Parallel and Concurrent Programming

Jacob Sparre Andersen

JSA Research & Innovation

October 2017

Parallel and concurrent programming

What it is, and how to do it with programming language support.

Using Ada as the demonstration language, I will give a quick course in how to write concurrent and parallel programs. Both language constructs and conceptual issues will be covered.

Terminology

- Parallel programming
- Concurrent programming

Terminology: Parallel programming

Programming for actual **simultaneous execution**.

(Single | Multiple) Instruction (Single | Multiple) Data:

- SISD - how we learn to program
- SIMD - vector processors
- MIMD - multi-core processors
- MISD - space shuttle avionics software

Typically seen as way to speed a program up:

$$t' = t \cdot (1 - p + \frac{p}{n})$$

Terminology: Concurrent programming

Programming **logically simultaneous** execution.

A way to abstract away consideration about how many cores are actually available for simultaneous execution.

The actual execution may be sequential (i.e. on a single core computer).

An extension of how programming languages (mostly) abstract away unnecessary details about how the processors we're going to run our programs on work.

Basic concepts

- **Process:** A sequence of instructions which executes concurrently with other sequences of instructions.
- **Shared resource:** Any kind of resource (memory area, I/O port, etc.) which may be used by more than one *process*.
- **Mutual exclusion:** Preventing two processes from simultaneously accessing a resource.
- **Atomic action:** An operation on a *shared resource* which can not be interleaved with other operations on the same resource.
- **Synchronisation:** Coordination of processes to reach a common goal.

Failure-modes

- **Deadlock:**
  Processes not making progress (typically because of competition about shared resources).
- **Starvation:**
  Indefinite postponement of processes (typically because other processes prevent their access to shared resources).
- **Race condition:**
  When the behaviour of the program depends on the sequence or timing of external events (in an unintended manner).

Failure-modes: Deadlock

Two processes (p_1 and p_2) and two resources (r_1 and r_2):

Step	p_1	p_2
1	Takes r_1.	Takes r_2.
2	Takes r_2.	Takes r_1.
3	Manipulates r_1 and r_2.	Manipulates r_1 and r_2.
3	Releases r_1.	Releases r_2.
4	Releases r_2.	Releases r_1.

If both processes finish step 1 before either of them start on step 2, both processes will be stuck in step 2 because the other process has the missing resource.

Failure-modes: Starvation

Indefinite postponement of a process.

If a process is waiting for access to a shared resource, and other processes continuously keeps it locked out, we say that the process is being **starved**.

Failure-modes: Race condition

Race conditions typically occur, when a sequence of operations is assumed to be atomic by the programmer, while this isn't the case in reality.

```
if [ ! -d "${directory}" ]; then
    mkdir "${directory}"
fi
```

Exercise:

How can this fail?

task

In Ada a process is called a “task”.

A trivial task declaration:

```
task Multiply;
```

And a matching body:

```
task body Multiply is
begin
   Product := 1.0;
   for Element of Values loop
      Product := Product * Element;
   end loop;
end Multiply;
```

Statically or dynamically created

Tasks can exist as stand-alone entities or as types.

```
task Multiply;
```

```
task type Receiver;
```

You create an actual task from a task type either by declaring an object of the type, or by allocating a task object on the heap:

```
One : Receiver; --  object
```

```
Two := new Receiver; --  heap
```

```
procedure Vector_Math (Values  : in      Vector;
                       Sum     :    out Scalar;
                       Product :    out Scalar) is
   task Multiply;

   task body Multiply is
   begin
      Product := 1.0;
      for Element of Values loop
         Product := Product * Element;
      end loop;
   end Multiply;
begin
   Sum := 0.0;
   for Element of Values loop
      Sum := Sum + Element;
   end loop;
end Vector_Math;
```

Protected object

In Ada you implement mutual exclusion using a “protected object”.

A protected object consists of a number of **operations**, and some private **data**.

To the extent that the operations only operate on the private data of the protected object, they are *atomic*.

