Concurrent Object-Oriented Programming

Bertrand Meyer, Volkan Arslan
Lecture 5: Classic Approaches to Concurrent Programming
Outline

Concurrent Programming
  ▪ Definition of concurrent programming
  ▪ Notion of process
  ▪ Process representation
  ▪ A simple concurrent (embedded) system

Shared variable-based synchronization and communication
  ▪ Mutual exclusion and condition synchronization
  ▪ Busy waiting
  ▪ (Suspend and resume)
  ▪ Semaphores
  ▪ Conditional critical regions
  ▪ Monitors
  ▪ Protected objects
  ▪ Synchronized methods
Concurrent Programming
Definition: Concurrent Programming (Ben-Ari, 1982)

- is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems

- Implementation of parallelism is a topic in computer systems (hardware and software) that is essentially independent of concurrent programming

- Concurrent programming is important because it provides an abstract setting in which we can study parallelism without getting into the implementation details
Work of Dijkstra (1968)
- A concurrent program is a collection of autonomous sequential processes, executing (logically) in parallel
- Each process has a single thread of control
- The actual implementation (i.e. execution) of a collection of processes usually takes one of three forms.

**Multiprogramming**
- processes multiplex their executions on a single processor

**Multiprocessing**
- processes multiplex their executions on a multiprocessor system where there is access to shared memory

**Distributed Processing**
- processes multiplex their executions on several processors which do not share memory (distributed systems)
Why do we need concurrency

- To fully utilise the processor
Parallelism between CPU and I/O devices

CPU
- Initiate I/O Operation
- Interrupt I/O Routine
- I/O Finished

I/O Device
- Process I/O Request
- Signal Completion
- Continue with Outstanding Requests
Run-Time Support System (RTSS)

- An RTSS has many of the properties of the scheduler in an operating system, and sits logically between the hardware and the application software.

- In reality it may take one of a number of forms:
  - A software structure programmed as part of the application. This is the approach adopted in Modula-2.
  - A standard software system linked to the program object code by the compiler. This is normally the structure with Ada programs.
  - A hardware structure microcoded into the processor for efficiency. An occam2 program running on the transputer has such a run-time system. The aJile Java processor is another example.
Processes and threads

- All operating systems provide processes
- Processes execute in their own virtual machine (VM) to avoid interference from other processes
- Modern OSs provide mechanisms for creating threads within the same virtual machine; threads are sometimes provided transparently to the OS
- Threads have unrestricted access to their VM
- The programmer and the language must provide the protection from interference
- Long debate over whether language should define concurrency or leave it up to the O.S.
  - Ada, Java and C# provide concurrency
  - C, C++ do not
Process states

- Non-existing
- Created
- Initializing
- Executable
- Terminated

1. Process states:
   - Non-existing
   - Created
   - Initializing
   - Executable
   - Terminated
Concurrent Programming Constructs

Allow

- The expression of concurrent execution through the notion of process (or/and threads)
- Process synchronization
- Inter-process communication

Processes may be

- independent
- cooperating
- competing
Concurrent Execution

Processes differ in
- **Structure** — static, dynamic
- **Level** — nested, flat

<table>
<thead>
<tr>
<th>Language</th>
<th>Structure</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrent Pascal</td>
<td>static</td>
<td>flat</td>
</tr>
<tr>
<td>occam2</td>
<td>static</td>
<td>nested</td>
</tr>
<tr>
<td>Modula</td>
<td>dynamic</td>
<td>flat</td>
</tr>
<tr>
<td>Ada</td>
<td>dynamic</td>
<td>nested</td>
</tr>
<tr>
<td>C/POSIX</td>
<td>dynamic</td>
<td>flat</td>
</tr>
<tr>
<td>Java</td>
<td>dynamic</td>
<td>nested</td>
</tr>
</tbody>
</table>
Concurrent Execution

- **Granularity** — coarse (Ada, POSIX processes/threads, Java), fine (occam2)
- **Initialization** — parameter passing, IPC
- **Termination**
  - completion of execution of the process body;
  - suicide, by execution of a self-terminate statement;
  - abortion, through the explicit action of another process;
  - occurrence of an untrapped error condition;
  - never: processes are assumed to be non-terminating loops;
  - when no longer needed.
Nested Processes

