Implementing Locks with Minimal Busy Waiting Using Test&Set – Better Solution

Lock::Acquire()
{
    while (test&set(value) == 1);
}

Lock::Release()
{
    value := 0;
}

With busy-waiting

Lock::Acquire()
{
    While (test&set(guard) == 1);
    if (value == 1) {
        Put TCB on wait queue for lock;
        set guard = 0 and call switch();
    } else (value = 1; guard = 0;)
}

Lock::Release()
{
    While (test&set(guard) == 1);
    if (wait queue is not empty)
    { Move a waiting thread to ready queue; }
    else ( value = 0; )
    guard = 0;
}

With minimal busy-waiting

Lock::Acquire()
{
    While (test&set(guard) == 1);
    if (wait queue is not empty)
    { Move a waiting thread to ready queue; }
    else ( value = 0; )
    guard = 0;
}

Lock::Release()
{
    While (test&set(guard) == 1);
    if (wait queue is not empty)
    { Move a waiting thread to ready queue; }
    else ( value = 0; )
    guard = 0;
}

Implementing Locks: Summary

- Locks are higher-level programming abstraction
  - Mutual exclusion can be implemented using locks
- Lock implementation generally requires some level of hardware support
- Two common implementation approaches
  - Disable interrupts
    - Uni-processor architectures only
  - Atomic read-modify-write instructions
    - Uni- and multi-processor architectures
- Other implementation alternatives
  - Busy-waiting implementation
  - Minimal-wait implementation

Beyond Locks

- Locks ensure mutual exclusion
- Is this sufficient?
  - What if you want to synchronize on a condition?
  - Example: Producer-consumer problem

What is wrong with this?

What is wrong with this?
Introducing Condition Variables

- Correctness requirements for bounded buffer producer-consumer problem
  - Only one thread manipulates the buffer at any time (mutual exclusion)
  - Consumer must wait for producer when the buffer is empty (scheduling/synchronization constraint)
  - Producer must wait for the consumer when the buffer is full (scheduling/synchronization constraint)

- Solution: condition variables
  - An abstraction that supports conditional synchronization
  - Key idea:
    - Enable threads to wait inside a critical section by atomically releasing lock at the same time

Condition Variables: Operations

- Three operations
  - Wait() 
    - Release lock
    - Go to sleep
    - Reacquire lock upon return
  - Signal() 
    - Wake up a waiter, if any
  - Broadcast() 
    - Wake up all the waiters

- Implementation
  - Requires a per-condition variable queue to be maintained
  - Threads waiting for the condition wait for a signal()

Implementing Wait() and Signal()

```c
Condition::Wait(lock){
  numWaiting++;
  lock.release();
  Put TCB on the waiting queue for the CV;
  switch();
  lock.acquire();
}
```

```c
Condition::Signal(){
  if (numWaiting > 0) {
    Move a TCB from waiting queue to ready queue;
    numWaiting--;
  }
}
```

Using Condition Variables: An Example

- Coke machine as a shared buffer
- Two types of users
  - Producer: Restocks the coke machine
  - Consumer: Removes coke from the machine

- Requirements
  - Only a single person can access the machine at any time
  - If the machine is out of coke, wait until coke is restocked
  - If machine is full, wait for consumers to drink coke prior to restocking

- How will we implement this?
  - What is the class definition?
  - How many lock and condition variables do we need?
Coke Machine Example

```cpp
class CokeMachine {
    Lock lock;
    int count = 0;
    Condition notFull, notEmpty;
}

CokeMachine::Deposit() {
    lock.acquire();
    while (count == 0) {
        notFull.wait(&lock);
    }
    Add coke to the machine;
    count++;
    notEmpty.signal();
    lock.release();
}

CokeMachine::Remove() {
    lock.acquire();
    while (count == 0) {
        notEmpty.wait(&lock);
    }
    Remove coke from the machine;
    count--;
    notFull.signal();
    lock.release();
}
```

Summary

- Non-deterministic order of thread execution → concurrency problems
  - Multiprocessing
  - A system may contain multiple processors → cooperating threads/processes can execute simultaneously
  - Multi-programming
  - Thread/process execution can be interleaved because of time-slicing

- Goal:
  - Ensure that your concurrent program works under ALL possible interleavings

- Approach:
  - Define synchronization constructs and programming style for developing concurrent programs
    - Locks to provide mutual exclusion
    - Condition variables → provide conditional synchronization

- Next time: high-level programming constructs - a historical perspective
  - Semaphores and monitors