Example (without mutual exclusion)

```
type Population_Size is range 0 .. 100;
--  Defined by fire regulations.

package Population is
   procedure Increment;
   procedure Decrement;
   function Current return Population_Size;
private
   Count : Population_Size := 0;
end Population;
```

Exercise:
What can go wrong, if you have multiple processes (tasks) accessing the `Population` package?

Example

Here as a protected object with proper mutual exclusion:

```
type Population_Size is range 0 .. 100;
--  Defined by fire regulations.

protected Population is
   entry Increment;
   procedure Decrement;
   function Current return Population_Size;
private
   Count : Population_Size := 0;
end Population;
```

Explanations

```
function Current return Population_Size;
```

functions have joint read access and blocks writing to the private data.

```
procedure Decrement;
```

A **procedure** has exclusive read/write access to the private data.

```
entry Increment;
```

An **entry** has exclusive read/write access to the private data, but calls can be blocked based on the internal state.

Example (implementation)

```
protected body Population is
   entry Increment
     when Count < Population_Size'Last
   is
   begin
     Count := Count + 1;
   end Increment;

   procedure Decrement is
   begin
      Count := Count - 1;
   end Decrement;

   function Current return Population_Size is (Count);
end Population;
```

Deadlock: Practical help

Finding deadlocks is NP-complete problem.

But a research group in Vienna has developed a technique using Kronecker algebra for proving the absence of deadlocks[1].

There can still be deadlocks hidden in dead code, which are not detected, but as the code will never be executed, it is not a problem in practice.

[1] Which is really what we want to know as developers.

Deadlocks: A simple protocol

A basic procedure for avoiding potential deadlocks due to competition for shared resources is that all processes takes/locks the shared resources in the same order.

Swapping the order of taking the resources in one of the two processes in the deadlock example earlier would prevent a deadlock from ever occurring there.

Synchronisation

In Ada we can implement synchronisation between tasks either indirectly using *protected objects* or directly using a "rendezvous".

The rendezvous is a client-server message passing construction:

- The server task declares a number of *entries*, which it may accept and execute at its leisure. (There is no guarantee that the server task will ever accept its published entries.)
- The client task can then call any of these entries (it looks like a plain procedure call).

Note that any task can function both as a client and a server.

Synchronisation (continued)

If a server is ready to accept an entry, but no client has called it yet, the server task will be blocked (waiting) until somebody calls the entry[2].

If a client calls an entry, but the server isn't ready to accept it, then the client task will be blocked (waiting) until the server is ready to accept the entry call[3].

[2] We skip selective accepts for now.

[3] The client-side too has a bit more options than this.

Synchronisation (continued)

Specification of a task with a single entry:

```
task Worker is
   entry Set_ID (Value : in     IDs);
end Worker;
```

Implementation of the same task:

```
task body Worker is
   ID : IDs;
begin
   accept Set_ID (Value : in     IDs) do
      ID := Value;
   end Set_ID;

   Do_Something (ID);
end Worker;
```

Synchronisation: Semaphore

A protected object with a counter and two operations:

- **Wait**: Wait until the counter is greater than zero, then decrement the counter.
- **Signal**: Increment the counter.

Semaphores are used to provide mutual exclusion in cases where you can't put the whole protected block into a *protected object*.

Synchronisation: Semaphore in Ada

The specification of a semaphore type in Ada:

```
package Semaphore is
   protected type Instance (Initial_Value : Natural) is
      entry Wait;
      procedure Signal;
   private
      Count : Natural := Initial_Value;
   end Instance;
end Semaphore;
```

Synchronisation: Semaphore in Ada

The implementation of the semaphore type:

```
package body Semaphore is
   protected body Instance is
      procedure Signal is
      begin
         Count := Count + 1;
      end Signal;

      entry Wait when Count > 0 is
      begin
         Count := Count - 1;
      end Wait;
   end Instance;
end Semaphore;
```

Synchronisation: Semaphore use

```
Lock : Semaphore.Instance (Initial_Value => 1);

procedure Put_Line (Item : in     String) is
begin
   Lock.Wait;
   Ada.Text_IO.Put_Line (Item);
   Lock.Signal;
end Put_Line;
```

The procedure `Ada.Text_IO.Put_Line` is potentially blocking, so we are not allowed to call it from inside a protected object. Instead we use the semaphore (`Lock`) to ensure that only one task at a time is calling `Ada.Text_IO.Put_Line` (through this package).

