CSCE 313-200
Introduction to Computer Systems
Spring 2016

Synchronization II
Dmitri Loguinov
Texas A&M University

February 11, 2016
Chapter 5: Roadmap

5.1 Concurrency
   Appendix A.1
5.2 Hardware mutex
5.3 Semaphores
5.4 Monitors
5.5 Messages
5.6 Reader-Writer
Mutex

• Where to get mutex functionality?
  • Two options
    – Make the kernel do it
    – Implement in user space
  • Techniques are similar with a few exceptions
    – Some may require privileged instructions
• Next, we’ll review classical algorithms and hardware support

• Here, we use global variables for simplicity of explanation
  • Mutex v1.0: naïve

```java
taken = false
Mutex.Lock () {
    while (taken == true) ;
    taken = true // we own mutex
}
// ------------
Mutex.Unlock (){ 
    taken = false
}
```

• Any problems?
Mutex

Main issue:
• Read followed by write is not an atomic operation!
• Two threads arrive simultaneously to mutex
  – Both check and see that taken is false
  – Both proceed inside
• Result
  – Failed mutual exclusion
  – Long hours debugging

• Mutex v2.0: Strict alternation
  – Do not enter until access is granted by other threads

```c
// N = number of threads
turn = 0
Mutex.Lock (i){
    while (turn ! = i) 
        ;  // do nothing
    // someone gave us the turn
}
// ------------
Mutex.Unlock (){ 
    turn = (turn + 1) % N
}
```

• Problems?
Mutex

Drawbacks of Mutex 2.0
- Threads forced to own mutex even if not needed
  - Wait time can be arbitrarily high

Classroom analogy
- No mutex: ask question as soon as ready
  - Keep talking concurrently with instructor and other students asking their questions
- Correct mutex: raise your hand if you have a question
  - Instructor finishes sentence, selects the order in which raised hands are polled
- Mutex 2.0: only person holding a token can ask question
  - When question asked, token is passed to next person
Mutex

- Mutex v3.0
  - Mutex cannot be solved with one variable, try two

```
bool want [2] = {false,false}
Mutex.Lock (i){
    j = 1-i  // other threadID
    want [i] = true
    while (want [j] == true)
        ;      // do nothing
}
// ------------
Mutex.Unlock (i){
    want [i] = false
}
```

- Only one thread can enter
  - But deadlock possible if both want it at same time

- Mutex v3.1
  - If not two, maybe three?
  - Dekker’s algorithm (1965) for 2 threads

```
bool want [2] = {false,false}
int turn = 0   // break ties
Mutex.Lock (i){
    j = 1-i  // other threadID
    want [i] = true
    while (want [j] == true) {
        if (turn == j) {
            want [i] = false
            while (turn == j)
                ;      // do nothing
            want [i] = true
        }
    }
// ------------
Mutex.Unlock (i){
    turn = 1-i
    want [i] = false
}
```
Mutex

- Mutex 3.1 guarantees that only one thread enters
  - Deterministically avoids deadlock and inconsistency

- Only competing threads are given access to mutex
  - Quite efficient

Drawbacks

- Somewhat complex

- Starvation (lack of fairness)
  - No guarantee that each thread eventually enters

Mutex v3.2

- Petersen’s algorithm (1981) for 2 threads

```c
bool want [2] = {false, false}
int turn // break ties

Mutex.Lock (i){
    j = 1-i // other threadID
    want [i] = true
    turn = j // give away turn
    while (want [j] == true
           && turn == j)
        ;  // do nothing
}

// ------------Mutex.Unlock (i){
    want [i] = false
}
```

- Fair, efficient, consistent
Mutex

- Mutex v3.2 without contention

```cpp
bool want[2] = {false, false}
int turn   // break ties
Mutex.Lock(0) {
    want [0] = true
    turn = 1    // give away turn
    while (want [1] == true
           && turn == 1)
        ;
    // owns mutex
}
// ------------
Mutex.Unlock (0){
    want [0] = false
}
```

```cpp
bool want[2] = {false, false}
int turn   // break ties
Mutex.Lock(1) {
    want [1] = true
    turn = 0    // give away turn
    while (want [0] == true
           && turn == 0)
        ;
    // owns mutex
}
// ------------
Mutex.Unlock (1){
    want [1] = false
}
```
Mutex

