Comparison of the Concurrency Concepts of Ada, CHILL, Erlang, and Java

Case Studies—2

DIPLOMARBEIT

zur Erlangung des akademischen Grades Diplom-Informatiker

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA
Fakultät für Mathematik und Informatik
Institut für Informatik
Lehrstuhl für Programmiersprachen und Compiler

eingereicht von: Frank Ecke
geboren am 22. Oktober 1974 in Sondershausen

Betreuer: Prof. Dr. Jürgen Winkler (FSU Jena)
Dr. Stefan Kauer (FSU Jena)

14. Juli 1999

Extended Abstract

The objective of this paper, which is an extension to (Brömel, 1998), is to further elaborate on the comparison of the concurrency concepts of four programming languages—Ada, CHILL, Erlang, and Java. We will use case studies which cover a variety of application domains to carry out the comparison. Although all four languages offer the programmer a rich set of facilities to tackle problems, none of them can be seen as a "panacea." Whenever communication is a vital part of an application, the asynchronous communication protocols in CHILL and Erlang are hampering in that they make it necessary to impose synchroneity explicitly. Additional effort is thus needed. Java's concurrency model is not adequate for direct inter-process communication. Ada, originally developed for realtime systems, is weak in this very domain.

A classical assignment in concurrent programming is to ensure mutually exclusive access to critical resources. Generally, two approaches spring to mind: a resource manager, which accepts calls from external clients and which ensures that, at any time, at most one request is being dealt with; or a passive entity that has associated with it an exclusive lock. To use this passive construct, the lock must be acquired beforehand. There are cases in which one method may have advantages over the other, which is why a programming language intended for concurrent programming should indeed offer both techniques. Erlang, however, does not know passive entities that provide for mutually exclusive access to their components—we only have the *active* solution here. *Abstraction inversion* might ensue—things which are inherently passive have to be modeled as active components. A good case in point are the forks in the problem of the Dining Philosophers. They represent a critical resource and, therefore, have to be protected, and they are passive. Nonetheless, we must use a process, which is active, to implement them. The possible effect is that there is more overhead at runtime. Java's lacking of direct inter-actor communication makes the development of a resource manager more complicated.

More often than not, communication plays an important role in an application. If an actor communicates with more than one partner, it is often desirable (if not necessary) that the communication comes about in an ordered manner. Resource budgeting, for instance, should be carried out in such a way that the temporal ordering of the satisfaction of resource requests reflects the temporal ordering of the requests. Alternatively, priorities could be taken into account, etc. A prerequisite for this, however, is that the set of actors waiting for resources is ordered. Unfortunately, only in Ada and Erlang do we find ordered wait sets. If a (waiting) CHILL or Java actor is reawakened, then there is no guarantee that the desired resource is available. If not, then the wrong actor has been resumed—in vain, of course, and it must be included into the wait set again. This has to be done manually, which entails additional effort. There is nothing which prevents the CHILL or Java runtime system to reawaken the (or a) wrong actor the next time a resource item becomes available. The process of false promotion might happen time and time again. We call this a less severe form of *busy waiting*. There is no quick and easy solution to the problem—a certain minimal effort is obviously always required: we have to evaluate the conditions in order to find out whether a resource item is available. However, this effort can be minimized. If we think about *who* performs this additional work *at what time*, and if we construct the runtime system according to our findings, we are able to keep the effort within bounds. Clearly, if actors check the conditions on their own (necessarily after they have been made active again), then this means extra context switches. If we assigned this kind of work to the runtime system, then these could be avoided. That is the way things go in Ada and Erlang: *first*, a check is made that the desired resource is free and *then* (and only then) the corresponding actor is reawakened. The use of a passive

construction in the preceding sentence shall express the fact that all this is done by the runtime system; we are only required to state the resource/condition.

We look at two special applications of concurrent programming: *interrupt handling* and *process scheduling*. Interrupt handling is a frequently encountered topic in embedded systems development. It poses a variety of problems to the programmer. Not only are these problems hardware specific (we have to target the application onto the particular architecture), but also the very nature of interrupt handlers is, upon second glance, different from that of ordinary subprograms. The call protocols might be varying, and there might be restrictions as to which statement is allowed inside such a handler. From the four languages, only one—Ada—is helpful in this special domain. While this is true, it is not entirely fair towards Java. The intended application area for this language is not to be sought in the realms of realtime or embedded systems. Given how important Java has become in recent years, it is worthwhile looking beyond the horizon. And indeed, the author did find something appropriate. PERC, which is a dialect of the Java language, has provisions for interrupt handling. Furthermore, PERC is the only language in our quartette/quintette that truly lends us a hand when it comes to process scheduling. Normally, it is not of paramount importance when a process/a set of processes is executed. The functional correctness of a program must not be dependent upon this. Unfortunately, there is another form of correctness—*temporal* correctness. We can often ignore this one, but there are situations in which we cannot indulge in this kind of luxury: realtime systems spring to mind immediately. In these systems, delivering an exact result too late (or too early) is worse than delivering a slightly imprecise result in time. When developing realtime systems, much time is spent and painstaking effort is undertaken in order to make certain that process scheduling works. Processes are equipped with *temporal scopes*, which indicate when a process is to be started, how long it may run (or how often), when it must have finished, etc. It looks good on paper certainly, but sooner or later, we have to implement the processes. What, we have to ask ourselves, does the programming language chosen offer as regards the specification of temporal scopes? If we have chosen Ada, CHILL, Erlang, or Java, the answer is intimidating: nothing! Things that work in theory have to be realized in a clumsy and troublesome way. The temporal scopes have to be simulated. It is no longer apparent to anyone reading the program that particular processes have to execute within stringent temporal bounds. Perhaps even worse, this is not apparent to the runtime system either, which is supposed to supervise the temporal behavior of processes. The light at the end of the tunnel is not that of an oncoming freight train, but PERC. This language allows the specification of temporal scopes and has also means that help in the supervision of the temporal behavior of processes.

Deutsche Zusammenfassung der Arbeit

Ziel der Arbeit, die eine Fortsetzung und Erweiterung von (Brömel, 1998) darstellt, ist es, durch Fallstudien den Vergleich der Nebenläufigkeitsaspekte von vier Programmiersprachen—Ada, CHILL, Erlang und Java—zu erweitern und zu vertiefen. Die Beispiele sind so gewählt, daß sie ein möglichst breites Feld abdecken. Es zeigt sich, daß—trotz der Vielfalt der von den Sprachen gebotenen Ausdrucksmittel—keine Sprache allmächtig ist. In den Beispielen, bei denen Kommunikation eine wichtige Rolle spielt, tritt die Asynchronität, die in CHILL und Erlang zu finden ist, negativ auf. Sie muß unterbunden

werden, was natürlich zusätzlichen Entwicklungs- und Programmieraufwand bedeutet. Das Nebenläufigkeitsmodel von Java beispielsweise erlaubt keine direkte Kommunikation zwischen Akteuren. Ada, eigentlich für Echtzeitsysteme entwickelt, ist gerade in diesem Bereich nicht überzeugend.

Eine Standardaufgabe beim nebenläufigen Programmieren ist es, abzusichern, daß gegenseitiger Ausschluß beim Zugriff auf kritische Ressourcen gewährleistet ist. Um dies zu erreichen, gibt es im wesentlichen zwei Möglichkeiten: einen Ressourcenmanager, der Anrufe von der Außenwelt entgegenimmt und sicherstellt, daß immer nur maximal ein Anruf bearbeitet wird; oder ein passives Konstrukt, für das, um es zu verwenden, vorher eine exklusive und nur einmal vorhandene Berechtigung erworben werden muß. Beide Methoden haben ihre Anwendungsgebiete, weswegen eine Programmiersprache, die Nebenläufigkeit unterstützt, auch beide anbieten sollte. Im Falle von Erlang finden wir aber nur die *aktive* Lösung. *Abstraktionsinversion* ist die Folge—Dinge, die an sich passiv sind, müssen mit aktiven Komponenten realisiert werden. Dies ist sehr schön sichtbar beim Problem der speisenden Philosophen, bei denen die Gabeln eine passive kritische Ressource darstellen. Eine Auswirkung in diesem Fall ist, daß möglicherweise erhöhter Verwaltungsaufwand zu Laufzeit entsteht. Bereits erwähnt wurde das Fehlen von direkter Kommunikation in Java, was die Entwicklung eines Ressourcenmanagers hier erschwert.

Kommunikation ist oftmals ein zentrales Thema einer Anwendung. Tritt die Kommunikation mit mehreren Partnern auf, so ist es häufig wünschenswert (ja sogar notwendig), daß sie in einer bestimmten Reihenfolge abläuft; z.B. sollte bei der Ressourcenverteilung so verfahren werden, daß sich die zeitliche Vergabe der Ressourcen in der zeitliche Ordnung der Anforderungen widerspiegelt oder daß Prioritäten beachtet werden usw. Dies setzt jedoch voraus, daß die Menge der Wartenden geordnet ist. Leider finden wir das nur in Ada und Erlang. Wird ein wartender Akteur in CHILL oder Java aufgeweckt, so heißt dies nicht notwendigerweise, daß die Ressource, auf die er gewartet hat, auch tatsächlich verfügbar ist. Falls nicht, so wurde der falsche Akteur geweckt, und er muß wieder in die Wartemenge eingebracht werden. All dies ist händisch auszuführen, d.h., erfordert zusätzlichen Aufwand. Niemand verbietet dem Laufzeitsystem von CHILL oder Java jedoch, beim nächsten Male wieder den (oder einen) falschen Akteur aufzuwecken usw. Wir nennen das eine abgeschwächte Form von *busy waiting* und wissen, daß es nicht gut ist. Eine Lösung des Problems ist nicht einfach; es ist offenbar stets Aufwand zu betreiben, der zum Prüfen der Bedingung nämlich. Verringert werden kann dieser Aufwand allerdings, wenn wir uns überlegen, *wer* ihn *wann* am besten betreibt und das Laufzeitsystem entsprechend bauen. Wenn alle Akteure für sich selbst prüfen, und—dann notwendigerweise—nachdem sie aufgeweckt wurden, so bedeutet das zusätzliche Prozeßwechsel, die vermieden werden könnten, würden wir dem Laufzeitsystem die Aufgabe des Prüfens übertragen. In Ada und Erlang funktioniert die Sache so—es wird *zuerst* geprüft, ob die gewünschte Ressource verfügbar ist, *bevor* (und nur dann) der Akteur geweckt wird. Die Passivkonstruktion im vorangegangenen Satz soll ausdrücken, daß dies vom Laufzeitsystem erledigt wird, wir geben nur die gewünschte Ressource/Bedingung an.

Zwei der Fallstudien befassen sich mit Spezialanwendungen der nebenläufigen Programmierung: *Interruptbehandlung* und *Zeitliches Einplanen von Prozessen.* Interruptbehandlung tritt oftmals im Zusammenhang mit eingebetteten Systemen auf und stellt die Programmierer vor einige Probleme. Nicht nur sind diese hardwarespezifisch (es müssen im allgemeinen die Besonderheiten der zugrundeliegenden Hardware berücksichtigt werden), sondern auch die Natur von Interruptbehandlungsroutinen unterscheidet sich bei genauerem Hinsehen von der normaler Unterprogramme. Die Aufrufmodelle sind unter Umständen verschieden, und nicht jede Anweisung ist geeignet innerhalb einer solchen

Routine. Von den vier untersuchten Sprachen ist nur eine, Ada, hilfreich auf diesem Gebiet. Dies ist zwar wahr, aber nicht ganz gerecht Java gegenüber. Das Anwendungsgebiet dieser Sprache ist nicht im Bereich der Echtzeit- oder eingebetteten Systeme zu suchen. Bei der Beliebtheit, der sich Java momentan erfreut, war es jedoch angebracht, sich auch einmal jenseits des Tellerrandes umzusehen. Und in der Tat, der Autor ist fündig geworden. PERC, ein Dialekt der Sprache Java, bietet Mittel, um Interruptbehandlung in den Griff zu bekommen. Mehr noch, PERC ist die einzige Sprache in dem Quartett/ Quintett, die uns beim zeitlichen Einplanen von Prozessen wirklich zur Hand geht. Normalerweise ist es nicht so wichtig, wann ein bestimmter Prozeß/eine Menge von Prozessen ausgeführt wird. Die funktionale Korrektheit eines Programms darf davon nicht abhängig sein. Unglücklicherweise gibt es aber auch noch eine weitere Form der Korrektheit— die *zeitliche*. Wie gesagt, oftmals kann diese ignoriert werden, aber es gibt natürlich auch Situationen, in denen wir uns diesen Luxus nicht erlauben können: in Echtzeitsystemen beispielsweise. Hier ist ein zu spät abgeliefertes (genaues) Ergebnis schlimmer als ein rechtzeitig erbrachtes, das vielleicht ein bißchen ungenau ist. Beim Entwickeln von Echtzeitsystemen wird viel Zeit und Mühe darauf verwandt, sicherzustellen, daß die zeitliche Einplanung funktioniert. Für Prozesse werden sogenannte *temporal scopes* festgelegt, die angeben, wann ein Prozeß zu starten ist, wie lange er laufen darf oder wie oft, wann er beendet sein muß usw. Auf dem Papier sieht das alles sehr gut aus, irgendwann jedoch müssen die Prozesse aber einmal implementiert werden. Was hat die gewählte Programmiersprache zu bieten bezüglich der Spezifikation von *temporal scopes*? Fiel die Wahl auf Ada, CHILL, Erlang oder Java, so ist die Antwort ernüchternd: nichts! Das, was in der Theorie so schön funktioniert, muß jetzt mehr oder weniger mühsam in die Praxis umgesetzt werden. Die *temporal scopes* müssen simuliert werden. Niemand sieht den Prozessen mehr an, daß sie bestimmte zeitliche Rahmenbedingungen einzuhalten haben. Am wenigsten das Laufzeitsystem, was ja eigentlich diese Bedingungen überwachen soll. Licht am Ende des Tunnels verspricht PERC, das die Angabe von *temporal scopes* erlaubt und über Mechanismen verfügt, die auch deren Überwachung gewährleisten.

Table of Contents

List of Programs

List of Figures

List of Tables

0 Preface and Preliminaries

> Every paper published in a respectable journal
> should have a preface by the author stating why he
> is publishing the article, and what value he sees in it.
> I have no hope that this practice will ever be adopted.
> -- Morris Kline

General Introduction

This paper extends the comparison of the concurrency concepts of Ada, CHILL, and Java given in (Brömel, 1998) in two ways: firstly, a new language, Erlang, is included and—secondly—this time, we focus on more practical material. Case studies will be used to carry out the comparison. Each chapter, except this one, is devoted to one case study in which we will pose a problem and propose solutions in the four languages. We will present the precise problem specification, the program design, and the program itself.

The case studies we look at in this paper are the following:

- Bounded Buffer
- Computing Prime Numbers
- The Dining Philosophers
- Process Scheduling
- Interrupt Handling

The following list defines the criteria upon which the comparison of the respective solutions will be based:

- Readability
- Program Size
- Liveness
- Absence of Deadlock
- Absence of Starvation
- Necessity of a Fair Machine
- Degree of Concurrency

Definitions

We give the following definitions of the criteria above (adopted from (Brömel, 1999)):

Readability. How easy (or difficult) is it to comprehend a program given in source code? This is, of course, a subjective issue and depends, to a great deal, on one's familiarity with a certain programming language. To be objective, we employ test persons and ask them to rank the readability on a scale ranging from 1–5. The interpretation is then as follows:

1 = very well readable
2 = well readable
3 = readable with minor difficulties
4 = readable with major difficulties
5 = not readable

Program Size. We count net lines of code—that is, comments and empty lines are excluded and there is one statement per line. We should bear in mind, however, that this technique is only a coarse method and suffers from many drawbacks. For example, the code fragment

```
if Reachable(Network_Socket) then
   Put_Line("Yes");
else
   Put_Line("No");
end if;
```

consists of five net lines of code. Yet what about the computation that goes on "behind the curtain," e.g., how is `Reachable` defined?

We thus see that net lines of code can only be used to gain an overview and should only be used in conjunction with other criteria.

Communication Effort measures the effort required to achieve actor synchronization by means of explicit blocking and resumption of actors. The effort is measured by counting the statements needed to synchronize.

Degree of Concurrency. Let $M = \{ a_1, a_2, \ldots, a_n \}$ be the set of actors in a program. Define the degree of concurrency to be the (average) number of actors in M that are runnable:

$$\text{Degree of Concurrency} = |\{ a_i \mid a_i \in M \wedge a_i \text{ is runnable} \}|$$

Speedup. Given p processors and an algorithm A with input size n, we define

$$S(p, n) = \frac{T(1, n)}{T(p, n)}$$

to be the speedup. $T(1, n)$ denotes the time required to perform A on a mono-processor system and $T(p, n)$ is the time required on a system consisting of p processors.

Cyclic Programs. Liveness, deadlock, starvation, and necessity of a fair machine are only pertinent to *cyclic* programs. We call a program P a cyclic program if certain actions, $da_1, da_2, \ldots, da_n$, are to be executed indefinitely often. For the sake of simplicity, we assume that an actor, a_i, contains *one* particular such action, da_i. We thus see that P is comprised of n actors, $a_1, a_2, \ldots, a_n$, and each a_i executes its desired action, da_i.

Since P contains actors that execute indefinitely often (more precisely, the desired action da_i of actor a_i of P is executed indefinitely often), we are only concerned with indefinite executions of P. We distinguish two flavors of indefinite executions:

- desired ones: all da_i occur indefinitely often
- undesired ones: at least one da_i occurs only a fixed number of times

Armed with this knowledge, we define:

Liveness. An execution, e, of P is *lively* with regard to an actor, a_i, iff da_i occurs indefinitely often in e. P is lively with regard to an actor, a_i, iff any execution of P is lively with regard to a_i. P is lively iff P is lively with regard to all actors in P.

Deadlock. Given a program, P, we distinguish three different levels of deadlock:

1. free of deadlock: there are only desired executions[1]
2. partial deadlock: both desired and undesired executions occur
3. total deadlock: only undesired executions occur

Starvation. An actor, a_i, *starves* in an execution, e, of P if this actor's desired action, da_i, only occurs a fixed number of times in e.

Necessity of a Fair Machine. A machine is called *fair* if it warrants that, for each execution, e, of P, no runnable actor, a_i, is missed out indefinitely often.

The following compilers were used:

Ada: GNU Ada Translator, v3.10p

CHILL: VISION O.N.E. CHILL Support Software, Toolset TL13U

Erlang: Erlang System/OTP R4B, v4.3.7

Java: Java Development Kit, v1.0.2

The presented programs can be obtained by contacting the author, see below. In the case of CHILL, however, we face the following problem: VISION O.N.E. is not fully CHILL 96-compliant—for example, task objects are not implemented. The CHILL programs presented in this paper are written in CHILL 96.

Audience

Readers are assumed to have a fundamental grasp of concurrency and its problems, and are expected to be basically familiar with the four languages, especially with the concepts they provide for concurrency. In the cases of Ada, CHILL, and Java, this knowledge can be acquired, for example, from (Brömel, 1998). A wonderful introduction to Erlang is given in (Armstrong, 1996).

As intimated by the title of this paper, there is a related paper which pursues a similar aim. The main difference, besides being written in German, is the set of case studies. Said paper is written by Peter Brömel, a colleague of mine, and referred to as (Brömel, 1999) throughout this paper.

Acknowledgements

I would like to express my gratitude to my supervisors—Prof. Dr. Jürgen Winkler and Dr. Stefan Kauer—who aroused my interest in this field; and who did the bulk of proof-reading, never getting tired of giving invaluable hints. I also wish to thank Michael Hartmeier for proof-reading and giving input for the discussions on readability (see the list above).

Being a non-native speaker of English, I have asked Elizabeth Ahrens-Kley to keep a sharp eye on my English. She did a wonderful job, and I am extremely grateful.

It goes without saying that I appreciate all the information given in the books I have used for writing this paper. On page 99, a bibliography can be found.

[1] We conclude that if P is free of deadlock, then P is lively. In fact, both notions are equivalent.

This paper was typeset in plain TEX—I would like to thank Donald Ervin Knuth for providing the world with this wonderful program.

The quotes at the title page of each chapter and the quote on page 34 are taken from `fortune`, a Unix program that prints out adages.

This paper can be obtained in a variety of formats (plain TEX, `DVI`, PostScript, PDF, and—of course—in print) by contacting the author: `franke@informatik.uni-jena.de`

Jena, July 13, 1999

1 Bounded Buffers

The world is coming to an end ... SAVE YOUR BUFFERS!!!

Introduction and Problem Description

In this chapter, we will see how we can implement a bounded buffer for use as a means of (indirect) communication between actors. This is a common technique in concurrent programming as it allows two (or more) actors that interchange data to perform better by being de-coupled. Fluctuations in the speeds at which the two actors are working are smoothed. The general approach is to employ a communication protocol by which one actor deposits a data item into the buffer and another actor retrieves an item from the buffer.

In a client-server environment, it is customary for a client that wishes to communicate with the server to pass data to a buffer from which the server can retrieve the data later. Usually, the number of clients is considerably larger than the number of servers and thus, there is the risk of a server being overloaded by too many simultaneous requests. Throwing away a request, in the case of high load, is not an option, of course, and so the server has to keep track of outstanding requests anyway. By de-coupling this buffering mechanism from the server, changes to the buffer (increasing its size, for example) can be made without "touching" the server.

Being bounded, the buffer must guard against being overfilled or under-emptied—that is, neither shall an actor be permitted to put a data item into an already full buffer, nor can an actor be allowed to take an item from a buffer that is empty (this would result in garbage being retrieved). Furthermore, the order of arrival of requests has to be preserved—hence, the buffer must operate in FIFO manner. It goes without saying that the buffer must ensure mutually exclusive access.

We will look at both active and passive buffers. An active buffer is an entity with activity—an actor in its own right, having a thread of control. A passive buffer does not possess a thread of control. It does not execute, it just provides the features described above. As regards the insurance against being overfilled and under-emptied, we will also examine two strategies: telling the client that the buffer is full (or empty), and letting the client simply wait until conditions are such that deposition or retrieval can be sensibly performed. Finally, if all actors are to be decommissioned, no more communication is required and it is time to shut down the buffer and the clients. This must be done in an orderly fashion.

The type of data to be stored in or retrieved from the buffer is immaterial here—we restrict ourselves, however, to homogeneous buffers. Though bounded, the buffer's size is also not relevant for our purposes (it is, of course, relevant in practical applications since the size chosen will undoubtedly affect the throughput).

1.1 Bounded Buffers in Ada

1.1.1 Active Buffers in Ada—Design

An active entity in Ada is a task; hence, we must implement an active bounded buffer as a task. The communication between a client and the buffer will be accomplished via the rendezvous—that is, a client calls a buffer operation (an entry of the task) and waits until

the buffer has completed (or rejected) the operation. It then carries on. To bound the buffer pool, we will use an array which is accessed by two indexes: one is for the next free slot and the other is for the last used slot. The buffer is used cyclically, so Ada's modular types are appropriate for the indexes. Since we abstract from the actual type of data, we will use generics. Attempts to store/retrieve a value in/from a buffer that is already full/empty will be handled thusly: we use exceptions to tell a client that the buffer is full or empty. For the wait solution in this case, let us employ guarded select statements to let the client simply wait until the conditions are appropriate (Ada's requeue facility could be used here instead, but it appears to be more expensive in terms of overhead). The former technique requires an intelligent client who must react on the exceptions being propagated to it whereas the wait solution is more straightforward. We present the exception solution in conjunction with active buffers and use the wait mechanism for passive buffers.