Synchronisation: Finalised lock (specification)

```
with Ada.Finalization;

generic
package Finalized_Lock is
   --  An object of this type will automatically wait on
   --  the semaphore declared inside the package when it
   --  is created, and signal the semaphore when it goes
   --  out of scope.
   --
   type Instance is new Ada.Finalization.
      Limited_Controlled with null record;
private
   overriding
   procedure Initialize (Item : in out Instance);
   overriding
   procedure Finalize (Item : in out Instance);
end Finalized_Lock;
```

Synchronisation: Finalised lock (implementation)

```
package body Finalized_Lock is
   Lock : Semaphore.Instance (Initial_Value => 1);

   overriding
   procedure Initialize (Item : in out Instance) is
      pragma Unreferenced (Item);
   begin
      Lock.Wait;
   end Initialize;

   overriding
   procedure Finalize (Item : in out Instance) is
      pragma Unreferenced (Item);
   begin
      Lock.Signal;
   end Finalize;
end Finalized_Lock;
```

Synchronisation: Finalised lock (use)

```
task body GULCh is
   Key : Lock.Instance with Unreferenced;
begin
   Ada.Text_IO.Put ("Linux ");
   delay 0.2;
   Ada.Text_IO.Put ("Day ");
   delay 0.2;
   Ada.Text_IO.Put ("2017 ");
   delay 0.2;
   Ada.Text_IO.Put ("a ");
   delay 0.2;
   Ada.Text_IO.Put_Line ("Cagliari");
end GULCh;
```

Synchronisation: Futures

Futures is a popular concept in non-concurrent programming languages.

Basically your apparently non-concurrent program hands off some processing to an invisible process to get the result back sometime in the future.

The equivalent Ada construct is to use concurrency explicitly, and exchange results using a protected object or a direct rendezvous.

Synchronisation: Interrupts

Interrupts represent events detected by the hardware.

In Ada you use protected procedures as interrupt handlers:

```
protected Converter is
   entry Read (Value :      out Voltage);
private
   procedure Handler;
   pragma Attach_Handler (Handler, SIGINT);

   entry Wait_For_Completion (Value :      out Voltage);

   Busy                : Boolean := False;
   Conversion_Complete : Boolean := False;
end Converter;
```

Tips and tricks

Some Ada specific tips for parallel and concurrent programming.

Task attributes

The package `Ada.Task_Attributes` allows the developer to attach an attribute (i.e. a variable) to all tasks in a system.

This can for example be useful if you want to let each task have its own database connection, but you don't want to make that visible in the source text for individual tasks.

Task termination

The package `Ada.Task_Termination` allows the developer to attach a termination handler to all dependent tasks (or to a specific task).

In one application I work on, we are using it to ensure that it is logged if a task terminates in an unexpected manner.

CPU

Ada defines the aspect `CPU` for tasks and task types.

You can use it to assign specific tasks to run on specific CPU's.
The problem with this is that you end up hard-coding hardware architecture in the source code, thus writing concrete, parallel programs instead of abstracted, concurrent programs.
But there are certainly cases where it is the correct approach.

References

If you want to learn more, I suggest the book “Building Parallel, Embedded, and Real-Time Applications with Ada” [1].

Several of the examples in this presentation were inspired by it.


John W. McCormick, Frank Singhoff, and Jérôme Hugues. *Building Parallel, Embedded, and Real-Time Applications with Ada*.
Cambridge, 2011.

Contact information

Jacob Sparre Andersen
JSA Research & Innovation
jacob@jacob-sparre.dk
http://www.jacob-sparre.dk/

Examples:
http://www.jacob-sparre.dk/programming/parallel_programming/linux-day-2017-examples.zip