On the multiple facets of synchronization

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

Evaluating language features – 1

- The features of a programming language may be evaluated from two angles
- **Expressive power**
  - How close they get to addressing the user needs
- **Usability**
  - How well they do on their own (*efficacy*) versus how well they interact with each other (*coherence*)

Evaluating language features – 2

- The synchronization constructs are an important ambit of such evaluation
  - Toby Bloom, "*Evaluating synchronisation mechanisms*", 7th ACM Symposium on Operating System Principles (1979) https://doi.org/10.1145/800175.806566
- The cited work singles out 6 types of conditions that may control synchronization
  - Over and above exclusion synchronization

Conditions on synchronization – 1

1. **Contingent on the synchronization state of the resource ☑**
   - Number of current users or number of enqueued calls (`'Count`) in relation to the resource multiplicity
2. **Contingent on the logical state of the resource ☑**
   - No-write-on-full, no-read-from-empty
3. **Contingent on the history of service ☑**
   - For fairness, load balance, energy efficiency, …

Conditions on synchronization – 2

4. **Contingent on the type of request ☑**
   - Preferential treatment for some requests (e.g., writes over reads)
5. **Contingent on the time of the request ☑**
   - Reflected in queuing policies for calls and callers
6. **Contingent on the request parameters ☒**
   - Where serviceability depends on whether the server can dispense as much as requested
     - E.g., as in paging or heap management

The resource allocation problem – 1

- Recurrent problem in any concurrent system
  - It involves all of Toby Bloom's 6 dimensions
    - Our current model is unable to handle it properly
- **Example**
  a) A resource manager dispenses a statically fixed number N of resources $\{R_{j=1,\ldots,N}\}$
  b) A number of concurrent clients $\{C_{i=1,\ldots,M}\}$ may request any subset of such resources
  c) Accepted requests shall be satisfied fully
    - (That is: requests *cannot* return until satisfied)
  d) Clients return resources after use

The resource allocation problem – 2

- Let us analyse the problem specification
  - Client interaction with resource manager must be *synchronous*
    - Wait-until-satisfied (Requirement c)
  - Volume of request specified as a parameter
    - This is the only plausible interface of the server
- What happens when the server finds itself unable to satisfy the request being examined
  - It cannot return to the caller prematurely
    - Hence it must keep that request on hold
  - How can it do that while continuing to serve others?
    - Serving others allows releases (Requirement d)

The resource allocation problem – 3

- Do guards help?
  - They prevent synchronization when the requested service cannot be executed
  - But guards operate **before** synchronization takes effect, hence **before** the request parameters can be examined
  - *Hence, guards as we know them, do not help*!
- Two alternatives are possible, which both need enhanced capabilities
  1. To allow guards expressions to access request parameters *without* engaging synchronization
  2. To *transfer to another queue* the request that presently cannot be satisfied
     - Beginning service but then holding it up until further notice
     - Becoming able to serve other pending requests

Alternative 1

1 resource per request is the trivial case

```
protected Controller is
 entry Allocate (R : out Resource);
 procedure Release (R : Resource);
private
 Free : Natural := Full_Capacity;
 …
end Controller;
protected body Controller is
 entry Allocate (R : out Resource)
  when Free > 0 is
 begin
  Free := Free - 1;
  …
 end Allocate;
 procedure Release (R : Resource) is
 begin
  Free := Free + 1;
 end Release;
end Controller;
```

