Chapter 13

Concurrency
Chapter 13 Topics

• Introduction
• Introduction to Subprogram-Level Concurrency
• Semaphores
• Monitors
• Message Passing
• Ada support for Concurrency
• Java Threads
• C# Threads
• Concurrency in Functional Languages
• 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
- A primary focus of this chapter is shared memory MIMD machines (multiprocessors)
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)

- Coroutines (**quasi-concurrency**) have a single thread of control
- A **thread of control** in a program is the sequence of program points reached as control flows through the program
Motivations for the Use of Concurrency

• Multiprocessor computers capable of physical concurrency are now widely used
• Even if a machine has just one processor, a program written to use concurrent execution can be faster than the same program written for nonconcurrent execution
• Involves a different way of designing software that can be very useful—many real-world situations involve concurrency
• Many program applications are now spread over multiple machines, either locally or over a network
Introduction to Subprogram–Level Concurrency

• A *task* or *process* or *thread* 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 – more efficient
- 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

Task A: \( \text{TOTAL} = \text{TOTAL} + 1 \)
Task B: \( \text{TOTAL} = 2 \times \text{TOTAL} \)

Depending on order, there could be four different results.
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 now continue (usually waiting for some event to occur)
- **Dead** – no longer active in any sense
Task Execution States (continued)
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 can an application influence task scheduling
• How and when tasks start and end execution
• How and when are tasks created
  * The most important issue
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
  - A task descriptor is a data structure that stores all of the relevant information about the execution state of the task
- 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
- The buffer is implemented as an ADT with the operations `DEPOSIT` and `FETCH` as the only ways to access the buffer
- Use two semaphores for cooperation: `emptyspots` and `fullspots`
- The semaphore counters are used to store the numbers of empty spots and full spots in 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 and Release Operations

```plaintext
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 and Consumer Tasks

```plaintext
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;
```
Competition Synchronization with Semaphores

- 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*
- Note that wait and release must be atomic!
Producer Code for Semaphores

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;
Consumer Code for Semaphores

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

• Ada, Java, C#
• The idea: encapsulate the shared data and its operations to restrict access
• A monitor is an abstract data type for shared data
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.
- A task that sends a message is suspended while waiting for the message to be accepted and during the rendezvous.
- Entry points in the spec are described with **accept** clauses in the body:

  ```
  accept entry_name (formal parameters) do
  ...
  end entry_name;
  ```
Example of a Task Body

task body Task_Example is

begin

loop

accept Entry_1 (Item: in Float) 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

(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
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

```vhdl
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 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

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

- An `accept` clause 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 `select` with Guarded `accept` Clauses:

- `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 `select` clause can include an `else` clause to avoid the error
  - When the `else` clause completes, the loop repeats
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
Partial Shared Buffer Code

```vhdl
-- task body Buf_Task is
Bufsize : constant Integer := 100;
Buf : array (1..Bufsize) of Integer;
Filled : Integer range 0..Bufsize := 0;
Next_In, Next_Out : Integer range 1..Bufsize := 1;
begin
  loop
    select
      when Filled < Bufsize =>
        accept Deposit(Item : in Integer) do
          Buf(Next_In) := Item;
          Next_In := (Next_In mod Bufsize) + 1;
          Filled := Filled + 1;
        end Deposit;
      or
        ...
    end loop;
  end BufTask;
end Buf_Task;
```
A Consumer Task

task Consumer;
task body Consumer is
    Stored_Value : Integer;
begin
    loop
        Buf_Task.Fetch(Stored_Value);
        -- consume Stored_Value -
    end loop;
end Consumer;
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
Evaluation of the Ada

• 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` is a request from the running thread to voluntarily surrender the processor
  - The `sleep` 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
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`
Competition Synchronization with Java Threads

• A method that includes the `synchronized` modifier disallows any other method from running on the object while it is in execution

  ...
  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 thru `synchronized statement`

  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
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
  – 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
  - **The `lock` statement**
    - Used to mark a critical section of code in a thread
      ```
      lock (expression) {
          ...
      }
      ```
  - **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
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
  \(!\text{HPF$ PROCESSORS procs } (n)\)

• Distribution of data
  \(!\text{HPF$ DISTRIBUTE (kind) ONTO procs :: identifier_list}\)
  
  - kind can be \text{BLOCK} (distribute data to processors in blocks) or \text{CYCLIC} (distribute data to processors one element at a time)

• Relate the distribution of one array with that of another
  \text{ALIGN array1_element WITH array2_element}
Statement–Level Concurrency Example

REAL list_1(1000), list_2(1000)
INTEGER list_3(500), list_4(501)
!HPF$ PROCESSORS 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)
• **FORALL** statement is used to specify a list of statements that may be executed concurrently

  
  FORALL (index = 1:1000)
  
  \[
  \text{list}_1(\text{index}) = \text{list}_2(\text{index})
  \]

• 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