As a matter of convenience, we encapsulate the buffer in a (generic) package and make it a type.

1.1.2 Active Buffers in Ada—The Program

Following below is the source code of an active buffer in Ada. It can be found in the files `active_buffer.ads` and `active_buffer.adb`. The former contains the specification of the package and the latter holds the package body. Note that `Buffer_Pool_Size` cannot be made generic—generic constants cannot be specified, and the expression following the reserved word `mod` in a modular type definition is to be static:

```
generic
   type Data_Item is private;

package Active_Buffer is

   task type Buffer is
      entry Store(What : in Data_Item);
      entry Retrieve(What : out Data_Item);
   end Buffer;

   Buffer_Overflow, Buffer_Underflow : exception;

end Active_Buffer;


package body Active_Buffer is

   task body Buffer is
      Buffer_Pool_Size : constant Positive := 8;
      type Pool_Index is mod Buffer_Pool_Size;
      -- 0 .. Buffer_Pool_Size - 1, wraps around
      Buffer_Pool : array(Pool_Index) of Data_Item;
      Last_Used_Slot, Next_Free_Slot : Pool_Index := 0;
      Count : Natural range 0 .. Buffer_Pool_Size := 0;

   begin
      Administrative_Loop : loop
```

```
         Work_Block : begin
            Work_Loop : loop
               select
                  accept Store(What : in Data_Item) do
                     if Count >= Buffer_Pool_Size then
                        raise Buffer_Overflow; -- tell client
                     end if;
                     Buffer_Pool(Next_Free_Slot) := What;
                  end Store;

                  Next_Free_Slot := Next_Free_Slot + 1;
                  Count := Count + 1;
               or
                  accept Retrieve(What : out Data_Item) do
                     if Count = 0 then
                        raise Buffer_Underflow; -- tell client
                     end if;
                     What := Buffer_Pool(Last_Used_Slot);
                  end Retrieve;

                  Last_Used_Slot := Last_Used_Slot + 1;
                  Count := Count - 1;
               end select;
            end loop Work_Loop;

         exception
            when Buffer_Underflow | Buffer_Overflow =>
               null;  -- server forgets, but client is informed
            when others =>
               -- something else went wrong, reraise the exception
               raise;
         end Work_Block;
      end loop Administrative_Loop;
   end Buffer;
end Active_Buffer;
```

Program 1.1: Bounded Active Buffer in Ada

For the exception solution (in the case of overflow or underflow) to work correctly, the exception must be propagated into the client; hence, it cannot be handled locally in the `accept` statement. The server, however, does not care about these exceptions and so a `null` handler is required. `Administrative_Loop`, `Work_Block`, and `Work_Loop` are needed for the server to operate correctly. The latter is for the normal server operation and the former two allow the server to recover gracefully from overflow and underflow exceptions.

A sample program is shown below: two actors, each of which operates at a different speed, communicate via an active buffer—whereby the variation in the speeds is smoothed.

```
with Ada.Text_IO, Active_Buffer, Ada.Numerics.Discrete_Random,
  Ada.Numerics.Float_Random;
use Ada.Text_IO, Ada.Numerics.Float_Random;
```

```
procedure Buffer_Client is

   package Integer_Buffer is new Active_Buffer(Data_Item => Integer);
   use Integer_Buffer;
   package Integer_Random is new Ada.Numerics.Discrete_Random(Integer);
   use Integer_Random;

   G : Integer_Random.Generator;
   H : Ada.Numerics.Float_Random.Generator;

   My_Buffer : Buffer;

   task Producer;

   task body Producer is
   begin
      Reset(G); -- roll the dice
      loop
         begin
            loop -- a hotspur he is
               My_Buffer.Store(Integer_Random.Random(G));
               Put_Line("Successfully stored a value");
            end loop;

         exception
            when Buffer_Overflow =>
               Put_Line("Buffer is crammed!");
               delay Duration(Random(H));
         end;
      end loop;
   end Producer;

   task Consumer;

   task body Consumer is
      X : Integer;

   begin
      Reset(H);
      loop
         begin
            loop
              delay Duration(Random(H));
              My_Buffer.Retrieve(X);
              Put_Line("Found this in the buffer: " & Integer'Image(X));
            end loop;

         exception
            when Buffer_Underflow =>
               Put_Line("There is nothing to be harvested");
```

```
         end;
      end loop;
   end Consumer;

begin
   null;
end Buffer_Client;
```

Program 1.2: Sample buffer clients for the active buffer

1.1.3 Passive Buffers in Ada—Design

We use a protected object to represent a passive buffer. Communication here is via calls, by a client, to the protected operations of the protected object. The caller executes, under mutual exclusion, the operation. Note that the buffer cannot, in this case, be called a server since it does not execute anything (it has no thread of control after all). The internal representation of the buffer is equivalent to that given in Program 1.1.

As promised in Subsection 1.1.1, page 5, we will use the wait method to guard against the buffer being overfilled or under-emptied—that is, a client is simply forced to wait until conditions are such that it can execute the desired operation. Fortunately, we do not need to spend much developmental effort on this issue since protected entries and entry barriers provide exactly the required functionality. As a reminder, a protected entry that has been called can only be executed if the associated barrier evaluates to true—if not, the caller is placed into a wait queue. If the barrier eventually becomes true, the caller is removed from that queue and allowed to perform the operation.

As before with the active buffer, we create a generic package and make the buffer a type.

1.1.4 Passive Buffers in Ada—The Program

Please find below, and in `passive_buffer.ads` and `passive_buffer.adb`, the source code of the passive buffer.

```
generic
   type Data_Item is private;

package Passive_Buffer is

   protected type Buffer is
      entry Store(What : in Data_Item);
      entry Retrieve(What : out Data_Item);
   end Buffer;

end Passive_Buffer;


package body Passive_Buffer is
   -- we must define the following external to the buffer
```

```
   Buffer_Pool_Size : constant Positive := 8;
   type Pool_Index is mod Buffer_Pool_Size;
   -- 0 .. Buffer_Pool_Size - 1, wraps around
   Buffer_Pool : array(Pool_Index) of Data_Item;
   Last_Used_Slot, Next_Free_Slot : Pool_Index := 0;
   Count : Natural range 0 .. Buffer_Pool_Size := 0;

   protected body Buffer is
      entry Store(What : in Data_Item)
        when Count < Buffer_Pool_Size is
      begin
         Buffer_Pool(Next_Free_Slot) := What;
         Next_Free_Slot := Next_Free_Slot + 1;
         Count := Count + 1;
      end Store;

      entry Retrieve(What : out Data_Item)
        when Count > 0 is
      begin
         What := Buffer_Pool(Last_Used_Slot);
         Last_Used_Slot := Last_Used_Slot + 1;
         Count := Count - 1;
      end Retrieve;
   end Buffer;
end Passive_Buffer;
```

Program 1.3: Bounded Passive Buffer in Ada

A protected body cannot contain type declarations, so the buffer's internals have to be declared in the package body. Since the package is used solely to encapsulate the protected type, this should not be too serious an issue. Note, however, that in a task body, we do not face this irregularity.

In the case of overflow or underflow (look at the barriers associated with `Store` and `Retrieve`), the client is now placed into a wait queue where it remains until conditions are appropriate—there is no polling involved and the barriers are re-evaluated at well-defined points only. If there is only one client in our system, and if this client tries to continuously fill (or empty) the buffer, the waiting would even last up to Judgement Day.[2] The buffer client below pays heed to this by utilizing Ada's feature of asynchronous transfer of control. The buffer is continuously filled in an endless loop. If the buffer is full, `My_Buffer.Store` blocks. This blocking would last forever were it not for the ATC statement—which supervises the filling—with the delay trigger of two seconds. After two seconds, the buffer is assumed to be full and the loop is aborted—thereby allowing the client to begin emptying the buffer, which is supervised by the same timing mechanism. Incidentally, this example assumes that the buffer can be filled/emptied within two seconds.

Readers are encouraged to think of more useful examples.

```
with Ada.Text_IO, Passive_Buffer;
use Ada.Text_IO;
```

[2] which some computer scientists believe is to come on January 1, 2000

```
procedure Buffer_Client2 is
   package Integer_Buffer is new Passive_Buffer(Data_Item => Integer);
   use Integer_Buffer;

   My_Buffer : Buffer;
   X : Integer;

begin
   loop
      select
         delay 2.0;
      then abort
         loop
            My_Buffer.Store(123);
            Put_Line("Filling ...");
         end loop;
      end select;

      Put_Line("Full");

      select
         delay 2.0;
      then abort
         loop
            My_Buffer.Retrieve(X);
            Put_Line("Emptying ...");
         end loop;
      end select;

      Put_Line("Empty");

   end loop;
end Buffer_Client2;
```

Program 1.4: Sample buffer client for the passive buffer

This concludes the presentation of bounded buffers in Ada. We have shown an active buffer using exceptions to inform clients about its state, and a passive buffer that places clients in wait queues if it is full or empty. We could have done it the other way around which would result in less convoluted code for the active buffer (one loop would suffice then), but the author feels that protected objects and entries (and barriers, of course) form a perfect coalition. As an aside, note that we could have also used Ada's requeue facility for the passive buffer. But, as already mentioned, this might lead to overhead in terms of busy waiting—something that is prohibitive on a mono-processor and should be avoided whenever possible.

Furthermore, typical servers in Ada should be implemented as protected objects rather than as tasks—there are at least two reasons: a protected object has no thread of control associated with it; hence, no context switches are required to execute a protected operation. This is not trivial since practical experience with Ada 83 has shown that implementing servers as tasks (the only way in Ada 83) led to poor performance. To create low level components, such as semaphores and monitors, tasks had to be used as

well. The rendezvous, clearly a high level concept, introduced what is being referred to as *abstraction inversion* (Rat, 1995, II.9). Secondly, there is no problem with protected objects if their services are no longer necessary—they are passive, so we do not need to fret about them lurking around and wasting processor cycles. In the case of tasks, this situation is different as discussed in the next subsection.

1.1.5 Coordinated Shutdown of Clients and the Buffer

Clients may communicate for a while—via the buffer—but sometime, there might be no further need for communication. If all the clients have finished their work duties, then this will obviously be the case—they just complete. Being of no use anymore, we would like the buffer to terminate as well (but not earlier, of course!). Generally, we only need to ponder on termination of an active buffer—that is, a task—since terms like activation and termination do not apply to passive entities. There is another reason why the active buffer should terminate if it is no longer in use: in Ada, a unit cannot be left until all dependent tasks have terminated. There is good reason to require this: leaving a unit results in all locally declared data being disposed; in the case of a task, this is far too disruptive. But as servers are typically implemented as endless loops (looking at the active buffer, on page 6, we discover that this server has got two endless loops), we face the problem: if a unit declares such a buffer, then this unit can never be left! Thus, even without the endless loops in Program 1.2, this program can never terminate because the buffer does not terminate.

We could extend the task in Program 1.1 by adding a further entry, `Shut_Down`, whose `accept` statement simply `exits` `Administrative_Loop`:

```
Administrative_Loop : loop
   ...
   or
      accept Shut_Down;
      exit Administrative_Loop;
   end select;
   ...
end Administrative_Loop;
```

Note that the `exit` statement is not part of the `accept` statement. This solution has the drawback that `Shut_Down` must be called by some client. But how can we be sure that it is the right time to call it? We would like the task to terminate itself automatically if it is of no further use. The proper approach is to use the `terminate` alternative in the `select` statement:

```
Administrative_Loop : loop
   ...
   or
      terminate;
   end select;
   ...
end Administrative_Loop;
```

We might combine both approaches, though.

1.2 Bounded Buffers in CHILL

1.2.1 Active Buffers in CHILL—Design

At first glance, CHILL task objects appear to be well suited to implement an active buffer. Recall (or see (Brömel, 1998, Chapter 4)) that communication between a task object and a client is via calls to the *guarded procedures* of the task object (by the client). The task, having a thread of control, then executes the operation called. However, while the task does this, the caller is allowed to carry on—that is, calls to task components are asynchronous. The caller, on the other hand, needs to know whether its request could be handled or was rejected (this, once more, indicates that communication always requires a certain/minimal amount of synchronization). Thus, the asynchronous communication provided by CHILL tasks is inappropriate here.

In (Brömel, 1999, Chapter 2), a solution is given to this problem: imposing synchronization upon asynchronous communication. A third party object, a CHILL buffer (P), is used by the task T and the client C to store and retrieve, respectively, the result of the operation *and* to synchronize. Hence, T and C synchronize via P. But there is a price to pay: each time C makes a call to a procedure of T, this call must be followed by a call to P. By encapsulating T and P in a module and by providing appropriate access methods (which perform the two calls), this should not be a problem. The client calls the module's `Store`, which calls T's `Store` (asynchronously) and then the synchronization operation on P. Since, at the time the synchronization operation (a `RECEIVE`) is called on P, there might not yet be a message present, the client becomes delayed (until a message arrives), which, after all, is exactly the synchronization we require. P eventually contains a message produced by T indicating whether or not the operation was successful. Based upon the message found in P, the client is then informed. See Figure 1.1 for an illustration.

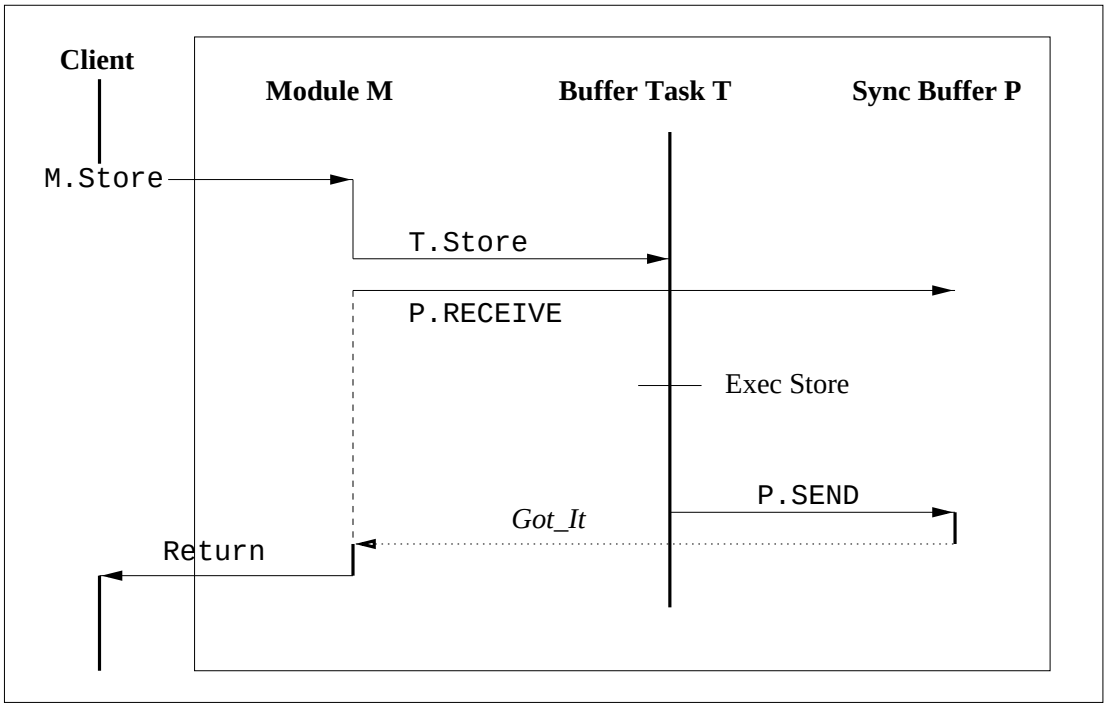

Figure 1.1: Imposing synchronization upon an asynchronous protocol

We take over the remaining design decisions from Subsection 1.1.1, except that we do not make the buffer a type; we provide the required functionality by utilizing CHILL's module features. A CHILL module can be a mode (that is, a type).

1.2.2 Active Buffers in CHILL—The Program

```
GENERIC
   MODE Data_Item = ANY_ASSIGN; /* as we need assignment */

SYNMODE Active_Buffer = MODULE SPEC
   GRANT Store, Retrieve;

   Store : PROC(What Data_Item IN) EXCEPTIONS(Buffer_Overflow) END Store;
   Retrieve : PROC(What Data_Item OUT)
     EXCEPTIONS(Buffer_Underflow) END Retrieve;

   SYNMODE Buffering = TASK SPEC
      GRANT Store, Retrieve;

      Store : PROC(What Data_Item IN) END Store;
      Retrieve : PROC(What Data_Item IN OUT) END Retrieve;

      SYN Buffer_Pool_Size = 8;
      NEWMODE Pool_Index = INT(0 : Buffer_Pool_Size - 1);
      DCL Buffer_Pool ARRAY(Pool_Index) Data_Item,
          Last_Used_Slot, Next_Free_Slot Pool_Index INIT := 0;
          Count RANGE(0 : Buffer_Pool_Size) INIT := 0;
   END Buffering;
END Active_Buffer;

SYNMODE Active_Buffer = MODULE BODY

   NEWMODE Message_Mode = STRUCT
      (Type SET (Buffer_Full, Buffer_Empty, Got_It, Have_It),
      CASE Type OF
         (Buffer_Full, Buffer_Empty) : ,
         (Got_It) : ,
         (Have_It) : What Data_Item
      ESAC),
      Sync_Buffer_Mode = BUFFER(1) Message_Mode;
   DCL Sync_Buffer Sync_Buffer_Mode;

   SYNMODE Buffering = TASK BODY

      Store : PROC(What Data_Item IN)
         IF Count >= Buffer_Pool_Size THEN
            SEND Sync_Buffer([Buffer_Full]);
         ELSE
            Buffer_Pool(Next_Free_Slot) := What;
```

```
            Next_Free_Slot := (Next_Free_Slot + 1) MOD Buffer_Pool_Size;
            Count := Count + 1;
            SEND Sync_Buffer([Got_It]);
         FI;
      END Store;

      Retrieve : PROC(What Data_Item IN OUT)
         IF Count = 0 THEN
            SEND Sync_Buffer([Buffer_Empty]);
         ELSE
            What := Buffer_Pool(Last_Used_Slot);
            Last_Used_Slot := (Last_Used_Slot + 1) MOD Buffer_Pool_Size;
            Count := Count - 1;
            SEND Sync_Buffer([Have_It, What]);
         FI;
      END Retrieve;
   END Buffering;

   DCL The_Buffer Buffering; /* is up now */

   Store : PROC(What Data_Item IN) EXCEPTIONS (Buffer_Overflow)
      The_Buffer.Store(What);
      RECEIVE(Sync_Buffer IN Message);
      CASE Message.Type OF
         (Got_It) : ;
         (Buffer_Full) : CAUSE Buffer_Overflow;
      ESAC;
   END Store;

   Retrieve : PROC(What Data_Item OUT) EXCEPTIONS(Buffer_Underflow)
      The_Buffer.Retrieve(What);
      RECEIVE(Sync_Buffer IN Message);
      CASE Message.Type OF
         (Have_It) : What := Message.What;
         (Buffer_Empty) : CAUSE Buffer_Underflow;
      ESAC;
   END Retrieve;
END Active_Buffer;
```

Program 1.5: Active Buffer in CHILL

Consider the following simple buffer client:

```
Buffer_Client : MODULE
   SEIZE Active_Buffer;
   My_Buffer : MODULE = NEW Active_Buffer
      SYNMODE Data_Item = INT;
   END My_Buffer;
   DCL X INT;

   My_Buffer.Store(1234);
```

```
   My_Buffer.Retrieve(X);

   ON
      (Buffer_Overflow, Buffer_Underflow) : /* do what is appropriate */
   END;
END Buffer_Client;
```

Program 1.6: A sample buffer client for the active buffer

1.2.3 Passive Buffers in CHILL—Design

We will use a region to create a passive buffer and two events for the waiting in the case of overflow or underflow. As in the case of protected objects in Ada, a client calls an operation and executes this operation under mutual exclusion. No synchronization buffer is required here since the client itself executes the operation. There are no operation barriers but events provide roughly the same functionality. There is, however, nothing comparable to Ada's "eggshell" model and delayed processes are re-activated in an implementation defined way (in Ada, the entry queues are FIFO-ordered, by default). Priorities cannot be used to impose FIFO ordering since the priority with which a process was delayed on an event[3] cannot be changed afterwards (which is what we were required to do, sooner or later).

1.2.4 Passive Buffers in CHILL—The Program

Incidentally, VISION O.N.E. does support regions, so the following program, which can be found in `passive_buffer.chill`, could be actually run after it has been adapted to VISION O.N.E.'s syntax (Program 1.7 is written in CHILL 96).

```
GENERIC
   MODE Data_Item = ANY_ASSIGN; /* as we need assignment */

SYNMODE Passive_Buffer = REGION SPEC
   GRANT Store, Retrieve;

   Store : PROC(What Data_Item IN) END Store;
   Retrieve : PROC(What Data_Item OUT) END Retrieve;

END Passive_Buffer;

SYNMODE Passive_Buffer = REGION BODY
   SYN Buffer_Pool_Size = 8;
   NEWMODE Pool_Index = INT(0 : Buffer_Pool_Size - 1);
   DCL Buffer_Pool ARRAY(Pool_Index) Data_Item,
       Last_Used_Slot, Next_Free_Slot Pool_Index INIT := 0;
       Count RANGE(0 : Buffer_Pool_Size) INIT := 0;
   DCL Buffer_Empty, Buffer_Full EVENT;
```

[3] this is *not* the priority used for scheduling and dispatching

```
   Store : PROC(What Data_Item IN)
      IF Count >= Buffer_Pool_Size THEN
         DELAY Buffer_Full;
      ELSE
         Buffer_Pool(Next_Free_Slot) := What;
         Next_Free_Slot := (Next_Free_Slot + 1) MOD Buffer_Pool_Size;
         Count := Count + 1;
         CONTINUE Buffer_Empty;
      FI;
   END Store;

   Retrieve : PROC(What Data_Item OUT)
      IF Count = 0 THEN
         DELAY Buffer_Empty;
      ELSE
         What := Buffer_Pool(Last_Used_Slot);
         Last_Used_Slot := (Last_Used_Slot + 1) MOD Buffer_Pool_Size;
         Count := Count - 1;
         CONTINUE Buffer_Full;
      FI;
   END Retrieve;
END Passive_Buffer;
```

Program 1.7: Passive Buffer in CHILL

And the buffer client is then similar to that presented in Program 1.6:

```
Buffer_Client2 : MODULE
   SEIZE Passive_Buffer;
   My_Buffer : REGION = NEW Passive_Buffer
      SYNMODE Data_Item = INT;
   END My_Buffer;
   DCL X INT;

   /* initiate a time supervision
      to bound the waiting */

   AFTER SECS(5) IN
      My_Buffer.Store(1234);
   TIMEOUT
      /* buffer too busy, do something else instead */
   END;

   My_Buffer.Retrieve(X);

END Buffer_Client;
```

Program 1.8: A buffer client for the passive buffer

We have presented an active buffer which uses exceptions to signal bad conditions and a passive buffer that makes use of CHILL's events to delay a client. In the case of the active buffer, we had—due to the asynchronous nature of a call to a task component—to use a CHILL buffer as a means of synchronization. The passive buffer, implemented with a region, is more straightforward, giving further evidence for the claim that buffers should be implemented as passive entities. A client has to wait for the completion of the operation anyway, so why not use the client's thread of control then?

