CISC 3320 Hardware Support for Synchronization

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

 These slides are a revision of the slides provided by the authors of the textbook via the publisher of the textbook

Outline

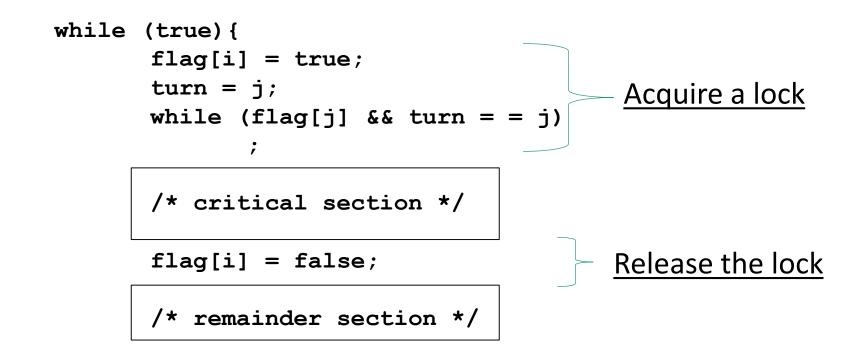
- Concept of lock
- Uniprocessor and multiprocessor system
- Memory barrier
- Special instructions
- Atomic variables

Synchronization

- Conceptually, any solution to the criticalsection problem can be viewed as to constructing a simple tool, called a "lock"
- A process must <u>acquire a lock</u> before entering a critical section, and <u>releases</u> <u>the lock</u> when it exits the critical section

Recall Peterson's Solution: Algorithm for Process P_i

Notice notations of "i" and Process P_i



}

Synchronization Hardware

- As discussed, software-based solutions (like Peterson's Solution) are not guaranteed to work on modern computer architectures
- Many systems provide hardware support for synchronization
 - Uniprocessor systems
 - Multiprocessor systems

Uniprocessor Systems

- Disable interrupts
 - Currently running code would execute without preemption
 - Generally too inefficient on multiprocessor systems

Hardware Support for Synchronization

- We will look at three forms of hardware support:
 - 1. Memory barriers
 - 2. Hardware instructions
 - 3. Atomic variables

Memory Barriers

- Memory model are the memory guarantees that a computer architecture makes to application programs.
- Memory models may be either:
 - Strongly ordered where a memory modification of one processor is immediately visible to all other processors.
 - Weakly ordered where a memory modification of one processor may not be immediately visible to all other processors.
- A memory barrier is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Recall Peterson's Solution: 2-Thread Example

• Two threads share the data:

```
boolean flag = false;
int x = 0;
```

• Thread 1 performs

```
while (!flag)
   ;
print x
```

Thread 2 performs

x = 100;flag = true

• What is the expected output?

10/23/2019

Recall Peterson's Solution: After Instruction Reordering

- 100 is the expected output.
- However, the operations for Thread 2 may be reordered:

flag = true; x = 100;

- If this occurs, the output may be 0!
 - The effects of instruction reordering in Peterson's Solution
 - This allows both processes to be in their critical section at the same time!

Solution using Memory Barrier

- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
    memory_barrier();
print x
```

Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

Questions?

Concept of memory barrier

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words **atomically** (uninterruptibly.)
- Test-and-Set instruction
- Compare-and-Swap instruction

$\texttt{test_and_set} \ \textbf{Instruction}$

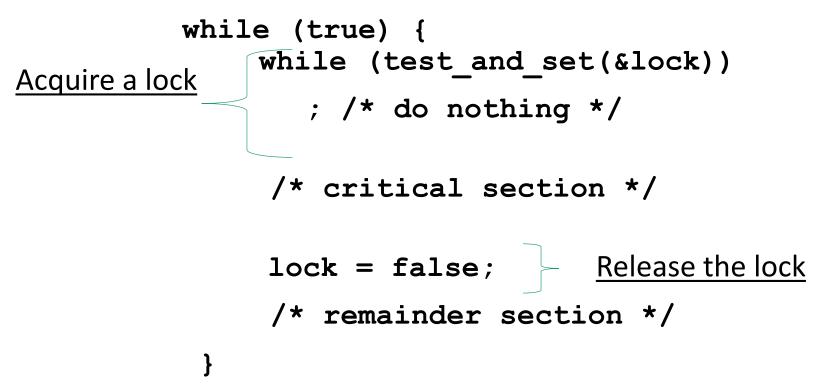
```
Definition:
```

```
boolean test_and_set (boolean *target)
{
    boolean rv = *target;
    *target = true;
    return rv:
```

- 1. Executed atomically
- 2. Returns the original value of passed parameter
- 3. Set the new value of passed parameter to true

Solution using test_and_set()

- Shared Boolean variable lock, initialized to false
- Solution:



compare_and_swap Instruction Definition:

```
int compare_and_swap(int *value, int expected, int
    new_value) {
```

```
int temp = *value;
if (*value == expected) *value = new_value;
return temp;
```

}

- 1. Executed atomically
- 2. Returns the original value of passed parameter value
- Set the variable value the value of the passed parameter new_value but only if *value == expected is true. That is, the swap takes place only under this condition.

Solution using

```
compare and swap
```

- Shared integer lock initialized to 0;
- Solution:

```
while (true) {
          while (compare and swap(&lock, 0, 1) != 0)
Acquire a
               ; /* do nothing */
lock
               /* critical section */
               lock = 0; <u>Release the lock</u>
               /* remainder section */
        }
```

Bounded Waiting?

- Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement
- Why?

Bounded-Waiting Mutual Exclusion

- Demonstrate it using with compare-andswap
- Two variables
 - boolean waiting[n];
 - int lock;
- The elements in the waiting array are initialized to false, and lock is initialized to follow to 0.

Bounded-Waiting Mutual Exclusion with compare-and-

swap

```
while (true)
  waiting[i] = true;
                                        /* critical section */
   key = 1;
   while (waiting[i] && key == 1)
                                        // scan (i+1,i+2,...n-1,0,...,i-1)
      key =
                                        j = (i + 1) \% n;
        compare and swap(&lock,0,1);
                                        while ((j != i) && !waiting[j])
   waiting[i] = false;
                                                i = (i + 1) \% n;
                                       if (j == i)
                                              lock = 0;
                                        else
                                              waiting[j] = false;
                                           /* remainder section */
                                        }
```

Bounded Waiting

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering (i + 1, i + 2, ..., n 1, 0, ..., i 1).
- It designates the first process in this ordering that is in the entry section (waiting[j] == true) as the next one to enter the critical section.
- Any process waiting to enter its critical section will thus do so within n – 1 turns.

Atomic Variables

- Typically, instructions such as compare-andswap are used as <u>building blocks</u> for other synchronization tools.
- One tool is an **atomic variable** that provides atomic (uninterruptible) updates on basic data types such as integers and booleans.
- For example, the increment() operation on the atomic variable sequence ensures sequence is incremented without interruption:

increment(&sequence);

Solution using Atomic Variables

• The increment() function can be implemented as follows:

```
void increment(atomic_int *v)
{
    int temp;
    do {
        temp = *v;
     }
     while (temp !=
     (compare_and_swap(v,temp,temp+1));
}
```

Questions?

- Concept of "lock"
- Synchronization hardware
 - Concept of lock
 - Uniprocessor and multiprocessor system
 - Memory barrier
 - Special instructions
 - Atomic variables