- Hierarchies of processes can be created and inter-process relationships formed.
- For any process, a distinction can be made between the process (or block) that created it and the process (or block) which is affected by its termination.
- The former relationship is known as **parent/child** and has the attribute that the parent may be delayed while the child is being created and initialized.
- The latter relationship is termed **guardian/dependent**. A process may be dependent on the guardian process itself or on an inner block of the guardian.
- The guardian is not allowed to exit from a block until all dependent processes of that block have terminated.
Nested Processes

- A guardian cannot terminate until all its dependents have terminated
- A program cannot terminate until all its processes have terminated
- A parent of a process may also be its guardian (e.g. with languages that allow only static process structures)
- With dynamic nested process structures, the parent and the guardian may or may not be identical
Process States

- Non-existing
- Created
- Initializing
- Terminated
- Waiting Child Initialization
- Executable
- Waiting Dependent Termination

Concurrent Object-Oriented Programming
Processes and Objects

- **Active** objects — undertake spontaneous actions
- **Reactive** objects — only perform actions when invoked
- **Resources** — reactive but can control order of actions
- **Passive** — reactive, but no control over order
- **Protected** resource — passive resource controller
- **Server** — active resource controller
Process Representation

- Coroutines
- Fork and Join
- Cobegin
- Explicit Process Declaration
Coroutine Flow Control

Coroutine A

1 -> Coroutine B
6 -> resume B

7 -> resume B
11 -> resume C
12 -> resume C

Coroutine B

2 -> Coroutine C
3 -> resume C
8 -> resume B
9 -> resume A
10 -> Coroutine A
13 -> Coroutine A
14 -> Coroutine A
15 -> Coroutine B

Coroutine C

4 -> Coroutine B
5 -> Coroutine A

Resume A

Coroutines A, B, C

Resume B

Resume C
Note

- Modula-2 supports coroutines
- No return statement — only a resume statement
- The value of the data local to the coroutine persist between successive calls
- The execution of a coroutine is suspended as control leaves it, only to carry on where it left off when it resumed

Do coroutines express true parallelism?
Fork and Join

- The fork specifies that a designated routine should start executing concurrently with the invoker.
- Join allows the invoker to wait for the completion of the invoked routine.

```
function F return is ...;
procedure P;
...
C:= fork F;
...
J:= join C;
...
end P;
```

- After the fork, P and F will be executing concurrently. At the point of the join, P will wait until the F has finished (if it has not already done so).
- Fork and join notation can be found in Mesa and UNIX/POSIX.
Cobegin

- The cobegin (or parbegin or par) is a structured way of denoting the concurrent execution of a collection of statements:

  ```
  cobegin
    S1;
    S2;
    S3;
    ...
    Sn
  coend
  ```

- S1, S2 etc, execute concurrently
- The statement terminates when S1, S2 etc have terminated
- Each Si may be any statement allowed within the language
- Cobegin can be found in Edison and occam2.
Explicit Process Declaration

- The structure of a program can be made clearer if routines state whether they will be executed concurrently.

- Note that this does not say when they will execute.

```
  task body  Process  is
  begin
    . . .
  end;
```

- Languages that support explicit process declaration may have explicit or implicit process/task creation.
Tasks and Ada

- The unit of concurrency in Ada is called a **task**

- Tasks must be explicitly declared, there is no fork/join statement, COBEGIN/PAR etc.

- Tasks may be declared at any program level; they are created implicitly upon entry to the scope of their declaration or via the action of an allocator.

- Tasks may communicate and synchronise via a variety of mechanisms: rendezvous (a form of synchronised message passing), protected units (a form of monitor/conditional critical region), and shared variables.
Robot Arm example

type Dimension is (Xplane, Yplane, Zplane);
task type Control(Dim : Dimension);
C1 : Control(Xplane);
C2 : Control(Yplane);
C3 : Control(Zplane);

task body Control is
    Position : Integer; -- absolute position
    Setting : Integer; -- relative movement
begin
    Position := 0; -- rest position
    loop
        New_Setting (Dim, Setting);
        Position := Position + Setting;
        Move_Arm (Dim, Position);
    end loop;
end Control;
Concurrent Object-Oriented Programming

- Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads (processes) are created.

