORD*) HeapAlloc
           (heap, HEAP_NO_SERIALIZE,
                    sizeof(DWORD));
   HeapDestroy (heap);
}
```

36M/s

```
DWORD __stdcall HeapThread (...) {
   HANDLE heap = HeapCreate
          (HEAP_NO_SERIALIZE,
          4 * 1024 * sizeof(DWORD), 0);

   DWORD **arr = new (DWORD *) [ITER];
   for (int i=0; i < ITER; i++)
          arr[i] = (DWORD*) HeapAlloc
           (heap, HEAP_NO_SERIALIZE,
                    sizeof(DWORD));

   for (int i=0; i < ITER; i++)
          HeapFree (heap,
              HEAP_NO_SERIALIZE, arr[i]);
}
```

12M/s

Chapter 5: Roadmap

5.1 Concurrency

5.2 Hardware mutex

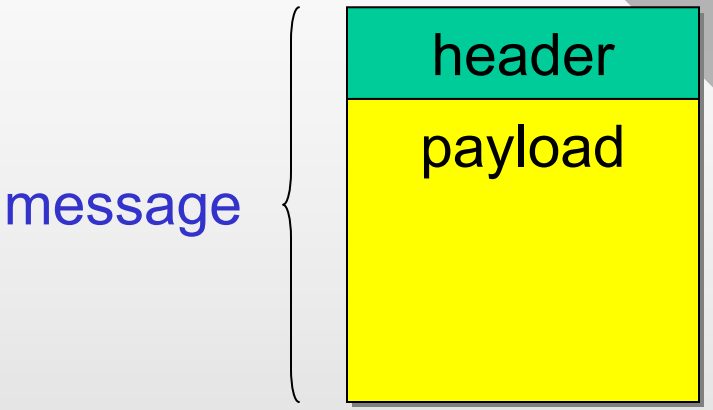
5.3 Semaphores

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
- Where used
  - 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

```
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();
}
```

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

- Infinite stream of readers?
  - Writers never get access
- RW 1.0 gives readers priority and starves writers

Reader-Writer

increasing writer thread priority
may help against being starved

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

```
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();
}
```

```
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

```
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();
}
```

OS chooses between one writer and M readers

```
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

Reader-Writer

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

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

- Textbook solution
  - Works even if semaphore is unfair

Reader-Writer

- What about the next solution that eliminates one lock and rearranges some of the lines
  - RW 1.4

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

- Find a problem at home