On concurrent programming

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

Premise – 1

- What a language is able to “say” determines its expressive power, but also sets its limit
  - What the language is unable to say, does **not** exist for the speaker of it
  - Cit.: “*The limits of my language are the limits of my mind. All I know is what I have words for.*”

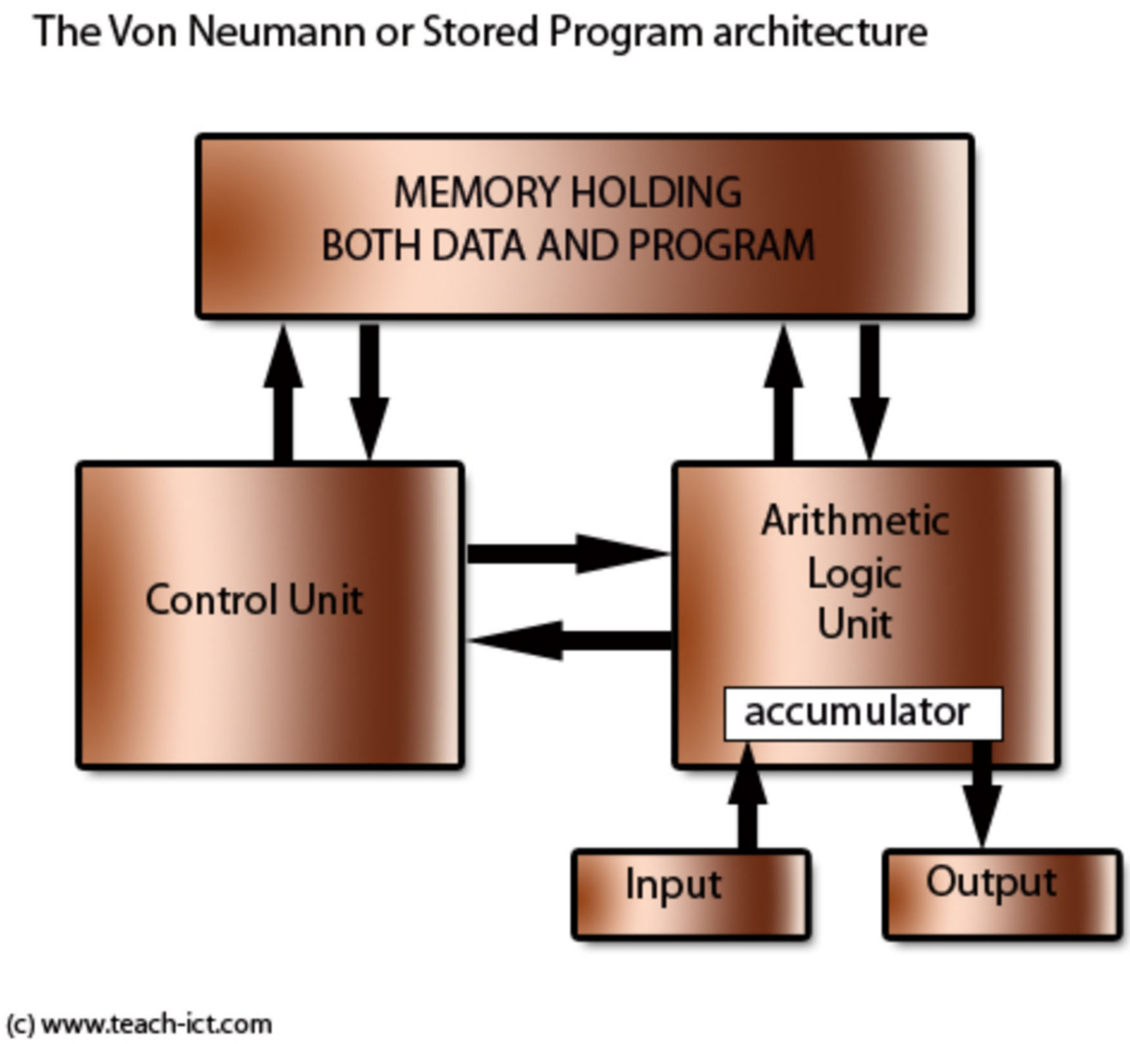
    Ludwig Wittgenstein, Tractatus Logico-Phisolophicus, 1922
- Programming languages are no different
  - Procedures, classes, threads, …, may “exist” in some languages and not in others

Premise – 2

- Most (historical and current) programming languages are **sequential**
  - Their nature is to reflect the execution model of the underlying processor

    Note: for now, we assume single-CPU computers
- All "traditional" processors conform to the von Neumann architecture
  - They are **stored-program computers**
    - A single memory for code and data, and one CPU whose duty cycle forever revolves around a fetch-decode-read-execute-write pipeline
  - Strictly sequential execution model