$1 < n \leq N$ resources per request is a much harder problem

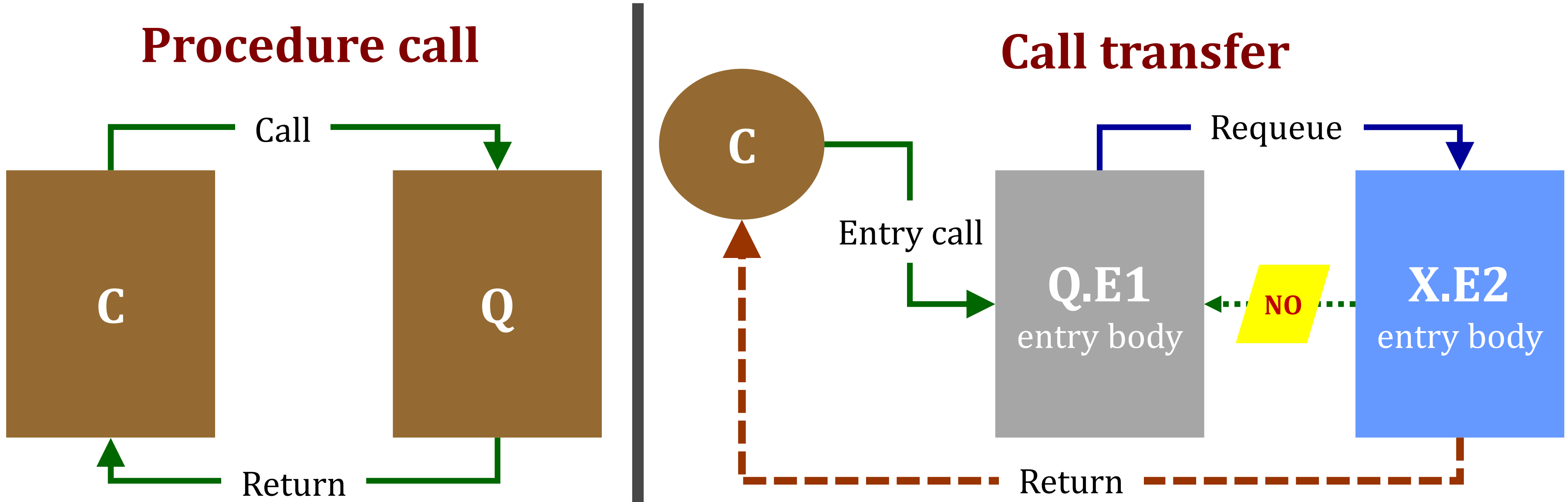
```
type Request is range 1..Max_Requests;
protected Controller is
 entry Allocate
  (R      :     out Resource;
   Amount : in      Request);
 procedure Release
  (R      : Resource;
   Amount : Request);
private
 Free : Request := Request'Last; …
end Controller;
protected body Controller is
 entry Allocate
  (R      :     out Resource;
   Amount : in      Request)
  when Amount <= Free is
 begin
  Free := Free – Amount;
 end Allocate;
 procedure Release (…) is …
end Controller;
```

Critique of alternative 1

- Requests that fail the guard are enqueued in the corresponding event queue (aka entry queue)
- Applying the eggshell model here would cause traversing <u>the entire event queue</u> every time a R/W access to the server state completes
  - Seeking any enqueued request that passes the guard
  - Untenably costly in the general case
- This solution causes each request to have its own "state-change event"
  - But the entry queue model that we know caters for a *single-condition* queue only

Alternative 2 – 1

- Transferring the call to another queue (**`requeue`**) is *not* a normal procedure call



When Q.E1 calls **`requeue`** to X.E2, the call to E1 gets finalized and left; the return to C happens only after X.E2 completes

Alternative 2 – 2

- A sophisticated feature, with challenging requirements on the runtime …
  - Transferring the call to another queue should *not* suspend the server on a closed guard
  - *Nor* it should awake the client during the transfer
  - Hence, transfer should occur atomically, *without* undergoing guard evaluation at the target queue
- This raises two "feature-interaction" issues
  1. Which entry queues can be allowable targets
  2. What happens to a time-out set on the call

Alternative 2 – 3

- **Issue 1: allowable targets**
  - Any entry with a *compatible* interface, anywhere, even outside of the server
    - The entry interface shall be either identical or with additional parameters all with default values, or with no parameters
- Implications
  - Transferring to a queue in the same server fits the eggshell model semantics nicely
    - In addition to yielding good functional cohesion
  - Transferring to an entry queue outside of the server requires releasing the R/W lock held on it
    - Without the eggshell model knowing exactly what to do

