Software Solutions to Critical Sections

- Critical Sections and Locks
- Buggy Software Implementations
- Peterson's Algorithm
- Practicality Issues:
  - Busy Waiting
  - What about Out-of-Order Memory Operations?

Recall: Critical Sections & Locks

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.
- Easy to do with locks.

inter-task invariant violated

```
Lock lck;
while (TRUE) {
   lck.lock();
   critical section;
   lck.unlock();
   remainder section;
}
```

Recall: Critical Sections & Locks

- Execution of critical section by threads must be mutually exclusive.
- Need protocol to enforce mutual exclusion.
- Easy to do with locks.

```
class Lock {
  public void lock() {???}
  public void unlock() {???}
};
```

```
Lock lck;
while (TRUE) {
  lck.lock();
  critical section;
  lck.unlock();
  remainder section;
}
```

Locks: Software Implementation 1

```
class Lock {
  private int turn = 0;   /* who's turn is it?            */
                          /* Thread 0 goes first.         */
  public void lock() {
    while (turn != self); /* loop until it's your turn   */
                          /* self is the current thread */
  }
  public void unlock() {
    turn = (turn+1) % 2;  /* it's next thread's turn     */
                          /* we have 2 threads           */
  }
};
```

"Works" for 2 threads only!

Locks: Software Implementation 1

Thread 0

```
...
turn == 0?
critical section;
-
-
turn = 1;
BYE!
```

Thread 1

```
-
-
-
turn == 1?
turn == 1?
-
-
turn == 1?;
critical section;
turn = 0;
...
turn == 1?
turn == 1?
```

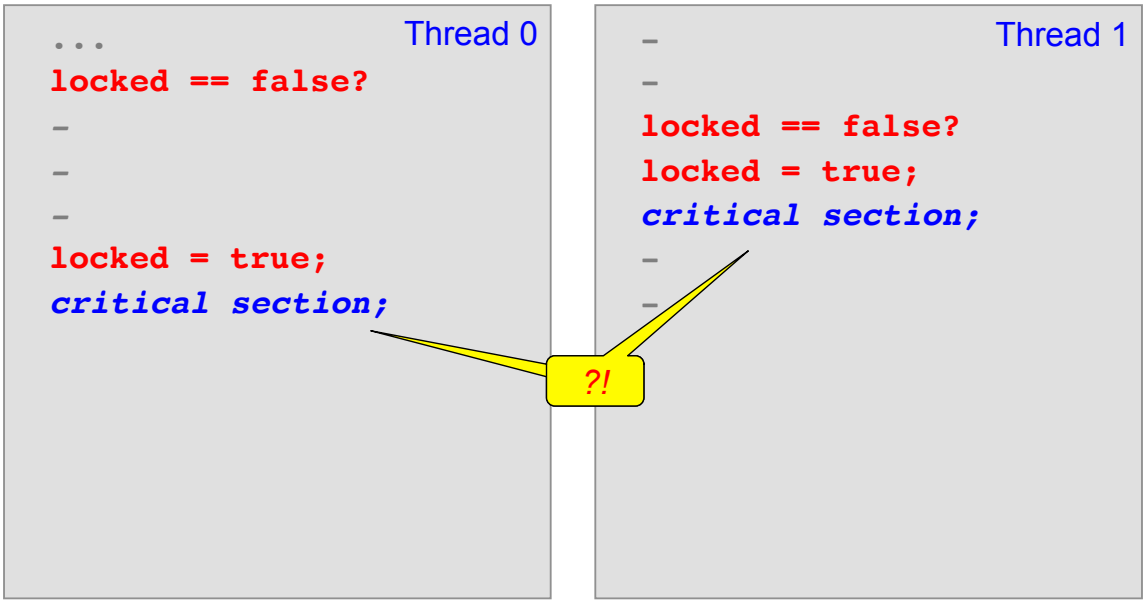
?!

Locks: Software Implementation 2

```
class Lock {
  private bool locked = false; /* locked? */

  public void lock() {
    while (locked);      /* busy loop until not locked */
    locked= true;
  }
  public void unlock() {
    locked = false;      /* mark lock as free          */
  }
};
```