1.2.5 Coordinated Shutdown of Clients and the Buffer

Again, the passive buffer does not cause trouble, but neither does the active one. Why? As pointed out in (Brömel, 1998, Chapter 3), a task object, once started, waits until one of its guarded procedures is called, it then executes this procedure and becomes quiescent again. Then, only a quiescent task mode location may be destroyed. To destroy a task mode location, its defining block must be left (or the `TERMINATE` function must be applied, in the case of a dynamically created task). Leaving a block and the effect of `TERMINATE` are deferred until the task is no longer busy. The rules can be found in (Brömel, 1998, 2.2.2.2) and will not be repeated here. Thus, a task gracefully dies if it is no longer in use and its scope is left.

1.3 Bounded Buffers in Erlang

1.3.1 Active Buffers in Erlang—Design

Looking at Chapter 5 of (Armstrong, 1996), we find that an active entity in Erlang is represented by a *process* and that communication is carried out via *message passing*. This message passing is, like calls to a task object's procedures in CHILL, of asynchronous nature. We thus face the same problem of imposing synchronization upon the asynchronous protocol.

Luckily, we can fall back on the solution pointed out in (Brömel, 1999, Chapter 2)—we use two asynchronous send operations to accomplish one synchronous one. Of course, the client does not need to worry about this as we shall provide for appropriate interface functions to the server. Further details (which are also pertinent to Subsection 1.2.1) can be found in (Brömel, 1999).

Erlang provides *tuples* to store a *fixed* number of Erlang objects;[4] we shall make use of tuples. For the exception solution (to the problem of overflow and underflow) to work, we cannot `throw` an exception in the server since it is impossible to transfer this exception occurrence from the server's thread of control to the client's (where it ultimately belongs). We use a similar approach to that in Program 1.5—passing a message from the server to the client (who has called one of the interface functions) and, thereby, informing the client of the status quo. The exception, if any, gets `thrown` in the interface function concerned (based upon evaluating the message received from the server). The client then, upon return from the interface function call, `catch`es "whatever comes out" of this interface function. Granted, this is a trifle imprecise, but since a function may return any Erlang

[4] An Erlang object is any Erlang term—genericity is implicit in Erlang (moreover, we can create heterogenous buffers without much ado).

term, we cannot be more definite. On the other hand, Erlang surely possesses the means required to deal with this matter.

It has been mentioned already, on page 3, that this paper can only sporadically present Erlang's features—it definitely cannot serve as an introduction to the language. Questions arising from the material presented above had best be answered by consulting (Armstrong, 1996).

1.3.2 Active Buffers in Erlang—The Program

We now present the program that implements an active bounded buffer in Erlang. It can be found in `active_buffer.erl`.

```
-module(active_buffer).
-export([start/1, store/1, retrieve/0, server/4]).

range(N, N) -> [N];
range(Min, Max) when Min < Max ->
   [Min | range(Min + 1, Max)];
range(Max, Min) when Max > Min ->
   range(Min, Max).

%% the interface functions

start(Buffer_Pool_Size) ->
   register(buffer_server,
     spawn(active_buffer, server,
       [list_to_tuple(range(0, Buffer_Pool_Size - 1)), 0, 0, 0])).

store(What) ->
   request({store, What}).

retrieve() ->
   request(retrieve).

%% the protocol

request(Request) ->
   buffer_server ! {self(), Request},
   receive
      {buffer_server, Reply} ->
         case Reply of
            buffer_full ->
               throw(buffer_overflow);
            store_ok ->
               store_ok;
            buffer_empty ->
               throw(buffer_underflow);
            {retrieve_ok, What} ->
               What;
            Other ->
```

```
                Other
          end
   end.

%% the server

server(Buffer_Pool, Last_Used_Slot, Next_Free_Slot, Count) ->
   receive
      {From, {store, What}} ->
         if
            Count >= size(Buffer_Pool) ->
               From ! {buffer_server, buffer_full},
               server(Buffer_Pool, Last_Used_Slot,
                 Next_Free_Slot, Count);
            Count < size(Buffer_Pool) ->
               From ! {buffer_server, store_ok},
               server(setelement(Next_Free_Slot + 1, Buffer_Pool, What),
                 Last_Used_Slot,
                 (Next_Free_Slot + 1) rem size(Buffer_Pool), Count + 1)
         end;
      {From, retrieve} ->
         if
            Count == 0 ->
               From ! {buffer_server, buffer_empty},
               server(Buffer_Pool, Last_Used_Slot,
                 Next_Free_Slot, Count);
            Count /= 0 ->
               From ! {buffer_server, {retrieve_ok,
                 element(Last_Used_Slot + 1, Buffer_Pool)}},
               server(Buffer_Pool,
                 (Last_Used_Slot + 1) rem size(Buffer_Pool),
                 Next_Free_Slot, Count - 1)
      end
   end.
```

Program 1.9: Active Buffer in Erlang

Just for fun, we do not provide a buffer client as a self-contained program—let us play a bit with the Erlang shell instead:

```
(lpc01) $ erl
Erlang (JAM) emulator version 4.7.3

Eshell V4.7.3  (abort with ^G)
1> c(active_buffer).
{ok,active_buffer}
2> catch active_buffer:start(2).
true
3> catch active_buffer:retrieve().
buffer_underflow
4> catch active_buffer:store(1).
```

```
store_ok
5> catch active_buffer:store('Mr Jones').
store_ok
6> catch active_buffer:store(2).
buffer_overflow
7> catch active_buffer:retrieve().
1
8> catch active_buffer:store(2).
store_ok
9> catch active_buffer:retrieve().
'Mr Jones'
10> catch active_buffer:retrieve().
2
11> catch active_buffer:retrieve().
buffer_underflow
12> halt().
(lpc01) $
```

1.3.3 Passive Buffers in Erlang?

No, since Erlang does not provide passive entities that guarantee mutually exclusive access to their components (which is what we need, of course). A method `exported` from a `module` can be called by multiple processes, which plays havoc with the internal data structures. Only processes, having exactly one thread of control, can give that guarantee. But processes are active.

At least, we can demonstrate the wait solution to buffer overflow/underflow. The full syntax of the `receive` statement reads thuswise:

```
receive
   Message_1 [when Guard_1] ->
      Action_1;
   Message_2 [when Guard_2] ->
      Action_2;
   ...
after Timeout_Expression ->
   Action_T;
end
```

We see that the individual branches can be guarded and the semantics is such that an incoming message matches branch i iff it matches `Message_I` *and* `Guard_I` is open (that is, true). If an incoming message does not match any of the branches, it remains in the mailbox of the process executing the `receive` statement—with appropriate interface functions (that achieve synchronous access to the server), a client can be forced to wait. This is because a process executing a `receive` is suspended until a message matches.[5] The client, according to the protocol defined in Program 1.9, waits for a message from the server and is hence also blocked.

[5] We thus take it that the matching is performed by the runtime system, not the process in question.

We also understand that a timeout (the `Timeout_Expression` is expected to evaluate to an integer expression which is interpreted as a time given in milliseconds) can be imposed upon the receipt of messages, an issue that is not of paramount interest here; it has been mentioned for the sake of completeness only.

Thusly enlightened, we can give a sketch of the server loop (Program 1.9) making use of guards:

```
server(...) ->
   receive
      {From, {store, What}} when Count < size(Buffer_Pool) ->
         From ! {buffer_server, store_ok},
         server(...)  %% as before
      {From, retrieve} when Count > 0 ->
         From ! {buffer_server, ...},  %% as before
         server(...)  %% as before
   end.
```

As the clients wait for the various `From ! {buffer_server, ...}` messages, they must wait until the server can send them, i.e., until the respective guards are true.

1.3.4 Coordinated Shutdown of Clients and the Buffer

Unlike in Ada or CHILL, there is no inherent hierarchy between processes in Erlang. (Armstrong, 1996, Chapter 7) tells us: "A process is a self-contained, separate unit of computation which exists concurrently with other processes in the system." The designer of an application may explicitly create a hierarchy, however. Therefore, a process is not bound to a particular scope and the question as to whether it is safe to exit a scope if some processes are still running is not relevant here.

A process terminates (normally) if it has finished the evaluation of the function given to it as an argument during creation and start. A server function (or the evaluation of which), ideally, is not meant to finish, though. However, it might be appropriate for the server to be shut down sometime. Erlang lacks an equivalent of Ada's `terminate` alternative in a `select` statement and so we must explicitly ask the server to become quiescent. A new branch in the `receive` statement of Program 1.9 and a new interface function will do:

```
%% an additional interface function (must be exported as well!)

shut_down() ->
   request(shut_down).

%% modified server

server(Buffer_Pool, Last_Used_Slot, Next_Free_Slot, Count) ->
   receive
      {From, shut_down} ->
         %% this message is, in request, of class ``Other''
         From ! {buffer_server, shut_down_ok},
         unregister(buffer_server);
```

```
      %% do not loop around

   %% as before, handling store and retrieve
end.
```

1.4 Bounded Buffers in Java

1.4.1 Active Buffers in Java—Design

Active entities in Java are of class `Thread`* and sensibly re-implement the `run()` method. But as explained in (Brömel, 1998, Chapter 3), Java threads are not suitable to implement a server. It is not possible for a client to send a request to a Java thread and to expect that this request be handled by the thread using Java's concurrency primitives. Of course, we could mimic a server by equipping a thread with a queue to which requests are sent and let `run()` examine this queue, extract requests, and handle them. But this has to be directly programmed. Although not complicated,[6] this is beyond the scope of this paper—we examine *existing* features for concurrency.

As a compensation, we show both the exception solution and the wait solution in connection with passive buffers in the next subsubsection.

1.4.2 Passive Buffers in Java—Design

We begin with the wait solution for overflow/underflow, using `wait/notifyAll` for suspension and resumption of a client and a sequential class with `synchronized` access methods to the buffer.

1.4.3 Passive Buffers in Java—The Program

The file `Passive_Buffer.java` contains the program for the passive buffer in Java.

```
class Passive_Buffer {

   public Passive_Buffer() {
      this(1);
   }

   public Passive_Buffer(int Buffer_Pool_Size) {
      this.Buffer_Pool_Size = Buffer_Pool_Size;
      Buffer_Pool = new Object[this.Buffer_Pool_Size];
      Last_Used_Slot = 0;
      Next_Free_Slot = 0;
      Count = 0;
   }
```

[6] Michael Hartmeier pointed out that the classes `EventObject` and `EventListener` are perhaps a good starting point.

```
   private static int Buffer_Pool_Size;
   private Object[] Buffer_Pool;  // genericity and heterogeneity
   private int Last_Used_Slot, Next_Free_Slot, Count;

   public synchronized void Store(Object What) {
      while (Count == Buffer_Pool_Size) {
         try {
            wait();
         }
         catch (InterruptedException E) {}
      }
      notifyAll();
      Buffer_Pool[Next_Free_Slot] = What;
      Next_Free_Slot = (Next_Free_Slot + 1) % Buffer_Pool_Size;
      Count = Count + 1;
   }

   public synchronized Object Retrieve() {
      while (Count == 0) {
         try {
            wait();
         }
         catch (InterruptedException E) {}
      }
      notifyAll();
      Object What = Buffer_Pool[Last_Used_Slot];
      Buffer_Pool[Last_Used_Slot] = null;
      Last_Used_Slot = (Last_Used_Slot + 1) % Buffer_Pool_Size;
      Count = Count - 1;
      return What;
   }
}
```

Program 1.10: Passive Buffer in Java

A buffer client is left to the imagination of the reader. As promised in Subsection 1.4.1, we now present the exception solution. Only a few modifications to Program 1.10 are necessary.

We need two kinds of exceptions,

```
   class Buffer_Overflow extends Exception {}
   class Buffer_Underflow extends Exception {}
```

which should appear in the method header of `Store` and `Retrieve` and, of course, in the respective bodies (note that there is no more need for `wait` and `notifyAll`),

```
public synchronized void Store(Object What) throws Buffer_Overflow {
  if (Count == Buffer_Pool_Size) {
     throw new Buffer_Overflow();
  }
```

```
  Buffer_Pool[Next_Free_Slot] = What;
  Next_Free_Slot = (Next_Free_Slot + 1) % Buffer_Pool_Size;
  Count = Count + 1;
}
```

`Retrieve` is to be modified in a similar way and the client is expected to `catch` the exceptions.

1.4.4 Coordinated Shutdown of Clients and the Buffer

Having only a passive buffer, the question is not relevant.

Even if we simulated an active buffer with the technique proposed in Subsection 1.4.1, there are no woes. Like in Erlang, there is no hierarchy between threads in Java. Leaving a scope only disposes the reference to a thread defined in that scope, not the thread object itself. Furthermore, the class `Thread` is equipped with methods to stop a thread (or we might send a special request to the server, asking it to become tranquil). Of course, we must know whether it is the right time to shut the server down; both in Erlang and Java.

1.5 Summary and Comparison

In this chapter, we have looked at bounded buffers, which are building blocks in many concurrent systems as they help smooth the variations in the speed at which communicating actors are operating. The emphasis was upon active and passive buffers which should react appropriately in case of overflow/underflow—namely, by means of exceptions, or by forcing the client to wait. In Ada and CHILL, the problems could be solved without difficulties as these languages provide all the means necessary. Erlang has no features that help develop passive buffers, and Java is not suitable for active buffers. All four languages support both the exception and the wait solution; and Ada, CHILL, and Erlang provide means to deal with the shutdown of active buffers. As the main objective of buffers is communication, which always requires synchronization, the asynchronous communication protocols of CHILL tasks and Erlang processes had to be circumvented. This did not prove difficult.

It is the author's belief that buffers should be implemented as passive entities. Since synchronization is required, the client is forced to wait anyway, so why not use the client's thread of control to perform the buffering?

In keeping with the requirements of this paper, we will now examine the points made in the list on page 1.

Readability

The code for the active buffer in Ada is, due to the exception irregularity, a trifle convoluted, and CHILL's active buffer as well as Erlang's buffer must deal with the asynchronous protocol, which introduces slight overhead in coding. The author feels that the Erlang code is easy to read (take the five-liner that represents a range of numbers, for example) although the exception solution appears to be distracting from the gist—using the wait solution is more straightforward (in all four languages).

The marks that were given to the readability can be found in Table 1.1 below.

Program Size

See Table 1.1.

Communication Effort

This is only relevant for CHILL tasks and Erlang processes. In each case, we have used, respectively, a pair of `SEND/RECEIVE`, and `!/receive`. We thus deduce that the communication effort is 2.

Degree of Concurrency

The buffers were designed so that, at any time, at most one client can store/retrieve a value. Strictly speaking, this was not a design issue in that we would have had the choice; it was rather predestined since we can equip a buffer with at most one thread of control (an active buffer has got exactly one, and a passive one has got none at all).

We thus have a degree of concurrency of 1.

Liveness

The buffers themselves are not really cyclic programs (admittedly, the active buffers in Ada and Erlang are, to a certain extent). They "execute" only when requested by a client; we thus examine liveness, deadlock, starvation, and necessity of a fair machine more in the light of the clients calling the buffers' procedures. We also assume a normal execution environment, that is, a system with clients that store and clients that retrieve.

As long as the buffer is neither full nor empty, no complications can arise: consumers are allowed to retrieve and producers can store values as they like. Should the buffer, for example, become full, i.e., the rate at which store requests arrive surpasses the rate at which they are "serviced" (retrieved from the buffer by consumers), then the producers are prevented from further accessing the buffer, which in effect throttles the store rate. Consumers, as they do not have to compete with producers for access to the buffer now, find perfect conditions for retrieving values. They do that and so the buffer reverts to its normal state, which allows the store rate to be increased again so that producers are free to execute their desired actions again.

We conclude that liveness is guaranteed.

Deadlock

Stating that a system is lively is just a paraphrase of saying that this system is free of deadlock.

Starvation

No actor can starve. Assuming the contrary, let a_i be an actor whose desired action, da_i, occurs only a fixed number of times in a given execution, e. Then e is not lively with regard to a_i—the desired contradiction.

Necessity of a Fair Machine

As long as the buffer is neither full nor empty, fairness is not an issue since all clients are granted access to the buffer.

Let us look at the problem case then and examine each language in turn.

Ada: Fairness is always guaranteed for active buffers; it is guaranteed for passive buffers once clients of the passive buffer have penetrated the outer layer of the protected object, i.e., when they have acquired the lock of the protected object and are placed into entry queues (which causes them to release the lock). This is because requests are by default stored in FIFO order in the entry queues of, respectively, the task and the protected object (this happens irrespective of whether the buffer is full/empty or not). However, a subtle difference does exist between tasks and protected objects: a client of a task can call an entry without acquiring the lock of the task (there is none!) whereas to call an entry of a protected object, the caller must obtain the lock beforehand. The wait set for this lock is unordered—thus, a client might fail repeatedly in its attempt to obtain said lock. It is for this reason that we need a fair machine for passive buffers.

CHILL: Although it is possible to attach a priority to a call to a task component which will be used to determine which request to handle at a time, it is difficult to utilize this feature to impose an ordering on the client set. For one thing, CHILL does not define the set of priorities available and the selection mechanism. If there are more clients than priority values, then some of the calls have to be issued with the same priority, which nullifies the ordering. Furthermore, this technique requires each client to acquire a priority beforehand; hence, we must have a sort of priority dispatcher that hands out priorities and manages them.

For regions in CHILL, an ordering of the client set does not exist. Whenever there is competition, the selection of a client is carried out in an implementation defined way. A fair machine is generally necessary.

Erlang: An Erlang process is equipped with a mailbox in which all messages sent to it are stored in FIFO order; furthermore, messages found in that mailbox are received in the order they were sent (Armstrong, 1996, p. 69). We do not require a fair machine here.

Java: To execute a `synchronized` method in Java, the lock associated with the object containing the method must be obtained. The wait set is again unordered, which means that a fair machine is needed. Once a client has snatched the lock, it may find the buffer full (or empty) in which case it either waits or raises an exception. No matter what it does, the lock is released, and the client must compete again next time—there are no method queues. This is to be contrasted to the case of protected objects in Ada: once inside one of the protected object's entry queues, a caller does not need to reapply for the lock, and the queues are ordered.

For convenience, we present a tabular summary of this chapter in Table 1.1.

Language	Ada	CHILL	Erlang	Java
Active Buffer	Yes	Yes	Yes	No
Passive Buffer	Yes	Yes	No	Yes
Readability	2	3^+	3	3^+
Program Size: active/passive	51/36	68/35	61/n.a.	n.a./37
Communication Effort	n.a.	2	2	n.a.
Degree of Concurrency	trivially 1			
Fair Machine: active/passive	No/Yes	Yes/Yes	No/n.a.	n.a./Yes

Table 1.1: Tabular Summary of Chapter 1

2 Computing Prime Numbers

> Once upon a time, when I was training to be a mathematician, a group of us bright young students taking number theory discovered the names of the smaller prime numbers.
>
> 2: The Odd Prime—It's the only even prime, therefore it's odd. QED. 3: The True Prime—Lewis Carroll: "If I tell you three times, it's true." 31: The Arbitrary Prime—Determined by unanimous unvote. We needed an arbitrary prime in case the prof asked for one, and so had an election. 91 received the most votes (well, it *looks* prime) and 3 + 4i the next most. However, 31 was the only candidate to receive none at all.
>
> Since the composite numbers are formed from primes, their qualities are derived from those primes. So, for instance, the number 6 is "odd but true," while the powers of 2 are all extremely odd numbers.

Introduction and Problem Description

Ever since ancient times, humans have been fascinated by prime numbers. This is evidenced by the fact that they are called *prime numbers* and that so many scientists have tried their hand at them. Number theory is a branch of mathematics that has contributed significantly to the comprehension of the nature of numbers, prime numbers in particular. We know, for example, that the set of prime numbers, hereinafter denoted by P, is enumerable but not finite. Every prime number is a natural number but not vice versa, $P \subset \mathbb{N}$. 2 is the only prime number that is even. It is unknown whether or not the set of twin prime numbers is finite. Two prime numbers, p_1, $p_2 \in P$, are twins iff either $p_1 = p_2 + 2$ or $p_2 = p_1 + 2$. And, of course, P is given by the following definition

$$P = \{\, p : p \in \mathbb{N},\, p > 1 \text{ so that only } 1 \mid p \text{ and } p \mid p \,\} \tag{2.1}$$

Nowadays, prime numbers are, beyond their pure mathematical beauty, of interest in many applications. Hash tables function better if their size is prime (and not in the neighborhood of a power of two). Encryption systems such as RSA use prime number keys to achieve security. They rely on the fact that it is fiendishly time-consuming to test whether a large number is prime or not. Naturally, we are interested in algorithms that determine whether a given natural number is prime (not that the incentive were to crack the RSA algorithm). For $n \in \mathbb{N}$, the brute force method is to test whether

$$n \bmod i \neq 0, \qquad \text{for } i = 2, 3, \ldots, n-1 \tag{2.2}$$

This algorithm runs in $O(n)$,[7] but a little thought given to the matter reveals that it suffices to test for divisibility up to $i = \lfloor \sqrt{n} \rfloor$. Naturally, we are also interested in fast algorithms to compute prime numbers. This endeavour is not new—the Greek philosopher Eratosthenes (276–195 B.C.) has already devised an algorithm to compute all prime numbers up to a given number $n \in \mathbb{N}$ and whose complexity is better than $O(n\sqrt{n})$:

[7] This assumes that all arithmetic operations can be carried out in $O(1)$. This is a theoretical bound, and since we have mentioned RSA a little earlier, it is worth pointing out that arithmetic operations involving large numbers take more than one step to execute (yet another reason why the RSA algorithm is so hard to crack).

1. Write down all numbers from 1 to n. Cross out the number 1.

2. Let i be the smallest number that is not marked. Mark i and cross out all multiples of i.

3. Repeat step 2 until $i^2 > n$.

4. Look at the numbers that are marked but not crossed out—those are the prime numbers from 1 to n.

This is known as the Sieve of Eratosthenes, which runs in $O(n \log n)$.

We observe "that if we have a list of all the primes below n so far, then n is also prime if none of these divide exactly into it. So we can try the existing primes in turn and as soon as one divides n, we discard it and try again with $n + 1$. On the other hand, if we get to the end of our list of primes without dividing n, then n must be also prime, so we add it to our list and also start again with $n+1$." (Barnes, 1996, p. 440) We can speed up the computation by utilizing the principle of *pipelining*. Why wait before testing the next number until the candidate number being tested either gets discarded somewhere in the list (it gets sifted out) or is identified as a new prime number? If the current candidate number, i, has passed the kth ($k \geq 1$) position in the list (it has either been discarded or passed on to position $k + 1$), the next candidate can be tested right away at position k.[8] Thus, if at position k, $i \bmod k \neq 0$, then i can be sent to position $k + 1$ and the next candidate can be examined at position k. Since most primes are odd, it suffices to test odd numbers.[9]

This chapter will look at how to implement the pipelined Sieve of Eratosthenes in our four languages. The pipeline comprises of actors, each of which is linked to its successor (except for the last). Actor j in the list represents the jth prime number, $p_j \in P$, and tests an incoming candidate number for divisibility by j. If the candidate, i, passes the test, it is discarded on the grounds that it cannot be prime (Figure 2.1); if the candidate fails, it is passed on to actor $j + 1$, representing p_{j+1}, for further examination (Figure 2.2). Note that $j < i$. If i is not dropped by the last actor (representing the largest prime number so far, p_n), it must be the newly found prime, p_{n+1}. A new actor is created, representing p_{n+1}. This actor is linked to that representing p_n, and so forth (Figure 2.3). In short, each actor acts as a filter removing multiples of its own prime value.

