Datastructures and Algorithms

Concurrency

Introduction

- A PROCESS or THREAD is a potentially-active execution context
- Classic stored program model of computing has single thread of control
- Parallel programs have more than one thread of control
- A process can be considered as an abstraction of a physical processor.
  - But here, only one processor will run multiple threads

Historical Usage

- Scientific computing, where parallelizing computations is important.
- Web development, which introduced the need for parallel servers and concurrent client programs.

Programming Notation

- Two main classes of programming notation
  - synchronized access to shared memory
  - message passing between processes that don't share memory
- Similarities
  - Both approaches can be implemented on hardware designed for the other, though shared memory on message-passing hardware tends to be slow
- Principle difference
  - the message passing type requires active participation of 2 processors at either end - one to send and one to receive - while on a multiprocessor, reading or writing only needs one processor to control

Cache Coherence Problem

- Multiprocessor may usually has a local cache and a shared main memory
- If two processors page in a common bit of memory, they may be operating under old or invalid data at the same time.
  - This is known as the **cache coherence problem.**
- Bus-based system
  - processors can eavesdrop
  - when it needs to change its copy, it requests an exclusive copy and **waits** for the other processors to invalidate their copies

Race conditions

- A race condition occurs when actions in two processes are not synchronized and program behavior depends on the order in which the actions happen
- If any of the possible program outcomes are ok → Race condition is not bad
  - e.g. workers taking things off a task queue

Bad Race Conditions

- Suppose processors **A** and **B** share memory, and both try to increment variable **X** at more or less the same time
- Very few processors support arithmetic operations on memory, so each processor executes
    - LOAD X
    - INCREMENT
    - STORE X
- Suppose X is initialized to 0.
    - If both processors execute these instructions simultaneously, what are the possible outcomes?
    - could be increased by 1 or by 2

Synchronization

- An act of ensuring that events in different processes happen in a desired order
- Can be used to eliminate race conditions
- In the previous example we need to synchronize the increment operations to enforce MUTUAL EXCLUSION on access to **X**
- Most synchronization can be regarded as either:
  - **Mutual exclusion** -- making sure that only one process is executing a CRITICAL SECTION
  - **Condition synchronization**-- making sure that a given process does not proceed until some condition holds (e.g. that a variable contains a given value)

Mutual Exclusion Vs Condition Synchronization

- The distinction is existential vs universal quantification
  - Mutual exclusion requires multi-process consensus
- **Mutual exclusion** -- making sure that only one process is executing a CRITICAL SECTION → One process access it at a time
- **Condition synchronization**-- making sure that a given process does not proceed until some condition holds → **No** restriction to access the variable by only one process at a time.

Over-Synchronization

- We do not in general want to over-synchronize
  - That eliminates parallelism, which we generally want to encourage for performance
- Basically, we want to eliminate "bad" race conditions, i.e., the ones that cause the program to give incorrect results

Implement Synchronization

- To implement synchronization you have to have something that is ATOMIC
  - that means it happens all at once, as an indivisible action
  - In most machines, reads and writes of individual memory locations are atomic (note that this is not trivial; memory and/or busses must be designed to arbitrate and serialize concurrent accesses)
  - In early machines, reads and writes of individual memory locations were all that was atomic

Synchronization

- Synchronization is generally implicit in memory passing models, since a message must be sent before it can be received.
- However, in shared-memory, unless we do something special, a new “receiving thread” will not necessarily wait, and could read an “**old**” value of a variable before it has been written by the “sending” thread.
- Usually implement (in either model) by **spinning** or **blocking**.

Threads

- There are two ways to create a Thread:
  - Define a class that extends Thread
    - Supply a public void run() method
    - Create an object o of that class
    - Tell the object to start: o.start();
  - Define a class that implements Runnable (hence it is free to extend some other class)
    - Supply a public void run() method
    - Create an object o of that class
    - Create a Thread that “knows” o: Thread t = new Thread(o);
    - Tell the Thread to start: t.start();

The **synchronized** statement

- Synchronization is a way of providing exclusive access to data
- You can synchronize on any Object, of any type
- If two Threads try to execute code that is synchronized on the **same** object, only one of them can execute at a time; the other has to wait
  - synchronized (someObject) { /* some code */ }
  - This works whether the two Threads try to execute the same block of code, or different blocks of code that synchronize on the same object
- Often, the object you synchronize on bears some relationship to the data you wish to manipulate, but this is not at all necessary

synchronized methods

- Instance methods can be synchronized:
  - ```
    synchronized public void myMethod( /* arguments */) {
        /* some statements */
    }
    ```
- This is equivalent to
  - ```
    public void myMethod( /* arguments */) {
        synchronized(this) {
            /* some statements */
        }
    }
    ```
- Static methods can also be synchronized
  - They are synchronized on the class object (a built-in object that represents the class)

Locks

- When a Thread enters a synchronized code block, it gets a lock on the monitor (the Object that is used for synchronization)
- The Thread can then enter other code blocks that are synchronized on the same Object
  - That is, if the Thread already holds the lock on a particular Object, it can use any code also synchronized on that Object
- A Thread may hold a lock on many different Objects
- One way deadlock can occur when
  - Thread A holds a lock that Thread B wants, and
  - Thread B holds a lock that Thread A wants

Atomic actions

- An operation, or block of code, is atomic if it happens “all at once,” that is, no other Thread can access the same data while the operation is being performed
- x++; *looks* atomic, but at the machine level, it’s actually three separate operations:
  1. load x into a register
  2. add 1 to the register
  3. store the register back in x
- Suppose you are maintaining a stack as an array:

```
void push(Object item) {
    this.top = this.top + 1;
    this.array[this.top] = item;
}
```

  - You need to **synchronize** this method, *and every other access to the stack*, to make the push operation atomic
- Atomic actions that maintain data invariants are thread-safe; compound (non-atomic) actions are not
- This is another good reason for encapsulating your objects

Check-then-act

- A Vector is like an ArrayList, but is synchronized
- Hence, the following code *looks* reasonable:
  - ```
    if (!myVector.contains(someObject)) {   // check
        myVector.add(someObject);           // act
    }
    ```
- But there is a "gap" between checking the Vector and adding to it
  - During this gap, some other Thread may have added the object to the array
  - Check-then-act code, as in this example, is **unsafe**
- You must ensure that no other Thread executes during the gap
  - ```
    synchronized(myVector) {
        if (!myVector.contains(someObject)) {
            myVector.add(someObject);
        }
    }
    ```
- So, what good is it that Vector is synchronized?
  - It means that each *call* to a Vector operation is atomic