Concurrency

- Concurrency - the expression of an algorithm in which parts may be executed in parallel
- Scalable - algorithm is scalable if addition of processors increases speed
- Levels
  - Instruction level
  - Statement level
  - Unit (routine) level
  - Program level
- Physical vs logical concurrency

Thread of control - the sequence of program points reached as control flows through a program
- Coroutines have a single thread of control
- Concurrency allows for multiple threads of control

Uses
- Multiple processors
- Blocking (e.g., I/O)
- Expressibility
- Distributed computation
Unit-level Concurrency

- Task (process) – a unit that can be in concurrent execution with other units in the same program
  - Characteristics
    - Implicit initiation
    - No suspension of initiator
    - No explicit return of control
  - Heavyweight vs lightweight

Communication
- Shared memory
- Message passing
- Parameters

Synchronization
- Required when tasks share data
- Cooperative vs competitive synchronization
  - E.g., producer/consumer problem
    - Shared buffer
      - Cooperative synchronization
        - Producer doesn't add when full
        - Consumer doesn't remove when empty
      - Competitive synchronization
        - Cannot update buffer at same time
          - Race condition
  - Competition synchronization requires mutually exclusive access
    - Shared resource can be possessed by only one task
      - Task must request access and release possession

Figure 11.3
The need for competition synchronization.
Synchronization requires delaying task execution
- E.g. to gain access to resource
- Scheduler
- States
  - New
  - Ready
  - Running
  - Blocked
  - Dead
- Ready queue
- Deadlock

Design Issues
- Choice to include concurrency
  - Language or library
- Level
  - Statement, unit
- Mechanism for synchronization
  - Semaphore, monitor, message passing
- Can application influence scheduling
  - E.g. priorities
- Task creation
- Task initiation/termination
Semaphores

- Guard – a mechanism to allow code to execute only when a condition is true
- Semaphore is a guard
  - Counter
  - Task queue
- Operations
  - wait
  - release
- Producer/consumer
  - Cooperative synchronization
    - emptyspots/fullspots
  - Competitive synchronization
    - access
      - Binary semaphore
- Evaluation
  - Typically libraries
  - Reliability – race condition, synchronization errors

```c
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = 0;

task producer;
loop
  -- produce VALUE --
  wait(emptyspots); { wait for a space }
  deposit(VALUE);
  release(fullspots); { increase filled spaces }
end loop;
end producer;

task consumer;
loop
  wait(fullspots); { make sure it is not empty }
  fetch(VALUE);
  release(emptyspots); { increase empty spaces }
  -- consume VALUE --
end loop
end consumer;
```
Monitors

- Encapsulation of shared data and the operations upon them
  - Like data abstraction
  - Run-time handles competitive synchronization
- Competitive synchronization
  - Shared data in monitor not individual task
  - Monitor guarantees exclusive access
    - Routine calls blocked
- Cooperative synchronization
  - Responsibility of programmer
    - Typically use semaphores
- Evaluation
  - Ada, Java, C#
Message Passing

- Note not same as message passing in OO
- Synchronous vs asynchronous
  - Interrupting a task
  - Synchronous
    - Task A wants to send message to task B
    - Task A waits until task B ready to receive message
    - Rendezvous – interaction between tasks (information sharing)
    - Tasks A & B continue
  - Multiple requests to same task
    - Guarded commands
    - Non-determinism

Ada

- Synchronous message passing
- Heavyweight task
- Task declaration like a package
  - Specification
    - entry points
    - With parameters
  - Body
    - accept clause
    - Code for rendezvous
    - Executes by receiver
- Blocking
- Asymmetric message passing
- Initiation/Termination
  - Initiated when unit in which it is declared begins execution
  - Terminated at end of body and all dependent tasks completed
- **select clause**
  - If willing to accept any of a number of messages
    - Extended accept
  - Cooperative synchronization
    - Guard on accept
    - Usually in select
  - Competitive synchronization
    - Task as monitor
      - Shared data embedded in task
      - Only one accept can be executed at a time
  - Task priorities
  - E.g., buffer as task
  - Protected objects
    - Lightweight task
    - Essentially a monitor
    - Synchronous communication between tasks
      - E.g., buffer as protected object
  - Evaluation
    - Supports both message passing and monitors
task body Teller is
begin
  loop
    select
      accept Drive_Up(formal parameters) do
        ... end Drive_Up;
      or
      accept Walk_Up(formal parameters) do
        ... end Walk_Up;
    end select;
  end loop;
end Teller;