In Figure 2.4, the situation is shown when the primes 2, 3, and 5 have been found. The 5 actor is examining 7 (which has failed the test for divisibility in the 2 actor and the 3 actor, and will prove to be a new prime), the 3 actor is currently quiescent and awaiting a new candidate from the 2 actor. The 2 actor (having just discarded 8) is testing 9 (which it will pass to the 3 actor presently, where it will—of course—be discarded).

The ideas presented above are not due to the author. A multiprocessing view of the Sieve of Eratosthenes was given by Shapiro and later Bokhari in, respectively, (Shapiro, 1995) and (Bokhari, 1997) to assess load balancing on shared-memory multiprocessors. The primary design decisions for the pipelined version were taken from (Barnes, 1996, p. 440) and will be used throughout this chapter.

[8] Note that if $k > 1$, then the next candidate is not $i + 1$.

[9] Thus, if i is the current candidate at position k, the next candidate at this position is never $i + 1$. Whenever we talk about candidates, $i + 1$ shall denote the candidate following i.

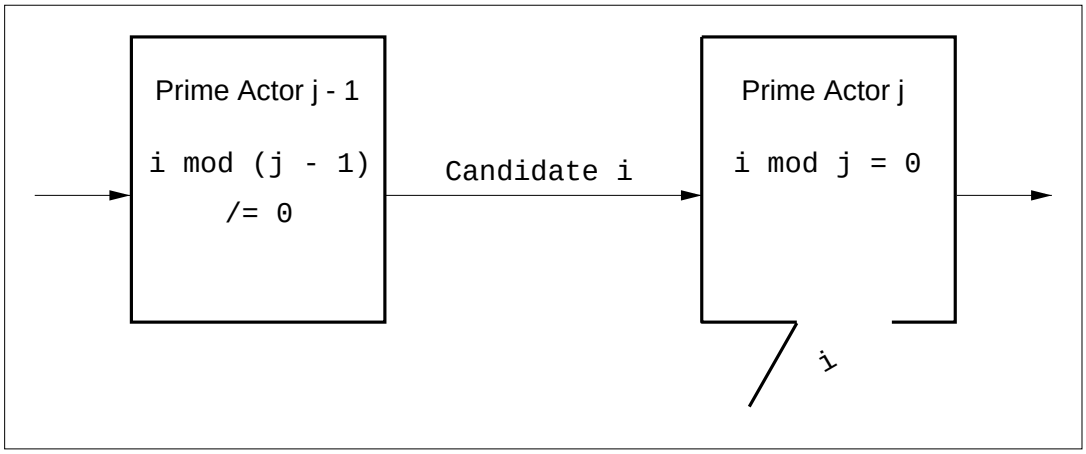

Figure 2.1: A candidate is being discarded

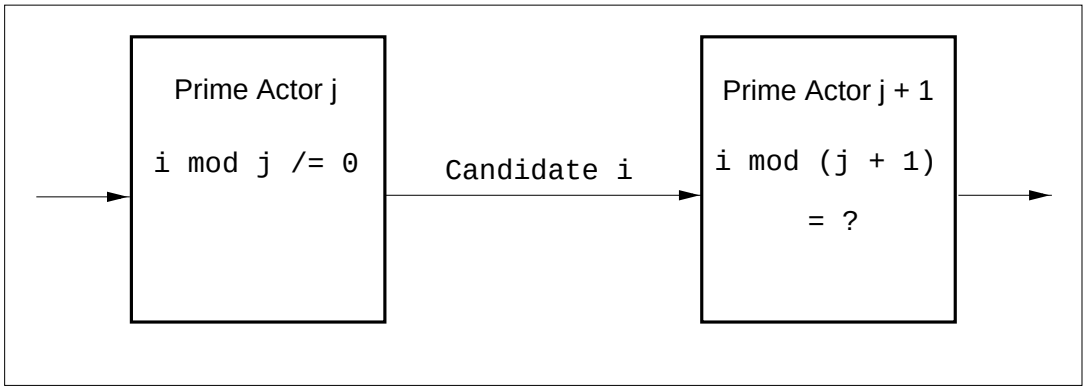

Figure 2.2: A candidate that looks prime to prime actor j is passed on

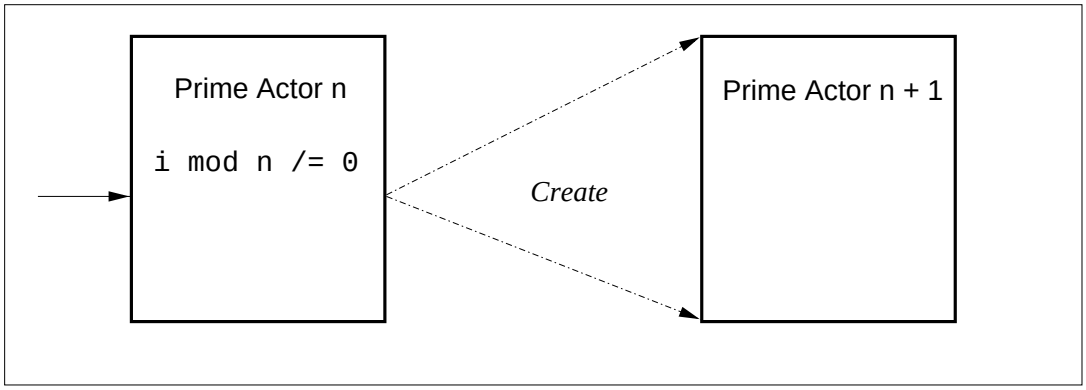

Figure 2.3: Eureka, a new prime number has been found!

2.1 The Pipelined Sieve of Eratosthenes in Ada

2.1.1 Design

We will follow the design decisions given above. The pipeline will be represented by a list of dynamically created tasks, and communication, that is, passing on candidate numbers, is via the rendezvous. Note that the rendezvous is a synchronous form of

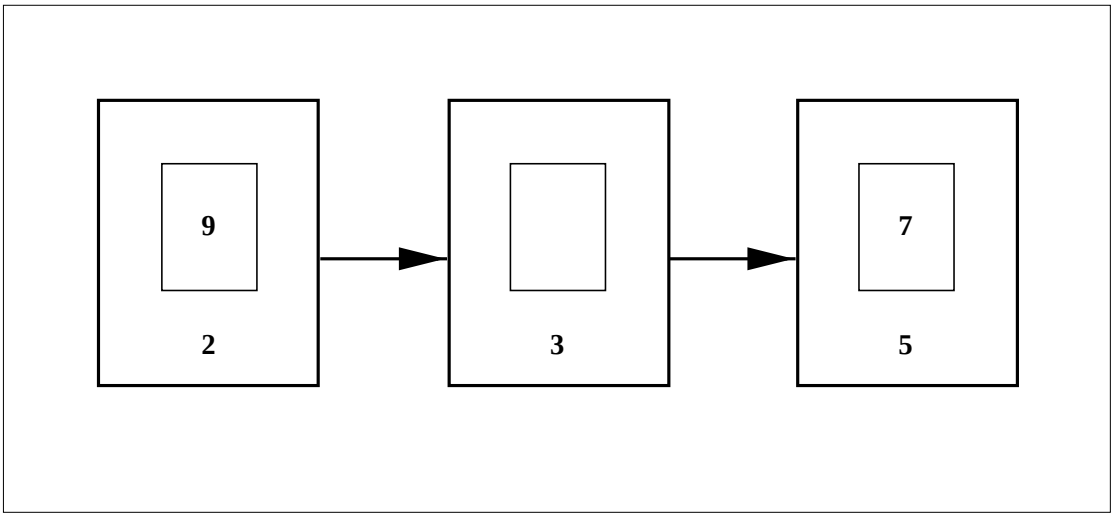

Figure 2.4: Pipelined Sieve of Eratosthenes

communication while we have said that passing on the numbers asynchronously is, from the perspective of speeding up the computation, of benefit here. But as Ada does not support asynchronous communication at all, we cannot meet this requirement. It is open to debate as to whether this is a real requirement. Put bluntly, it does not hurt if the candidate numbers are passed on synchronously. The speed and throughput of the computation might be affected, however, so we shall make the rendezvous as short as possible. Note that the flow through the pipeline is excellent, there are no tailbacks whatsoever. We will stick to the synchronous rendezvous here since we cannot do anything about it. In Subsection 2.2.1, we will focus our attention again upon this topic and learn that passing on the candidates asynchronously is not really an option. Note that Subsection 2.2.1 will also justify the aforementioned claim regarding the tailbacks.

If the issue of speed and throughput is crucial, then we can place a buffer between two prime tasks that helps smooth the variations in the speed and thereby accounts for more decoupling.

The main program will produce a stream of odd natural numbers which are fed into the pipeline, initially consisting of the task T_{p_1} representing $p_1 = 2$ only. After 3 has been fed into the pipeline, T_{p_1} finds that 3 is prime and creates a new task, T_{p_2}, and links itself to it. T_{p_1} then receives 5 as a candidate to test and finds that 5 looks prime and passes it on to T_{p_2}, which confirms that observation by creating T_{p_3}, representing $p_3 = 5$. And so forth.

Computers are finite machines—P is not finite. We let the user provide an upper bound for the computation.

2.1.2 The Sieve Program in Ada

The code below represents an implementation of the pipelined Sieve of Eratosthenes in Ada. It can be found in `sieve.adb`.

```
with Ada.Text_IO, Ada.Command_Line;
use Ada.Text_IO, Ada.Command_Line;

procedure Sieve is
```

```
   task type Prime_Task(Who_Am_I : Positive) is
      entry Pass_On(Candidate : in Positive);
   end Prime_Task;

   type Prime_Task_Ptr is access Prime_Task;

   function New_Pipeline_Stage(Who_Am_I : Positive)
     return Prime_Task_Ptr is
   begin
      return new Prime_Task(Who_Am_I);
   end New_Pipeline_Stage;

   task body Prime_Task is
      Examinee : Positive;
      Next_Stage : Prime_Task_Ptr;

   begin
      if Who_Am_I = 2 then
         Put_Line("It is a widely held belief that 2 is prime.");
      end if;
      loop
         select
            accept Pass_On(Candidate : in Positive) do
               Examinee := Candidate;
            end Pass_On;

            if Examinee mod Who_Am_I /= 0 then -- it *looks* prime
               if Next_Stage = null then -- it *is* prime
                  Put_Line("It has been confirmed that" &
                    Positive'Image(Examinee) & " is prime.");
                  Next_Stage := New_Pipeline_Stage(Examinee);
               else -- let's see
                  Next_Stage.all.Pass_On(Examinee);
               end if;
            else
               Put_Line("Though odd," & Positive'Image(Examinee) &
                 " is a counterfeit prime" &
                 " (try" & Positive'Image(Who_Am_I) & ").");
            end if;
         or
            terminate;
         end select;
      end loop;
   end Prime_Task;

   First_Stage : Prime_Task_Ptr := new Prime_Task(2);
   N : Positive := 3;
   Upper_Bound : Positive := Positive'Value(Argument(1));

begin
```

```
   while N <= Upper_Bound loop
      First_Stage.all.Pass_On(N); -- feed the pipeline
      N := N + 2;
   end loop;
end Sieve;
```

Program 2.1: Pipelined Sieve of Eratosthenes in Ada

Why do we need the function `New_Pipeline_Stage`? Why not simply write

```
Next_Stage := new Prime_Task(Examinee);
```

The reason is that `Prime_Task` cannot be used as a subtype mark *within* the body of the task `Prime_Task`. It refers to the current execution of the task, something that is known in OO as `this` or `self`, i.e., `abort Prime_Task` does work inside `Prime_Task`. Note that `Ada.Task_Identification.Current_Task` is not a solution. Yes, it is clumsy.

The `terminate` alternative in the `select` statement comes in useful once the list of primes requested by the user has been produced.

There are (at least) two points in `sieve.adb` (in all programs presented in this chapter, for that matter) where some sort of optimization could be applied. One of them provides only minor performance improvements, but the other is bound to considerably increase the speed of the computation. It is left as an exercise for the inclined reader to spot those points.[10]

A sample run of `sieve` is shown below; it computes the primes up to 34:

```
(lpc01) $ ./sieve 34
It is a widely held belief that 2 is prime.
It has been confirmed that 3 is prime.
It has been confirmed that 5 is prime.
It has been confirmed that 7 is prime.
Though odd, 9 is a counterfeit prime (try 3).
It has been confirmed that 11 is prime.
Though odd, 15 is a counterfeit prime (try 3).
It has been confirmed that 13 is prime.
Though odd, 21 is a counterfeit prime (try 3).
It has been confirmed that 17 is prime.
It has been confirmed that 19 is prime.
Though odd, 25 is a counterfeit prime (try 5).
Though odd, 27 is a counterfeit prime (try 3).
It has been confirmed that 23 is prime.
Though odd, 33 is a counterfeit prime (try 3).
It has been confirmed that 29 is prime.
It has been confirmed that 31 is prime.
(lpc01) $
```

[10] Premature optimization is the root of all evil. –– D.E. Knuth

2.2 The Pipelined Sieve of Eratosthenes in CHILL

2.2.1 Design

As in the case of Ada, we create a list of task objects, each of which represents a prime task, to implement the pipeline. Calls to such a task object are used to pass on candidates, and, having reached the end of the list so far, a new task object is created and appended to the list. Straightforward then.

A problem occurs with the asynchronous nature of calls to a task object's procedures. While such an approach surely speeds up the computation and improves the throughput, it suffers from the problem of tailbacks. Note that it is of utmost importance that a prime task, T_{p_i}, process candidates **in exactly the same** order they were sent to it by $T_{p_{i-1}}$. Only thereby can we ensure that, at any time, the pipeline has the form $T_{p_1}, T_{p_2}, \ldots, T_{p_n}$. Thus, if $T_{p_{i-1}}$ sends a bulk of candidates asynchronously to T_{p_i}, T_{p_i} must process them in FIFO order. The wait set of T_{p_i}, as of any task object in CHILL, is unordered, however. If there is an unordered tailback "in front" of T_{p_i} (see Figure 2.5), then it is not clear which candidate (strictly, which task's request; note that all requests stem from the same task in our case) will be handled next. Thus, T_{p_i}—having just processed candidate j—might choose to receive just any candidate instead of candidate $j+1$. This, of course, causes the computation to become erroneous.

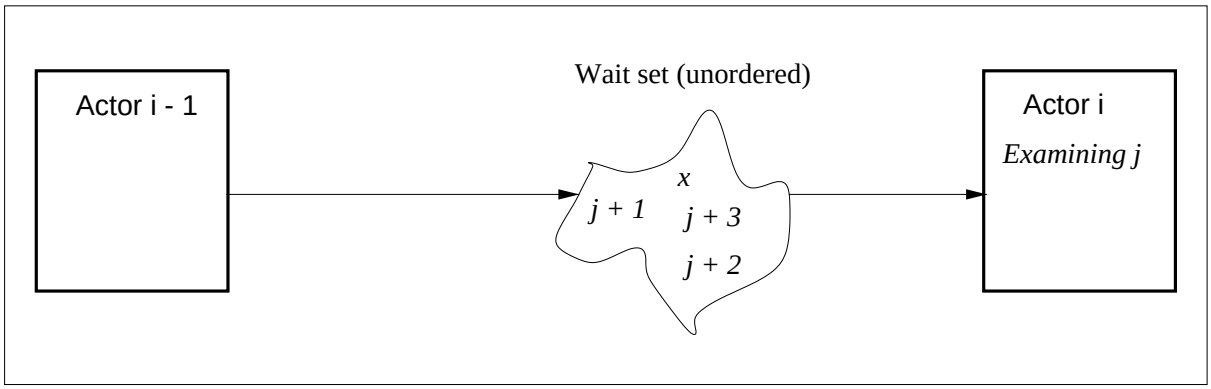

Figure 2.5: An unordered tailback

More elaborate thought given to the matter, however, shows that the FIFO-requirement is perhaps a bit too rigid. We (i.e., T_{p_i}) are, to a certain extent, allowed to take an arbitrary candidate out of the wait set for processing. If the candidate proves to be divisible by p_i, then it is thrown away by T_{p_i} anyway—so we are on the safe side here. At the beginning of this chapter, it was pointed out that it is sufficient to test a number $n \in \mathbb{N}$ for divisibility up to $i = \lfloor \sqrt{n} \rfloor$. Thus (and this is the solution to one of the exercises on page 34), a candidate, c, can be identified as a prime number by T_{p_i} if $c \bmod p_i \neq 0$ and ${p_i}^2 \geq c$. We do not really have to push it to the end of the pipeline. (Caution is needed, however: since several prime tasks now could identify numbers as being prime simultaneously, the end of the pipeline becomes a critical resource—the newly found primes must be appended to the (end of the) pipeline in the correct order! Besides other issues in this case, that of a fair machine emerges and becomes obnoxious.) So T_{p_i} can just take a candidate out of the wait set and test these conditions: if $c \bmod p_i = 0$, then c is thrown away; if not, T_{p_i} checks whether ${p_i}^2 \geq c$. If this is the case, then we have found a new prime number; however, if ${p_i}^2 < c$, then c must be passed on to $T_{p_{i+1}}$. But what if there is a candidate, d, in the wait set so that $d \bmod p_i \neq 0$ and ${p_i}^2 < d$, but $d < c$? In this

case, d must be processed before c. But it is too late, c has already been sent off to $T_{p_{i+1}}$. Note that there might also be a problem with the order of the identification of new prime numbers: if the wait set contains two prime numbers, p_x and p_y so that $p_x < p_y$, then identifying p_y as a prime number prior to p_x disrupts the structure of the pipeline. We must then ask ourselves the question as to whether we have indeed found the *next* prime number.

Whether all this can actually happen, depends on the frequency and distribution of prime numbers in $\mathbb{N}$, an issue we do not know enough about. Except for identifying a natural number as not being prime (which is always foolproof), this is a risky business. The situation could be be made more secure by having T_{p_i} peek at the candidates in the wait set and take them out in the correct order. Besides the fact that it is not clear how to do that in our languages (in Ada, we could probably use the `requeue` statement and sort the candidates locally), an ordering of the wait set in the first place (and pushing the candidates through the whole pipeline so that only the last prime actor identifies new prime numbers) is perhaps more adequate. FIFO and pushing the candidates through the whole pipeline are safe—the rest is optimization.

Continuing with the dangers of unordered tailbacks, let us look at Figure 2.6 in which the situation is shown where candidate x gains access to T_{p_i} instead of candidate $j+1$. Now let $x \bmod i \neq 0$, for $i = 2, 3, 5, \ldots, p_{\text{so\_far}}$. Sometime, x is passed on, finally reaching the end of the pipeline, where it is identified as a new prime. The question is: Is x really prime? Even if it were, is it really $p_{\text{so\_far}+1}$?

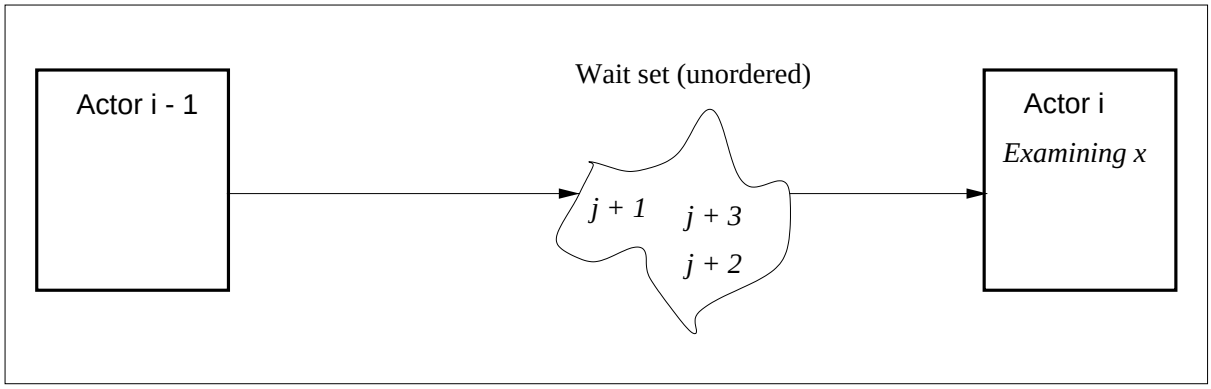

Figure 2.6: The wrong candidate has been promoted

As an example, suppose we have found so far

$$p_1 = 2,\ p_2 = 3,\ p_3 = 5, \text{ and } p_4 = 7$$

To each of the corresponding prime tasks, the candidate $x = 60769$ looks prime. T_{p_4} thus finds that $p_5 = 60769$ and creates T_{p_5}. This would be perfect were it not for the two flaws: $p_5 = 11$ and $67 \mid 60769$.

Note that putting a buffer between two tasks leads to synchronous communication (a CHILL buffer cannot be used for it does not guarantee FIFO access).

Note further that the author has indulged in the technique of deliberate lying, which was first brought to his attention in (Knuth, 1984), when he pointed out that the wait set of a CHILL task is unordered: there *is* a way to impose an ordering upon the wait set of a task. All $T_{p_{i-1}}$ has to do is attach a distinct priority to each call to T_{p_i} when transmitting a candidate. This priority is taken into consideration by T_{p_i} when it comes to choosing

the next request to handle. But, alas, this technique cannot be sensibly applied here for the following reasons. First, the CHILL specification is very unhelpful in this respect. It does not state the priorities available (we **assume** that the set of which is finite) and it lacks details on how the selection is done. Furthermore, since there are more candidates than priority values, most calls have to be made with the same priority anyway, which nullifies the ordering.

We conclude that asynchronous message passing is not an option here.

The author followed, quite amused, a discussion the other day on the suitability of Ada's concurrency features for realtime systems in `comp.lang.ada` and `comp.realtime`. In the latter newsgroup, it was lamented that the synchronous rendezvous does not scale well in large realtime systems with lots of inter-actor communication. Asynchronous message passing was identified as being more beneficial in terms of speed and throughput. There is no doubt about that but realtime systems, as one must understand, are systems where one must make certain that the number of actors does not exceed a limit (n). This can be accomplished by either disallowing dynamic actor creation or by including mechanisms that deal with erratically created actors. Without this compromise, any attempts to predict schedulability are in vain. The problem is that in realtime systems, which are often also embedded and autonomous, we have to guarantee (beforehand) that the physical memory available is sufficient for all actors to operate as required, especially as regards their behavior in time. It is clear that if the system is busy shutting down malfunctioning actors, then valuable time is lost. Thus, there is the risk for actors with hard deadlines to miss their stringent deadlines. Often, there is also an upper bound imposed upon the number of messages (m) an actor is permitted to send. But if $n \times m$ is a constant, then we can send messages asynchronously without being overly concerned about overfull mailboxes (of course, the messages received by an actor must still be stored)—we just arrange for the size of each mailbox to be (at least) $n \times m$. Of course, thought must be given to the question as to whether it is practical to provide each actor with a possibly large buffer. In short, asynchronous message passing makes sense only if we know beforehand how many messages will be sent in a bulk to an actor. And that is something we do not know in our case.

