

Concurrency – Topics

- Introduction
- Introduction to Subprogram–Level Concurrency
- Semaphores
- Monitors
- Message Passing
- Java Threads

1

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

2

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

3

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

4

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

5

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

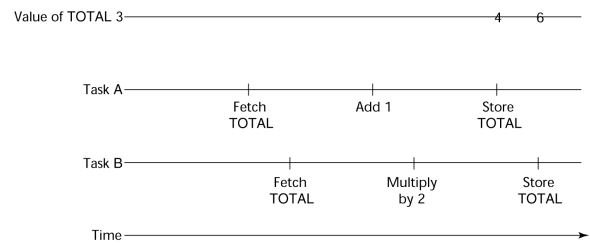
6

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)

7

Need for Competition Synchronization



8

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

9

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

10

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

11

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

12

Methods of Providing Synchronization

- Semaphores
- Monitors
- Message Passing

13

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

14

Cooperation Synchronization with Semaphores

- Example: A shared 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
- The buffer is implemented as an ADT with the operations `DEPOSIT` and `FETCH` as the only ways to access the buffer

15

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`

16

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

17

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
```

18

Semaphores: Release Operation

```
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
```

19

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;
```

20

Producer Consumer Code

```
task consumer;
  loop
  wait (fullspots);{wait till not empty}}
  FETCH(VALUE);
  release(emptyspots); {increase empty}
  -- consume VALUE --
  end loop;
end consumer;
```

21

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!

22

Producer Consumer Code

```
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;
```

23

Producer Consumer Code

```
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;
```

24

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

25

Monitors

- The idea: encapsulate the shared data and its operations to restrict access
- A monitor is an abstract data type for shared data
- Java

26

Competition Synchronization

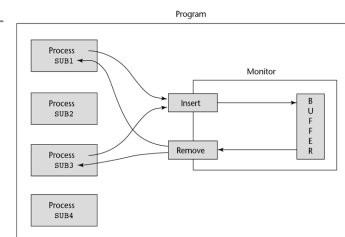
- Shared data is resident in the monitor (rather than in the client units)
- All access resident in the monitor
 - Monitor implementation guarantee synchronized access by allowing only one access at a time
 - Calls to monitor procedures are implicitly queued if the monitor is busy at the time of the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

27

Cooperation Synchronization

- Cooperation between processes is still a programming task
 - Programmer must guarantee that a shared buffer does not experience underflow or overflow

Figure 13.2
A program using a monitor to control access to a shared buffer



28

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

29

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

30

Message Passing Rendezvous

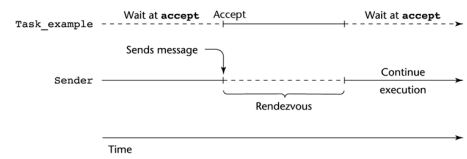
- To support concurrent tasks with message passing, a language needs:
 - A mechanism to allow a task to indicate when it is willing to accept messages
 - A way to remember who is waiting to have its message accepted and some "fair" way of choosing the next message
- When a sender task's message is accepted by a receiver task, the actual message transmission is called a rendezvous

31

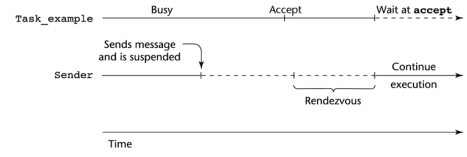
Rendezvous Time Lines

Figure 13.3

Two ways a rendezvous with TASK_EXAMPLE can occur



(a) Task_example waits for Sender



(b) Sender waits for Task_example

32

Message Passing: Server/Actor Tasks

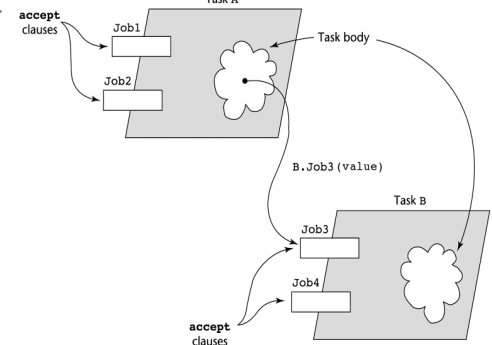
- A task that has `accept` clauses, but no other code is called a **server task** (the example above is a server task)
- A task without `accept` clauses is called an **actor task**
 - An actor task can send messages to other tasks
 - Note: A sender must know the `entry` name of the receiver, but not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