when not Pull BUFFER 
  accept Deposit(New Value) do
    ... end

task Bu_Task is
  entry Deposit(Item : in Integer); entry Fetch(Item : out Integer);
  end Bu_Task;
task body Buf_Task is
  BufSize : constant Integer := 100;
  Buf : array [1..BufSize] of Integer;
  Filled : Integer range 0..BufSize := 0;
  Next_In : Integer range 1..BufSize := 1;
  Next_Out : Integer range 1..BufSize := 1;
begin
  loop
    select
      when Filled < BufSize =>
        accept deposit(item : in Integer) do
          Buf(Next_In) := item;
          Filled := Filled + 1;
        end accept;
      when Filled = 0 =>
        accept fetchitem : out Integer do
          item := Buf(Next_Out);
          Filled := Filled - 1;
        end accept;
    end select;
    end loop
  end Buf_Task;

  task Producer; task Consumer;
  task body Producer is
    New_Value : Integer;
  begin
    loop
      -- produce New_Value --
      Buf_Task.deposit(New_Value);
    end loop
  end Producer;
  task body Consumer is
    Stored_Value : Integer;
  begin
    loop
      Buf_Task.fetch(Stored_Value);
      -- consume Stored_Value --
    end loop
  end Consumer;

protected Buffer is
  entry Deposit(item : in Integer);
  entry Fetch(item : out Integer);
private
  BufSize : constant Integer := 100;
  Buf : array [1..BufSize] of Integer;
  Filled : Integer range 0..BufSize := 0;
  Next_In : Integer range 1..BufSize := 1;
  Next_Out : Integer range 1..BufSize := 1;
end Buffer;
Java

- Threads are lightweight tasks
  - Subclass Thread
  - Implement Runnable interface
  - Pass instance to Thread constructor
  - Concurrent process is the method run
  - Completes when finished body
- main method and the GUI run on their own thread
- Thread methods
  - start - executes the run method concurrently
  - yield - surrender processor
  - sleep(millis) - blocks for at least millis
  - t.join(millis) - block until completed or waited millis
  - t.interrupt - sets t’s interrupted bit
  - t.isInterrupted
  - InterruptedException

```java
class MyThread extends Thread {
    public void run() {
        ...        
    }
    ...
    Thread myTh = new MyThread();
    myTh.start();

    public void run() {
        ...
        Thread myTh = new Thread();
        myTh.start();
        // Do part of the computation of this thread
        myTh.join(); // Wait for myTh to complete
        // Do the rest of the computation of this thread
    }
}
```
• Priority of thread can be set
• Semaphores
  – Library package
  – Counting semaphore
  – acquire, release
• Competitive synchronization
  – synchronized method
    * Must complete before any other synchronized method on
      same object can start
  – If all methods are synchronized then class is a monitor
  – Every object has a lock which must be (implicitly) acquired by
      synchronized method, released on termination
  – Every object has a queue of blocked threads
  – synchronized block
    * Block (implicitly) acquires lock

```java
fullpot = new Semaphore(0);
emptypot = new Semaphore(0);`
```
Cooperative synchronization
- Every object has a wait list
  - wait
    - Thread put on object's wait list
  - notify, notifyAll
    - One or all waiting threads released from object's wait list
  - Methods can only be called within a synchronized method/block
    - Uses object's lock
    - Only one queue
      - Typically notifyAll and each thread retests condition
- E.g. Java producer/consumer with buffer
  - Producer/consumer may implement Runnable

```java
try {
  while (!theCondition)
    wait();
    // Do whatever is needed after theCondition comes true
} catch (InterruptedException myProblem) {
...
}
```

```java
class Queue {
  private int [] q;
  private int head, tail, numItems, filled, drainList;
  public Queue(int size) {
    q = new int [size];
    filled = 0;
    head = -1;
    tail = -1;
    queues = size;
  }
  //** end of Queue constructor
  public synchronized void deposit (int item) throws InterruptedException {
    try {
      while (filled == queues)
        wait();
      q[head] = item;
      head = (head + 1) % queues;
      filled--;
      notifyAll();
    } catch (InterruptedException myProblem) {
...
    } //** end of deposit method
  }
}
```
public synchronized int fetch() throws InterruptedException {
    int item = 0;
    try {
        while (filled == 0) {
            wait();
            if (counter > queueSize) {
                throw new QueueException();
                filled = 0;
            } else
                notifyAll();
        }
        item = buffer.fetch();
        return item;
    }
}

