Chapter 13

Concurrency
Chapter 13 Topics

- Introduction
- Introduction to Subprogram-Level Concurrency
- Semaphores
- Monitors
- Message Passing
- Ada Support for Concurrency
- Java Threads
- C# Threads
- Statement-Level Concurrency
Introduction

• Concurrency can occur at four levels:
  – Machine instruction level
  – High-level language statement level
  – Unit level
  – Program level

• Because there are no language issues in instruction– and program–level concurrency, they are not addressed here
Multiprocessor Architectures

• Late 1950s – one general-purpose processor and one or more special-purpose processors for input and output operations
• Early 1960s – multiple complete processors, used for program-level concurrency
• Mid-1960s – multiple partial processors, used for instruction-level concurrency
• Single-Instruction Multiple-Data (SIMD) machines
• Multiple-Instruction Multiple-Data (MIMD) machines
  – Independent processors that can be synchronized (unit-level concurrency)
Categories of Concurrency

- Categories of Concurrency:
  - Physical concurrency – Multiple independent processors (multiple threads of control)
  - Logical concurrency – The appearance of physical concurrency is presented by time-sharing one processor (software can be designed as if there were multiple threads of control)

- A thread of control in a program is the sequence of program points reached as control flows through the program

- Coroutines (quasi-concurrency) have a single thread of control
Motivations for Studying Concurrency

• Involves a different way of designing software that can be very useful—many real-world situations involve concurrency
• Multiprocessor computers capable of physical concurrency are now widely used
Introduction to Subprogram-Level Concurrency

• A task or process is a program unit that can be in concurrent execution with other program units.

• Tasks differ from ordinary subprograms in that:
  – A task may be implicitly started.
  – When a program unit starts the execution of a task, it is not necessarily suspended.
  – When a task’s execution is completed, control may not return to the caller.

• Tasks usually work together.
Two General Categories of Tasks

- Heavyweight tasks execute in their own address space
- Lightweight tasks all run in the same address space
- A task is disjoint if it does not communicate with or affect the execution of any other task in the program in any way
Task Synchronization

- A mechanism that controls the order in which tasks execute
- Two kinds of synchronization
  - Cooperation synchronization
  - Competition synchronization
- Task communication is necessary for synchronization, provided by:
  - Shared nonlocal variables
  - Parameters
  - Message passing
Kinds of synchronization

- Cooperation: Task A must wait for task B to complete some specific activity before task A can continue its execution, e.g., the producer–consumer problem
- Competition: Two or more tasks must use some resource that cannot be simultaneously used, e.g., a shared counter
  - Competition is usually provided by mutually exclusive access (approaches are discussed later)
Need for Competition Synchronization

Value of TOTAL 3

Task A
- Fetch TOTAL
- Add 1
- Store TOTAL

Task B
- Fetch TOTAL
- Multiply by 2
- Store TOTAL

Time
Scheduler

- Providing synchronization requires a mechanism for delaying task execution
- Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors
Task Execution States

• New – created but not yet started
• Ready – ready to run but not currently running (no available processor)
• Running
• Blocked – has been running, but cannot continue now (usually waiting for some event to occur)
• Dead – no longer active in any sense
Liveness and Deadlock

• Liveness is a characteristic that a program unit may or may not have
  – In sequential code, it means the unit will eventually complete its execution

• In a concurrent environment, a task can easily lose its liveness

• If all tasks in a concurrent environment lose their liveness, it is called deadlock
Design Issues for Concurrency

- Competition and cooperation synchronization
- Controlling task scheduling
- How and when tasks start and end execution
- How and when are tasks created
CUDA

Compute Unified Device Architecture

- Parallel computing architecture by NVIDIA
- Developers have access to the virtual instruction set and memory of the parallel computational elements in CUDA GPUs.
- Using GPUs as CPUs
- GPUs emphasize executing many concurrent threads slowly, CPUs: executing a single thread very fast.
- Solving general purpose problems on GPUs is known as GPGPU.

