Concurrency

Hwansoo Han
Process, Thread, and Processor

- A *process* or *thread* is a potentially-active execution context
  - Classic von Neumann (stored program) model of computing has single thread of control
  - Parallel programs have more than one

- A process can be thought of as an abstraction of a physical *processor*
Multi-Process and Multi-Thread

- Multiple processes/threads can come from
  - Multiple CPUs
  - Kernel-level multiplexing of single physical machine
  - Language or library level multiplexing of kernel-level abstraction

- Multiple processes/threads
  - Run in true parallel
  - Can be unpredictably interleaved
  - Run-until-block
Programming Models

- Two main classes of programming models
  - Synchronized access to *shared memory*
  - *Message passing* between processes that do not share memory

- Hardware can be designed for both approaches
  - Both approaches can be implemented with
    - Software on hardware designed for the other
    - But shared memory on message-passing hardware tends to be slow
  - *Shared memory machines* are typically *multiprocessors*
  - *Message-passing based machines* are typically *clusters* now
Parallel Computer Architectures

- **Shared memory**
  - Access all data within a single address space
  - SMP, UMA, cc-NUMA
  - Popular programming model
    - Thread APIs (pthread, …)
    - OpenMP

- **Distributed memory**
  - Access only partial data. Others are accessed via communication
  - NUMA, Cluster
  - Popular programming model
    - MPI (de facto standard)
    - Hadoop (map-reduce model)
Machine Abstraction for Program

- **Shared-memory**
  - Single address space for all CPUs
  - Communication through regular load/store (implicit)
  - Synchronization using locks and barriers
  - Ease of programming
  - Complex HW for cache coherence

- **Message-passing**
  - Private address space per CPU
  - Communication through message send/receive over network interface (explicit)
  - Synchronization using blocking messages
  - Need to program explicit communication
  - Simple HW (no cache coherence supporting HW)
Cache Coherence

- Assume the following sequence
  - P0 loads A (A is in P0’s $)
  - P1 loads A (A is in P1’s $)
  - P0 writes a new value to A
  - P1 loads A (Can P1 get a new value?)

- Cache coherence
  - Allow *multiple read-only* copies or an *exclusive modified* copy
  - Invalidate all the other copies when a CPU needs to update a cache line
Race Conditions

- A race condition occurs
  - When actions in two processes/threads are racing to touch some common objects and
  - Program behavior depends on the order in which the actions happen

- Race condition can be resolved by synchronizations
  - Make a sequence of instructions to be atomic
  - Critical section
  - Mutual exclusion
Example of Race Condition

- Suppose processes A and B share memory, and both try to increment variable X at the same time
  - Very few processors support arithmetic operations on memory, so each processor executes
    
    ```
    LOAD r1, X
    INC r1  // increase by 1
    STORE r1, X
    ```
  
- Suppose X is initialized to 0. If both processors execute these instructions simultaneously, what are the possible outcomes?
  - Could go up by one or by two
Synchronization

- *Synchronization* ensures that events in different processes happen in a desired order.

- Synchronization can be used to eliminate race conditions.
  - Synchronize the increment operations in the example to enforce the *mutual exclusion* on the accesses to $X$. 
Synchronization (cont’d)

- Most synchronization can be regarded as either mutual exclusion or condition synchronization

- Mutual exclusion
  - Makes sure that only one process is executing a critical section at a time
    (e.g., increment a shared variable)

- Condition synchronization
  - Makes sure that a given process does not proceed until some condition holds
    (e.g., a variable contains a given value)
Synchronization (cont’d)

- We do not in general want to over-synchronize
  - Synchronize can eliminate parallelism, which we generally want to encourage for performance

- Basically, we want to eliminate *bad* race conditions
  - i.e., eliminate the ones that cause the program to give incorrect results
Parallel Programming

- Shared memory machines
  - Languages – Java, C#
  - Extension – OpenMP
  - Library – pthreads, Win32 threads

- Distributed computing
  - Message passing interface library – MPI
  - Remote procedure calls (RPC)
  - Google map-reduce, Apache Hadoop
Lifetime of Concurrent Threads

- With co-begin, parallel loops, or launch-at-elaboration
  - Threads are always properly nested – (a)
- With fork/join
  - More general patterns are possible – (b)
Two-level Implementation of Threads

- Many threads on some processes
Coroutines allow multiple execution contexts, only one of which is active
- Can transfer to other coroutine and back to the current
- The argument of the transfer is the pointer to the context blocks of other and current

Create a new coroutine in a state that looks like it is blocked in transfer
- Manipulate the context of a new coroutine
Thread Scheduler for Uniprocessor

- Run-until-block threads on a single process
  - Scheduler selects the thread to run (no need of transfer)
  - Ready list: keeps the contexts of the threads that are runnable but not running

- Can be blocked for fairness or synchronization
  - If blocked for fairness, current thread yields and enqueues itself on the ready list (next time it can run again)
  - If blocked for synchronization, current thread adds itself to a queue associated with the awaited condition
    - Some other thread will move the thread to the ready list when the awaited condition is met and we can continue
Contexts for Scheduler