The bottom line is that we will use task objects for the prime tasks and impose synchronization upon the asynchronous call protocol using the technique proposed in (Brömel, 1999, Chapter 2). Painstaking thought reveals that T_{p_i} can pass on a candidate to $T_{p_{i+1}}$ only after it can be sure that $T_{p_{i+1}}$ has processed the previous candidate. Thus, each prime task can send/receive at most one candidate at a time, which means that a mailbox of size 1 is sufficient.[11] That, in turn, paves the way for CHILL buffers being used here—a buffer of size 1 is trivially FIFO; additionally, we are completely rid of unordered tailbacks outside this buffer (T_{p_i} communicates only with $T_{p_{i+1}}$). We let T_{p_i} `SEND` the candidate into a buffer and pass (a reference of) this buffer on to $T_{p_{i+1}}$. $T_{p_{i+1}}$, then, `RECEIVE`s from this buffer the candidate and processes it—this is because $T_{p_{i+1}}$ is suspended at a `RECEIVE` statement, waiting to extract a candidate from the buffer, a reference of which was given as a parameter in the pass-on call. Since the buffer's size is 1, we have also the net effect of synchronization here: after T_{p_i} has wrapped the candidate in the buffer, the buffer is full—hence if T_{p_i} wants to pass on the next candidate, it is blocked and can succeed only after $T_{p_{i+1}}$ has taken the previous candidate out of the buffer.

It is the author's hope that the discussion above has provided strong evidence to support the claim that the Ada pipeline is free of tailbacks.

[11] Upon second thought, we realize that we can actually send/receive $n \geq 1$ candidates, for some constant n and with a mailbox of adjusted size, synchronously.

Interestingly, especially with respect to the CHILL compiler at our disposal, we could have used processes instead of tasks with virtually the same underlying mechanism for communication (a processes has no interface functions, however; we would link two processes by a buffer and use `SEND/RECEIVE` for communication as well). But then, of course, we would not have had to discuss asynchronous matters, which the author felt was vital. We present both solutions in the next subsection.

2.2.2 The Sieve Programs in CHILL

Below is the code for the pipelined Sieve of Eratosthenes in CHILL using a task solution, `sieve.chill`, and processes, `sieve_processes.chill`. Note that since CHILL tasks have no code that is run when the task is activated, we have put all of a prime task's activity into the `Pass_On` entry.

```
Sieve : MODULE

   NEWMODE Mailbox = BUFFER(1) INT;
   NEWMODE Mailbox_Ptr = REF Mailbox;

   SYNMODE Prime_Task = TASK SPEC
      GRANT Prime_Task, Pass_On;

      Prime_Task : PROC(Who_Am_I INT IN) CONSTR END;
      Pass_On : PROC(Candidate Mailbox_Ptr IN) END;

      DCL Id INT;
      DCL Next_Stage Prime_Task_Ptr;
      DCL MBox Mailbox_Ptr;
   END Prime_Task;

   NEWMODE Prime_Task_Ptr = REF Prime_Task;

   SYNMODE Prime_Task = TASK BODY

      Prime_Task : PROC(Who_Am_I INT IN) CONSTR
         Id := Who_Am_I;
         Next_Stage := NULL;
         MBox := ALLOCATE(Mailbox);
      END Prime_Task;

      Pass_On : PROC(Candidate Mailbox_Ptr IN)
         RECEIVE(Candidate-> IN Examinee);
         IF Examinee MOD Id /= 0 THEN
            IF Next_Stage = NULL THEN
               Next_Stage := ALLOCATE(SELF, Examinee);
               -- Examinee is prime
            ELSE
               SEND MBox->([Examinee]);
               Next_Stage->.Pass_On(MBox);
               -- Examinee looks prime
```

```
            FI;
         ELSE
            -- Examinee is not prime
         FI;
      END Pass_On;
   END Prime_Task;

   DCL First_Stage Prime_Task_Ptr := ALLOCATE(Prime_Task, 2);
   SYN Upper_Bound = ...;  -- some appropriate value
   DCL Envelope Mailbox_Ptr := ALLOCATE(Mailbox);

   DO FOR N := 3 BY 2 TO Upper_Bound;
      SEND Envelope->([N]);
      First_Stage->.Pass_On(Envelope); -- feed the pipeline
   OD;
END Sieve;
```

Program 2.2: Pipelined Sieve of Eratosthenes in CHILL (task solution)

```
Sieve : MODULE

   NEWMODE Mailbox = BUFFER(1) INT;
   NEWMODE Mailbox_Ptr = REF Mailbox;

   -- this is needed to keep track of the
   -- end of the pipeline safely

   SYNMODE Counter_Mode = REGION SPEC
      GRANT Increase, Value;

      Increase : PROC(By INT IN) END Increase;
      Value : PROC() RETURNS(INT) END Value;

      DCL Counter INT := 1;
   END Counter_Mode;

   SYNMODE Counter_Mode = REGION BODY

      Increase : PROC(By INT IN)
         Counter := Counter + By;
      END Increase;

      Value : PROC() RETURNS(INT)
         RETURN Counter;
      END Value;
   END Counter_Mode;

   DCL Counter Counter_Mode;

   Start_Prime_Process : PROC(Who_Am_I INT IN, Envelope Mailbox_Ptr IN,
     Number INT IN)
```

```
      START Prime_Process(Who_Am_I, Envelope, Number);
      Counter.Increase(1);
   END New_Pipeline_Stage;

   Prime_Process : PROCESS(Who_Am_I INT IN, Envelope Mailbox_Ptr IN,
     Number INT IN);

     DCL MBox Mailbox_Ptr := ALLOCATE(Mailbox);

     DO FOR EVER;
        RECEIVE(Envelope-> IN Examinee);
        IF Examinee MOD Who_Am_I /= 0 THEN
           IF Number = Counter.Value() THEN
              Start_Prime_Process(Examinee, MBox, Number + 1);
              -- Examinee is prime
           ELSE
              SEND MBox->([Examinee]);
              -- Examinee looks prime
              FI;
        ELSE
           -- Examinee is not prime
        FI;
     OD;
   END Prime_Process;

   DCL Upper_Bound INT := ...;
   DCL Envelope Mailbox_Ptr := ALLOCATE(Mailbox);

   START Prime_Process(2, Envelope, 1);

   DO FOR N := 3 BY 2 TO Upper_Bound;
      SEND Envelope->([N]);
   OD;
END Sieve;
```

Program 2.3: Pipelined Sieve of Eratosthenes in CHILL (process solution)

As intimated in Subsection 1.2.5, task objects do not cause trouble when it comes to shutting down the pipeline. In the process solution, more care is needed: each process runs in an endless loop. Granted, if the pipeline is left devoid of candidates, each process is blocked at its `RECEIVE` statement and does not consume processor cycles anymore. A special candidate, say 0, could be inserted into the pipeline, and with the process loop modified accordingly, we shall be able to tackle the problem.

2.3 The Pipelined Sieve of Eratosthenes in Erlang

2.3.1 Desgin

The author proposes three different plans to tackle the problem. All will make use of Erlang's `send/receive` primitives, but in radically different ways.

1. Utilizing `send/receive` *to achieve synchronous communication.* T_{p_i} sends a candidate (or n candidates, as long as n is fixed) to $T_{p_{i+1}}$ (asynchronously) and waits for a notification from $T_{p_{i+1}}$ stating that the candidate has reached $T_{p_{i+1}}$, thereby achieving synchronization. Thus, T_{p_i} first uses a `send` and then a `receive` while $T_{p_{i+1}}$ tries to `receive` a value and then `send`s an acknowledgement.

2. Equipping each prime task with a (bounded) FIFO buffer. A prime task, T_{p_i}, and its successor, $T_{p_{i+1}}$, are linked via a bounded buffer of appropriate size. Communication is thus via this buffer. Using the wait solution in the case of overflow/underflow, we achieve synchronization. The buffer, which we will borrow from Chapter 1, is implemented by means of `send` and `receive`.

3. Using `send/receive` *to achieve asynchronous communication.* How will this work? Reading (Armstrong, 1996, p. 69), we learn that messages "are always delivered ... in the same order they were sent" and that each "process has a mailbox and all messages which are sent to the process are stored in the mailbox in the same order as they arrive." We conclude from this that FIFO is guaranteed by the Erlang runtime system and that, therefore, explicit synchronization is superfluous.

Now, let us assess the approaches. Since Erlang does not know passive entities that guarantee mutually exclusive access, the buffer in *2* has to be implemented as a process (see Chapter 1). Hence, we end up with a buffer process for each prime process. Though functionally correct, this is felt as being riddled with overhead. Note that we have used the basic idea behind this approach in Section 2.2; we had no alternative, however, and the buffer was passive.

The technique underlying *1* is in keeping with the approach taken in Section 2.1, and is both functionally correct and acceptable in terms of overhead.

The astute reader undoubtedly has spotted the blemish of *3*. It does not mention what happens should the mailbox become full. This blemish must be attributed to the authors of (Armstrong, 1996) for they do not mention this vital aspect at all. The mailbox is the backbone of the correctness of the computation. Even if it were implemented as a storage pool that can grow dynamically, it might fail: computers are finite machines.

In the author's conviction, it is essential that every Software Engineer have the endeavour to first make a program comply with the specification before applying optimization. This provides justification to drop proposal *3* and to adopt *1* in favour of *2*.

In summary, we will spawn a process for each stage of the pipeline. Each process, then, waits for a candidate to be sent to it from its preceding process. The candidate is then examined, and either dropped, passed on to the process' successor, or, if there is no successor, identified as a new prime number, in which case a new prime process is created.

Sending a candidate is made synchronous by waiting for an acknowledgement from a process' successor.

2.3.2 The Sieve Program in Erlang

Here is the code of `sieve.erl` then:

```
-module(sieve).
-export([start/0, examine/2, feed_pipeline/1]).

start() ->
   register(first_stage, spawn(sieve, examine, [2, null])),
   io:format("Yes, 2 is prime~n").
```

```
examine(Who_Am_I, Next) ->
   receive
      {From, Candidate} ->
         From ! ok,  %% got it
         if Candidate rem Who_Am_I /= 0 ->  %% it *looks* prime
            if Next == null ->  %% it *is* prime
               io:format("This is a prime number: ~w~n", [Candidate]),
               Last = spawn(sieve, examine, [Candidate, null]),
               examine(Who_Am_I, Last);
            true ->
               Next ! {self(), Candidate},  %% who knows?
               receive
                  ok ->
                     true
               end,
               examine(Who_Am_I, Next)
            end;
         true ->
            io:format("~w is not prime, try ~w~n",
              [Candidate, Who_Am_I]),
            examine(Who_Am_I, Next)
         end
   end.

feed_pipeline(Candidate) ->
   first_stage ! {self(), Candidate},
   receive
      ok ->
         true
   end,
   %% sleep(1000),
   feed_pipeline(Candidate + 2).  %% most of them are odd, you know

sleep(Time) ->
   receive
   after Time ->
      true
   end.
```

Program 2.4: Pipelined Sieve of Eratosthenes in Erlang

The alert reader has surely noticed that the line

```
Last = spawn(sieve, examine, [Candidate, null])
```

contains an assignment. In Erlang, however, we are allowed to assign a value to a variable only once—this is called the *single assignment rule*. The following dialogue with the Erlang shell illustrates this (a variable has got the first letter of its name capitalized, by the way):

```
6> Foo = hello.
```

```
hello
7> Foo = hallo.
** exited: {{badmatch,hallo},{erl_eval,expr,3}} **
8>
```

`Foo` has been bound to `hello` in line 6 and is thus no longer unbound. Assignment (called *binding* in Erlang parlance) can only occur to an unbound variable. Nevertheless, Program 2.4 compiles and executes without error even though each process executes the code of `examine` many, many times. The secret behind this is, of course, that `Last = ...` is executed only once. In this code fragment, we create a new prime process whose Pid is assigned to `Last`. Let T_k be the last prime task so far. Then

$$\texttt{Last} = \begin{cases} \textit{non-existent}, & \text{for } T_k \\ \texttt{Pid}(T_k), & \text{for } T_{k-1} \\ \texttt{Pid}(T_{k-1}), & \text{for } T_{k-2} \\ \quad\vdots & \quad\vdots \\ \texttt{Pid}(T_2), & \text{for } T_1 \end{cases} \tag{2.3}$$

Thus, for most processes, `Last` is not the last process in the pipeline but the next process. Since for a given process the (direct) successor is unique, the code involving the assignment to `Last` is executed indeed only once; hence, we get away with impunity.

From the standpoint of optimization, however, the assignment to `Last` is superfluous—a simple

```
examine(Who_Am_I, spawn(sieve, examine, [Candidate, null]))
```

would do. But the author feels that the more verbose formulation is much better in terms of readability and has resisted the temptation to optimize.

Let us now test our program by playing again with the Erlang shell:

```
1> c(sieve).
./sieve.erl:46: Warning: function sleep/1 not called
{ok,sieve}
2> sieve:start().
Yes, 2 is prime
ok
3> sieve:feed_pipeline(3).
This is a prime number: 3
This is a prime number: 5
This is a prime number: 7
9 is not prime, try 3
This is a prime number: 11
15 is not prime, try 3
This is a prime number: 13
21 is not prime, try 3
This is a prime number: 17
This is a prime number: 19
25 is not prime, try 5
27 is not prime, try 3
```

```
This is a prime number: 23
33 is not prime, try 3
35 is not prime, try 5
This is a prime number: 29
39 is not prime, try 3
This is a prime number: 31
...
```

Readers should not worry about the function `sleep`; it has been included for test purposes only.

We have not paid heed to the issue of termination here, but the technique proposed in Subsection 2.2.2, with the obvious modifications to `start`, `feed_pipeline`, and `examine`, should be adequate to solve the problem.

2.4 The Pipelined Sieve of Eratosthenes in Java

2.4.1 Desgin

It has been lamented, several times already, that Java lacks direct actor communication. It is for this reason that we must resort to indirect actor communication, fundamental concepts of which can be found in (Brömel, 1998, Chapter 5). The use of `suspend/resume`, quite appealing at first glance, suffers from at least two major drawbacks:[12] values—i.e., candidates—cannot be transmitted this way, and synchronization is not guaranteed. The tactics would be to let T_{p_i} send $T_{p_{i+1}}$ (which has suspended itself) a candidate and execute $T_{p_{i+1}}$`.resume()` and then T_{p_i}`.suspend()`. T_{p_i} will then be reawakened by $T_{p_{i-1}}$, and so on. Careful thought reveals that there is a race condition—if $T_{p_{i+1}}$ is "not quick enough," it might miss a candidate, i.e., T_{p_i} sends a new candidate before $T_{p_{i+1}}$ has had a chance to process the preceding one. This could be fixed by having $T_{p_{i+1}}$ issue an acknowledgement message to T_{p_i} (as in Subsection 2.3.1). But how to send this acknowledgement? `suspend/resume` is inappropriate since it would confuse T_{p_i} (because T_{p_i} is also waiting for a `resume` from $T_{p_{i-1}}$). Further, how to send/transmit a candidate from one thread to another? A third party object, a buffer, must be used. But why not use this buffer for both passing on candidates and synchronization?

So we shall equip each prime thread with a passive buffer (from Chapter 1) which is used to pass on a candidate synchronously. Thus, T_{p_i} and $T_{p_{i+1}}$ are linked via this buffer. Each time a new prime thread is created, its predecessor passes on a reference to this buffer.

Readers surely have noticed that this approach, which is illustrated in Figure 2.7, bears some resemblance to that used in Subsection 2.2.1.

A solution which uses means provided by Java directly was proposed by Michael Hartmeier. Java knows streams, which can be used as a communication facility between threads (note that this is still indirect communication). A sample implementation (by Mr Hartmeier) of this approach is available from the author.

[12] To tell the whole truth, there is a third obstacle: `suspend/resume` have been deprecated in Java 1.2. "The problem is that if a thread is the target of a `suspend()`, then it does not release any monitor locks. If the thread that is to perform the `resume()` needs to acquire one of the locks held by the suspended thread, then the two threads will be deadlocked." (Brosgol, 1998, 8.2)

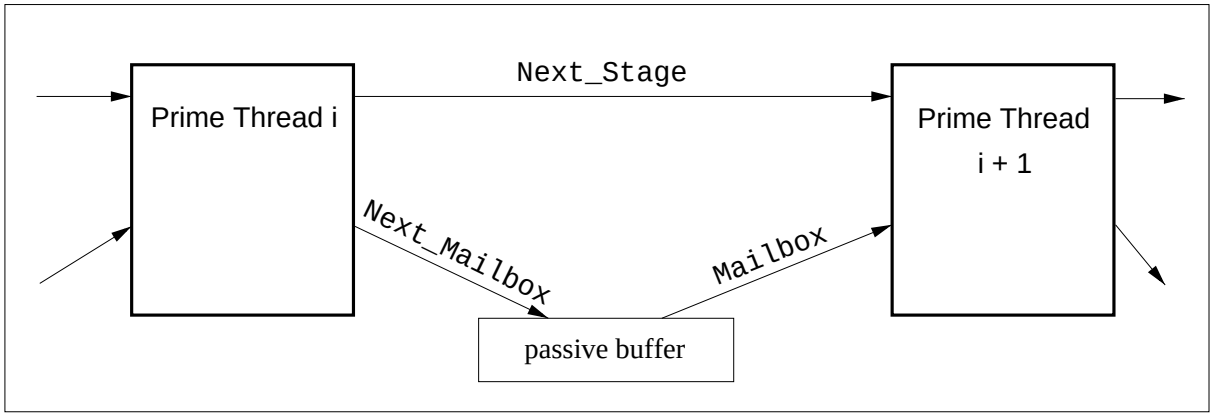

Figure 2.7: The pipeline in Java—two prime threads share a passive buffer

2.4.2 The Sieve Program in Java

Presented below is the code of `Sieve.java`.

```
class Pipeline_Stage extends Thread {

   public Pipeline_Stage(int Who_Am_I, Passive_Buffer Mailbox) {
      this.Mailbox = Mailbox;
      this.Who_Am_I = Who_Am_I;
      this.Next_Stage = null;
      this.Next_Mailbox = new Passive_Buffer();
   }

   private Passive_Buffer Mailbox;
   private Pipeline_Stage Next_Stage;
   private Passive_Buffer Next_Mailbox;
   private int Who_Am_I;

   public void run() {

      int Examinee;

      if (Who_Am_I == 2) {
         System.out.println("Mind you, 2 is the only prime number" +
           " that's even---hence, it's odd!");
      }

      while (true) {
         Examinee = ((Integer) Mailbox.Retrieve()).intValue();
         if (Examinee == Sieve.Special_Candidate) {
            System.out.println("Whereupon prime thread " + Who_Am_I +
              " says, ``Yeah, we've had enough!''");
            // tell the fellow workers to go home
            Next_Mailbox.Store(new Integer(Sieve.Special_Candidate));
            return;
```

```
         }
         if (Examinee % Who_Am_I != 0) {
            if (Next_Stage == null) {
               Next_Stage = new Pipeline_Stage(Examinee, Next_Mailbox);
               Next_Stage.start();
               System.out.println("Prime number found: " + Examinee);
            } else {
                 Next_Mailbox.Store(new Integer(Examinee));
              }
         } else {
              System.out.println(Examinee + " is not prime, try " +
                Who_Am_I);
           }
      }
   }
}

class Sieve {

   public static final int Special_Candidate = 0;
   // the special candidate used to shut down the pipeline

   public static void main(String[] Args) {
      Passive_Buffer Mailbox =
        new Passive_Buffer(); // size defaults to 1

      Pipeline_Stage First_Pipeline_Stage =
        new Pipeline_Stage(2, Mailbox);
      First_Pipeline_Stage.start();

      // let the user provide the upper bound and get
      // it from the command line
      int Upper_Bound = new Integer(Args[0]).intValue();

      // and off we go
      for (int I = 3; I <= Upper_Bound; I = I + 2) {
         Integer Candidate = new Integer(I);
         Mailbox.Store(Candidate);
      }
      System.out.println("The boss says, ``Let's call it a day!''");
      Mailbox.Store(new Integer(Special_Candidate));
   }
}
```

Program 2.5: Pipelined Sieve of Eratosthenes in Java

Note that we have included a way of shutting down the pipeline explicitly—this is necessary. Since there is no dependence between threads in Java (unless the programmer creates one), there is no need to equip the language with an equivalent of Ada's `terminate`. Note also that, strictly speaking, `Sieve.main()` terminates after the `for` loop has been executed. It is only the prime threads (not really dependent upon `Sieve.main()`) that

run in an endless loop and, therefore, prevent the Java Virtual Machine from terminating, as can be verified by consulting (JLS, 1996, 12.9):

> A Java Virtual Machine terminates all its activity and exits when one of two things happens:
>
> - All the threads that are not daemon threads (§ 20.20.24) terminate.
> - Some thread invokes the `exit` method (§ 20.16.2) of class `Runtime` or class `System` and the exit operation is not forbidden by the security manager (§ 20.17.13).

Thus, the JVM waits for the prime threads to terminate, not for `Sieve.main()` (which, in turn, does not wait for the prime threads).

Although a quick solution to the problem is given above, the `exit` method, we have chosen another way: feeding a *special* candidate, 0, into the pipeline. By having every prime thread check whether it has received this special candidate, and—if so—propagate it to its successor, we shut the pipeline down, stage by stage. Once a prime thread has sent the special candidate to its successor, it returns from its `run()` method (in a hearty manner). Essentially the same approach can be used in Program 2.3 and Program 2.4.

Proponents of optimization might argue that the continuous creation of `Integer` objects throughout the program is a waste of storage. Indeed, it would be more efficient to use one `int` variable for the candidates. But, as can be seen in Program 1.10, the passive buffer is not fixed in terms of the component type; it supports heterogeneity by allowing entities of type `Object`* to be stored. A special purpose buffer (for `int`) *could* be created—later, when optimization is undertaken. There is a price to pay for additional functionality, of course.

Let us look at a sample run of our program

```
(lpc01) $ javac Sieve.java && java Sieve 20
Mind you, 2 is the only prime number that's even---hence, it's odd!
Prime number found: 3
Prime number found: 5
Prime number found: 7
9 is not prime, try 3
Prime number found: 11
15 is not prime, try 3
Prime number found: 13
The boss says, ``Let's call it a day!''
Whereupon prime thread 2 says, ``Yeah, we've had enough!''
Whereupon prime thread 3 says, ``Yeah, we've had enough!''
Whereupon prime thread 5 says, ``Yeah, we've had enough!''
Prime number found: 17
Whereupon prime thread 7 says, ``Yeah, we've had enough!''
Whereupon prime thread 11 says, ``Yeah, we've had enough!''
Prime number found: 19
Whereupon prime thread 13 says, ``Yeah, we've had enough!''
Whereupon prime thread 17 says, ``Yeah, we've had enough!''
Whereupon prime thread 19 says, ``Yeah, we've had enough!''
(lpc01) $
```

which computes the prime numbers up to 20.