Locks: Software Implementation 2

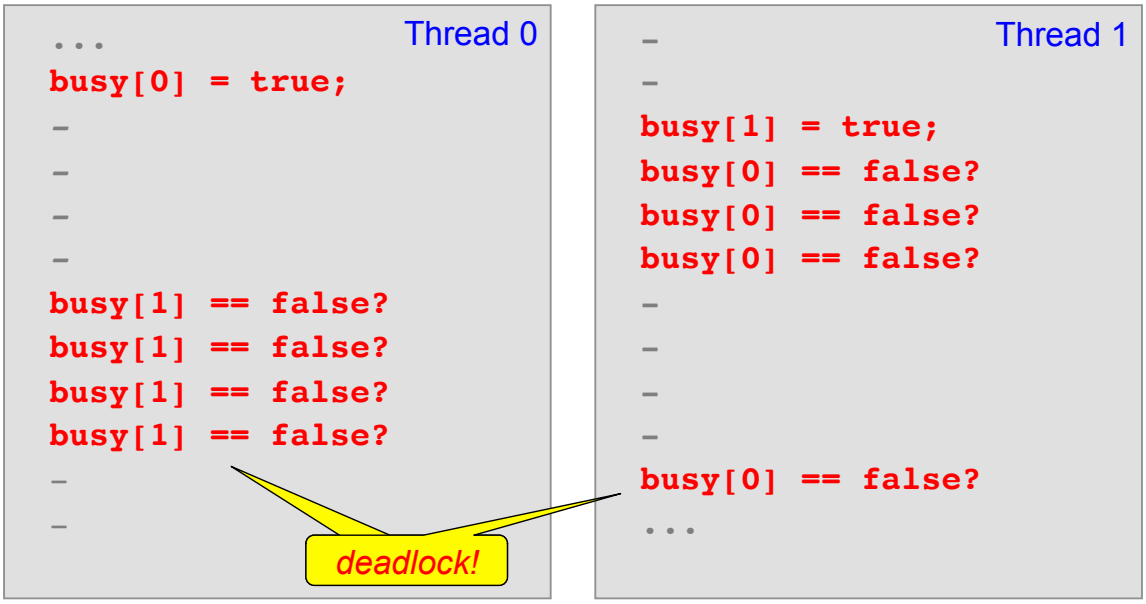


Locks: Software Implementation 3

```
class Lock {
  private bool busy[2];      /* initialize to false     */
  public void lock() {
    busy[self] = true;       /* 'self' is current thread*/
    int other = 1 - self;    /* 'other' is other thread */
    while(busy[other]);      /* busy loop               */
  }
  public void unlock() {
    busy[self] = false;      /* mark lock not-busy      */
  }
};
```

"Works" for 2 threads only!

Locks: Software Implementation 3



Locks: Software Implementation 4 (Peterson Algorithm)

```
class Lock {
  private int   turn;
  private bool busy[2];     /* initialize to false          */
  public void lock() {
    busy[self] = true;      /* 'self' is current thread     */
    int other = 1 - self ; /* 'other' is other thread      */
    turn = other;
    while(busy[other] && turn != self); /* busy loop        */
  }
  public void unlock() {
    busy[self] = false;     /* mark lock not-busy           */
  }
};
```

Works for 2 threads only!

Peterson Algorithm: Considerations

Pros:

- Provides Mutual Exclusion: at most 1 thread in CS.
- Provides Progress: Only threads wanting to enter CS participate in selection of who gets to enter next. Selection cannot be indefinitely postponed.
- Provides Bounded Waiting: A thread never has to wait for more than one turn to enter CS.

Cons:

- Works for 2 threads only.
  - Alternatives: Filter algorithm, Baker's algorithm
- Busy waiting

Eliminating Busy Loop in Peterson's Algorithm

```
class Lock {
  private int   turn;
  private bool busy[2];     /* initialize to false        */
  public void lock() {
    busy[self] = true;      /* 'self' is current thread   */
    int other = 1 - self ; /* 'other' is other thread    */
    turn = other;
    while(busy[other] && turn != self); /* busy waiting  */
  }
  public void unlock() {
    busy[self] = false;     /* mark lock not-busy         */
  }
};
```

Eliminating Busy Loop in Peterson's Algorithm

```
public void lock() {
  busy[self] = true;
  int other = 1 - self ;
  turn = other;
  while(busy[other] && turn != self); /* busy waiting  */
}
```

Eliminating Busy Loop in Peterson's Algorithm

```
public void lock() {
  busy[self] = true;
  int other = 1 - self ;
  turn = other;
  while(busy[other] && turn != self) {
    scheduler->yield();
  }
}
```

No more busy waiting!
When lock is busy, give up CPU and try again later.

Peterson Algorithm: Considerations

Pros:

- Provides Mutual Exclusion: at most 1 thread in CS.
- Provides Progress: Only threads wanting to enter CS participate in selection of who gets to enter next. Selection cannot be indefinitely postponed.
- Provides Bounded Waiting: A thread never has to wait for more than one turn to enter CS.

Cons:

- Works for 2 threads only.
  - Alternatives: Filter algorithm, Baker's algorithm
- Busy waiting
- Does not work on most modern CPUs, due to out-of-memory operations.

Peterson's Algorithm and Memory Order

```
public void lock() {
  busy[self] = true;
  int other = 1 - self ;
  turn = other;
  while(busy[other] && turn != self) {
    scheduler->yield();
  }
}
```

Peterson's Algorithm and Memory Order

```
public void lock() {

  busy[self] = true;
  int other = 1 - self ;
  turn = other;



  while(busy[other] && turn != self) {

    scheduler->yield();
  }
}
```

Peterson's Algorithm and Memory Order

```
public void lock() {

  busy[self] = true;
  int other = 1 - self ;
  turn = other;



  while(busy[other] && turn != self) {

    scheduler->yield();
  }
}
```

Peterson's Algorithm and Memory Order

```
public void lock() {

  busy[self] = true;
  int other = 1 - self ;
  turn = other;

  __sync_synchronize();    /* memory fence */

  while(busy[other] && turn != self) {

    scheduler->yield();
  }
}
```

Software Solutions to Critical Sections

- Critical Sections and Locks
- Buggy Software Implementations
- Peterson's Algorithm
- Practicality Issues:
  - Busy Waiting
  - What about Out-of-Order Memory Operations?