- current_thread is running
- Contexts of other threads are kept
  - ready_list, blocked_list for waiting for various conditions
Preemption

- Need to preempt the current thread
  - Without explicit yield, current thread switches to other
  - Use *timer signals* to trigger involuntary yields
    - Request the OS to deliver a signal at a specific time
    - Signal handler manipulates current thread’s context to make it appear *yield* has just been called
    - Signal handler “returns” to *yield* and switch to other thread
  - Thread scheduler must be called with signals disabled, and must re-enable them upon its return
    - Thread scheduler is not *reentrant*
Multiprocessor Scheduling

- Schedulers in multiprocessors
  - Share data structures related to scheduler e.g., current_thread, ready_list and blocked_list

- Multiple kernel threads/processes access the scheduler structures
  - Disabling signals does not suffice
  - Synchronization is needed for the critical section

```
Disable_signals
Acquire_lock(scheduler_lock)
... // scheduler actions
Release_lock(scheduler_lock)
Re-enable_signals
```
Implementing Synchronization

- Atomic read and write instruction is used
  - test-and-set: sets a variable to 1 (true) and returns if the variable was previously 0 (false)

- Condition synchronization
  - Loop until “location X contains Y” – atomic read X

- Mutual exclusion
  - Dekker (early 1960), Dijkstra (1965), Peterson (1981): algorithms without atomic instruction
  - Spin lock with a special atomic instruction

while (test_and_set(L))
  ; // spin
Spin Locks

- Repeatedly reading a shared location until it reaches a certain value is known as spinning or busy-waiting
  - A busy-wait mutual exclusion is known as a spin lock

- Spin locks waste processor cycles
  - Scheduler can put a process to sleep and run something else instead of spinning – scheduler-based synchronization
  - Spin locks are still valuable for certain things and widely used
    - In particular, it is better to spin than to sleep when the expected spin time is less than the rescheduling overhead
Barriers

- Data-parallel algorithms need barriers
  - Often structured with a series of high-level steps
  - Every thread needs to complete the previous step before any thread moves on to the next

Implementing a busy-wait barrier
- Use globally shared counter
- Modify with an atomic `fetch_and_decrement`

```
parallel for I = 1, 100
  A[i] = B[i] + C[i];
sum = 0;
parallel_reduce for I = 1, 100
  sum = sum + A[i];
```
Semaphore

- Semaphores are the oldest scheduler-based synchronization
- A semaphore is a special counter
  - It has an initial value (C) and
  - two operations, P and V, for changing that value
  - A semaphore keeps track of the difference between the number of P and V operations that have occurred
  - A P operation is delayed (the process is de-scheduled) until \( #P - #V \leq C \), the initial value of the semaphore
Semaphores (cont’d)

- If $C = 1$, used for mutual exclusion
  - P, V operations act as if lock/unlock (binary semaphore)
- If $C > 1$, used for bounded buffer
  - P, V operations controls available slots

Problems with semaphores

- They're pretty low-level
  - P, V pair is used – one of them may be left out
- Their use is scattered all over the place
  - Make it difficult to track them down
Monitors

- Monitors are language-level mechanism
  - Attempt to address the two weaknesses of semaphores
  - Suggested by Dijkstra, developed more thoroughly by Hansen, and formalized by Hoare in the early 1970s
- Monitors are incorporated into many languages
  - Concurrent Pascal, Modula, Mesa
  - None incorporates the precise semantics of Hoare's formalization
Monitors (cont’d)

- A monitor is a shared object
  - With operations, internal state, and a number of condition variables (one queue per condition variable)
  - Only one operation of a given monitor may be active at a given point in time

- Semantics
  - If a process calls a busy monitor, delayed until the monitor is free
    - On behalf of its calling process, any monitor operation may suspend itself by waiting on a condition
    - An monitor operation may also signal a condition and one of the waiting processes is resumed, usually the one that waited first
Conditional Critical Regions (CCR)

- A region is permitted to access a *protected variable* when a *boolean condition* is true

  ```
  region protected_variable, when boolean_condition do
  ...
  end region
  ```

- Proposed by Hansen (1973)
- Incorporated in Edison, a concurrent language
- Influenced the synchronization mechanism of Ada95, Java, C#
- Synchronization in Java/C# is a blend of monitor and CCR
Synchronized in Java

- Every object can be accessed by multiple threads
  - Implement implicit mutual exclusion lock
    ```java
    synchronized (shared_obj) {
      ...
    }
    ```

- Method can be synchronized
  - The object for the synchronized method is mutually excluded
    ```java
    class list {
      ...
      synchronized void void insert(item a);
      synchronized item remove();
    }
    ```
Summary

- Programming models for parallelism
  - Shared memory – pthread, OpenMP, etc.
  - Distributed memory – MPI, Hadoop, etc.

- Synchronization
  - Mutual exclusion
  - Event synchronization

- Race condition
  - Incorrect/inconsistent results from improper synchronization
  - Atomic operation needs to be enforced properly

- Thread implementation
  - Coroutine – transfer (yield control explicitly)
  - Preemptive scheduler (context switch)

- Synchronization implementation
  - Spin lock, barrier, semaphore, monitor, conditional critical region