2.5 Summary and Comparison

This chapter has been concerned with the concurrent computation of prime numbers. After a short introduction, the algorithm of Eratosthenes was brought to the reader's attention. We have then looked at the pipelined version of this algorithm, the realization of which has been adopted as the primary goal of this chapter. Following a design proposal by (Barnes, 1996, p. 440), we have decided to represent the pipeline as a chain/list of actors, each of which symbolizes a prime task that accepts candidates from its predecessor. Examining the candidate, it was either rejected, passed on, or identified as a new prime number. Each prime actor can be conceived of as a filter removing multiples of its own prime value.

Questions arose as to whether the candidates should be passed on synchronously or asynchronously, the latter alternative offering possibly better speedup and throughput. But in the end, we have rejected the asynchronous approach on the grounds that it is important that the order of the pipeline be preserved—candidates should be stored and processed in FIFO order. Besides the fact that only CHILL and Erlang support asynchronous communication, it was not clear as to how to ensure this vital aspect. Not paying heed to this has been identified as being perilous for the correctness of the computation. Consequently, we have decided to stick to synchronous communication, allowing each prime actor to pass on at most one candidate at a time and avoiding any tailbacks whatsoever.

In all four languages, it did not prove difficult to reach the aforementioned goal. In Ada, tasks are the natural choice for the pipeline stages and the rendezvous is adequate for passing on the candidates as desired. CHILL offers even two alternatives for the prime actors—processes or tasks. Erlang's `send/receive` primitives are also straightforward for the purpose needed. Note that in CHILL and Erlang, additional effort was required to ensure synchronous communication between a prime actor and its successor. By using, respectively, a CHILL buffer (of size 1) and an acknowledgement message, this could be accomplished without difficulty: communication and synchronization have been effectively (and efficiently) coupled. Though it lacks the classical client/server model (with an active server), Java did not present an obstacle. By linking two "adjacent" prime threads via a passive buffer (from Chapter 1) and using this buffer for both communication and synchronization, the direction was clear.

Though the nature of the pipelined algorithm makes an application of concurrency very natural, it must be understood that true performance improvements can only be gained if a lot of prime actors can be mapped onto physical processors—that is, a MIMD computer is of true benefit here.

Readability

See Table 2.1 below.

Program Size

Again, refer to Table 2.1.

Communication Effort

Only in Erlang do we need to explicitly suspend a process in order to achieve synchronization. This has been accomplished via a pair of `send/receive` per actor. In Ada,

synchronization is provided by the rendezvous, and buffers are used in CHILL and Java, which come "equipped with synchronization" in that they force clients to wait should the buffers be empty or full.

Thus the communication effort in Erlang is 2.

Degree of Concurrency

Even if we only allow each prime actor to send/receive at most one candidate at a time, it is not the case that only one candidate, at a time, is being worked on in the pipeline; the emphasis is on *each prime actor*.

Let l be the current length of the pipeline and define n to be the number of candidates currently in the pipeline. Then the *utilization* of the pipeline is given by the ratio $\frac{n}{l}$. The higher the utilization (whose upper bound is 1), the higher the degree of concurrency (which is n, with upper bound l). Determining n is not easy, however. If a lot of candidates get sifted out early in the pipeline, then those prime actors towards the end of the pipeline are idle. By feeding only odd numbers into the pipeline, we already try to increase the utilization.

Thorough thought reveals that the utilization of the pipeline is high if a long chain of prime twins is fed into the pipeline (in which case the pipeline will also grow, that is, l is getting larger). Unfortunately, we do not know much about prime twins, especially about their distribution and frequency in $\mathbb{N}$. It might be the case that "somewhere in $\mathbb{N}$," there is a long chain of prime twins; long enough so that the utilization of the pipeline is high.

For the degree of concurrency, we have already mentioned that its upper bound is l and that its current value is n. A good approximation is the average length of prime twin chains.

Liveness

The pipeline is, of course, vulnerable: if some prime actor breaks down, then the flow of communication is interrupted and prime actors following the faulty actor are left devoid of candidates. Passing on candidates to a defunct prime actor causes an exception to occur, which can be used to handle this case (i.e., to rebuild the pipeline stage concerned).

Otherwise, since the stream of candidates is continuous and since there are infinitely many prime numbers, each prime actor is allowed to execute its desired action as often as required (since it is bound to receive candidates)—liveness is guaranteed then.

Deadlock

Disregarding the case in which a prime actor breaks down, we deduce that the pipeline is free of deadlocks.

Starvation

Using the same antecedent as in the preceding subsubsection, the author does not see any cause for starvation.

Necessity of a Fair Machine

Surprisingly, fairness is not of importance here. Though one might expect that if a prime actor were not scheduled/dispatched properly, this would amount (in the outcome) to a

breakdown of this actor with all the fateful consequences (much harder to cure), this is not the case.

Assume, without loss of generality, that T_{p_i} is the only prime actor that is *not* treated fairly by the machine. Then, sooner or later, all prime actors "to the right of" T_{p_i} will run out of candidates to test. As a result, they block. $T_{p_{i-1}}$ finds that it cannot pass on a candidate to T_{p_i} and blocks as well. Likewise, $T_{p_{i-2}}$ blocks because it cannot send a candidate to $T_{p_{i-1}}$. And so forth. After a while, we end up with T_{p_i} being the only actor that is runnable. The machine now must schedule/dispatch T_{p_i} for execution.

Generally, the issue of fairness is not only a vital one in concurrent programming (disregarding the exotic cases of the execution of error handlers), but also a tricky one: it is fiendishly intricate to give a precise definition of *fairness* and, therefore, to assess whether or not a machine is fair. Fortunately, we do not need a fair machine here.

Table 2.1 below summarizes and completes this chapter.

Language	Ada	CHILL	Erlang	Java
Readability	2	3^+	3	3^+
Program Size	52	45/54	41	63
Communication Effort	n.a.	n.a.	2	n.a.
Degree of Concurrency	*hard to deduce*			

Table 2.1: Tabular Summary of Chapter 2

3 The Dining Philosophers

> The men sat sipping their tea in silence. After a while, the klutz said, "Life is like a bowl of sour cream."
>
> "Like a bowl of sour cream?" asked the other. "Why?"
>
> "How should I know? What am I, a philosopher?"

Introduction and Problem Description

Once upon a time, in a secluded rural village, there lived five great philosophers. So profoundly wise were they that people from everywhere would send letters to them in which they posed problems and asked for help. Only by strictly following a rigid ritual could the philosophers gain the wisdom: each philosopher's existence was entirely made up of either thinking or eating (Cogito ergo sum: ergo deinde consumo!). Thus, they would think a while until they became hungry; they would eat then, and delve into the problems again, etc.

They would, however, only eat spaghetti at the local restaurant called Frederici's. Inside this restaurant, there was a big round table with five sets of forks, plates, and chairs. As it happened, the philosophers found it difficult to eat with only one fork (the spaghetti were delicious but hopelessly entangled). Furthermore, they were too polite to lean across the table to obtain forks and thus they would use the forks on either side of them only (i. e., adjacent forks).

In one curious episode, back in the days when Frederici allowed up to five gurus inside, the gurus were all seated at the table, ready to grab their forks. Each guru reached to their left to grab their neighbor's fork, thinking that their own fork was safe. Soon after grabbing the nearby fork, the gurus looked down, and noticed that their own forks were missing. Seeing this, each of them waited until a second fork was free.

It turned out to be a *very long* wait.

This is a hearty description (with a bit of folklore) of the problem of the Dining Philosophers which was originally posed in (Dijkstra, 1971). The purpose of this chapter is to present a way that ensures the survival of the philosophers, i.e., to make sure that every hungry philosopher will eventually be allowed to eat and that situations as described above cannot occur. A prerequisite for this is to have enough spaghetti, of course. But we shall not get distracted by such things—let there be ample noodles once and for all.

Frederici was very unhappy about this situation, and he knew that every so often, all five gurus would turn in after a rough day. After painstaking thought, he came to realize that the above situation would always occur if he admitted five philosophers at a time to have their meals. He figured out that allowing at most only four gurus to dine simultaneously would ameliorate the situation: one philosopher would always find two free forks, which they could use to actually eat. Having finished the meal, the philosopher would put down their forks, thereby allowing another hungry fellow to start eating. A philosopher who had finished their meal was expected to leave the restaurant immediately, which enabled a philosopher waiting outside, if any, to enter and join the round.

But that meant of course a possible loss in profit; having only four gurus dine instead of five could reduce the tips per round and might upset the one waiting outside (what if it

rained?), he thought. Frederici, the shrewd business man he was, came up with a brilliant idea: he introduced even more table manners. He reckoned that it would be better if every philosopher reached out to grab their forks only if they could be sure that neither their left nor right fork was currently being used by their left or right neighbor. He thus decreed that the left and right fork were not to be obtained in two individual grabs but only in one. This way, all five gurus could indeed be seated at the table and no one would have had to wait outside in the cold. Furthermore, the dilemma described above could not occur: a philosopher would either pick up both forks in an indivisible action or none at all. At no time would a philosopher sit at the table holding one fork and waiting for the other to become free.

Frederici was immediately hailed as a great hero, thinker, and diplomat. He ensured the survival of the philosophers. He was offered to join their cult, but he declined and kept right on serving spaghetti in his restaurant.

The story goes that they all lived happily ever after!

3.1 The Dining Philosophers in Ada

3.1.1 Design

The first thing to ponder is whether we admit five gurus at a time or restrict ourselves to only four. This is important for it seals the fate of Frederici. With the restriction to four philosophers, he prevents the fifth from entering the restaurant. Should one of the four philosophers inside leave the restaurant, the fellow waiting outside is given permission to enter. All this is taken care of by Frederici, who is then modelled as a task. In this case, both the restaurant and the forks constitute critical resources—the former will be protected by Frederici.

Having up to five gurus at the table leaves no real purpose for Frederici—the forks will be represented by a protected object as described below. So there is no need to include Frederici into the scenario at all. In terms of speed and throughput, this approach is more helpful—we will follow that path. At the end of Subsection 3.1.2, however, we will outline a modification to the scenario which justifies Frederici's existence.

The philosophers are naturally modelled as tasks, and the life of one of them is fairly simple and can be expressed using the following pseudocode:

```
loop
   Enter the restaurant
   Pick up a pair of forks
   Enjoy the meal
   Put down the pair of forks
   Leave the restaurant
   Delve into the problems of humankind
end loop
```

As mentioned above, the forks represent a critical resource. Since they are passive, they can (and will) be represented by a protected object. We provide operations to acquire and to release a pair of forks. Strictly speaking, releasing a pair of forks is not critical and can be done via a procedure; acquiring a pair, however, must be implemented by

means of an entry with an appropriate barrier. *Appropriate* means that we have to check whether a philosopher's forks (left and right) are available. We number our philosophers from 0 to $N-1$ and say that the forks of philosopher i are available if both fork i and fork $(i+1) \bmod N$ are currently not in use.[13] That is, philosopher i will wait if at least one needed fork is not free. The status of the forks is kept in a Boolean array, called `Available`. We thus come up with the following extension of the pseudocode above for philosopher i:

```
loop
   Enter the restaurant
   Forks.Pick_Up_Pair(I)
   Enjoy the meal
   Forks.Put_Down_Pair(I)
   Leave the restaurant
   Delve into the problems of humankind
end loop
```

Now, the barrier of `Forks.Pick_Up_Pair` would simply check whether `Available(I) and Available(I + 1)` evaluates to true. Almost straightforward—alas, we are not allowed to reference formal entry parameters in barriers. But it is easy to circumvent this restriction: we use a `True` barrier and employ the `requeue` facility:

```
entry Pick_Up_Pair(Whose : in Philosopher_Range) when True is
begin
   if Available(Whose) and then Available(Whose + 1) then
      -- OK, grab them
   else
      requeue Please_Get_Pair;  -- a private entry
   end if;
end Pick_Up_Pair;
```

or we could use an entry family. Thus, each philosopher would have their own `Pick_Up_Pair` entry, which they call with their identification. This identification establishes the family index which, in turn, can be referenced by the corresponding barrier. The author favours the former approach, and Subsection 3.5.1 will point out why.

As an aside, notice that in both approaches, the call syntax is identical so that the program can be easily adapted to changing needs.

The purpose of `Enter the restaurant` and the like is not of significance here; readers are invited to think of meaningful semantics.

3.1.2 The Ada Gurus

Read on or examine the file `dining.adb` to find out how the gurus would appease their hunger in Ada.

[13] Numbering them from 0 to $N-1$ is handy since we will use modular types; remember that the table in Frederici's was round.

```
with Ada.Text_IO, Ada.Numerics.Float_Random;
use Ada.Text_IO, Ada.Numerics.Float_Random;

procedure Dining is
   Philosophers_Count : constant Natural := 5;
   type Philosopher_Range is mod Philosophers_Count;
   type Available_Array is array(Philosopher_Range) of Boolean;
   Id : Integer := -1;
   G : Generator;

   function Next_Id return Philosopher_Range is
   begin
      Id := Id + 1;
      return Philosopher_Range(Id);
   end Next_Id;

   procedure Enter_The_Restaurant(Who : in Philosopher_Range) is
   begin
      Put_Line("Philosopher" & Philosopher_Range'Image(Who) &
        " says ``Hello'' to Frederici.");
   end Enter_The_Restaurant;

   procedure Enjoy_The_Meal(Who : in Philosopher_Range) is
   begin
      Put_Line("Philosopher" & Philosopher_Range'Image(Who) &
        " is currently not available for comment.");
      delay Duration(Random(G));
   end Enjoy_The_Meal;

   procedure Leave_The_Restaurant(Who : in Philosopher_Range) is
   begin
      Put_Line("Philosopher" & Philosopher_Range'Image(Who) &
        " asks Frederici to be excused.");
   end Leave_The_Restaurant;

   procedure Think(Who : in Philosopher_Range) is
   begin
      delay Duration(2.0 * Random(G));
   end Think;

   -- the forks then

   protected Forks is
      entry Pick_Up_Pair(Whose : in Philosopher_Range);
      procedure Put_Down_Pair(Whose : in Philosopher_Range);

   private
      Available : Available_Array := (others => True);
      -- this is a private entry used for the requeue mechanism
      -- the idea is a steal from (Brosgol, 1996)
      entry Please_Get_Pair(Whose : in Philosopher_Range);
```

```
   Waiting_Count : Natural := 0;
end Forks;

protected body Forks is

   entry Pick_Up_Pair(Whose : in Philosopher_Range) when True is
   begin
      if Available(Whose) and then Available(Whose + 1) then
         Available(Whose) := False;
         Available(Whose + 1) := False;
      else
         requeue Please_Get_Pair;
      end if;
   end Pick_Up_Pair;

   entry Please_Get_Pair(Whose : in Philosopher_Range)
     when Waiting_Count > 0 is
   begin
      Waiting_Count := Waiting_Count - 1;
      if Available(Whose) and then Available(Whose + 1) then
         Available(Whose) := False;
         Available(Whose + 1) := False;
      else
         requeue Please_Get_Pair;
      end if;
   end Please_Get_Pair;

   procedure Put_Down_Pair(Whose : in Philosopher_Range) is
   begin
      Available(Whose) := True;
      Available(Whose + 1) := True;
      Waiting_Count := Please_Get_Pair'Count;
   end Put_Down_Pair;

end Forks;


task type Philosopher(Who_Am_I : Philosopher_Range := Next_Id) is
   -- they will eat together but they
   -- won't talk to one another!
end Philosopher;


task body Philosopher is
begin
   loop
      Enter_The_Restaurant(Who_Am_I);
      Forks.Pick_Up_Pair(Who_Am_I);
      Enjoy_The_Meal(Who_Am_I);
      Forks.Put_Down_Pair(Who_Am_I);
      Leave_The_Restaurant(Who_Am_I);
```

```
         -- and, finally:
         Think(Who_Am_I); -- ;-)
      end loop;
   end Philosopher;

begin -- of Dining
   Put_Line("Well, let's get it on ...");
   Reset(G);

   declare
      Gurus : array(Philosopher_Range) of Philosopher;
   begin
      null;
   end;
end Dining;
```

Program 3.1: The Dining Philosophers in Ada

We use the function `Next_Id` to provide each guru with a unique identity. Since `Next_Id` is only called in the default expression of a philosopher task's discriminant, this is safe. This way, there is no possibility for `Next_Id` to be called simultaneously by two or more tasks (the astute reader has undoubtedly sensed that this function manipulates global data[14]). Although the order of the evaluation of the components of the array `Gurus` is not defined by the Ada language, there is the requirement that these elaborations be performed in a sequential order, see (ARM, 1995, 1.1.4[18]). This does not guarantee that the jth philosopher gets the identification j as the ARM speaks of "some sequential order." But this not really a problem since all we are interested in is that a philosopher gets a unique identification. If there were the order requirement, then we could use an entry in the philosopher task that sets this identification.

It is easy to alter the scenario so that there is a reason for Frederici's existence. If we increase the number of philosophers but keep the number of pairs of forks available a constant, then if there are already five gurus inside, there is need for some philosophers to be prevented from entering the restaurant (if it rains, they are allowed to come in, but not permitted to take a seat at the table). After a philosopher has left the restaurant, another one waiting outside can enter and join the round. Thus, Frederici could act as a sort of dispatcher. A philosopher willing to enter would greet Frederici who would then decide whether there is still room for another guru. A philosopher who is about to leave would say goodbye to Frederici. This way, Frederici could be modelled as a task offering two entries, `Ask_For_Admittance` and `Goodbye`, that keeps track of the number of philosophers currently inside the restaurant. The details are left as an exercise for the reader.

Strictly speaking, even with this extended version, there is no *real* reason for Frederici to exist—the protected object which is the foundation of the forks does its jobs irregardless of the number of philosophers competing for the forks. But including Frederici appears to be more natural (at least in terms of the folklore).

[14] Proponents of functional programming might find yet another stumbling block.

3.2 The Dining Philosophers in CHILL

3.2.1 Design

We can either use task objects or processes to represent the philosophers (we could use both). In both cases, the philosophers have to be awakened explicitly—so this is not really a criterion. Tasks appear to be more advantageous since there is a type underlying. We can create arrays of tasks, for example, which is more convenient. Also, should the life in the little community change so that our gurus would be required to communicate with one another, tasks are the better alternative.

In the author's humble opinion, a CHILL region is ideally suited for the forks and an event can be used to accomplish the blocking of a philosopher in case their forks are not free. This is, to some extent, comparable to the use of the `requeue` statement in Ada (within an entry body) or the use of `wait/notify` in Java. A simple

```
...
IF Forks_Not_Free THEN
   DELAY Forks_Occupied;
FI;
-- grab them
...
```

in the `Pick_Up_Pair` attempt is not sufficient. If a delayed philosopher is allowed to continue (by a `CONTINUE` (which, of course, does not need to be guarded) executed in `Put_Down_Pair`), then it is not clear whose forks have actually just been freed: *theirs* or just two. Instead, a guru must incessantly poll their forks' status, i.e.,

```
...
DO WHILE Forks_Not_Free;
   DELAY Forks_Occupied;
OD;
-- grab them
...
```

This, of course, gives rise to the suspicion of busy waiting and to the necessity of a fair machine. There is nothing comparable to Ada's entry barriers and ordered queues in CHILL—the wait set of both regions and events is unordered: priorities cannot be used in calls to region components (they can be used in `DELAY` statements, however, but the unclear semantics of priorities in CHILL almost precludes that). The possibility of busy waiting in Ada exists, but is less likely. Because the `requeue` puts the callers in FIFO order before the private entry, whose barrier is only re-evaluated after a guru has put down their forks, the order in which the requests have been made is preserved, and since a requeued caller, C, is likely to have other callers "in front" of it (which have the potential to free C's forks), there is an increased likelihood that C's forks will be free when it is permitted to execute `Please_Get_Pair` (again).

Note that the use of several events (one for each philosopher, one for each fork, or one for each pair of forks?) does not better the situation.

We will return to that topic in Section 3.5.2 and discuss a possible workaround.

3.2.2 The CHILL Gurus

Since a CHILL task's execution must be triggered explicitly by calling one of the task components, we have to provide such a component interface and put the whole life cycle of a philosopher into it. Executing this procedure in the main program causes the task (and hence the philosopher) to start its activity (recall that this procedure call is asynchronous). This is depicted below and in the file `dining.chill`.[15]

```
Dining : MODULE
   SYN Philosophers_Count = 5;
   NEWMODE Philosopher_Range = INT(0 : Philosophers_Count - 1);

   Enter_Restaurant : PROC(Who_Am_I Philosopher_Range IN)
      -- ...
   END Enter_Restaurant;

   Enjoy_Meal : PROC(Who_Am_I Philosopher_Range IN)
      -- ...
   END Enjoy_Meal;

   Leave_Restaurant : PROC(Who_Am_I Philosopher_Range IN)
      -- ...
   END Leave_Restaurant;

   Think : PROC(Who_Am_I Philosopher_Range IN)
      -- Uh-oh ...
   END Think:

   SYNMODE Forks_Mode = REGION SPEC
      GRANT Pick_Up_Pair, Put_Down_Pair;

      Pick_Up_Pair : PROC(Whose Philosopher_Range IN) END Pick_Up_Pair;
      Put_Down_Pair : PROC(Whose Philosopher_Range IN) END Put_Down_Pair;

      DCL Forks_Occupied EVENT;
      DCL Available ARRAY(Philosopher_Range) BOOL
        INIT := ((Philosopher_Range) := TRUE);
   END Forks_Mode;

   SYNMODE Forks_Mode = REGION BODY

      Pick_Up_Pair : PROC(Whose Philosopher_Range IN)
         DO WHILE NOT Available(Whose) ORIF
           NOT Available((Whose + 1) MOD Philosophers_Count);
             DELAY Forks_Occupied;
         OD;
         Available(Whose) := FALSE;
         Available((Whose + 1) MOD Philosophers_Count) := FALSE;
```

[15] Alternatively, we could put the life cycle into the constructor; this saves us the explicit wake up call.

```
      END Pick_Up_Pair;

      Put_Down_Pair : PROC(Whose Philosopher_Range IN)
         Available(Whose) := TRUE;
         Available((Whose + 1) MOD Philosophers_Count) := TRUE;
         CONTINUE Forks_Occupied;

         -- As intimated in the text, it is not clear which
         -- philosopher will be allowed to continue.
         -- A fair scheduler is required here.
      END Put_Down_Pair;
   END Forks_Mode;

   DCL Forks Forks_Mode;

   SYNMODE Philosopher = TASK SPEC
      GRANT Wake_Me_Up;

      Wake_Me_Up : PROC(Who_Am_I Philosopher_Range IN) END Wake_Me_Up;
   END Philosopher;

   SYNMODE Philosopher = TASK BODY
      Wake_Me_Up : PROC(Who_Am_I Philosopher_Range IN)
         DO FOR EVER;
            Enter_The_Restaurant(Who_Am_I);
            Forks.Pick_Up_Pair(Who_Am_I);
            Enjoy_The_Meal(Who_Am_I);
            Forks.Put_Down_Pair(Who_Am_I);
            Leave_The_Restaurant(Who_Am_I);
            Think(Who_Am_I);
         OD;
      END Wake_Me_Up;
   END Philosopher;

   DCL Gurus ARRAY(Philosopher_Range) Philosopher;

   DO FOR I IN Philosopher_Range;
      Gurus(I).Wake_Me_Up(I);  -- asynchronous call
   OD;
END Dining;
```

Program 3.2: The Dining Philosophers in CHILL

Note that this time, we have chosen to provide each philosopher with an identity by means of an explicit entry call, `Wake_Me_Up`. This is handy since we have to call `Wake_Me_Up` to start the tasks anyway.