class Producer extends Thread {
    private Queue buffer;
    public Producer(Queue que) {
        buffer = que;
    }
    public void run() {
        while (true) {
            // Create a new item
            buffer.enqueue(new_item);
        }
    }
}

class Consumer extends Thread {
    private Queue buffer;
    public Consumer(Queue que) {
        buffer = que;
    }
    public void run() {
        while (true) {
            // Consume the inserted item
            item = buffer.dequeue;
        }
    }
}

Queue buffer = new Queue(100);
Producer producer1 = new Producer(buffer);
Consumer consumer1 = new Consumer(buffer);
producer1.start();
consumer1.start();

class Producer implements Runnable {
    
    Producer producer1 = new Producer(buffer);
    Thread producerThread = new Thread(producer1);
    producerThread.start();
Atomic types
- Atomic add, inc, dec, getters and setters

Explicit lock
- Lock type and Condition Interface
- Can have multiple queues (Condition) on same lock

Evaluation
- Simple and effective
- Doesn't support distributed processing

C#

- Threads like Java
- Delegate mechanism allows any method to run as a thread
  - ThreadStart is delegate
- Thread methods
  - Start, Sleep, Join like Java
  - Abort
    - Throws exception thread can catch
- Server threads
  - Only runs when delegate called
  - Synchronous
    - Calling method blocks caller
  - Asynchronous
    - BeginInvoke, IAsyncResult, EndInvoke

```csharp
public void MyRun() {...
    Thread myThread = new Thread(new ThreadStart(MyRun));
    ...
    public float MyMethod(int x);
    ...
    Thread myThread = new Thread(new ThreadStart(MyMethod));
    IAsyncResult result = myThread.BeginInvoke(int, null, null);
    float returnValue = EndInvoke(result);
```
11.15

Synchronization
- Interlocked class
  - Atomic inc/dec
  - lock statement
    - In private instance method
      - token is this
    - In public instance method
      - token is an object created in the class
- Monitor class
  - Enter, Wait, Pulse, PulseAll, Exit

Evaluation
- Supports actors & servers
- Cleaner thread termination
- Locks implicitly unlocked

```csharp
Interlocked.Increment(ref counter);

lock(token) {
  // The critical section
}
```
Functional Languages

- MultiLisp
  - Adds concurrency to Scheme
  - `pcall`
    - Parameters evaluated concurrently
  - `future`
    - Function call in separate thread
    - Caller waits if result needed before computed
  - Not threadsafe if functions have side effects
- Concurrent ML
  - Adds threads and message passing to ML
  - `spawn`
    - Function runs in new thread
  - `channel`
    - Synchronous communication between threads
    - Non-deterministic choice of message to receive

```ml
(pcall f a b c d)

let val mychannel = channel()

send(mychannel, ?)
```

- F#
  - Part of .NET so similar to C#
    - Delegate is implicitly created
  - Synchronized update
    - `lock function`
      - Variable to be updates and function that updates
Statement-level Concurrency

- High-performance Fortran
  - Directives for optimization on multi-processor systems
  - `PROCESSORS`
    - Number of processors
  - `DISTRIBUTE`
    - Distributed elements of array to processors
      - Assuming non-shared memory
    - `BLOCK/CYCLIC`
  - `ALIGN`
    - Assures corresponding elements of array on same machine
  - `FORALL` statement
    - Assignment statements that can be executed concurrently

```fortran
real list_1 (1000), list_2 (1000)
integer list_3 (500), list_4 (500)
opps processors proc [10]
opps distribute (block) onto proc 
  :: list_1, list_2
opps align list_3 (index) with list_4 (index)
  list_1 (index) = list_2 (index)
  list_3 (index) = list_4 (index)

forall (index = 1:1000)
  list_1 (index) = list_2 (index)
endforall
```