Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1400-1 semester

Zeinab Zali

Session 14: Synchronization

Memory Barrier

- **Memory model** are the memory guarantees 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.

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- 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
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Check Critical section requirements

```
P1:
while ( true )  {
        flag [1] = true ;
        turn = 1 ;
        while ( flag [2 ] and turn=2);
        <Critical - Section >
       flag [1] = false ;
       < remainder >
}
```

```
P2:
 while ( true ) {
       flag [2 ] = true ;
       turn = 2 ;
        while ( flag [1] and turn=1);
        < Critical - Section >
         flag [2 ] = false ;
       < remainder >
   }
```

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Hardware instructions
2. Atomic variables

Hardware Instructions

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

The test_and_set Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to `true`

Solution Using test_and_set()

- Shared boolean variable **`lock`**, initialized to **`false`**
- Solution:

```
do {
        while (test_and_set(&lock))
        ; /* do nothing */

            /* critical section */

    lock = false;
            /* remainder section */
 } while (true);
```

While(lock==true);
lock=true;

- Does it solve the critical-section problem?

This solution does not satisfy bounded waiting: starvation

The 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;
 }
```

- Properties
  - Executed atomically
  - 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.

Intel x86 instruction
lock cmpxchg <destination operand>, <source operand>

Solution using compare_and_swap

- Shared integer **`lock`** initialized to 0;
- Solution:

```
while (true){
        while (compare_and_swap(&lock, 0, 1) != 0)
        ; /* do nothing */

        /* critical section */

        lock = 0;

    /* remainder section */
}
```

- Does it solve the critical-section problem?

Bounded-waiting with compare-and-swap

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks 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:
  - Let **`sequence`** be an atomic variable
  - Let **`increment()`** be operation on the atomic variable **`sequence`**
  - The Command:

    `increment(&sequence);`

    ensures **`sequence`** is incremented without interruption:

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));
}
```

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **`acquire()`** a lock
  - Then **`release()`** the lock
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is mutex lock
- Protect a critical section by first **`acquire()`** a lock then **`release()`** the lock
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

```
do {
    acquire lock
       critical section
    release lock
      remainder section
 } while (true);
```

acquire() and release()

```
acquire() {
    while (!available) ; /* busy wait */
    available = false;
 }
release() {
    available = true;
}
```

-
```
  do {
   acquire lock
      critical section
   release lock
     remainder section
} while (true);
```

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the **`wait()` operation**

```
wait(S) {
    while (S <= 0)
        ; // busy wait
    S--;
}
```

- Definition of the **`signal()` operation**

```
signal(S) {
    S++;
}
```

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems

Semaphore Usage Example

- Solution to the CS Problem
  - Create a semaphore "**mutex**" initialized to 1

```
wait(mutex);
    CS
signal(mutex);
```

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2
  - Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Example : synchronization problems

- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

```
P1:
    S1;
    signal(synch);
P2:
    wait(synch);
    S2;
```

Example: Mistakes using semaphores

- Let ***S*** and ***Q*** be two semaphores initialized to 1, What happens executing P0 and P1 ?

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

Example: Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

- Waiting queue

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

Implementation with no Busy waiting (Cont.)

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

signal(semaphore *S) {
    S->value++;
    if (S->value <= 0) {
        remove a process P from S->list;
        wakeup(P);
    }
}
```