It should not be too difficult to adapt Program 3.2 to the requirement of including Frederici in the scenario; again, an exercise for the inclined reader.

3.3 The Dining Philosophers in Erlang

3.3.1 Design

Erlang lacks passive entities that guarantee mutually exclusive access, which is why we must make use of processes for both the forks and the philosophers. This is somehow inconvenient since the forks themselves are passive—they are *used* to eat. Since the primary objective of concurrency is speedup, the question of efficiency is always present. The additional context switches that are required for the fork server process might incur runtime penalties. Furthermore, as the only means of communication between the philosophers and the forks is asynchronous message passing, we have to impose synchronization explicitly; a philosopher must of course know when their forks are free.

Luckily, we do not need to worry about the order of requests to acquire forks in Erlang. That is because all messages sent to a process are stored in the process' mailbox in FIFO order. Thus, if philosopher i makes an attempt to grab their forks, but if they are not free at the moment, then—later, when they are available,—philosopher i will be allowed to grab their forks—even if in the meantime, philosopher $i+1$ and/or philosopher $i-1$ themselves have (unsuccessfully) tried to pick up their own forks (you know, philosopher i's left fork is the right fork of philosopher $i+1$ (clockwise direction) and so on).

So what we will do is the following: we create a fork server, which manages the forks and handles the incoming requests from the philosophers. Should it be necessary for a philosopher to be blocked, then we can utilize Erlang's guarded receive statement. A philosopher sends their `pick_up_pair` request to the fork server and waits for an acknowledgement. The fork server checks whether the needed forks are free and, if so, grants them to the hungry guru by sending the acknowledgement. This way, the fork server works similarly to the active buffer in Program 1.9. Since putting the forks down is not critical, a philosopher can send this request to the fork server without waiting for a reply.

3.3.2 The Erlang Gurus

```
-module(dining).
-export([start/1, pick_up_pair/1, put_down_pair/1, forks/1,
  set_true/1, philosopher/1, traverse_and_start/1,
  eat/1, think/1]).

range(N, N) -> [N];
range(Min, Max) when Min < Max ->
   [Min | range(Min + 1, Max)];
range(Max, Min) when Max > Min ->
   range(Min, Max).

set_true(N) -> 1. %% the N is a dummy value, but it's needed
                  %% because of the way this function is used

traverse_and_start([H | T]) ->
   [apply(erlang, spawn, [dining, philosopher, [H]]) |
     traverse_and_start(T)];
```

```
traverse_and_start([]) ->
   [].

start(Philosophers_Count) ->
   register(fork_server,
     spawn(dining, forks,
       [list_to_tuple(lists:map({dining, set_true},
         range(0, Philosophers_Count - 1)))])),
   Id = range(0, Philosophers_Count - 1),
   Gurus = traverse_and_start(Id).

pick_up_pair(Whose) ->
   request({pick_up_pair, Whose}).

put_down_pair(Whose) ->
   fork_server ! {self(), {put_down_pair, Whose}}.
   %% we don't need an acknowledgement

request(Request) ->
   fork_server ! {self(), Request},
   receive
      {fork_server, Reply} ->
         case Reply of
            ok ->
               ok
         end
   end.

%% the forks

forks(Available) ->
   io:format("Look at the forks: ~w~n", [Available]),
   receive
      {From, {pick_up_pair, Whose}} when element(Whose + 1,
        Available) == 1, element(((Whose + 1) rem size(Available)) + 1,
          Available) == 1 ->
         From ! {fork_server, ok},
         forks(setelement(((Whose + 1) rem size(Available)) + 1,
           setelement(Whose + 1, Available, 0), 0));

      {From, {put_down_pair, Whose}} ->
         forks(setelement(((Whose + 1) rem size(Available)) + 1,
           setelement(Whose + 1, Available, 1), 1))
   end.

sleep(Time) ->
   receive
      after Time ->
         true
   end.
```

```
random() ->
   element(1, statistics(wall_clock)).
   %% Anyone have a better idea?  E-mail the author.

eat(Who_Am_I) ->
   sleep(((Who_Am_I + 1) * random()) rem 4000).

think(Who_Am_I) ->
   sleep(((Who_Am_I + 1) * random()) rem 5000).

philosopher(Who_Am_I) ->
   io:format("Philosopher ~w enters the restaurant~n", [Who_Am_I]),
   pick_up_pair(Who_Am_I),
   io:format("Philosopher ~w is not available (munch, munch!)~n",
     [Who_Am_I]),
   eat(Who_Am_I),
   put_down_pair(Who_Am_I),
   io:format("Philosopher ~w leaves the restaurant~n", [Who_Am_I]),
   io:format("Philosopher ~w is in deep meditation~n", [Who_Am_I]),
   think(Who_Am_I),
   philosopher(Who_Am_I).
```

Program 3.3: The Dining Philosophers in Erlang

Some of Erlang's concepts are peculiar. On the one hand, many things can be solved with striking elegance: to represent the interval $[a, b]$, for instance, five lines of Erlang code are sufficient. On the other hand, user-defined functions are not allowed in guard expressions (according to the Erlang people, this is largely for fear of side effects). That is why the author had to write such convoluted code in the fork server. It would have been nice if we could use `Fork_Available(Whose)` and such. Also, for the sake of readability, it is perhaps more beneficial to fill the array[16] `Available` with values such as `Occupied` vs. `Free` instead of with `0` vs. `1`.

How, the author wonders, do such restrictions get on well together with the claim that Erlang is a language that is intended for large industrial systems? Clearly, the issue of side effects is a serious one, but being overly conservative can prevent us from seeing obvious and straightforward solutions to a problem. Programmers are expected to be aware of the risks of side effects and to employ their skills carefully.

Perhaps notable is the way we start our gurus here. The line

```
Gurus = traverse_and_start(Id).
```

traverses the tuple $[0, 1, 2, \ldots,$ `Philosophers_Count` $- 1]$, and for each element of the tuple, a new philosopher process is started. Since the `spawn` function returns the Pid of the activated process, this approach makes `Gurus` a tuple that contains the Pids of the processes. This can be used to verify that the gurus have indeed been awakened. We can see that in a sample run of Program 3.3 (it is high time that we played again with the Erlang shell anyway):

[16] In Erlang, "array" is spelled "tuple," but the first index is always 1 while, for modulo computations, 0 seems to be more appropriate—another issue which adds to the confusion.

```
(lpc01) $ erl
Erlang (JAM) emulator version 4.7.3

Eshell V4.7.3  (abort with ^G)
1> c(dining).
{ok,dining}
2> dining:start(5).
Look at the forks: {1,1,1,1,1}
Philosopher 4 enters the restaurant
Philosopher 3 enters the restaurant
Philosopher 2 enters the restaurant
Philosopher 1 enters the restaurant
Philosopher 0 enters the restaurant
[<0.39.0>,<0.38.0>,<0.37.0>,<0.36.0>,<0.35.0>]
Look at the forks: {0,1,1,1,0}
Philosopher 4 is not available (munch, munch!)
Look at the forks: {0,1,0,0,0}
Philosopher 2 is not available (munch, munch!)
Philosopher 2 leaves the restaurant
Look at the forks: {0,1,1,1,0}
Philosopher 2 is in deep meditation
Look at the forks: {0,0,0,1,0}
Philosopher 1 is not available (munch, munch!)
Philosopher 4 leaves the restaurant
Look at the forks: {1,0,0,1,1}
Philosopher 4 is in deep meditation
Look at the forks: {1,0,0,0,0}
Philosopher 3 is not available (munch, munch!)
Philosopher 1 leaves the restaurant
Look at the forks: {1,1,1,0,0}
Philosopher 1 is in deep meditation
Look at the forks: {0,0,1,0,0}
Philosopher 0 is not available (munch, munch!)
BREAK: (a)bort (c)ontinue (p)roc info (i)nfo (l)oaded
       (v)ersion (k)ill (D)b-tables (d)istribution
(lpc01) $
```

Hence, the line

```
[<0.39.0>,<0.38.0>,<0.37.0>,<0.36.0>,<0.35.0>]
```

tells us that 5 processes have been started and it also reveals the Pids of these processes.

We conclude this section with the remark that Frederici's affairs can easily be incorporated into the setting. In order to avoid further overhead, it is perhaps more reasonable to extend the given fork server instead of introducing yet another layer of abstraction.

3.4 The Dining Philosophers in Java

3.4.1 Design

Java comes with everything we need to build the system. Threads will be used to implement the philosophers and a sequential class with synchronized methods will serve as the protection layer for the forks. With `suspend/resume` being deprecated in Java 1.2, we have to fall back on `wait/notify` to control the access to the forks. One drawback of this solution is that the wait set of an object's lock is unordered which requires that

- ◇ a guru who has been `notify`'ed check the status of their forks, see the discussion in Subsection 3.2.1 (page 57), and that
- ◇ the underlying machine be fair.

We can cope with the first challenge, possibly at the expense of efficient waiting.

3.4.2 The Java Gurus

They reside in `Dining.java`, but traces of their existence are clearly visible below.

```
class Forks_Guardian {

   public Forks_Guardian(int Philosophers_Count) {
      this.Philosophers_Count = Philosophers_Count;
      Available = new boolean[Philosophers_Count];
      for (int I = 0; I < Philosophers_Count; I = I + 1) {
         Available[I] = true;
      }
   }

   public synchronized void Pick_Up_Pair(int Whose) {
      while (!Available[Whose] ||
        !Available[(Whose + 1) % Philosophers_Count]) {
         try {
            wait();
         }
         catch (InterruptedException E) {}
      }
      Available[Whose] = false;
      Available[(Whose + 1) % Philosophers_Count] = false;
   }

   public synchronized void Put_Down_Pair(int Whose) {
      Available[Whose] = true;
      Available[(Whose + 1) % Philosophers_Count] = true;
      notifyAll();
   }

   private boolean[] Available;
   private int Philosophers_Count;
```

```
}

class Philosopher extends Thread {

   public Philosopher(int Who_Am_I) {
      this.Who_Am_I = Who_Am_I;
      this.Forks = Dining.The_Forks;
   }

   public void run() {
      while (true) {
         System.out.println("Philosopher " + Who_Am_I + " enters the" +
           " location");
         Forks.Pick_Up_Pair(Who_Am_I);
         System.out.println("Philosopher " + Who_Am_I + " finds that" +
           " the noodles are really soggy");
         try {
            sleep((int) Math.random() * 1000);
         }
         catch (InterruptedException E) {}
         Forks.Put_Down_Pair(Who_Am_I);
         System.out.println("Philosopher " + Who_Am_I + " leaves");
         System.out.println("Says philosopher " + Who_Am_I + ",");
         System.out.println("``A great many people think they are" +
           " thinking when they are merely");
         System.out.println("rearranging their prejudices.''");
         System.out.println("                -- William James");
         try {
            sleep((int) Math.random() * 3000);
         }
         catch (InterruptedException E) {}
      }
   }

   private int Who_Am_I;
   private Forks_Guardian Forks;
}


class Dining {

   static Forks_Guardian The_Forks;

   public static void main(String[] Args) {
      int Philosophers_Count = new Integer(Args[0]).intValue();
      The_Forks = new Forks_Guardian(Philosophers_Count);
      Philosopher[] Gurus = new Philosopher[Philosophers_Count];

      for (int I = 0; I < Philosophers_Count; I = I + 1) {
         // up and away
         Gurus[I] = new Philosopher(I);
```

```
            Gurus[I].start();
        }
    }
}
```

Program 3.4: The Dining Philosophers in Java

This time, we have decided to provide each philosopher with their identification by means of a constructor.

The question as to whether it is more efficient to use `notify` instead of `notifyAll` in `Put_Down_Pair`[17] is not quite appropriate. Note that in both cases, at most one thread is allowed to grab the lock; the only difference between the two techniques is that in the case of `notifyAll`, *all* threads of the wait set are scheduled/made runnable. But only the lucky one is dispatched to the processor and allowed to *execute* the code of `Pick_Up_Pair`. The use of `notify` results in but one thread to be scheduled and subsequently dispatched. Although it stands to reason that scheduling all threads versus scheduling only one is more expensive, this is not the primary issue concerning efficiency. Definitely far more performance-degrading is the aspect of busy waiting that arises from the necessity that every notified philosopher must check their forks in a `while` loop. The root of all evil is the lack of an ordering of the wait set, of course. From that viewpoint, the use of either `notifyAll` or `notify` is merely a matter of taste. It is a fallacy to assume that this choice will help increase the performance significantly.

3.5 Summary and Comparison

The problem of the Dining Philosophers is a classical exercise for concurrent programming. It can be stated completely without mentioning computers at all, yet it contains many aspects of importance for this branch of computer science, including

- ◇ **Concurrency**—each philosopher is an entity with a thread of control
- ◇ **Access to critical resources**—a pair of forks can only be used by one philosopher at a time
- ◇ **Synchronization**—a philosopher must wait until both their forks are free

Because each philosopher is thought to be immortal, a number of additional requirements are imposed. The solution should be

- ◇ **free from deadlock** and
- ◇ **free from starvation**

The objective of this chapter was to present solutions that satisfy these criteria in our four languages. After initial discussions about Frederici's role in the scenario, we have come to the core of the problem and addressed ways to tackle it. Deviating from the usual pattern, we will now examine each language in turn and discuss the relevant points. Table 3.1 will then give a concise summary.

[17] or, for that matter, in Program 1.10

3.5.1 The Solution in Ada

On the conceptual level, Ada is ideally suited to implement the exercise. The philosophers (and, perhaps, Frederici) can be mapped onto tasks, and the forks are best represented by a protected object; so there is no problem. Neither is there one concerning the topic of deadlock—each guru acquires their forks in an all-or-nothing manner: they either pick them up or they do not. A situation in which (at least) one philosopher holds only a single fork and waits for the other one to be freed cannot occur. And since, having finished the meal, a philosopher will put down their forks, another guru will eventually be allowed to eat. The nightmare described on page 51 is avoided.

From the absence of deadlock follows the presence of liveness; the simple life cycle of each philosopher is a cyclic program and the respective desired actions can potentially be executed indefinitely. Starvation is avoided because the only way to starve `Philosopher(J)` is if the neighboring philosophers are able to repeatedly call `Pick_Up_Pair` and `Release_Pair`. Assume that the neighbors have obtained their forks. Then `Philosopher(J)`'s unsuccessful attempt will result in `Philosopher(J)` being placed into the `Please_Get_Pair` queue, which is served in FIFO order. At some point in time, the neighbors will put down their forks by calling `Release_Pair`. This, however, causes the availability condition tested in `Please_Get_Pair` to become true. Therefore, `Philosopher(J)` is permitted to grab their forks. The semantics of protected objects prescribes that pending calls whose barriers have become true (and `Please_Get_Pair`'s barrier has become true) must be serviced before any new protected action is allowed to start. Thus, `Philosopher(J)` will be able to call `Please_Get_Pair` before *any* other philosopher (the two neighbors in particular) is allowed to call `Pick_Up_Pair` again. And since both neighbors have put down their forks, `Philosopher(J)` can actually start eating. "The key is that there is basically only one queue, for `Please_Get_Pair`, and the default queuing policy is FIFO. Since the `Release_Pair` epilogue services pending callers on `Please_Get_Pair` while the protected object is still locked, `Philosopher(J)` is guaranteed to be awakened as a side effect of the neighbor's call to `Release_Pair`." (Brosgol, 1996).

Had we made use of the entry family approach outlined on page 53, things would be more complicated. Since we have as many `Pick_Up_Pair` entries as there are philosophers, we also have more than one entry queue—we have one per `Pick_Up_Pair` entry. This, of course, makes the nice feature of the FIFO ordering of entry queues in Ada useless since each `Pick_Up_Pair` queue contains at most one caller at a time. The individual queues *are* FIFO ordered, but the philosophers waiting for access to their forks (not for access to the protected object) are not. Thus, we might have some trouble with the claim that the order in which philosophers—whose forks are free—are granted their forks reflects the order in which they have made the corresponding requests. If all the gurus have the same priority (which they will), then the default scheduling policy does not specify which queue is to be served first should there be calls from two or more queues simultaneously eligible for selection, and the `Priority_Queuing` policy defined in (ARM, 1995, D.4(12)) prescribes that the scheduler give preference to the entry with the lowest index. This situation is somewhat awkward and requires either a fair machine/a solution with an explicit "politeness" policy ensuring that each fork request is timestamped and the forks are granted based on these timestamps or, in case of `Priority_Queuing`, the use of priorities, which is distracting. The use of the entry family techniques bears the risk of starvation.

With the `requeue` approach, everything is fine since there is only one entry queue (for there is only one `Pick_Up_Pair` entry), which is FIFO-ordered. Another good thing

is that we do not need to care about a fair machine—the order of requests is always preserved. Though an awakened philosopher must check the status of their forks, the risk of busy waiting is considerably reduced by the FIFO ordering of the entry queues.

We conclude that the Ada solution is fully compliant with the requirements.

3.5.2 The Solution in CHILL

Using tasks and a region for, respectively, the gurus and the forks, the principal solution is easily achieved. This time, not even the asynchronous nature of calls to task objects are woes—they are handy here, indeed! Citing the same reasons as in Subsection 3.5.1, we sustain the claim that deadlock is avoided and liveness is guaranteed. The lack of an ordering of the wait set of a signal (which was used in the region to achieve the blocking) makes it necessary that each philosopher poll the status of their forks. Since a `CONTINUE` issued by a guru who has just finished the meal does not necessarily wake up the philosopher waiting for one of the forks just put down, the awakened philosopher must verify that their forks are indeed free. This is not difficult to achieve, but it gives rise to busy waiting.

But the unordered wait set of the signal has even more negative consequences: there is no order in the requests to obtain a fork. Thus, starvation is possible and this leads to the belief that a fair machine is required. The details of scheduling are blurred in CHILL, but at least an implementation is allowed to provide its own scheduler. So, as in the case of Ada, we could use a fair scheduler,[18] or play around with timestamped requests or priorities. But again, the unclear semantics of priorities is likely to be an obstacle.

Another workaround would be to require each philosopher to register their call to `Pick_Up_Pair` to a server which stores them in FIFO order and have this server regulate the allocation of the forks to the philosophers based upon this ordering. We would get rid of the problem, but this registration server is not pre-existent in CHILL: this solution is thus against the spirit of this paper, as we compare *existing* features of concurrency—the set of which in CHILL must be regarded as being insufficient for our purpose.

We cannot meet all requirements of the problem.

3.5.3 The Solution in Erlang

Two things spring to mind: to achieve mutually exclusive access to data, Erlang offers only processes, which implies that the forks must be protected by an active entity. While this is not a threat to the functional correctness of the solution, it could be perceived as a sort of wrong abstraction; the forks are passive. It can be perceived as a cause for runtime overhead. An issue which poses a threat to the functional correctness of a system (not necessarily ours) is that the use of user-defined functions is severely restricted in areas where it is most beneficial, sigh!

Also, the communication—which is asynchronous in Erlang—between the gurus and their fork server requires more effort. We must explicitly make certain that a philosopher is made aware of the fact that their forks are free, that is, communication effort is needed to impose synchronization upon the message passing.

As before with Ada and CHILL, deadlock is not an issue but liveness is, and fortunately, starvation is avoided, and we do not need to worry about a fair machine. The

[18] provided that we have one at hand

FIFO mailbox of the fork server caters for this. If we look closely enough at this topic, we realize that there is not even a wait set of philosophers waiting for access to the server (not to the forks). If they wish, then all philosophers can make a fork request at the same time—there is no contention in sending messages to the fork server. Of course, requests that arrive simultaneously at the fork server are serialized (as they must be entered into a FIFO queue) by the runtime system. But at this time, all the requests so made are successful from the perspective of the philosophers as they are either waiting for an acknowledgement (in the case of `pick_up_pair`) from the server or they are leaving the restaurant.

All requirements pertaining to the problem can be met (but with less elegance than in Ada). One possible advantage of the active fork server is that there is no busy waiting involved. We conclude from this that a solution free from busy waiting could be built in Ada and CHILL by using a fork task, too.

3.5.4 The Solution in Java

With threads for the gurus and a sequential class with synchronized methods for the protection of the forks, we have been able to come up with a solution to the problem. As in the case of CHILL, the absence of an ordering of the wait set of the philosophers that have themselves suspended via `wait` makes it necessary that each awakened philosopher check the status of their forks. Hence, busy waiting might ensue. Thus, starvation cannot be ruled out (as for deadlock and liveness, refer to the preceding subsections on Ada, CHILL, and Erlang). In Java, things are in particular problematic as the language definition remains almost completely silent about the issue of scheduling and dispatching. Not only is the defining document contradictory (vague at best) as regards the effect of priorities (see (JLS, 1996, 17.12)), but it also seems to give no permission to an implementation to provide its own scheduling mechanisms. If starvation is observed, then the features of concurrency Java provides directly are not sufficient to cope with that problem. Additional effort by the programmer is needed to, for instance, use timestamped request or implement the workaround proposed at the end of Subsection 3.5.2.

We have to concede that the full repertoire of requirements cannot be satisfied.

Asked to give a ranking as regards the compliance with the requirements, the author would suggest

1. Ada
2. Erlang
3. CHILL
4. Java

To end the chapter and to provide the remaining details such as program size, readability, etc., refer to Table 3.1 below.

Language	Ada	CHILL	Erlang	Java
Readability	3^+	3	3	3^+
Program Size	98	62	74	72
Communication Effort	n.a.	n.a.	2	n.a.
Degree of Concurrency	NoG*	NoG	NoG+1	NoG
Starvation	No	Possible	No	Possible

* Number of Gurus

Table 3.1: Tabular Summary of Chapter 3

4 Interrupt Handling

The Roman Rule: The one who says it cannot be done should never interrupt the one who is doing it.

Introduction and Problem Description

When developing embedded systems, one is often faced with the difficulty of designing and implementing device drivers. A device driver is a component of a system that is responsible for controlling access to some external device by manipulating device registers and responding to interrupts.

An interrupt itself is an occurrence of an event which requires special attention by the system. We distinguish between *external* and *internal* interrupts. An external interrupt is, for example, a signal sent from a device (to the device driver) which indicates that the device is ready to transmit data. A clock interrupt, an update request from a graphical widget, or a keystroke by the user (assuming that a keyboard is an integral part of a computer) are examples of internal interrupts. It is important to realize that an interrupt occurrence is an asynchronous matter. We cannot say beforehand when one of our sample interrupts from above will occur.

As we are talking about high-level features, we omit the description of the general (low-level) model of interrupt handling. A good book on computer science will be helpful in this respect. From our standpoint, we would like to have a particular routine, called an *interrupt handler* or *interrupt service routine*, be executed as a response to the occurrence of an interrupt. In other words, one actor or the hardware system sends a notification and we are supposed to react.

We can imagine this routine to be executed by a hypothetical actor in the case of software interrupts. With hardware-generated interrupts, however, it is often more beneficial in terms of efficiency to execute the handler directly by the hardware, i.e., the interrupt handler is neither called by the runtime nor operating system in this case.

