

## **Ada Concurrency: a Bank Simulation**

**Michael B. Feldman**

**Department of Computer Science  
School of Engineering and Applied Science  
The George Washington University, Washington, DC 20052**

**(202) 994-5919  
mfeldman@gwu.edu**

### **Outline of Talk**

- **motivation, definitions and background**
- processes: the units of concurrency
- programming example: bank simulation

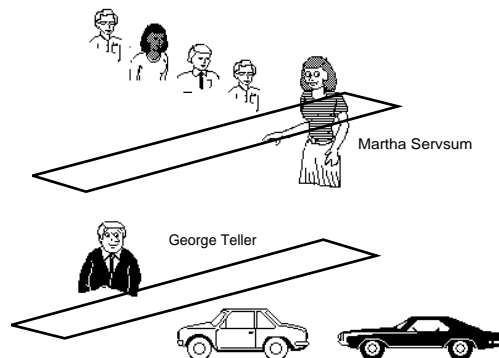
2

### Basic Definitions: Concurrent, Parallel, and Distributed Programs

- A *concurrent program* defines actions that may be performed simultaneously. Note: “*may be performed simultaneously*” does not necessarily mean “*are performed simultaneously*.”
- A *pseudoconcurrent program* is a concurrent program whose actions are performed on a single processor in interleaved fashion.
- A *parallel program* is a concurrent program that is designed for execution on parallel hardware.
- A *distributed program* is a parallel program designed for execution on a network of autonomous processors that do not share main memory.

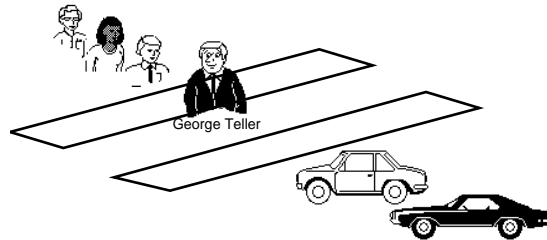
3

### Motivation: First Charlatan Savings and Loan



4

### First Charlatan Savings and Loan (after staff cuts)



5

### Concurrent Programming Today

Concurrent programming (explicit concurrency) is now recognized as a potentially powerful tool for modeling and simulation, not just a means for building the operating system kernel or controlling the computer.

The world is concurrent!

6

### Implicit vs. Explicit Concurrency

#### *Implicit concurrency*

- The program contains independent processing steps (at the block, statement, or expression level) that may be executed in parallel; or
- The program triggers device operations that may proceed in parallel with the execution of the program.

#### *Explicit concurrency*

- The program contains explicit constructs to control aspects of the concurrency; the concurrency is specified by the program designer.

7

### Capsule History of Concurrent Programming

- *1950s* — implicit concurrency introduced with overlapping of input, output, and processing (to get more productivity out of expensive CPUs)
- *1960s* — formal work on explicit concurrency, especially to develop more reliable multi-user operating systems (e.g. the Eindhoven OS of E.W. Dijkstra, 1968)
- *1970s* — experimental work on language constructs for explicit concurrency (for example Communicating Sequential Processes of C.A.R. Hoare, 1978)
- *1980s* — concurrent programming matures, with industrial-strength systems like VMS and UNIX, and “real” languages like Ada

8

### The Programmer's Interface to Explicit Concurrency

- operating system level subprogram libraries: UNIX, VMS, MVS, VM, OS/2, etc.
- language-oriented subprogram library, independent of operating system: Modula-2, Modula-3, etc.
- programming language constructs, independent of operating system: Communicating Sequential Processes (CSP), Ada, occam, Concurrent C, Co-Pascal, etc.

9

### Outline of Talk

- motivation, definitions and background
- **processes: the units of concurrency**
- programming example: bank simulation

10

## Processes: the Basic Unit of Concurrency

- A sequential program specifies sequential execution of a list of statements; its execution is called a *process*.
- A concurrent program specifies two or more sequential programs that may be executed (pseudo-) concurrently.
- In many languages, process is also the name of the construct used to describe process behavior; one notable exception is Ada, which uses the name *task* for this purpose

11

## Process States

*Elaborated* — the process's declaration has been carried out; memory for it has been allocated.

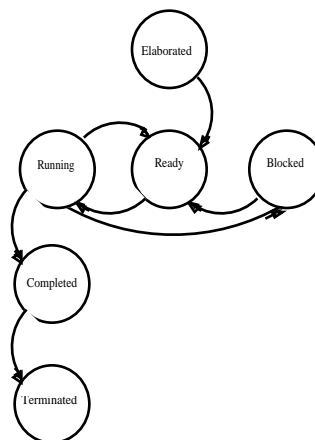
*Running* — the process has a processor and is actively executing instructions.

