On asynchronous communication

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2020/2021

Defects of the base model

- Dijkstra’s semaphores are a “poor” abstraction
  - They leave it to the user to decide where the critical sections are on a point basis
    - Without assuming or requiring language support …
  - They only address exclusion synchronization
  - Their implementation without busy wait requires nontrivial runtime support
- The monitor is a useful step forward
    - Per Brinch Hansen, “Structured Multiprogramming”, CACM 15(7):574-578 (1972)
  - The clever intuition is to **unite** exclusion synchronization and avoidance synchronization in a single abstraction
    - Unfortunately, it leaves the programming of condition (event) variables to the user, yielding an evident vulnerability

Addressing exclusion synchronization – 1

- Requirements
  1. Write access is exclusive to any other operation
  2. Read access does not conflict with other reads
- It is opportune to distinguish between R/O and R/W access

```
protected type Shared_Integer (Initial_Value : Integer) is
 function Read return Integer;
 procedure Write (Value : Integer);
private
 The_Integer : Integer := Initial_Value;
end Shared_Integer;
```

```
protected body Shared_Integer is
 function Read return Integer is
 begin
  return The_Integer;
 end Read;
 procedure Write (Value : Integer) is
 begin
  The_Integer := Value;
 end Write;
end Shared_Integer;
```

Parallel reads

Exclusive writes

Addressing exclusion synchronization – 2

- **Servers** are heavy-weight abstractions
  - Appropriate when the collaboration logic is algorithmically complex and its cost pays off
  - Wasteful otherwise
- **Protected resources** are lighter-weight and have a much simpler termination semantics
  - They simply go out of scope …
  - But they are unable to express complex synchronization logic