- Mutex v3.2 with contention

```c
bool want[2] = {false, false};
int turn // break ties
Mutex.Lock(0) {
    want[0] = true
    turn = 1
    while (want[1] == true && turn == 1)
        ;
    // owns mutex
}
// ------------
Mutex.Unlock(0);

bool want[2] = {false, false};
int turn // break ties
Mutex.Lock(1) {
    want[1] = true
    turn = 0
    while (want[0] == true && turn == 0)
        ;
    // owns mutex
}
// ------------
Mutex.Unlock(1);
```

true  want[0]  1  turn  false  want[1]
### Mutex

- **Mutex v3.2 avoiding starvation**

```c
bool want[2] = {false, false};
int turn; // break ties
Mutex.Lock(0) {
    // want[0] = true
    turn = 1;
    while (want[1] == true && turn == 1)
    {
        // owns mutex
    }
} // ---------
Mutex.Unlock(0);

bool want[2] = {false, false};
int turn; // break ties
Mutex.Lock(1) {
    // want[1] = true
    turn = 0;
    while (want[0] == true && turn == 0)
    {
        // owns mutex
    }
} // ---------
Mutex.Unlock(1);
```

<table>
<thead>
<tr>
<th>want[0]</th>
<th>turn</th>
<th>want[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>0</td>
<td>true</td>
</tr>
</tbody>
</table>
Mutex

- Mutex v3.2 with reversed order of want and turn
  - Allows both threads to enter

```c
bool want[2] = {false, false}
int turn    // break ties
Mutex.Lock(0) {
  turn = 1
  want[0] = true
  while (want[1] == true
        && turn == 1)
    ;
  // owns mutex
}
// ------------
Mutex.Unlock (0){
  want[0] = false
}

bool want[2] = {false, false}
int turn    // break ties
Mutex.Lock(1) {
  turn = 0
  want[1] = true
  while (want[0] == true
        && turn == 0)
    ;
  // owns mutex
}
// ------------
Mutex.Unlock (1){
  want[1] = false
}
```
Mutex Summary

Mutex v3.2 on modern computers

• Compiler optimization A
  – Compiler sees that the loop does not change any variables
  – Removes it from code

• Compiler optimization B
  – Variables may be kept in registers for loop duration or order of operations changed

• CPU cache coherency
  – Shared variables stored in L2 caches of different cores

• CPU memory fetch
  – Hardware may reorder read/write operations
  – Major problem for all algorithms:

// intended sequence
write want[i]
read want[j]
read turn

// actual sequence
read want[j]
read turn
write want[i]
Chapter 5: Roadmap

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

- Without CPU support, mutual exclusion is impossible
- One seemingly good approach is to disable interrupts
  - Assembler instructions cli (clear interrupts) and sti (set interrupts)

```assembly
       asm { cli }
// critical section
       asm { sti }
```

- Maybe worked fine in old single-CPU mainframes, but is unsuitable for modern computers
  - Privileged instruction, only the kernel can use
  - Masked interrupts on one CPU do not affect others
  - Cache coherency issues not resolved
**Hardware Mutex**

- A more powerful approach is to employ instructions that lock the memory bus and synchronize caches
  - x86 assembler prefix `lock`

- **Now mutex v4.0**

  ```c
  int AtomicSwap (int *ptr, int val) {
     __asm {
       mov    eax, val
       xchg   eax, [ptr]
       ret    eax
     }
  }
  ```

- `xchg` is always locked, so the keyword is redundant here

- Another low-level primitive is **Compare & Swap (CAS)**
  - Compares the target to some constant, swaps if equal
  - Maps to assembler instruction `CMPXCHG`
Hardware Mutex

- Mutex v4.1 using CAS:
  - Usually slower than AtomicSwap
    - Why use it then?
- Example where AtomicSwap doesn’t work
  - Suppose taken can be 0-2
    - If 0, set it to 1
    - If 1, set to 2; if 2, set to 0
- Windows APIs
  - Several versions: 32-bit, 64-bit, and pointers

```c
taken = 0
Mutex.Lock () {
    want = 0; newValue = 1
    // CAS returns the old value
    while (CAS (&taken, newValue, want) != want)
        ;
    // owns mutex
}
Mutex.Unlock ()
taken = 0;
```

InterlockedExchange = AtomicSwap
InterlockedCompareExchange = CAS
InterlockedIncrement = a++
InterlockedDecrement = a--
InterlockedAdd = a + constant
InterlockedXor = a ^ constant
InterlockedAnd = a & constant
InterlockedOr  = a | constant
InterlockedBitTestAndSet = set bit to 1
InterlockedBitTestAndReset = set bit to 0

all of these use 32-bit destinations
Hardware Mutex

- Mutexes 4.0-4.1 are called spinlocks
- Internally, OS uses them to mutex against itself
  - Tiny critical sections make this acceptable
- At the user level, spinlocks are used rarely
  - Mostly to achieve extreme levels of performance
  - We’ll have some benchmarks later

