CSCE 313-200
Introduction to Computer Systems
Spring 2024

Synchronization II
Dmitri Loguinov
Texas A&M University

February 7, 2024
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

- For now, assume
  - Each C line is atomic
  - No caching
- Use global variables for simplicity of explanation
- Mutex v1.0: naïve

```c
bool 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
- Can we do better?
- Mutex v2.0: Strict alternation
  - Do not enter until access is granted by other threads

```c
// N = number of threads
int turn = 0;
Mutex.Lock (i){
    while (turn != i)
        ;    // do nothing
        // someone gave us the turn
}
// -----------
Mutex.Unlock (){ turn = (turn + 1) % N
```
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
- Mutex 2.0: only person holding a token can ask question
  - When question asked, token is passed to next person
- Correct mutex: raise your hand if you have a question
  - Instructor finishes sentence, selects the order in which raised hands are polled
Mutex v3.0
- Consider just two threads

```c
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
- Need to break ties
- Dekker’s algorithm (1965) for two threads

```c
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
  - Efficient

Drawbacks
- Pretty complex
- Lack of *fairness*: one thread may enter multiple times while the other is waiting

Mutex v3.2
- Petersen’s algorithm (1981) for two 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

```c
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
}

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
}
```

false
want[0]

0
turn

true
want[1]
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){
    want[0] = false
}
```

```c
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){
    want[1] = false
}
```
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){
    want [0] = false
}
```

```c
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){
    want [1] = false
}
```
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 L1/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
write want[i]

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