Why use threads for concurrency?

- Threads are cheaper than processes in terms of creation, scheduling, communication...
- Concurrency (multithreading) can be very useful for
  - Utilizing multiple processors
  - Handling concurrent events (e.g., Web requests)
  - Driving slow devices (e.g., disks, networks, printers)
- Threads interact through accessing shared memory
  - Need to synchronize accesses
Lock

• A lock is an object in memory
  – Two states: locked, unlocked
  – Two operations: acquire(), release()
  – The implementation of acquire/release must be atomic

• A thread executing inside a critical section hold the lock

• Lock can be viewed as a simple resource scheduling mechanism
  – Resource: shared variables accessed inside the critical section
  – Scheduling policy: one thread at a time
Use of lock

• Lock is used to protect code or data?
  – Both…

• Example
  – Assume initially for \( i = 0 \) to 999 \( \text{table}[i] = 0 \)
  – If thread A calls \( \text{Insert}(x) \) concurrently with thread B calling \( \text{Insert}(y) \), what will happen?
  – We get different results depending on the order of execution!

```java
Insert (int r) {
    if (r != 0) {
        table [i] = r;
        i++;
    }
}
```

Execution sequences:
Invariant

- Lock protects the *invariant* of the associated data
- *Invariant*: a boolean function of the data that is true whenever the lock is **not held**
- Each thread must restore the invariant before releasing the lock

```java
Insert (int r) {
    if (r != 0) {
        table [i] = r;
        i++;
    }
}
```

Invariant: i is the index of the first ZERO element, and all elements beyond i are ZERO

```java
Insert (int r) {
    if (r != 0) {
        acquire(m);
        table [i] = r;
        i++;
        release(m);
    }
}
```
Cheating

• General rule: every access to shared data needs to be protected by a lock
• Violate the rule?
  – If data protected and the operation are simple
• Is the following code cheating?

```java
if (!initDone) {
    acquire (m);
    if (!initDone) {
        Initialize ();
        initDone = TRUE;
    }
    release (m);
}
```

• The procedure “Initialize()” is protected;
• But “initDone”, a global variable, is not protected by any lock!
• Use two “if” test initDone
Deadlock

• Deadlock occur when a thread tries to acquire a lock that it already holds
  – Thread A locks M1;
  – Thread B locks M2;
  – Thread A blocks trying to lock M2;
  – Thread B blocks trying to lock M1.

• Possible solutions:
  – Applying a partial order to the acquisition of locks
  – Partition the data into smaller pieces protected by separate locks
Poor Performance

- Thread A holding a lock stops another thread B from making progress.
- If thread A
  - blocking on another lock?
  - taking page fault?
  - waiting for an i/o device?
- To improve performance, we need to reduce lock conflicts
  - Lock at a finer granularity
  - But this introduces complexity…
Semaphores

• All the issues discussed for lock also true for mutex semaphores!
• Lock operations: acquire(), release()
• Semaphore operations: wait(), signal()
• What is the difference between locks and semaphores?
  – Semaphore can “remember” threads in waiting
  – Lock is more like a variable, while semaphore is more like a queue
  – Which one is more efficient?