- Besides spinlocks, one can ask the kernel to block thread until its event is ready
  - Useful for large critical sections and I/O operations
  - Kernel threads can be blocked this way too
- As the event is signaled
  - Threads are unblocked in FIFO order (unless priorities dictate otherwise)
Chapter 5: Roadmap

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

• Perhaps one of the most useful synchronization constructs was invented by Dijkstra in 1965

• **Definition**: semaphore v1.0 is a class shared between threads/processes that admits two atomic operations:

  ```
  class Semaphore1 {
    int s; // current state
    P(); V(); // operations
  }
  ```

  Semaphore1::P() {
    s--
    if (s < 0)
      // block current thread
  }

  Semaphore1::V() {
    s++
    if (s <= 0)
      // unblock one waiting thread
  }

  also called Lock or Wait

  also called Unlock or Release

• This version allows the state to be negative
  - Does not set any limits on its maximum or minimum value
  - Potential overflow issues
Semaphore

• **Semaphore v2.0** avoids incrementing s when there are pending threads and adds an upper bound on s

```cpp
Semaphore2::P() { // inside kernel
    if (s > 0)
        s--;
    else
        t = GetCurrentThread()
        blocked.add (t) // block thread t
}
```

```cpp
Semaphore2::V() { // inside kernel
    if (blocked.size() > 0)
        t = blocked.remove() // unblock thread t
    else
        s = min (s+1, max);
}
```

• Dijkstra defined semaphore 1.0 (abstract concept)
• Windows semaphores are 2.0 (kernel-mode)
  - Unless specified otherwise, assume this type
  - Initial state and max are set during creation
Semaphore

- POSIX semaphore v3.0 does not ensure that both operations P() and V() are atomic
  - Instead, it uses an internal mutex

```
Semaphore3::P() {  // user mode
  m.Lock()
  while (s <= 0)
    m.Unlock()
  sleep
  m.Lock();
  s--
  m.Unlock()
}
```

- Semaphore 3.0 does not enforce any order in which competing threads acquire semaphore
  - Potential for starvation/unfairness
- Inefficient due to sleep-spinning?
Semaphore

- Examples:

```cpp
Semaphore semaX = {15, 15}; // (s,max)
Thread () {
    semaX.Wait();  // P
    // some section
    semaX.Release(); // V
}
```

allows up to 15 concurrent threads in some section

Semaphore semaX = {0, 1}; // (s,max)
Thread1 () {
    semaX.Wait();  // P
}

thread1 waits for thread2 to finish initialization

Semaphore semaX = {0, 1}; // (s,max)
Semaphore semaY = {0, 1}; // (s,max)
Thread1 () {
    // initialize stuff
    semaX.Wait();  // P
    semaY.Release(); // V
}

Semaphore semaX = {0, 1}; // (s,max)
Semaphore semaY = {0, 1}; // (s,max)
Thread2 () {
    // initialize stuff
    semaY.Wait();  // P
    semaX.Release(); // V
}

deadlock

Semaphore semaX = {0, 1}; // (s,max)
Semaphore semaY = {0, 1}; // (s,max)
Thread1 () {
    // initialize stuff
    semaX.Wait();  // P
}
Semaphore semaX = {0, 1}; // (s,max)
Semaphore semaY = {0, 1}; // (s,max)
Thread2 () {
    // initialize stuff
    semaY.Wait();  // P
    semaX.Release(); // V
}
**Semaphore**

- **Examples (cont’d):**

```c
Semaphore semaX = {0, 1}; // (s,max)
Semaphore semaY = {0, 1}; // (s,max)
Thread1 () {
    // initialize stuff
    semaY.Release(); // V
    semaX.Wait(); // P
}
```

```c
Semaphore semaX = {0, 1}; // (s,max)
Semaphore semaY = {0, 1}; // (s,max)
Thread2 () {
    // initialize stuff
    semaX.Release(); // V
    semaY.Wait(); // P
}
```

both threads wait for the other to initialize

- **Most common use of semaphores:** allow entry of \( \leq s \) concurrent threads into some section of the code

- **Definition:** a semaphore is called *binary* if \( \text{max} = 1 \) and *counting* (general) otherwise
Wrap-up

• Definition: a semaphore is called strong if it unblocks threads in FIFO order and weak otherwise

• Semaphore v1.0
  – Not detailed enough to determine

• Semaphore v2.0:
  – If internal data structure List is a FIFO queue, then it is strong

• Some kernels (e.g., Windows) run semaphore queues through the CPU scheduler
  – This makes them weak, but only to the extent of yielding to higher-priority threads
  – Thus, if user threads all have the same priority, their unblocking order relative to each other is approx FIFO

• Semaphore v3.0
  – Weak