Operating Systems: Synchronization Tools and examples – Part III

Neerja Mhaskar

Department of Computing and Software, McMaster University, Canada

Acknowledgements: Material based on the textbook Operating Systems Concepts (Chapters 6 & 7)

Hardware Support for Synchronization

- Hardware Support for Synchronization is typically for
  - kernel developers and
  - implementation of high-level Synchronization tools.
- Some of them are
  - Memory barriers or Memory Fences
  - Atomic hardware instructions
    - `Test_and_Set()`
    - `Compare_and_swap()`
  - Atomic variables

Memory Barriers or Memory Fences

- Computer systems can reorder instructions for efficiency.
  - Unfortunately, leads to data inconsistency.
- **Memory model** – explains how a computer architecture determines what memory guarantees it will provide to an application program.
  - Varies by processor type and kernel developers cannot make any assumptions about it.
  - To address this issues computer architectures, provide memory barriers or memory fences.

Memory Barriers or Memory Fences

- **Memory barriers** or **Memory Fences**
  - Computer instructions that force any changes in memory to be propagated to all other processors in the system.
  - Executing a memory barrier instruction ensures that all loads and stores are completed before any subsequent load or store operations are performed in the system.
  - Example:

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

Hardware Support for Synchronization

- Modern machines provide special atomic hardware instructions
  - **`Test_and_Set()`**
  - **`Compare_and_swap()`**
- They are executed atomically.
- For instance, two `test_and_set()` instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Synchronization Hardware

Test_and_Set() instruction:

```
boolean test_and_set (boolean *target)
{     boolean rv = *target; //stores the target/
                            //locked value

      *target = TRUE; //sets it to true

      return rv:  //returns the old value!
}
```

- Executed atomically.
- Returns the original value of the passed parameter.
- ALWAYS sets its value to `TRUE`.

Solution using test_and_set()

- Shared Boolean variable `locked`
- Initially `locked = FALSE` (lock is available)
- Suppose a process **P**$_i$ wants to enter its CS

```
do {
     while (test_and_set(&locked));
          /* do nothing */
       /* critical section */
       locked = false;
       /* remainder section */
   } while (true);
```

Sets locked = True and returns False

Question - Implementing Mutex with Test and Set()

Implement a mutex lock using the `test_and_set()` atomic hardware instruction.

Assume that the following structure defining the mutex lock is available:

```
typedef struct {
  bool held;
} lock;
```

`held == false (true)` indicates that the lock is available (not available)

Using the `struct lock`, illustrate how the following functions can be implemented using the `test_and_set()` instructions:

- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.

Implementing Mutex with Test and Set() - Solution

```
 //initialization

Struct lock *mutex;

mutex->held = false;  //(lock is available)



// acquire using test_and_set()

void acquire(lock *mutex) {

  while (!test and set(&mutex->held));

  return;

  }
```

Implementing Mutex with Test and Set() - Solution

```
// release operation on

void release(lock *mutex) {

  mutex->held = false;

  return;

}
```

compare_and_swap Instruction

```
int compare _and_swap(int *value, int expected, int
  new_value)
{
    int temp = *value;
    if (*value == expected)
       *value = new_value;
    return temp;
}
```

- Executed atomically.
- Returns the original value of passed parameter “value”.
- Only resets `value`, if it is not equal to `expected`.

Solution using compare_and_swap

- Shared integer "lock" initialized to 0; (lock is available)
- Expected value = 0 (lock is available)
- New value = 1 (lock is unavailable)
- Suppose lock is available and process **P$_i$** wants to enter its CS
- Solution:

```
do {
     while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */
     /* critical section */
 lock = 0;
     /* remainder section */
} while (true); s
```

Checks if lock=0,
in this case it is,
therefore resets
lock = 1 and returns 0

Atomic Variables

- Incrementing or decrementing a shared integer value may produce a race condition.
- **Atomic variables** are a programming language construct that provide atomic operations on basic data types such as integers and booleans.

Example:

- In the bounded buffer problem declare `counter` as an atomic variable
- Incrementing (decrementing) counter will not result in race condition.

Alternative Approaches

- Transactional Memory
  - A **memory transaction** is a sequence of read-write operations to memory that are performed atomically.
- OpenMP
  - OpenMP is a set of compiler directives and API that support parallel programming.
  - The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.

Alternative Approaches

- Functional Programming Languages
  - Do not maintain state. Variables are treated as immutable and cannot change state once they have been assigned a value.