- However to avoid all threads having to be child classes of `Thread`, it also uses a standard interface:

  ```java
  public interface Runnable {
      public abstract void run();
  }
  ```

- Hence, any class which wishes to express concurrent execution must implement this interface and provide the `run` method.
public class UserInterface
{
    public int newSetting (int Dim) { ... }  
    ...  
}

public class Arm
{
    public void move(int dim, int pos) { ... }  
}

UserInterface UI = new UserInterface();

Arm Robot = new Arm();
public class Control extends Thread {
    private int dim;

    public Control(int Dimension) // constructor {
        super();
        dim = Dimension;
    }

    public void run() {
        int position = 0;
        int setting;

        while (true) {
            Robot.move(dim, position);
            setting = UI.newSetting(dim);
            position = position + setting;
        }
    }
}
final int xPlane = 0;  // final indicates a constant
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane);
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);

C1.start();
C2.start();
C3.start();
public class Control implements Runnable
{
    private int dim;

    public Control(int Dimension) // constructor
    {
        dim = Dimension;
    }

    public void run()
    {
        int position = 0;
        int setting;

        while(true)
        {
            Robot.move(dim, position);
            setting = UI.newSetting(dim);
            position = position + setting;
        }
    }
}
Alternative Robot Control

```java
final int xPlane = 0;
final int yPlane = 1;
final int zPlane = 2;

Control C1 = new Control(xPlane); // no thread created yet
Control C2 = new Control(yPlane);
Control C3 = new Control(zPlane);

// constructors passed a Runnable interface and threads created
Thread X = new Thread(C1);
Thread Y = new Thread(C2);
Thread Z = new Thread(C2);

X.start(); // thread started
Y.start();
Z.start();
```
Java Thread States

non-existing

Create thread object

new

start

executable

run method exits
stop, destroy

blocked

dead
Points about Java Threads

- Java allows dynamic thread creation
- Java (by means of constructor methods) allows arbitrary data to be passed as parameters
- Java allows thread hierarchies and thread groups to be created but there is no master or guardian concept; Java relies on garbage collection to clean up objects which can no longer be accessed
- The main program in Java terminates when all its user threads have terminated (see later)
- One thread can wait for another thread (the target) to terminate by issuing the `join` method call on the target's thread object.
- The `isAlive` method allows a thread to determine if the target thread has terminated
A Thread Terminates

- when it completes execution of its `run` method either normally or as the result of an unhandled exception;
- via its `stop` method — the `run` method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
  - the thread object is now eligible for garbage collection.
  - if a `Throwable` object is passed as a parameter to `stop`, then this exception is thrown in the target thread; this allows the `run` method to exit more gracefully and cleanup after itself
- `stop` is inherently unsafe as it releases locks on objects and can leave those objects in inconsistent states; the method is now deemed obsolete (depreciated) and should not be used
- by its destroy method being called — destroy terminates the thread without any cleanup (never been implemented in the JVM)
A Simple Embedded System

- Overall objective is to keep the temperature and pressure of some chemical process within well-defined limits.
package Data_Types is
    type Temp_Reading is new Integer range 10..500;
    type Pressure_Reading is new Integer range 0..750;
    type Heater_Setting is (On, Off);
    type Pressure_Setting is new Integer range 0..9;
end Data_Types;
with Data_Types; use Data_Types;
package IO is
    procedure Read(TR : out Temp_Reading); -- from ADC
    procedure Read(PR : out Pressure_Reading);
    procedure Write(HS : Heater_Setting); -- to switch
    procedure Write(PS : Pressure_Setting); -- to DAC
    procedure Write(TR : Temp_Reading); -- to screen
    procedure Write(PR : Pressure_Reading); -- to screen
end IO;
with Data_Types; use Data_Types;
package Control_Procedures is
  -- procedures for converting a reading into
  -- an appropriate setting for output.
procedure Temp_Convert (TR : Temp_Reading;
                         HS : out Heater_Setting);
procedure Pressure_Convert (PR : Pressure_Reading;
                            PS : out Pressure_Setting);
end Control_Procedures;
with Data_Types; use Data_Types; with IO; use IO;
with Control_Procedures; use Control_Procedures;
procedure Controller is