The von Neumann architecture



Concurrent languages – 1

- Concurrency can be “added” to a sequential programming language by tapping into the underlying OS
  - E.g., leveraging UNIX’s `fork()` / `exec()` model
- In that manner, the expression and the semantic control of the program’s concurrency are **outside** of the language
  - The program becomes not portable and its semantics weaken

Concurrent languages – 2

- The alternative is to enable the language to express multiple simultaneous “places of control”
  - The language runtime must virtualize them on the single program counter provided by the processor
  - This requires realizing the abstraction of “**thread**”, with its own context, within the context of the program’s **process**

  Note: we call “process” a program that executes under the hosting of a multi-programmed OS

Concurrent languages – 3

- Designing language-level concurrency should conform (aspire) to a **model of concurrency** that be coherent and consistent
  - For choice of abstractions (*what*) and of runtime semantics (*how*)
    - The alternative ☹ is to provide basic, low-level utensils and tell the user: *Do It Yourself*!
  - Expressive, efficient and verifiable: quite a challenge!
- Concurrent programming simplifies the application architecture helping to capture the pattern of collaboration inherent in the problem
  - Program-level concurrency is **collaborative**

Forms of concurrency – 1

- Seen from the outside (the host environment), an executing program is a process
  - To escape the sequential prison, the process abstraction requires an execution context that can be saved and restored across pre-emptions
- The process execution model may stipulate that
  a) All processes share the same processor
  b) Each process has its own processor and all processors share memory (*these would be today's multicores*)
  c) Each process has its own processors and processors do not share memory

Forms of concurrency – 2

- Each such solution implies different models of execution
  - We have **parallelism** when multiple processes “own” a CPU at the same point in time
  - We have **concurrency** when processes might execute in parallel, but do not need to
    - The application is able to make progress *without* parallelism
- When parallel hardware was not a commodity, concurrent programming was the privileged way to explore parallel solutions to a problem and assess their synchronization and communication overhead
  - In that regard, concurrency is more powerful than parallelism

Concurrency vs. Parallelism

Concurrency

- Allows simplifying the application logic with *long-lived threads* of control that capture patterns of <u>collaboration</u> in the solution space
  - *Heavier-weight* concurrency constructs (stack, context, …) can be acceptable
- Key trait: collaboration

Parallelism

- Allows a *divide-and-conquer* approach to the problem, using *short-lived threads* to work in parallel on <u>independent</u> parts of it
  - Concurrency constructs should be *light-weight* as they are used very often at run time
- Key trait: independence

Observations

- Given n processes and m processors
  - When $1 = m < n$, a concurrent solution to a problem may yield a quality software architecture and achieve high utilization of the CPU
  - When $1 < n \leq m$, any solution has **speed-up** $S \leq n$, contingent on the extent of effective parallelism that it can achieve
  - When $1 < n \ll m$, concurrency does not help anymore: extreme parallelism must be sought to make effective use of the processors

Precursors of concurrency – 1

- **Coroutine**: a bit of history
  - One of the first and most basic ways to express concurrency programmatically
    - M.E. Conway, *Design of a separable transition-diagram compiler*, Communications of the ACM, 6(7), July 1963
    - A *separable* program breaks into processing modules that communicate with adjacent modules as if they were input or output subroutines
  - There is main program anymore, but an explicit (programmed) alternation of execution among concurrent routines
    - Commands `yield[_to]` or resume
  - Very convenient to program discrete simulation: multiple events occurring at discrete points in time after some causal chain
    - Featured in SIMULA 67
    - Then carried over in Modula-2
  - More recently incorporated by Ruby (as of v1.9.0) and other mainstream languages for *asynchronous programming*

Precursors of concurrency – 2

```
var q := new queue
coroutine produce
 loop
  while (q not full)
   <create item>
   <add item to q>
   yield to consume
   <point of resumption>
  end while
 end loop
coroutine consume
 loop
  while (q not empty)
   <remove item from q>
   <use item>
   yield to produce
   <point of resumption>
  end while
 end loop
```

- The coroutines that relinquish the CPU preserve their context (their stack)
  - Normal procedures lose it on return
  - For this reasons, the coroutines are also terms "**continuations**"
- The coroutines have multiple points of entry
  - All the *points of resumptions*
  - Procedures only have one
- The execution of a coroutine may "return" multiple times before ending
  - Procedures return once

Nomenclature

- We call ***threads*** the flows of control that may exist within one process
  - **Remember**: a process is a "*program in execution*"
- In a concurrent language, such threads are managed by the language runtime
  - The OS **may** support threads within processes, but there need **not** be a 1:1 mapping between OS threads and runtime threads
  - **Remember**: the runtime's prime responsibility is to realize the language semantics, not to lazily piggyback on the OS's

Forms of concurrency – 3

- Declaring and activating threads

Algol68, CSP, Occam

```
cobegin
   P1;  P2;  P3;
coend;
```

Explicit activation
Separate from declaration:
what are `P1`**,** `P2`**,** `P3`**?**