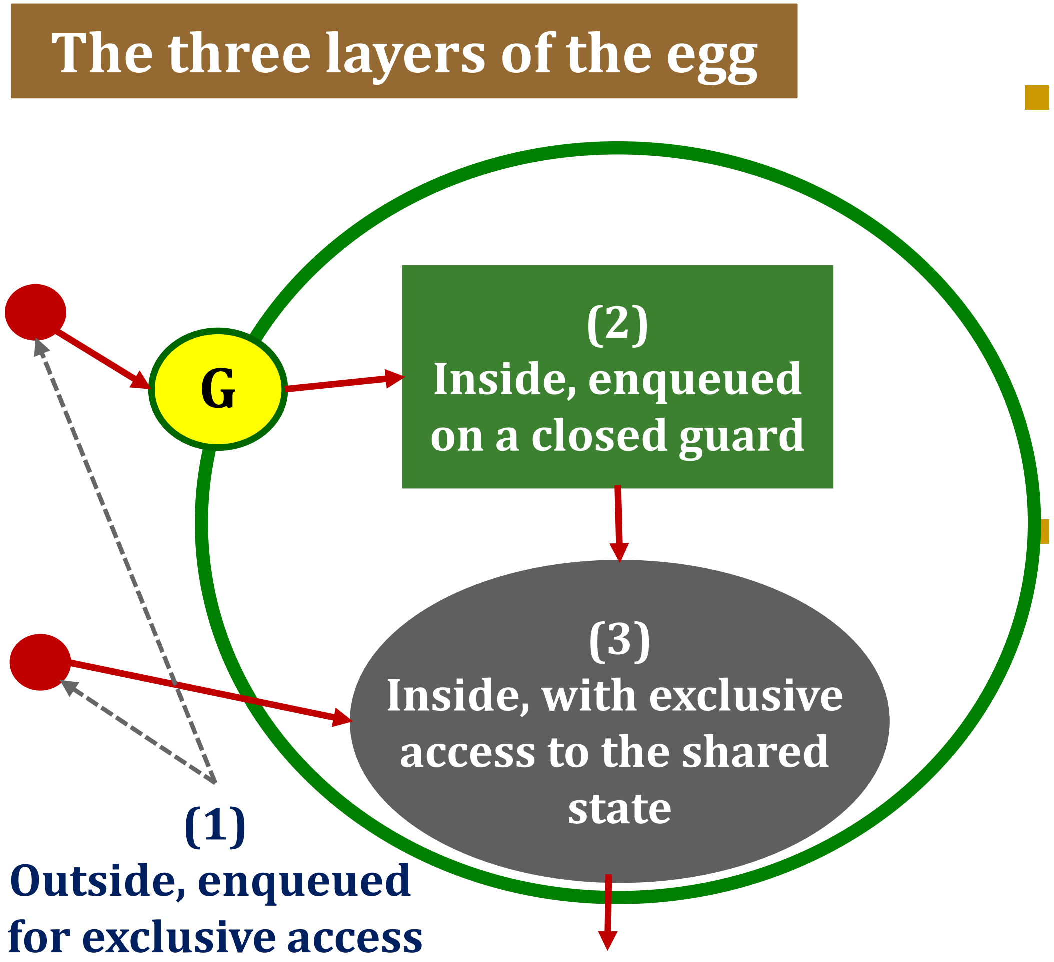
Addressing avoidance synchronization

- Requirements
  1. The caller shall be able to synchronize with an event determined by a state transition in the shared resource
     - Suspending until that event occurs
  2. The runtime shall take care of making suspension, event notification, and resumption happen
     - No direct involvement by the caller
- The resumption semantics is the most delicate piece of the puzzle
  - We have seen Java's blunder in addressing it …
  - The solution is known as the **eggshell model**

The eggshell model – 1

- Protected resource services that associate with state events are syntactically denoted
  - An **`entry`** prefixed by a Boolean guard
    - This signifies it is other than a procedure or a function
    - The Boolean guard represents the condition associated with the expected logical state of the resource
      - Orthogonal to be “free” for exclusive
- On a closed guard, the caller’s request is enqueued **within** the resource
  - Not outside of it
  - This saves resumption from the risk of starvation

The eggshell model – 2



- Guard evaluation requires exclusion synchronization
  - Relinquishing the lock on a closed guard enqueues the call *inside* the locked region
  - On the corresponding event queue
- State events can only occur on execution with exclusive access to the state
  - Guards shall be re-evaluated every time a write-access operation completes
  - Hence potentially also during guard evaluation

Example: bounded buffer – 1

```
Buffer_Size : constant Positive := 5;
type Index is mod Buffer_Size; -- tipo modulare
subtype Count is Natural range 0 .. Buffer_Size;
type Buffer_T is array (Index) of Any_Type;

protected type Bounded_Buffer is
   entry Get (Item :    out Any_Type);
   entry Put (Item : in     Any_Type);
private
   First : Index := Index'First; -- 0
   Last  : Index := Index'Last;  -- 4
   In_Buffer : Count := 0;
   Buffer : Buffer_T;
end Bounded_Buffer;
```

Public interface

Private part

Spec

Example: bounded buffer – 2

```
protected body Bounded_Buffer is
   entry Get (Item :    out Any_Type)
      when In_Buffer > 0 is
   begin -- first read then move pointer
      Item := Buffer(First);
      First := First + 1; -- free from overflow
      In_Buffer := In_Buffer - 1;
   end Get;
   entry Put (Item : in     Any_Type)
      when In_Buffer < Buffer_Size is
   begin --  first move pointer then write
      Last := Last + 1; -- free from overflow
      Buffer(Last) := Item;
      In_Buffer := In_Buffer + 1;
   end Put;
end Bounded_Buffer;
```

Guards

The eggshell model – 3

- A protected resource is under a read lock when one or more calls are executing a **`function`** on it
- A protected resource is under a write lock when a call is executing a **`procedure`** or an **`entry`** (including guard evaluation) on it
- All conflicting calls are enqueued <u>outside</u> of the protected resource
  - Exclusion queue

The eggshell model – 4

- Calls owning a write lock on a protected resource can call other services of the same PR *without* having to exit the eggshell
  - Such additional calls are serviced immediately
- Calls with targets *outside* of the protected resource that return to the PR compete for the lock on access to it
  - Very bad idea, exposed to the risk of stalling
  - A protected resource service that needs to call outside of itself shows bad encapsulation

Eggshell model evaluation rules – 1

1. When the PR is under read lock, a function call to it gets immediately executed; **goto** 8
2. When the PR under read lock, procedure or entry calls to it are held until the lock is relinquished
3. When the PR is under write lock, all calls to it are held until the lock is relinquished
4. When the PR is free, a function call sets it in read lock and executes it; **goto** 8
5. When the PR is free, procedure or entry calls set it to write lock, then
    a. If the call is to a procedure, it gets executed; **goto** 6
    b. If the call is to an entry, its guard is evaluated, then
        i. If the guard is open, the entry gets executed; **goto** 6
        ii. If the guard is closed, the call is enqueued (this is when any `select` clause on the call side gets evaluated); **goto** 6

