Synchronization Tools (cont'd)

Course Code: CSC 2209 Course Title: Operating Systems

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	**09**	**Week No:**	**09**	**Semester:**	
Lecturer:	*Name & email*				

Lecture Outline

1. Hardware Support for Synchronization
2. Mutex Locks
3. Semaphores

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section (CS) 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. Memory barriers

  2. Hardware instructions

  3. Atomic variables

Memory Barriers

- **Memory models** 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 <u>immediately visible to all other processors</u>.
  - **Weakly ordered** – where a memory modification of one processor may <u>not be immediately visible to all other processors</u>.

- A **memory barrier** is an <u>instruction</u> that forces any change in memory to be propagated (made visible) to all other processors.

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
```

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* (uninterruptibly.)
- **Test-and-Set** instruction
- **Compare-and-Swap** instruction

test_and_set Instruction

Definition:

```
    boolean test_and_set (boolean *target)
{
  1      boolean rv = *target;   //
  2     *target = true; //
  3   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:

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

           /* critical section */

    lock = false;
           /* remainder section */

 } while (true);
```

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`**
3. 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)
       ; /* do nothing */

  /* critical section */

  lock = 0;

  /* remainder section */
}
```

Bounded-waiting Mutual Exclusion 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, the **increment()** operation on the atomic variable **sequence** ensures **sequence** is incremented without interruption:

  increment(&sequence);

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 <u>software tools</u> to solve critical section (CS) 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.
- But this solution requires **busy waiting- waste CPU cycle**
  - This lock therefore called a **spinlock**

Solution to Critical-section Problem Using Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Lock Definitions

- ```
  acquire() {
       while (!available)

     ; /* busy wait */

   available = false;;

  }
  ```

```
critical section
```

- ```
  release() {

   available = true;

  }
  ```

These two functions must be implemented atomically. Both test-and-set and compare-and-swap can be used to implement these functions.

Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process 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 Usage

- **Counting semaphore** – integer value can range over an unrestricted domain (+ infinity to - infinity)
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various 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:

  S_1;

  signal(synch);

  P2:

  wait(synch);

  S_2;
- Can implement a counting semaphore ***S*** as a binary semaphore

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
- **typedef struct {**

  int value;

  struct process *list;

} semaphore;

Implementation with no Busy waiting (cont'd)

```
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);
   }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - **`signal (mutex)`** …. **`wait (mutex)`**
  - **`wait (mutex)`** … **`wait (mutex)`**
  - Omitting of **`wait (mutex)`** and/or **`signal (mutex)`**
- These – and others – are examples of what can occur when sempahores and other synchronization tools are used incorrectly.

Books

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9^{th}

References

- Operating Systems Concept
  - Written by Galvin and Silberschatz
  - Edition: 9th