Alternative 2 – 4

Issue 2: handling of time-out

- Two possible outcomes
  1) Call `B.E1` not accepted within `T1` gets aborted
  2) Call `B.E1` accepted and then transferred to `B.E2`, is aborted if not accepted there within `T1`
- Outcome 2) incurs an ugly temporal distortion

Example

```
-- client A
select
 B.E1;
or
 delay T1;
end select;
```

```
-- server B
select
 accept E1 do
  … -- T2 time units

  requeue E2 with abort;
 end E1;
or
 …
end select;
```

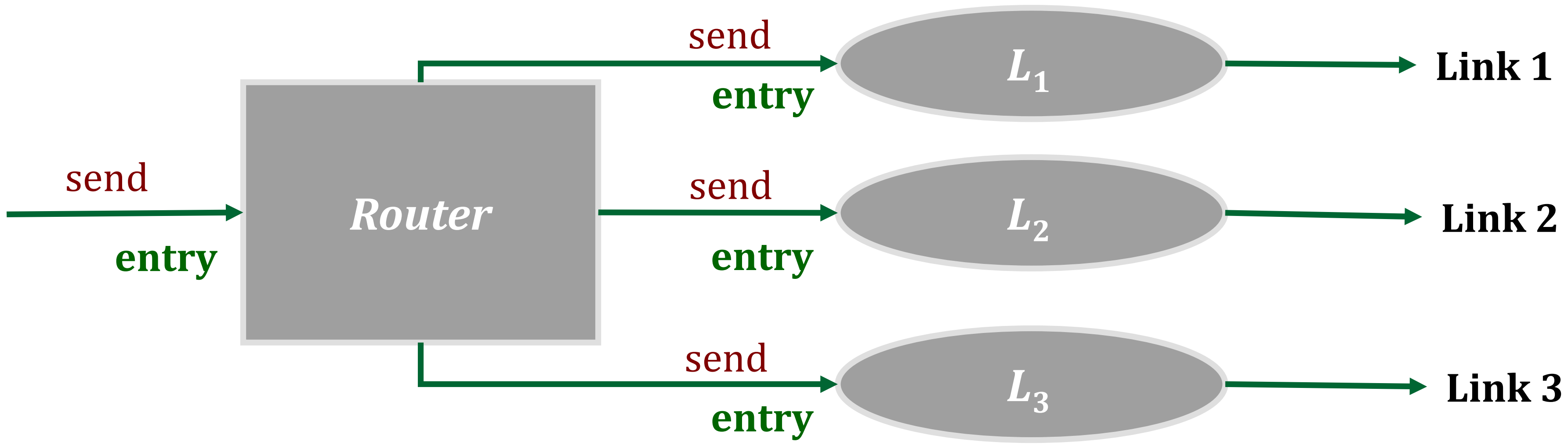
The **`with abort`** clause preserves the time-out effect upon call transfer

Use cases – 1

- Appropriate use of the `requeue` feature helps realize resource managers quite neatly
  - Check the implementation linked to today's lecture
- **Homework**
  1. Try and improve the given solution in a manner that avoids useless call transfers
  2. Try the same solution with a programming language of your choice

Use cases – 2

- A network router may forward inbound packets on to $N = 3$ outbound links
  - Link L_1 is the preferred choice, but the other links (first L_2 and then L_3) are used when L_1 risks overloading
- Likening packets to calls, and router and links to servers maps packet forwarding to a requeue



Flipped-class exercise

- Realize a circular-line metro service simulator
  - $M > 1, M \in \mathbb{N}$ train stations along a circular line
  - $N > M, N \in \mathbb{N}$ commuters who forever revolve around one and the same duty cycle
    1. Go from home to the nearest train station
    2. Board the first possible train
    3. Get off the train and go to work (and work as due)
    4. Go from work to the nearest train station
    5. Board the first possible train
    6. Get off the train and go home (and rest as allowed)
  - 1 commuter train with capacity $C < N$ (no prebooking)