# 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)

# POSIX Condition Variables

- **Condition variables** in Pthreads behave similarly to those described under Monitors.
- Since Pthreads is typically used in C programs—and since C does not have a monitor—we accomplish locking by associating a condition variable with a mutex lock.
- Condition variables in Pthreads use the
  - `pthread_cond_t` data type, and
  - `pthread_cond_init()` function to initialize it.
- The following code creates and initializes a condition variable as well as its associated mutex lock:

```
pthread_mutex_t mutex;

pthread_cond_t cond_var;

pthread_mutex_init(&mutex,NULL);

pthread_cond_init(&cond_var,NULL);
```

Null indicates default attributes passed

# POSIX Condition Variables Contd…

- `pthread_cond_wait()` function – used to wait/block on a condition variable. It takes two parameters
  - Pointer to condition variable
  - Pointer to mutex.
    - This mutex must be locked by the calling thread before the `pthread_cond_wait()` is used, or undefined behaviour will result.
- `pthread_cond_wait()` atomically unlocks the mutex and wait on the condition variable.
- When condition variable is signaled, before returning to the calling thread, the the mutex lock is acquired on entrance to **pthread_cond_wait()**

# POSIX Condition Variables Contd…

- Example: A thread waiting for the condition `a == b` to become true using a Pthread condition variable:

```
pthread_mutex_lock(&mutex);
while (a != b)
    pthread_cond_wait(&cond_var, &mutex);
pthread_mutex_unlock(&mutex);
```

# POSIX Condition Variables Contd…

- `pthread_cond_signal()` function
  - Takes in a pointer to the condition variable.
  - used to signal any one thread waiting on the condition variable.
  - If no thread waiting on the condition variable nothing happens.
- Example:

```
pthread_mutex_lock(&mutex);

a = b;

pthread_cond_signal(&cond_var);

pthread_mutex_unlock(&mutex);
```

# 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

- System can reorder instructions for efficiency - 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**
  - Computer instructions that force any changes in memory to be propagated to all other processors 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()`**:

```
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!
}
```

- Returns the original value of the passed parameter and sets its value to "TRUE".

# Solution using test_and_set()

- Shared Boolean variable `lock`.
- 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

## Solution using test_and_set() - Mutual Exclusion

- Suppose process **$P_j$** wants to enter its CS, while **$P_i$** is executing in its CS. It sees lock=True.

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

Returns True, and also, locked=True, so waits.

- Process waits to get the lock at the while loop.
- Therefore, when any process **$P_i$** is executing in its CS, another process cannot enter its CS and mutual exclusion is achieved.

## Solution using test_and_set() - Progress and Bounded waiting

- Initially `locked=FALSE`,
  - If multiple processes want to enter CS, only one of them gets to execute `test_and_set()` (also in finite time)
  - A process executing in its remainder section does not get to block another process from entering the CS, since it has already released the lock.
- However, the same process can enter CS repeatedly, without giving a turn to other processes. Hence, bounded wait is **<u>not</u>** achieved.

# 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”.
- Compares the integer stored in “value”  to the integer value stored in expected. If they are equal, then stores the integer value of “new_value” in “value”.

# 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 $\mathbf{P_i}$ 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

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