Eggshell model evaluation rules – 2

6. Any guard with a nonempty queue that may have changed since last evaluation, gets re-evaluated
   a. If any guard were open, one is selected and its entry is executed and the corresponding call is dequeued; **goto** 6
   b. Else **goto** 7
7. If no guard has a nonempty queue, **goto** 8
8. From all calls enqueued outside of the PR, select either one that requires write lock or all that require read lock; **goto** 4 or 5
   a. If no calls are enqueued outside of PR, the protocol ends

- Steps 6-7 are the heart of the eggshell model: they serve the event queues inside *in precedence* to outside calls

Further enhancement

- The number of calls enqueued on an entry queue (in servers and in protected resources) can be queried
  - By **`'Count`**, predefined function attribute of entry
  - This is a case of “read-only reflection”, whereby the program inquires information on a runtime abstraction
- This feature requires call enqueuing to need a write lock on the protected resource
  - Or the channel in servers
- Using **`'Count`** in a guard expression causes its re-evaluation every time a write-lock call gets executed
  - Runtime overhead versus interesting semantics

Example: group barriers – 1

A group-barrier check that lets $N \geq 1$ calls at once

```
protected Guardian is
 entry Let_In;
private
 Open : Boolean := False;
end Guardian;
protected body Guardian is
 entry Let_In
  when Let_In'Count = N or Open is
 begin
  if Let_In'Count = 0 then
   Open := False;
  else
   Open := True;
  end if;
 end Let_In;
end Guardian;
```

- The N^{th} caller will find the guard closed, but its enqueuing will change the value of **'Count** on that entry
- This will cause the re-evaluation of the guard, which now has become open
- The 1^{st} call in the queue will be resumed and will change the guard value so that it stays open until the N^{th} gets resumed, which will close the guard

Example: group barriers – 2

- Group barriers do **not** encapsulate shared state, but shield access to it
  - They can offer powerful semantics (as in the example) but leave it to the user to place the calls in the “right” place
- **Homework**: modify the logic of the group barrier so that <u>no more than N</u> callers ever be simultaneously within the protected space
  - Currently, there is no such control

Precautions of use

- Execution within protected resources should be rapid
  - More than with servers, which are used to realize more complex service logic
- Execution with exclusion-access rights should **not** make potentially blocking calls
  - Such calls are those that may cause the caller to relinquish the CPU <u>synchronously</u>
  - If the runtime detects this it raises program error exception

Semantics of use

- Protected resources are elaborated when their declaration is encountered in a declarative region being processed
- Protected resources are finalized when their master terminates
  - Obviously, this cannot happen until there are calls enqueued into it
    - Otherwise the calling threads should be terminated anomalously

Preferential ordering – 1

- Preferential ordering is useful when servicing certain calls yields more value than servicing others
  - The logic of that policy should be server-side, **transparent** to the client
  - Otherwise the client would have heavy coupling with it
- Exclusion synchronization alone does **not** suffice
- Guards are very well fit for it
  - Interestingly, protected resources allow realizing preferential ordering also on access to suspensive entities
  - However, such suspension should **never** occur within protected operations

Preferential ordering – 2

```
protected Access_Control is
 entry Start_Read;
 procedure Stop_Read;
 entry Start_Write;
 procedure Stop_Write;
private
 Readers : Natural:= 0;
 Writers : Boolean := False;
end Access_Control;
```

```
procedure Read (I : out Item) is
begin
 Access_Control.Start_Read;
 … -- actual read (suspensive)
 Access_Control.Stop_Read;
end Read;
```

```
procedure Write(I : in Item) is
begin
 Access_Control.Start_Write;
 … -- actual write (suspensive)
 Access_Control.Stop_Write;
end Write;
```

```
protected body Access_Control is
 entry Start_Read when not Writers and
   Start_Write'Count = 0 is
 begin
  Readers := Readers + 1;
 end Start_Read;
 procedure Stop_Read is
 begin
  Readers := Readers – 1;
 end Stop_Read;
 entry Start_Write when not Writers and
   Readers = 0 is
 begin
  Writers := True;
 end Start_Write;
 procedure Stop_Write is
 begin
  Writers := False;
 end Stop_Write;
end Access_Control;
```

Preferential ordering – 3

- The guard to entry `Start_Write` warrants exclusive access rights to write operations
  - As if they were encapsulated in a protected resource
- The guard to entry `Start_Read` warrants preference to writes over reads
  - Baseline use case for guards in this regard
- **Warning**: when a critical section *not encapsulated* in a protected resource fails without returning, the program becomes erroneous and the runtime cannot help