*Ready* — the process would be running, but isn't assigned to a processor. A process never volunteers for this state.

*Blocked* — the process is "sleeping" or waiting for an external event such as interprocess communication. This is always a voluntary state.

*Completed* — the process has finished its useful work but cannot terminate yet because it has "live" dependent tasks.

*Terminated* — the process is gone from the system.



12

### **Nondeterminism**

A sequential program imposes a total ordering on the actions it specifies. A concurrent program imposes a partial ordering, which means that there is uncertainty over the precise order of occurrence of some events; this property is referred to as *nondeterminism*. A consequence of nondeterminism is that when a concurrent program is executed repeatedly it may take different execution paths even when operating on the same input data.

13

### **Operations on Processes: the Process as Abstract Data Type**

- *creation* — declares a code segment that will later be associated with one or more threads of control
- *activation* — associates a code segment with a new thread of control; can be implicit or require an explicit operation
- *termination* — must parent must wait child's termination?
- *scheduling* — method by which processes are assigned to processors and move from *ready* to *running*
- *synchronization* — means by which shared data structures are protected or processes "touch base" to communicate
- *communication* — shared variables vs. synchronous or asynchronous message passing
- *nondeterministic constructs* — expressing or controlling nondeterminism via guarded commands

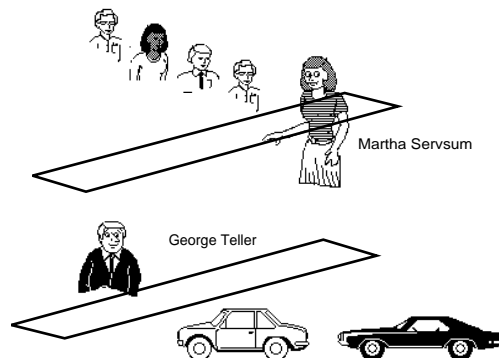
14

### Outline of Talk

- motivation, definitions and background
- processes: the units of concurrency
- **programming example: bank simulation**

15

### Motivation: First Charlatan Savings and Loan



16

### First Charlatan in Ada

- customer arrives at bank, makes a call like

```
-- Customer task
Martha.MakeDeposit
  (ID => 1273, Amount => 100.0, MyNewBalance);
```

- teller serves one customer per loop iteration; Ada's queuing system forces other customers to queue up

```
LOOP
  ACCEPT MakeDeposit
    (ID: Integer; Amt: Float; Balance: out Float) DO
    BalanceFile(ID) := BalanceFile(ID) + Amt;
    Balance := BalanceFile(ID);
  END MakeDeposit;
END LOOP;
```

17

### Defining the Teller Model

- specify task type with callable entries (services)

```
TASK TYPE TellerTemplate IS
  ENTRY MakeDeposit
    (ID: Integer; Amt: Float; Balance: OUT Float);
END TellerTemplate;
```

- declare some task objects (variables)

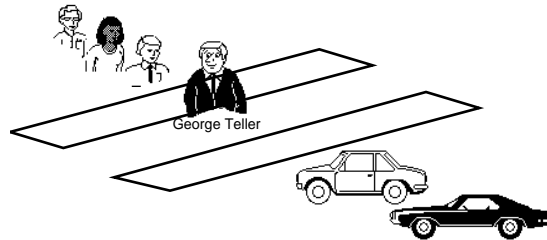
```
George, Martha: TellerTemplate;
```

- task body (implementation)

```
TASK BODY TellerTemplate IS
BEGIN
  LOOP
    -- as on previous page
  END LOOP;
END TellerTemplate;
```

18

**First Charlatan Savings and Loan  
(after staff cuts)**



19

**First Charlatan in Ada  
(after staff cuts)  
(new teller specification)**

```
TASK TYPE TellerTemplate IS
  ENTRY MakeWalkupDeposit
    (ID: Integer; Amt: Float; Balance: OUT Float);
  ENTRY MakeDriveupDepcsit
    (ID: Integer; Amt: Float; Balance: OUT Float);
END TellerTemplate;
```

20

### George's New Body

- Let's let George alternate between the queues

```
LOOP
  ACCEPT MakeWalkupDeposit
  (ID: Integer; Amt: Float; Balance: OUT Float) DO
    -- update balance as before
  END MakeWalkupDeposit;
  ACCEPT MakeDriveupDeposit
  (ID: Integer; Amt: Float; Balance: OUT Float) DO
    -- update balance as before
  END MakeDriveupDeposit;
END LOOP;
```

- Problem: George can't serve two cars in succession, even if there aren't any walkup customers (he'll wait at the first accept statement).