  task Temp_Controller;
  task body Temp_Controller is
    TR : Temp_Reading;
    HS : Heater_Setting;
    begin
      loop
        Read(TR);
        Temp_Convert(TR,HS);
        Write(HS); Write(TR);
      end loop;
    end Temp_Controller;

begin
  null;
end Controller;

begin
  task Pressure_Controller;
  task body Pressure_Controller is
    PR : Pressure_Reading;
    PS : Pressure_Setting;
    begin
      loop
        Read(PR);
        Pressure_Convert(PR,PS);
        Write(PS); Write(PR);
      end loop;
    end Pressure_Controller;
end Controller;
Definition: Concurrent Programming (Ben-Ari, 1982)

- is the name given to programming notation and techniques for expressing potential parallelism and solving the resulting synchronization and communication problems
Shared variable-based synchronisation and communication
Synchronization and Communication

- The **correct behaviour of a concurrent program** depends on **synchronisation** and **communication** between its processes.

  - **Synchronisation**: the satisfaction of constraints on the interleaving of the actions of processes (e.g. an action by one process only occurring after an action by another)

    also in the narrower sense: bringing two processes simultaneously into predefined states

  - **Communication**: the passing of information from one process to another
Synchronization and Communication

- Concepts are linked since communication requires synchronisation, and synchronisation can be considered as contentless communication.

- Data communication is usually based upon either shared variables or message passing.
Examples: busy waiting, semaphores and monitors

Unrestricted use of shared variables is unreliable and unsafe due to multiple update problems

Consider two processes updating a shared variable, \( x \), with the assignment: \( x := x + 1 \)
  - load the value of \( x \) into some register
  - increment the value in the register by 1 and
  - store the value in the register back to \( x \)

As the three operations are not indivisible, two processes simultaneously updating the variable could follow an interleaving that would produce an incorrect result
task body Helicopter is
  Next: Coordinates;
begin
  loop
    Compute_New_Cordinates(Next);
    Shared_Cordinates := Next;
  end loop
end;

task body Police_Car is
begin
  loop
    Plot(Shared_Cordinates);
  end loop;
end;

type Coordinates is
  record
    X : Integer;
    Y : Integer;
  end record;
end

Shared_Cordinate: Coordinates;

Shared_Cordinates := Next;
Villain's Escape Route

Villain's Escape Route
(seen by helicopter)

Villain Escapes!

Police Car's Pursuit Route

Chair of Software Engineering

Concurrent Object-Oriented Programming
Avoiding Interference

- The parts of a process that access shared variables must be executed *indivisibly* (*atomically*) with respect to each other.

- These parts are called **critical sections**.

- The required protection is called **mutual exclusion**.
Mutual Exclusion

- A sequence of statements that must appear to be executed indivisibly is called a critical section.

- The synchronisation required to protect a critical section is known as **mutual exclusion**.

- **Atomicity** is assumed to be present at the memory level. If one process is executing \( x := 5 \), simultaneously with another executing \( x := 6 \), the result will be either 5 or 6 (not some other value).

- If two processes are updating a structured object, this atomicity will only apply at the **single word element level**.
Condition synchronisation is needed when a process wishes to perform an operation that can only sensibly, or safely, be performed if another process has itself taken some action or is in some defined state.