33

Graphical Representation of a Rendezvous

Figure 13.4

Graphical representation of a rendezvous caused by a message sent from task A to task B



34

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;
```

35

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

36

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;
```

37

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

38

Cooperation Synchronization with Message Passing

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

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

- 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

39

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

40

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

```
task Gas_Station_Attendant is
  entry Service_Island (Car : Car_Type);
  entry Garage (Car : Car_Type);
end Gas_Station_Attendant;
```

41

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;
```

42

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

43

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

44

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

45

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

46

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

47

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();
```

48

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

49

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`

50

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 using the `synchronized` statement

```
synchronized (expression)
statement
```

51

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

52

Creating and Running Threads in Java

- There are three basic ways to use Threads in Java
 - Extend the Thread Class
 - Implement the Runnable Interface
 - Use anonymous inner classes

53

Extending the Thread Class

Extend the thread class and override the `run()` method

```
public class MyThread extends Thread
{
    public static void run()
    {
        System.out.println("Do something cool here.");
    }
}
```

Then create and start the thread:

```
Thread myThread = new MyThread();
myThread.start();
```

54

Implementing the Runnable Interface

Any object that implements the Runnable Interface can be passed as a parameter to the constructor of a Thread object.

```
public class MyClass extends SomeOtherClass implements Runnable{
    public MyClass(){
        Thread thread = new Thread(this);
        thread.start();
    }

    public void run(){
        System.out.println("Do something cool here.");
    }
}
```

The MyClass class implements Runnable, passes itself into a new thread, then starts that thread which executes MyClass.run() .

55

Using Anonymous Inner Classes

An anonymous inner class can be used to start a Thread when inheriting the Thread class or implementing the Runnable interface is not desirable.

```
new Thread() {
    public void run() {
        System.out.println("Do something cool here.");
    }
}.start();
```

This piece of code creates an instance of a nameless class that inherits the Thread class and overrides the run() method.

This technique should be used carefully because it can easily become hard to read.

56

The join() and sleep() Methods

- Thread.join();
 - If you are in one Thread and you want to wait for another Thread to finish then call the other Thread object's join() method. The current Thread will remain inactive until the outside Thread finishes its run() method.
- Thread.sleep(int);
 - The sleep(int) method causes a Thread to be inactive for the specified number of milliseconds during which it will take up no clock cycles.

57

Avoiding Deadlock

Deadlock is the result of two threads that stall because they are waiting on each other to do something.

Fig 1

- Thread A acquires lock 1
- Thread B acquires lock 2

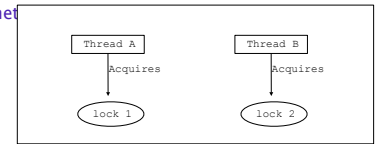
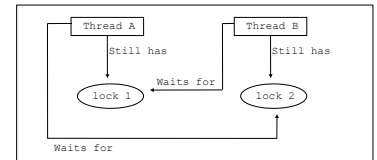


Fig 2

- Thread B waits for lock 1 to be released
- Thread A waits for lock 2 to be released



Both threads are now waiting for the other to finish so neither will continue. To avoid deadlock write your synchronization code so that deadlock will not occur.

There are no blanket fixes for deadlock, but there are detectors.

The Java Event Model

- When your program operates in a graphical environment it can be accessed by at least two threads even if you are not using threads explicitly.
- The two threads are the main thread that runs your program and the AWT event dispatch thread which handles user input in order to allow event driven program design.
- Because of this you should always keep synchronization in mind even if you are not creating and using threads of your own.

59

When to Use Threads

- For the purpose of game design threads are useful to prevent lengthy operations from hindering the playing experience. (The player will not be happy if hitting the quiksave button causes the game to freeze for several seconds)
- Other examples of smart thread use include:
 - Loading files from the disk
 - Network communication, such as sending high scores to a server
 - Massive calculations, such as terrain generation

When not to use Threads

- In games there are often many things happening at the same time. This does not mean that every bullet, spaceship, and flake of snow should have its own thread. In fact such a design would be very problematic and slow. Later in the book we will see different methods for “lots of stuff happening”.

61

Thread Pools

A thread pool is used to limit the number of threads being used at a time.

Here we have an example ThreadPool class and a test program for that class.

Java's Thread Evaluation

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

63

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

64