21

### Nondeterministic Selection

```
LOOP
  SELECT
    ACCEPT MakeWalkupDeposit
    (ID: Integer; Amt: Float; Balance: OUT Float) DO
      -- update balance as before
    END MakeWalkupDeposit;
  OR
    ACCEPT MakeDriveupDeposit
    (ID: Integer; Amt: Float; Balance: OUT Float) DO
      -- update balance as before
    END MakeDriveupDeposit;
  END SELECT;
END LOOP;
```

- Now George will serve whichever queue has activity
- If both queues have waiting callers, the Ada standard says "select the first caller from one of the queues," but doesn't specify how to do it.

22

### The Bank's Not Open 24 Hours a Day

```
LOOP
  SELECT
    WHEN CounterHours =>
      ACCEPT MakeWalkupDeposit
        (ID: Integer; Amt: Float; Balance: OUT Float) DO
        -- update balance as before
      END MakeWalkupDeposit;
    OR
    WHEN DriveupHours =>
      ACCEPT MakeDriveupDeposit
        (ID: Integer; Amt: Float; Balance: OUT Float) DO
        -- update balance as before
      END MakeDriveupDeposit;
  END SELECT;
END LOOP;
```

- CounterHours and DriveUpHours are conditions called "guards;" this is an example of a *guarded command*.

23

### Keeping the Teller Busy

```
LOOP
  SELECT
    WHEN CounterHours =>
      ACCEPT MakeWalkupDeposit
        (ID: Integer; Amt: Float; Balance: OUT Float) DO
        -- update balance as before
      END MakeWalkupDeposit;
    OR
    WHEN DriveupHours =>
      ACCEPT MakeDriveupDeposit
        (ID: Integer; Amt: Float; Balance: OUT Float) DO
        -- update balance as before
      END MakeDriveupDeposit;
    ELSE
      DoSomePaperWork; -- only if nothing else to do
  END SELECT;
END LOOP;
```

24

### The Timed Wait - a Timeout Scheme

```
LOOP
  SELECT
    DELAY 5.0 * Minutes; -- sets 5-minute timer
  OR
    WHEN CounterHours =>
      ACCEPT MakeWalkupDeposit
    (ID: Integer; Amt: Float; Balance: OUT Float) DO
      -- update balance as before
    END MakeWalkupDeposit;
  OR
    WHEN DriveupHours =>
      ACCEPT MakeDriveupDeposit
    (ID: Integer; Amt: Float; Balance: OUT Float) DO
      -- update balance as before
    END MakeDriveupDeposit;
  END SELECT;
END LOOP;
```

- SELECT is satisfied if a customer arrives on either queue or 5 minutes elapse, whichever comes first

25

### The Customer Also Has Options Besides Waiting Forever

- conditional call— customer gives up if George cannot serve him immediately

```
SELECT
  George.MakeDeposit
  (ID => 1273, Amount => 100.0, MyNewBalance);
ELSE
  --Holler about low service quality at the S&L
END SELECT;
```

- timed call— customer gives up if George cannot serve him within 5 minutes

```
SELECT
  George.MakeDeposit
  (ID => 1273, Amount => 100.0, MyNewBalance);
OR
  DELAY 5.0 * minutes;
END SELECT;
```

26

### Final Comments

- Concurrent programming is an essential part of modern real-time and interactive systems.
- For many applications, concurrency gives a more natural model of the world.
- Ease of understanding, and a high degree of machine independence, can be achieved by putting concurrency constructs in the programming language.
- Concurrent programming is also *fun!*

27

### Selected Reading List

- Ben-Ari, M. *Principles of Concurrent and Distributed Programming*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- Bustard, D., *Concepts of Concurrent Programming*. Module CM-24. Pittsburgh, Penna.: Software Engineering Institute, 1990.
- Dijkstra, E. W. "The Structure of the 'THE' Multiprogramming System." *Comm. ACM* 11, 5 (May 1968), 341-346.
- Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes." *Acta Informatica* (1971), 115-138.
- Feldman, M.B. *Ada Problem Solving and Program Design*. Reading, Mass.: Addison-Wesley, 1991.
- Feldman, M.B. *Language and System Support for Concurrent Programming*. Module CM-25. Pittsburgh, Penna.: Software Engineering Institute, 1990.
- Gehani, N. *Ada: Concurrent Programming (2nd ed.)*. Summit, NJ.: Silicon Press, 1991.
- Gehani, N., and A. D. McGettick, eds. *Concurrent Programming*. Reading, Mass.: Addison Wesley, 1988.
- U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A, 1983.

28