Ada

```
procedure Main is
    task A;
    task B;
    …
    task A is … ;
    task B is … ;
begin
    …
end Main;
```

Declaration
in Main's declarative region

Implementation
in Main's declarative region

Implicit activation
at start of Main's executive region

At this point,
`Main`**,** `A`**,** `B` **are**
concurrent threads

A model of concurrency – 1

- Concurrent entities can be
  - **Active**, able to execute without depending on others (if granted the necessary compute resources)
  - **Reactive**, only capable of executing in response to application-level triggers
    - **Resources**, with an internal state, and pre- and post-conditions holding on access to it
      - How to realize such access control?
    - **Passive**, with no internal state
      - A plain procedure (not a concurrency abstraction)

A model of concurrency – 2

- Realizing such model requires three distinct concurrency abstractions
  - **Threads**, for active entities
  - Active-control resources (**servers**)
    - More expressive: more sophisticated access protocols
    - Heavier weight: cost like a thread while often quiescent
  - Passive-control resources
    - May use **semaphores** or (better) "**monitors**"
    - Less expressive: more basic and inflexible access protocols
    - Lighter weight

Pros and cons

- Language-level concurrency allows for
  - More readable, better organized programs
  - Portable semantics, warranted by the runtime, independent of the underlying OS
    - Also good for embedded applications, which do not use general-purpose OS
- Predicates on the choice of a suitable model of concurrency (expressive, efficient, verifiable)
- Independence from the underlying OS is costly
  - At the extreme, it is like doing the same “thing” twice
- A well-defined model of concurrency is antagonistic to a “Do-It-Yourself” style
  - Choosing a model causes loss of generality

Time matters – 1

- The presence of pre-emption causes the progression of time to matter to program execution
  - With pre-emption, “time” advances faster than execution
- The runtime must support an abstraction of **time**
  - The processor **clock** is a composite entity
    - A HW down-counting register that raises an interrupt on zero
    - A SW up-counting register that advances on every HW zero
- The question is what should “time” mean
  - A wall clock (hours, minutes, seconds from a base epoch)
  - A source of monotonic time (which makes no back jumps)
  - A measure of bounded intervals (as quanta in time sharing)

Time matters – 2

Wall-clock time

```
declare
   Start, Finish : Ada.Calendar.Time;
   Interval : Ada.Calendar.Duration := 2.5;
begin
   Start := Ada.Calendar.Clock;
   … -- actual work
   Finish := Ada.Calendar.Clock;
   if Finish - Start > Interval then
      raise Overrun_Error;
   end if;
end;
```

This programming implicitly assumes single-threading!

Monotonic time

```
Ada.Real_Time.Time;
```

```
Ada.Real_Time.Time_Span := Ada.Real_Time.To_Time_Span(2.5);
```

```
Ada.Real_Time.Time_Span := Ada.Real_Time.Milliseconds(2500);
```

Time matters – 3

- A runtime clock may also be used to program time-dependent actions
  - **Relative** suspension

```
delay 10.0; -- type is Ada.Calendar.Duration
```

    - Counting from when the command is evaluated
    - Suspension is guaranteed to be no less than the required length (but no upper bound on it)
  - **Absolute** suspension

```
delay until A_Time; -- type is Ada.Real_Time.Time
```

    - The time of expiry is actual, independent of when the command is evaluated

Time matters – 4

A

```
Start := Clock;
First_Action;
delay until (Start + 10.0);
Second_Action;
```

B

```
Start := Clock;
First_Action;
delay (Start + 10.0 - Clock);
Second_Action;
```

- Fragments A and B do *not* have the same effect because the evaluation of the boxed statement in B takes very different results if preempted
  - We cannot know when the call to clock in **B** will be evaluated
    - The awake time is unknown
  - The evaluation of the “delay until” in **A** call is not effected by pre-emption
    - The awake time is known

Time matters – 5

- With a monotonic clock and absolute suspensions, programming **periodic threads** is straightforward

```
with Ada.Real_Time; use Ada.Real_Time;
…
task body Periodic_Task is
   Interval : constant Time_Span := Millisecond(10_000);
   Next_Time : Time := <System_Start_Time>;
begin
   loop
      delay until Next_Time;
      Periodic_Action;
      Next_Time := Next_Time + Interval;
   end loop;
end Periodic_Task;
```

Time matters – 6

- Regardless of how the clock abstraction is implemented in the runtime, keeping time in the program is exposed to two hazards
  - **Local drift**, the minimum time distance between two successive accesses to the clock
    - Inevitable: it depends on the costly complexity of supporting the abstraction of time
  - **Cumulative drift**, the chain effect caused by the accumulation of local drift and program naiveties
    - Evitable: using absolute delays links the time seen in the program to the actual progress of it