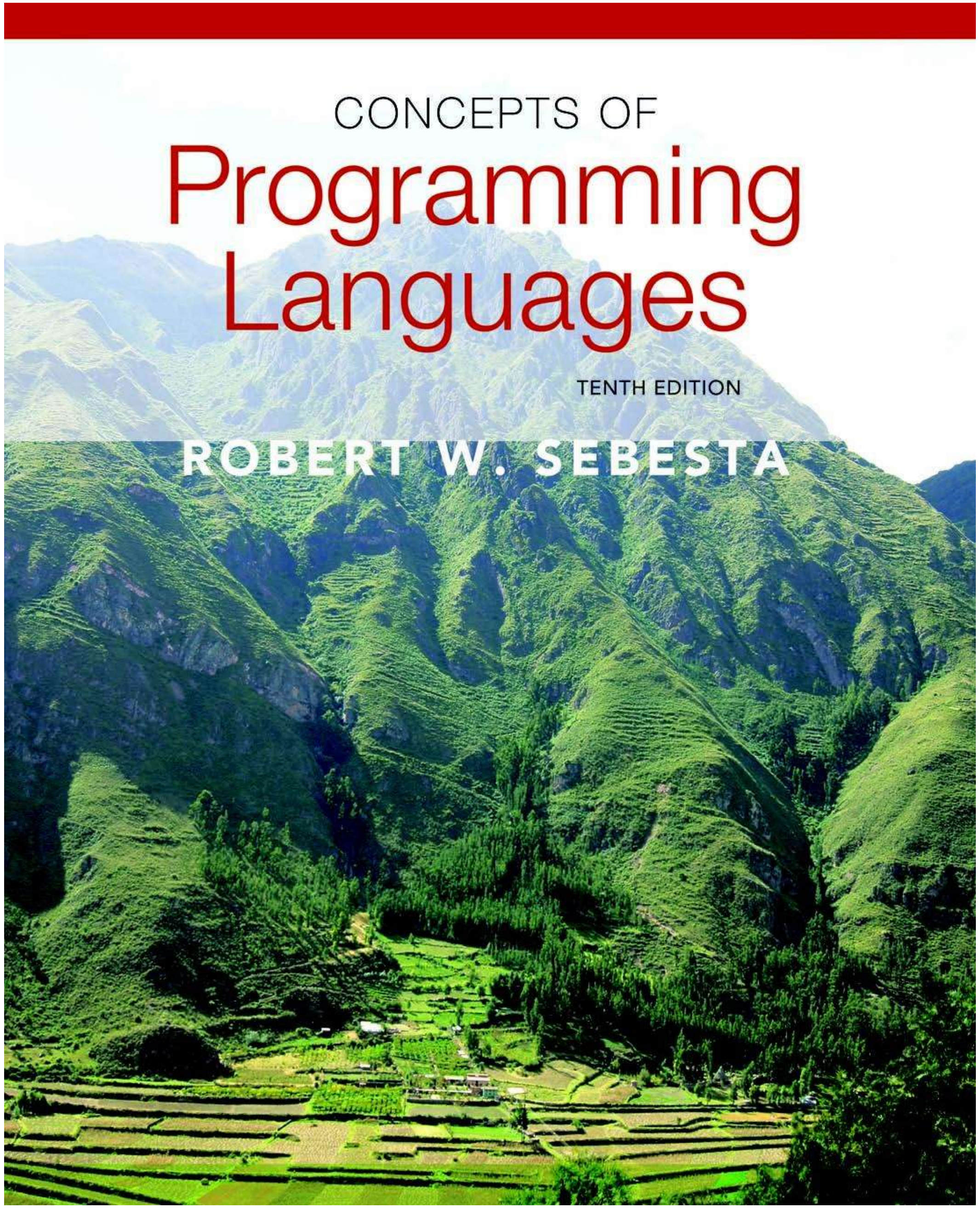
Chapter 13

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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)
- 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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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)

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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*

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

Producer and Consumer Tasks

```
semaphore fullspots, emptyspots;
fullstops.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;
```

Copyright © 2012 Addison-Wesley. All rights reserved.

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!

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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*

Copyright © 2012 Addison-Wesley. All rights reserved.

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:

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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;
```

Copyright © 2012 Addison-Wesley. All rights reserved.

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;
```

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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)

Copyright © 2012 Addison-Wesley. All rights reserved.

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 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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

Message Passing Priorities

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

  pragma `Priority (`static expression`);`

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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*

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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`

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

Java’s Thread Evaluation

- Java’s support for concurrency is relatively simple but effective
- Not as powerful as Ada’s tasks

Copyright © 2012 Addison-Wesley. All rights reserved.

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`

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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

Copyright © 2012 Addison-Wesley. All rights reserved.

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 united 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

Copyright © 2012 Addison-Wesley. All rights reserved.