E.g. a bounded buffer has 2 condition synchronisation:
- the producer processes must not attempt to deposit data onto the buffer if the buffer is full.
- the consumer processes cannot be allowed to extract objects from the buffer if the buffer is empty.
Busy Waiting

- One way to implement synchronisation is to have processes set and check shared variables that are acting as flags.
- This approach works well for condition synchronisation but no simple method for mutual exclusion exists.
- Busy wait algorithms are in general inefficient; they involve processes using up processing cycles when they cannot perform useful work.
- Even on a multiprocessor system they can give rise to excessive traffic on the memory bus or network (if distributed).
Busy waiting (spinning)

Busy waiting and condition synchronizing

```plaintext
process P1; (* waiting process *)
    while flag = down do
        null
    end
end P1;

process P2; (* signalling process *)
    flag = up;
end P2;
```
Busy waiting and mutual exclusion

```plaintext
process P;
    loop
        entry protocol
            critical section
        exit protocol
        non critical section
    end
end P;
```
Busy waiting and mutual exclusion (not correct)

**process** P1;
  loop
    flag1 := up (* announce intent to enter *)
    while flag2 = up do
      null (* busy wait if the other process is in *)
    end; (* its critical section *)
  <critical section>
  flag1 := down (* exit protocol *)
  <non critical section>
  end
end P1;

**process** P2;
  loop
    flag2 := up
    while flag1 = up do
      null
    end;
  <critical section>
  flag2 := down
  <non critical section>
  end
end P1;
Interleaving of P1 and P2

- P1 sets its flag (flag1 = up)
- P2 sets its flag (flag2 = up)
- P2 checks flag1 (it is up therefore P2 loops)
- \textbf{P2 enters its busy wait}
- P1 checks flag2 (it is up therefore P1 loops)
- \textbf{P1 enters its busy wait}

Result
- Both processes P1 and P2 will remain in their busy waits
- Neither can get out because the other cannot get out
  $\rightarrow \textbf{Livelock}$ (see later)
Busy waiting and mutual exclusion (not correct)

```plaintext
process P1;
  loop
    while flag2 = up do
      null (* busy wait if the other process is in *)
    end; (* its critical section *)
    flag1 := up (* announce intent to enter *)
    <critical section>
    flag1 := down (* exit protocol *)
    <non critical section>
  end
end P1;

process P2;
  loop
    while flag1 = up do
      null
    end;
    flag2 := up
    <critical section>
    flag2 := down
    <non critical section>
  end
end P1;
```
Interleaving of P1 and P2

- P1 and P2 are in their non-critical section
  flag 1 = flag 2 = down
- P1 checks flag2 (flag2 = down)
- P2 checks flag1 (flag1 = down)
- P2 sets its flag (flag2 = up)
  P2 enters critical section
- P1 sets its flag (flag1 = up)
  P1 enters critical section

- P1 and P2 are both in their critical section!
Semaphores — originally designed by Dijkstra (1968) — are a simple mechanism for programming

- mutual exclusion
- condition synchronisation

Semaphores have two benefits
- they simplify the protocols for synchronisation
- they remove the needs for busy-wait loops
Semaphores

- A semaphore is a non-negative integer variable that apart from initialization can only be acted upon by two procedures P (or wait) and V (or signal).
- \textit{wait} (S)
  If the value of \( S > 0 \) then decrement its value by one; otherwise delay the process until \( S > 0 \) (and then decrement its value).
- \textit{signal} (S)
  Increment the value of \( S \) by one.
- \textit{wait} and \textit{signal} are \textbf{atomic} (indivisible). Two processes both executing \textit{wait} operations on the same semaphore \textbf{cannot interfere} with each other and cannot fail during the execution of a semaphore operation.
var consyn : semaphore (* init 0 *)

process P1;
  (* waiting process *)
  statement X;
  wait (consyn)
  statement Y;
end P1;

process P2;
  (* signalling proc *)
  statement A;
  signal (consyn)
  statement B;
end P2;

In what order will the statements execute?
Mutual exclusion (mutex)

(* mutual exclusion *)
var mutex : semaphore; (* initially 1 *)

process P1;
  statement X
  wait (mutex);
  statement Y
  signal (mutex);
  statement Z
end P1;

process P2;
  statement A;
  wait (mutex);
  statement B;
  signal (mutex);
  statement C;
end P2;

In what order will the statements execute?