Methods of Providing Synchronization

• Semaphores
• Monitors
• Message Passing
Semaphores

• Dijkstra – 1965
• A semaphore is a data structure consisting of a counter and a queue for storing task descriptors
• Semaphores can be used to implement guards on the code that accesses shared data structures
• Semaphores have only two operations, wait and release (originally called P and V by Dijkstra)
• Semaphores can be used to provide both competition and cooperation synchronization
Cooperation Synchronization with Semaphores

• Example: A shared buffer

• Use two semaphores for cooperation: \texttt{emptyspots} and \texttt{fullspots}

• The semaphore counters are used to store the numbers of empty spots and full spots in the buffer

• The buffer is implemented as an ADT with the operations \texttt{DEPOSIT} and \texttt{FETCH} as the only ways to access the buffer
Cooperation Synchronization with Semaphores (continued)

• **DEPOSIT must first check emptyspots to see if there is room in the buffer**

• If there is room, the counter of emptyspots is decremented and the value is inserted

• If there is no room, the caller is stored in the queue of emptyspots

• **When DEPOSIT is finished, it must increment the counter of fullspots**
Cooperation Synchronization with Semaphores (continued)

• **FETCH must first check fullspots to see if there is a value**
  - If there is a full spot, the counter of fullspots is decremented and the value is removed
  - If there are no values in the buffer, the caller must be placed in the queue of fullspots
  - When **FETCH** is finished, it increments the counter of emptyspots

• **The operations of FETCH and DEPOSIT on the semaphores are accomplished through two semaphore operations named wait and release**
Semaphores: Wait Operation

wait(aSemaphore)
if aSemaphore’s counter > 0 then
decrement aSemaphore’s counter
else
put the caller in aSemaphore’s queue
attempt to transfer control to a ready task
// if the task ready queue is empty,
// deadlock occurs
end
release(aSemaphore)
if aSemaphore’s queue is empty then
    increment aSemaphore’s counter
else
    put the calling task in the task ready queue
    transfer control to a task from aSemaphore’s queue
end
Producer Consumer Code

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

task producer;
    loop
    // produce VALUE
    wait (emptyspots); {wait for space}
    DEPOSIT(VALUE);
    release(fullspots); {increase filled}
    end loop;
end producer;
```
task consumer;
    loop
    wait (fullspots); {wait till not empty}
    FETCH(VALUE);
    release(emptyspots); {increase empty}
    // consume VALUE
    end loop;
end consumer;
• A third semaphore, named `access`, is used to control access (competition synchronization)
  - The counter of `access` will only have the values 0 and 1
  - Such a semaphore is called a **binary semaphore** (also: mutex)

• Note that wait and release must be atomic!
Producer Consumer Code

```java
semaphore access, fullspots, emptyspots;
access.count = 0;
fullspots.count = 0;
emptyspots.count = BUFLEN;

task producer;
    loop
        // produce VALUE
        wait(emptyspots);  {wait for space}
        wait(access);     {wait for access}
        DEPOSIT(VALUE);
        release(access);  {relinquish access}
        release(fullspots); {increase filled}
    end loop;
end producer;
```
Producer Consumer Code

```plaintext
task consumer;
    loop
    wait(fullspots); {wait till not empty}
    wait(access); {wait for access}
    FETCH(VALUE);
    release(access); {relinquish access}
    release(emptyspots); {increase empty}
    // consume VALUE
    end loop;
end consumer;
```
Evaluation of Semaphores

- Misuse of semaphores can cause failures in cooperation synchronization, e.g., the buffer will overflow if the wait of fullspots is left out.
- Misuse of semaphores can cause failures in competition synchronization, e.g., the program will deadlock if the release of access is left out.
Monitors

- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data
- Ada, Java, C#
Competition Synchronization

- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
  - Monitor implementation guarantee synchronized access by allowing only one access at a time
  - Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the call
Cooperation Synchronization

• Cooperation between processes is still a programming task
  – Programmer must guarantee that a shared buffer does not experience underflow or overflow
Evaluation of Monitors

- A better way to provide competition synchronization than are semaphores
- Semaphores can be used to implement monitors
- Monitors can be used to implement semaphores
- Support for cooperation synchronization is very similar as with semaphores, so it has the same problems
Message Passing

• Message passing is a general model for concurrency
  – It can model both semaphores and monitors
  – It is not just for competition synchronization

• Central idea: task communication is like seeing a doctor—most of the time she waits for you or you wait for her, but when you are both ready, you get together, or rendezvous
Message Passing Rendezvous

- To support concurrent tasks with message passing, a language needs:
  - A mechanism to allow a task to indicate when it is willing to accept messages
  - A way to remember who is waiting to have its message accepted and some “fair” way of choosing the next message

- When a sender task’s message is accepted by a receiver task, the actual message transmission is called a rendezvous
Ada Support for Concurrency

• The Ada 83 Message-Passing Model

  - Ada tasks have specification and body parts, like packages; the spec has the interface, which is the collection of entry points:

```ada
task Task_Example is
  entry ENTRY_1 (Item : in Integer);
end Task_Example;
```
Task Body

• The **body** task describes the action that takes place when a rendezvous occurs

• Entry points in the spec are described with `accept` clauses in the body

  ```
  accept entry_name (formal parameters) do
    ...
  end entry_name
  ```

• A task that sends a message is suspended while waiting for the message to be accepted and during the rendezvous
Example of a Task Body

task body Task_Example is
  begin
    loop
      accept Entry_1 (Item: in Integer) do
        ...
        end Entry_1;
    end loop;
  end Task_Example;
Ada Message Passing Semantics

- The task executes to the top of the `accept` clause and waits for a message.
- During execution of the `accept` clause, the sender is suspended.
- `accept` parameters can transmit information in either or both directions.
- Every `accept` clause has an associated queue to store waiting messages.
Rendezvous Time Lines

Figure 13.3
Two ways a rendezvous with TASK EXAMPLE can occur:

(a) Task_example waits for Sender

(b) Sender waits for Task_example
Message Passing: Server/Actor Tasks

- A task that has accept clauses, but no other code is called a server task (the example above is a server task)
- A task without accept clauses is called an actor task
  - An actor task can send messages to other tasks
  - Note: A sender must know the entry name of the receiver, but not vice versa (asymmetric)
Graphical Representation of a Rendezvous

Figure 13.4
Graphical representation of a rendezvous caused by a message sent from task A to task B.
Example: Actor Task

```
task Water_Monitor;  -- specification task body
Water_Monitor is  -- body
  begin
    loop
      if Water_Level > Max_Level
        then Sound_Alarm;
      end if;
      delay 1.0;  -- No further execution
                  -- for at least 1 second
    end loop;
  end Water_Monitor;
```
Multiple Entry Points

- Tasks can have more than one entry point
  - The specification task has an entry clause for each
  - The task body has an accept clause for each entry clause, placed in a select clause, which is in a loop
A Task with Multiple Entries

task body Teller is
    loop
        select
            accept Drive_Up(formal params) do
                ...
            end Drive_Up;
            ...
        or
            accept Walk_Up(formal params) do
                ...
            end Walk_Up;
            ...
        end select;
    end loop;
end Teller;
Semantics of Tasks with Multiple `accept` Clauses

- If exactly one `entry` queue is nonempty, choose a message from it
- If more than one `entry` queue is nonempty, choose one, nondeterministically, from which to accept a message
- If all are empty, wait
- The construct is often called a `selective wait`
- Extended `accept` clause – code following the clause, but before the next clause
  - Executed concurrently with the caller
Cooperation Synchronization with Message Passing

- **Provided by** Guarded `accept` clauses

```plaintext
when not Full(Buffer) =>
    accept Deposit (New_Value) do
```

- **An `accept` clause with a with a `when` clause is either open or closed**
  - A clause whose guard is true is called open
  - A clause whose guard is false is called closed
  - A clause without a guard is always open
Semantics of \texttt{select} with Guarded \texttt{accept} Clauses:

- \texttt{select} first checks the guards on all clauses
- If exactly one is open, its queue is checked for messages
- If more than one are open, non-deterministically choose a queue among them to check for messages
- If all are closed, it is a runtime error
- A \texttt{select} clause can include an \texttt{else} clause to avoid the error
  - When the \texttt{else} clause completes, the loop repeats
Example of a Task with Guarded `accept` Clauses

- Note: The station may be out of gas and there may or may not be a position available in the garage

```plaintext
task Gas_Station_Attendant is
  entry Service_Island (Car : Car_Type);
  entry Garage (Car : Car_Type);
end Gas_Station_Attendant;
```
Example of a Task with Guarded `accept` Clauses

task body  Gas_Station_Attendant is
  begin
    loop
      select
        when Gas_Available =>
          accept Service_Island (Car : Car_Type) do
            Fill_With_Gas (Car);
          end Service_Island;
        or
          when Garage_Available =>
            accept Garage (Car : Car_Type) do
              Fix (Car);
            end Garage;
        else
          Sleep;
      end select;
    end loop;
  end Gas_Station_Attendant;

Competition Synchronization with Message Passing

- Modeling mutually exclusive access to shared data
- Example—a shared buffer
- Encapsulate the buffer and its operations in a task
- Competition synchronization is implicit in the semantics of `accept` clauses
  - Only one `accept` clause in a task can be active at any given time
Task Termination

• The execution of a task is completed if control has reached the end of its code body
• If a task has created no dependent tasks and is completed, it is terminated
• If a task has created dependent tasks and is completed, it is not terminated until all its dependent tasks are terminated
The `terminate` Clause

- A `terminate` clause in a `select` is just a `terminate` statement.
- A `terminate` clause is selected when no `accept` clause is open.
- When a `terminate` is selected in a task, the task is terminated only when its master and all of the dependents of its master are either completed or are waiting at a `terminate`.
- A block or subprogram is not left until all of its dependent tasks are terminated.
Message Passing Priorities

- The priority of any task can be set with the `pragma priority`

  `pragma Priority (expression);`

- The priority of a task applies to it only when it is in the task ready queue
Binary Semaphores

• For situations where the data to which access is to be controlled is NOT encapsulated in a task

```vhd
task Binary_Semaphore is
    entry Wait;
    entry release;
end Binary_Semaphore;

task body Binary_Semaphore is
    begin
        loop
            accept Wait;
            accept Release;
        end loop;
end Binary_Semaphore;
```
Concurrency in Ada 95

- Ada 95 includes Ada 83 features for concurrency, plus two new features
  - Protected objects: A more efficient way of implementing shared data to allow access to a shared data structure to be done without rendezvous
  - Asynchronous communication
Ada 95: Protected Objects

- A protected object is similar to an abstract data type
- Access to a protected object is either through messages passed to entries, as with a task, or through protected subprograms
- A protected procedure provides mutually exclusive read–write access to protected objects
- A protected function provides concurrent read–only access to protected objects
Asynchronous Communication

• Provided through asynchronous `select` structures

• An asynchronous `select` has two triggering alternatives, an entry clause or a delay
  – The entry clause is triggered when sent a message
  – The delay clause is triggered when its time limit is reached
Evaluation of the Ada concurrency

- Message passing model of concurrency is powerful and general
- Protected objects are a better way to provide synchronized shared data
- In the absence of distributed processors, the choice between monitors and tasks with message passing is somewhat a matter of taste
- For distributed systems, message passing is a better model for concurrency
Java Threads

• The concurrent units in Java are methods named **run**
  - A **run** method code can be in concurrent execution with other such methods
  - The process in which the **run** methods execute is called a **thread**

```java
Class myThread extends Thread
    public void run () {... }
}

... Thread myTh = new MyThread (); myTh.start ();```
Controlling Thread Execution

• The `Thread` class has several methods to control the execution of threads
  - The `yield` (no parameters) is a request from the running thread to voluntarily surrender the processor
  - The `sleep` (single parameter) method can be used by the caller of the method to block the thread
  - The `join` method is used to force a method to delay its execution until the run method of another thread has completed its execution
• The `join` method is used to force a method to delay its execution until the run method of another thread has completed its execution

```java
public void run ()
{
    Thread myTh = new MyThread ();
    myTh.start ();
    //do part of the computation for this thread
    myTh.join ();
    //do the rest of the computation for this thread
}
```
Thread Priorities

- A thread’s default priority is the same as the thread that create it
  - If `main` creates a thread, its default priority is `NORM_PRIORITY`
- Threads defined two other priority constants, `MAX_PRIORITY` and `MIN_PRIORITY`
- The priority of a thread can be changed with the methods `setPriority`
A method that includes the `synchronized` modifier disallows any other method from running on the object while it is in execution.

```java
public synchronized void deposit(int i) {...}
public synchronized int fetch() {...}
```

The above two methods are synchronized which prevents them from interfering with each other.

If only a part of a method must be run without interference, it can be synchronized using the `synchronized` statement:

```java
synchronized (expression) statement
```
Cooperation Synchronization with Java Threads

- Cooperation synchronization in Java is achieved via `wait`, `notify`, and `notifyAll` methods
  - All methods are defined in `Object`, which is the root class in Java, so all objects inherit them
- The `wait` method must be called in a loop
- The `notify` method is called to tell one waiting thread that the event it was waiting has happened
- The `notifyAll` method awakens all of the threads on the object’s wait list
Cooperation Synchronization with Java Threads

- try
  { 
    while (!theCondition)
      wait();
      //do whatever is needed after theCondition comes true
  }

  catch (InterruptedException myProblem)
  {
    ...
  }
Java’s Thread Evaluation

• Java’s support for concurrency is relatively simple but effective
• Not as powerful as Ada’s tasks
C# Threads

• Loosely based on Java but there are significant differences

• Basic thread operations

  – Any method can run in its own thread (only run in Java)
  – A thread is created by creating a Thread object
  – Creating a thread does not start its concurrent execution; it must be requested through the Start method
  – A thread can be made to wait for another thread to finish with Join
  – A thread can be suspended with Sleep
  – A thread can be terminated with Abort
Synchronizing Threads

• Three ways to synchronize C# threads
  – The **Interlocked class**
    • Used when the only operations that need to be synchronized are incrementing or decrementing of an integer, e.g. `Interlocked.Increment(ref counter)`
  – The **lock statement**
    • Used to mark a critical section of code in a thread
      `lock (expression) { //critical section... }`
  – The **Monitor class**
    • Provides four methods that can be used to provide more sophisticated synchronization
C#'s Concurrency Evaluation

- An advance over Java threads, e.g., any method can run its own thread
- Thread termination is cleaner than in Java
- Synchronization is more sophisticated
- Lightweight as with Java
Statement-Level Concurrency

• Objective: Provide a mechanism that the programmer can use to inform compiler of ways it can map the program onto multiprocessor architecture

• Minimize communication among processors and the memories of the other processors
High-Performance Fortran

• A collection of extensions that allow the programmer to provide information to the compiler to help it optimize code for multiprocessor computers

• Specify the number of processors, the distribution of data over the memories of those processors, and the alignment of data
Primary HPF Specifications

• Number of processors
  
  !HPF$ PROCESSORS procs (n)

• Distribution of data

  !HPF$ DISTRIBUTED (kind) ONTO procs :: identifier_list

  - kind can be BLOCK (distribute data to processors in blocks) or CYCLIC (distribute data to processors one element at a time)

• Relate the distribution of one array with that of another

  ALIGN array1_element WITH array2_element
Statement-Level Concurrency Example

```plaintext
REAL  list_1(1000), list_2(1000)
INTEGER list_3(500), list_4(501)

!HPF$  PROCESORS  proc (10)

!HPF$  DISTRIBUTE (BLOCK)  ONTO  procs :: list_1, list_2

!HPF$  ALIGN  list_1(index)  WITH  list_4 (index+1)

...  
list_1 (index) = list_2(index)
List_3 (index) = list_4(index+1)
```
Statement-Level Concurrency (continued)

• **FORALL** statement is used to specify a list of statements that may be executed concurrently

```
FORALL (index = 1:1000) list_1(index) = list_2(index)
```

• Specifies assignment of list_2 elements to the corresponding elements of list_1.

• Conceptually specifies that all 1,000 RHSs of the assignments can be evaluated before any assignment takes place
Summary

- Concurrent execution can be at the instruction, statement, or subprogram level
- Physical concurrency: when multiple processors are used to execute concurrent units
- Logical concurrency: concurrent units are executed on a single processor
- Two primary facilities to support subprogram concurrency: competition synchronization and cooperation synchronization
- Mechanisms: semaphores, monitors, rendezvous, threads
- High-Performance Fortran provides statements for specifying how data is to be distributed over the memory units connected to multiple processors