Thus, given a particular interrupt occurrence, we have to arrange for our interrupt handler code's start address to be that of the interrupt service routine that belongs by default to this interrupt (hence, we must override the default handling). Upon occurrence of said interrupt, then, the interrupt handling code is automatically executed by being invoked as a subprogram (often, a simple jump is made to the very address, which results in the interrupt handler being executed).

The intention of this chapter is to consider how our languages support the handling of interrupts.

4.1 Interrupt Handling in Ada

In Ada 83, task entries were used to implement interrupt handlers. With the help of a special representation clause, namely `X'Address`, with `X` being a task entry, the entry could be placed at the correct address location. This feature is now considered obsolete, (ARM, 1995, J.7.1), and the preferred method is to use a protected procedure which has been

marked as an interrupt handler. Nevertheless, the task entry approach is still retained for downward compatibility with Ada 83 applications. As a consequence, the programmer is offered two ways of implementing interrupt handlers. This section, however, will only concentrate on the modern approach, but only after we have looked at Ada's model of interrupts more closely. The description can be found in Annex C, Systems Programming, of (ARM, 1995).

4.1.1 Ada's interrupt model

Reading (ARM, 1995, C.3), we learn that:

"An *interrupt* represents a class of events that are detected by the hardware or the system software. Interrupts are said to occur. An *occurrence* of an interrupt is separable into generation and delivery. *Generation* of an interrupt is the event in the underlying hardware or system that makes the interrupt available to the program. *Delivery* is the action that invokes part of the program as [a] response to the interrupt occurrence. Between generation and delivery, the interrupt occurrence (or interrupt) is *pending*. Some or all interrupts may be *blocked*. When an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered.

Certain interrupts are *reserved*. The set of reserved interrupts is implementation defined. A reserved interrupt is either an interrupt for which user-defined handlers are not supported, or one which already has an attached handler by some other implementation-defined means. Program units can be connected to non-reserved interrupts. While connected, the program unit is said to be *attached* to that interrupt. The execution of that program unit, *the interrupt handler*, is invoked upon delivery of the interrupt occurrence.

While a handler is attached to an interrupt, it is called once for each delivered occurrence of that interrupt. While the handler executes, the corresponding interrupt is blocked. While an interrupt is blocked, all occurrences of that interrupt are prevented from being delivered. Whether such occurrences remain pending or are lost is implementation defined. Each interrupt has a *default treatment*, which determines the system's response to an occurrence of that interrupt when no user-defined handler is attached. The set of possible default treatments is implementation defined, as is the method (if one exists) for configuring the default treatments for interrupts."

Perhaps notable is the fact that an exception propagated from a handler which is invoked by an interrupt has no effect. The rationale behind this requirement is easily explained: given the asynchronous nature of interrupts, it is not clear which exception handler, if any, is in charge when the exception is being propagated from the corresponding handler. Incidentally, this rule is modelled after the rule about exceptions propagated out of task bodies. Also note that it is regarded as a bounded error to call `Ada.Task_Identification.Current_Task`, which returns a value identifying the calling task, from an interrupt handler. "A consequence of direct hardware invocation of interrupt handlers is that one cannot speak meaningfully of the 'currently executing task' within an interrupt handler." (Rat, 1995, C.3)

4.1.2 Interrupt Handlers in Ada

The idea is to use a protected procedure with a parameterless[19] profile as an interrupt handler. The corresponding protected object, given the asynchronous nature of interrupt occurrences, must be at library level. This is because when a protected object's scope is left, it is finalized. This includes the detachment of interrupt handlers, if any, from protected procedures, thereby restoring the handler previously in effect.

The Ada standard encourages an implementation to use a different mechanism for invoking a protected procedure in response to a hardware interrupt than is used for a call to that protected procedure from a task. Given the possible frequency of interrupt occurrences and the requirement to deliver them promptly, it is often more beneficial to call the interrupt handler directly by the hardware. As pointed out earlier, this is, in principle, achieved by placing the handler at the address of the hardware interrupt vector associated with the interrupt. The `Address` representation clause can be used for this purpose, but there is a better way in Ada 95. *Better*, here, means more portable. The expression in the `Address` representation clause is clearly machine-dependent and has to be changed each time the program is to be re-targeted.[20] Why not let the compiler do this job? Before we will see how this can be done, we should point out that there are *two* ways in which a handler can be attached to an interrupt. The programmer can choose between *permanent* and *dynamic* attachment.

Permanent Attachment of Interrupt Handlers

The pragma `Attach_Handler(`*Handler*`_Name, Expression)` is used to attach a handler permanently to an interrupt. The *Handler*`_Name` is expected to denote a protected procedure with a parameterless profile. The expected type for the `Expression` is `Ada.Interrupts.Interrupt_ID` (see below). The pragma itself is only allowed immediately within a protected definition, which—as explained—must be at library level.

Upon creation of a corresponding protected object, the handler is attached to the interrupt specified. If the interrupt is reserved, `Program_Error` is raised and the existing treatment is not affected.

Armed with this knowledge, we can give a brief example (taken from (Rat, 1995, C.3)) of the ideas presented above:

```
protected Alarm is
   procedure Response;
   pragma Attach_Handler(Response, Alarm_Int);
end Alarm;

protected body Alarm is
   procedure Response is
   begin
      -- what to do when it rings
   end Response;
end Alarm;
```

[19] an implementation is allowed to provide, additionally, protected procedures with parameters

[20] Portability, one has to admit, is of less crucial importance in embedded systems, though.

`Alarm_Int` is a fictitious name thought to identify a physical interrupt.

Dynamic Attachment of Interrupt Handlers

For this purpose, the pragma `Interrupt_Handler(`*Handler*`_Name)` has been included into the set of facilities. Again, the *Handler*`_Name` is expected to denote a protected procedure with a parameterless profile. The pragma itself is only allowed immediately within a protected definition.

Using this pragma marks a protected procedure as an interrupt handler (as does the `Attach_Handler` pragma) *and* prepares it to be attached, detached, and replaced by another handler dynamically. This functionality is offered by the package `Ada.Interrupts`, which is part of Annex C and which is reprinted verbatim below:

```
with System;
package Ada.Interrupts is
   type Interrupt_ID is implementation-defined;
   type Parameterless_Handler is
      access protected procedure;

   function Is_Reserved (Interrupt : Interrupt_ID)
      return Boolean;

   function Is_Attached (Interrupt : Interrupt_ID)
      return Boolean;

   function Current_Handler (Interrupt : Interrupt_ID)
      return Parameterless_Handler;

   procedure Attach_Handler
      (New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);

   procedure Exchange_Handler
      (Old_Handler : out Parameterless_Handler;
       New_Handler : in Parameterless_Handler;
       Interrupt   : in Interrupt_ID);

   procedure Detach_Handler
      (Interrupt : in Interrupt_ID);

   function Reference(Interrupt : Interrupt_ID)
      return System.Address;

private
   ... -- not specified by the language
end Ada.Interrupts;


package Ada.Interrupts.Names is
    implementation-defined : constant Interrupt_ID :=
       implementation-defined;
```

```
      ...
    implementation-defined : constant Interrupt_ID :=
       implementation-defined;
end Ada.Interrupts.Names;
```

The semantics of most operations in this package can be inferred from the respective names. We will concentrate on the vital points only. It is important that each protected procedure intended to be used as an interrupt handler be marked as such a handler, via the pragma `Interrupt_Handler`. If it is not, then `Program_Error` is raised by `Attach_Handler` and `Exchange_Handler`. Operations that are not allowed on reserved interrupts result in `Program_Error` being raised as well. Attempts to mix static and dynamic attachment of handlers to interrupts, i.e., using the procedures `Attach_Handler`, `Detach_Handler`, or `Exchange_Handler` on handlers statically attached (via the pragma `Attach_Handler`) are thwarted by raising `Program_Error`, too. In all cases where `Program_Error` is raised, the current interrupt treatment is not changed.

The `Reference` function returns a value of the type `System.Address`. This function can be used to bridge the gap between the interrupt handling of Ada 83 and Ada 95. The value delivered can be used to attach a task entry to an interrupt "manually."

There exists a child package of `Ada.Interrupts`, `Ada.Interrupts.Names`, which contains implementation-defined names for the interrupts that are supported by the implementation for a particular hardware configuration. To see how this looks like in an actual implementation of Ada, the author has reprinted below a couple of lines of said package as it appears in the GNAT compiler for a typical Unix environment:

```
package Ada.Interrupts.Names is
   SIGILL : constant Interrupt_ID :=
     System.OS_Interface.SIGILL;     -- illegal instruction (not reset)

   SIGFPE : constant Interrupt_ID :=
     System.OS_Interface.SIGFPE;     -- floating point exception

   SIGBUS : constant Interrupt_ID :=
     System.OS_Interface.SIGBUS;     -- bus error

   SIGSEGV : constant Interrupt_ID :=
     System.OS_Interface.SIGSEGV;    -- segmentation violation

   SIGALRM : constant Interrupt_ID :=
     System.OS_Interface.SIGALRM;    -- alarm clock

   SIGURG : constant Interrupt_ID :=
     System.OS_Interface.SIGURG;     -- urgent condition on IO channel

   SIGPOLL : constant Interrupt_ID :=
     System.OS_Interface.SIGPOLL;    -- pollable event occurred

   SIGIO : constant Interrupt_ID := -- input/output possible,
     System.OS_Interface.SIGIO;      -- SIGPOLL alias (Solaris)

   SIGVTALRM : constant Interrupt_ID :=
     System.OS_Interface.SIGVTALRM; -- virtual timer expired
```

```
      SIGPROF : constant Interrupt_ID :=
        System.OS_Interface.SIGPROF;   -- profiling timer expired

      SIGXCPU : constant Interrupt_ID :=
        System.OS_Interface.SIGXCPU;   -- CPU time limit exceeded
   end Ada.Interrupts.Names;
```

The `System.OS_Interface.SIGXYZ`s are constants which represent signals to be used in the ANSI C signal handling.

Our example from above can be rewritten, using dynamic attachment of interrupts (and an actually existing interrupt):

```
with Ada.Interrupts.Names;

protected Alarm is
   procedure Response;
   pragma Interrupt_Handler(Response);
end Alarm;

protected body Alarm is
   procedure Response is
   begin
      -- what to do when it rings
   end Response;
end Alarm;

Attach_Handler(Alarm.Response'Access, Ada.Interrupts.Names.SIGALRM);
```

4.2 Interrupt Handling in CHILL

The author has spent some time studying the defining document for CHILL 96, (Z200, 1996). Despite the effort undertaken, he was not able to spot anything appropriate for interrupt handling in CHILL. Even the set of implementation defined features, quite long a list in CHILL, did not bear even the faintest hint as regards this matter.

Remembering the compiler we have at hand, which comes with a truckload of documentation, the author has spent even more time scrutinizing a particular implementation of CHILL. Regrettably, to no avail.

4.3 Interrupt Handling in Erlang

The outcome of examining (Armstrong, 1996) was not encouraging. Erlang, too, seems to have no provisions for interrupt handling. However, to be fully assured,[21] the author sent

[21] The author knows of at least one incident where the defining document for Erlang and one actual implementation (by the authors) diverged.

an e-mail to the Erlang people the other day, asking them to straighten out the matter once and for all. The result of said inquiry is daunting. As it happens, software close to the hardware has to be implemented outside Erlang, for example, in C. This software then communicates with the control system (written in Erlang). The communication can basically be implemented in three ways:

1. Through ordinary TCP/UDP sockets.
2. Pipes. Erlang can start an executable program by using the `open_port` Built In Function (BIF). This will result in a new process being created (by `fork()`). The executable program is started (`exec()`) and communication takes place through pipes (created before the `fork()`).
3. A linked-in driver. A program written in C which conforms to the driver interface specified in the appendix of (Armstrong, 1996) can be dynamically linked into the Erlang emulator. The Erlang interface is still the `open_port` BIF. The difference to 2. is that the linked-in software now executes in the same process as the Emulator. Compared to 2., this method avoids context switches, which may give speed improvements. On the other hand, this method makes the Erlang emulator vulnerable (a bug in the linked-in driver will crash the Emulator).

We conclude that there are no inherent features for interrupt handling in Erlang. The suggestions above are a workaround at best.

4.4 Interrupt Handling in Java

Reading the initial lines of (JLS, 1996), it becomes obvious at once that the intended application area for Java is not (or, perhaps, no longer) that of embedded systems. It is thus not surprising that Java lacks means which help tackle the problems arising in this domain, interrupt handling in particular.

However, it would be inappropriate to end the chapter right here. Given how important Java has become in recent years, it is worthwhile looking beyond the horizon. Prompted by an article in (CACM, 1998), which has also been briefly mentioned in (Brömel, 1998), the author has decided to look further at the topic of PERC. The acronym is short for Portable Executive for Reliable Control, and is a programming language fully complying with the Java standard set forth by Sun Microsystems and intended to supplement Java with additional functionality. PERC has been developed by NewMonics Inc., `www.newmonics.com`. Besides offering features for realtime programming, which will be of interest in the next chapter, it provides facilities for interrupt handling, and it is the remainder of this chapter which will be devoted to these. The standard reference for PERC can be found in (Nilsen, 1998) and will be used heavily in what follows.

4.4.1 Interrupt Handling in PERC

PERC offers two packages to assist the programmer in the development of realtime systems: `com.newmonics.Embedded` and `com.newmonics.RealTime`. The latter contains

abstractions relevant to the budgeting of resources or the representation of realtime activities, and will be of interest in the next chapter. The `Embedded` package holds the tools we need to program embedded systems. The means include support for I/O ports, interrupt vectors, and persistent memory.

The idea is that a programmer utilizes the class `Interrupt` and the interface `ISR`, which obviously stands for interrupt service routine, to achieve the desired effect. The `ISR` interface has, as its sole component,[22] the following method

```
void handler()
```

which is to be implemented by the programmer with the code that is to be executed in response to an interrupt. The `handler()` method is then automatically invoked by the corresponding `Interrupt` object each time the interrupt occurs (hardware or software).

This is sufficient to create our own interrupt service routine. It remains to find out how we can couple an implementation of an `ISR` interface, i.e., an interrupt handler, with a particular interrupt. This is accomplished via the class `Interrupt`, which is an abstract representation of system interrupt vectors. The combination of an `ISR` object and an `Interrupt` object allows programmers to arrange for a Java thread to execute the interrupt handling code. To see how we can do this, let us look at `Interrupt`'s constructor, the semantics of which is described as follows in (Nilsen, 1998, p. 41):

> `public Interrupt(DeviceCapability token, int vector_number, ISR routine)`
>
> "Creates an `Interrupt` object representing the specified interrupt number and arranges for the interrupt to be serviced by the `ISR` object. The system executive may do a security manager and possibly other checks to make sure this is an acceptable request. If the system executive decides that the requested operation is not allowed, it throws a `SecurityException`. Each time the hardware interrupt is raised, the system executive arranges to execute the `routine.handler()` method. The thread that executes the `handler()` method has `Thread.NORM_PRIORITY` priority."

Note that a second constructor is offered which allows to adjust the priority "manually" via an additional parameter.

`DeviceCapability` is a class that represents permissions to access a particular device in a particular address range. The system executive ensures that only one `DeviceCapability` object exists for each resource. Thus, by acquiring a `DeviceCapability` object and making it private, a software component can make certain that it alone has permission to make use of particular interrupts or I/O ports. There are no public constructors or methods for `DeviceCapability` objects.

To find out which interrupt is mapped onto what `vector_number`, PERC offers the class `DeviceRegistry`. This is a clearing house for information regarding the devices (e.g., I/O ports and interrupts) that are associated with a particular hardware configuration. A lookup service is provided (via the `lookup()` method), which helps determine which devices are available in the system and where they are located within the address space. The device registry for a particular hardware configuration is accessible via the `static` constant

[22] Sadly, (Nilsen, 1998) does not contain an exhaustive listing of `ISR` (or, for that matter, of any of the classes/interfaces that make up PERC). The members are scattered throughout the paper, which is why the author suggests that said paper be read as a source of primary information. It can be downloaded from NewMonics' web site.

```
public static final DeviceRegistry registry
```

To obtain a `DeviceCapability` object for the use as the `token` parameter in the `Interrupt` constructor, the `DeviceRegistry` class offers the `acquire()` method.

Both the `lookup()` and the `acquire()` method take a string as their sole parameter which is meant to identify a particular device. An example of such a name is `com.newmonics.serial.com1.addr`, which might represent a fictitious serial device driver developed by NewMonics.

This describes the primary API to the handling of interrupts in PERC. The remaining methods in the `Interrupt` class are only mentioned briefly for the sake of completeness. To find out which `ISR` object is attached to a given `Interrupt` object, the method `isr()` can be used. The number of interrupt occurrences (of a particular interrupt) currently pending is returned by `pendingCount()` and can be reset with `clearPendingCount()`. Should we wish to execute the interrupt handler even if no interrupt is currently being serviced, we can do so by invoking the `trigger()` method, which—in turn—will invoke the `handler()` method.

Systems that support both the `Embedded` and the `RealTime` package, offer programmers the possibility to combine a `HardwareTriggeredTask` from the `RealTime` package with the interrupt handling capabilities provided by the `Embedded` package. The approach is to have particular realtime sporadic tasks be triggered by hardware interrupts. Although the details of the `RealTime` package are the subject of the next chapter, we can have a quick look at it. The class `HardwareTriggeredTask`, which is part of the `RealTime` package, has the following constructor

```
public HardwareTriggeredTask(DeviceCapability cap, int vector)
```

which creates a hardware triggered task that is to be triggered by hardware interrupt `vector`. The second parameter, `cap`, is used to represent permissions regarding `vector`.

This completes the description of the interrupt handling facilities of PERC and our first excursion into this language. A second, more elaborate one will be undertaken in the final chapter of this paper, when we will occupy ourselves with the problem of process scheduling.

4.5 Summary and Comparison

It is widely agreed that the area of embedded systems is a key application domain for concurrent programming. It is, however, an area which entails dealing with hardware specific details and configurations. Programs must be mapped onto particular hardware architectures and developers must give much more attention to low-level characteristics. One of the challenges that is being posed is the handling of interrupts: events that occur possibly at random must be recognized and dealt with. In former times, these embedded systems were the stronghold of assembler languages. The apparent disadvantages of this approach have led to the desire to equip high-level programming languages with facilities that help tackle the problem. This transition process is still under way.

This is, also, evidenced by the fact that, strictly speaking, only one—viz. Ada—of our languages provides features for interrupt handling. Although CHILL and Erlang claim to be suitable for large industrial systems, the author has not been able to identify

the facilities relevant to interrupt handling in these two languages. An alternative to interrupt-driven processing is status-driven processing, also known as polling, which has the advantages of being inexpensive and of not adding to the non-deterministic behavior of the system (which is sometimes required on safety grounds).

With Java, things are a bit different; its preferred application domain is the World Wide Web now, and after re-targeting the initial language proposal (which was intended to serve in the development of embedded systems), the language designers no longer feel the need to support the branch of embedded systems.

Having only one language in this chapter, then, has struck the author as peculiar, and he has therefore decided to look at a Java dialect intended to support the development of realtime systems. The two languages to compare are thus Ada and PERC.

The first thing that is convenient is that Ada and PERC use *existing* language features to approach the topic of interrupt handlers. In Ada, a protected procedure is used, and PERC provides a class and an interface to achieve the effect. Thus, there is no new syntax involved; the concept of an interrupt handler fits seamlessly into the set of concepts offered by the two languages in general.

Note that Ada carefully distinguishes between dynamic and permanent attachment of interrupt handlers (and forbids mixing the two). It allows handlers to be attached, detached, and exchanged. In order to exchange a handler in PERC, we must first detach the old handler and then associate a new one with the interrupt object. However, there is no explicit way to detach a handler in PERC; we have to create both a new `ISR` and a new `Interrupt` object and combine the two. In this sense, we do not really exchange the handler.

The idea that it might be more efficient to have an interrupt handler be executed directly by hardware support can be found in both languages. In both languages, interrupt handlers can be called "by hand," via a call to the corresponding protected procedure (Ada) vs. by means of the `trigger()` method of the `Interrupt` object (PERC).

By and large, both approaches to interrupt handling appear to be straightforward, and the author does not see any reason why either of the two languages should be superior to the other in this respect.

5 Process Scheduling

There cannot be a crisis next week. My schedule is already full.
-- Henry Kissinger

Introduction and Problem Description

In this final chapter, we will occupy ourselves with the topic of process scheduling. Having said in Chapter 0 that this paper is going to be an extension of (Brömel, 1998), we can now be even more precise. This chapter is a sequel of the sixth chapter of (Brömel, 1998). In said chapter, we have examined how three of our example languages—Ada, CHILL, and Java—support realtime programming.[23] We have looked at the issue of priorities and their effect on scheduling and dispatching, and have addressed the topic of deadline specification for actors. The generic term we have used there was *temporal scope* for time-related parts of an actor's behavior such as startup time, maximum execution time, period, frequency, etc. The idea is that by specifying temporal scopes as part of an actor's definition/creation, it should be possible to tell the system executive that these actors must execute within their timing constraints. For example, if we state that an actor is to be triggered every 20 μs, then we would like the system executive to take precautions so that this actually happens.

Unfortunately, none of our sample languages in (Brömel, 1998) would support the specification of temporal scopes. Workarounds had to be used—in the case of the periodic actor from above, a loop containing the relevant actions and an appropriate sleep statement. PEARL, however, proved to be adequate for the specification of temporal scopes. On the other hand, Java was extremely unhelpful since it was not even really possible to provide workarounds. Why? A basic requirement for the workarounds is that we must be able to suspend an actor for at least a given amount of time. The `sleep()` method in Java does not guarantee that, however, for there is the possibility that a sleeping thread is reawakened prematurely by an interrupt sent to it by another thread.[24]

Nothing has changed in the Ada and CHILL world since the author wrote the sixth chapter of (Brömel, 1998); so there cannot be a news update. But there have been tremendous changes taking place in the Java world—PERC has seen the light of the day, and, as promised in Chapter 4, we will take our second journey into the realm of this language. And, of course, there is Erlang.

5.1 Process Scheduling in Erlang

Section 5.6 of the Erlang manual sheds light on process scheduling, realtime topics, and priorities. After mentioning that scheduling is an implementation-dependent issue, we learn that

- "The scheduling algorithm must be *fair*, that is, any process which can be run will be be run; if possible, in the same order as they become runnable."

[23] Note that this chapter has also featured a brief presentation of PEARL.

[24] Readers should not confuse the concept of the method `Thread.interrupt()` with the material presented in Chapter 4.

- "No process will be allowed to block the machine for a long time. A process is allowed to run for a short period of time, called a *time slice*, before it is rescheduled to allow another runnable process to be run."

A time slice, in Erlang, is set to permit the process to perform about 500 reductions. *Reduction* is Erlang parlance for "function call." The manual concludes the topic of scheduling by stating that any scheduling algorithm which meets the above criteria is good enough for Erlang.

For the sake of completeness, let us look at priorities as well. There are only two priorities in Erlang, `normal` and `low`, with the former being the default for all processes. Runnable processes with priority `low` are run less often than those with `normal` priority. To change the priority of a process, the BIF `process_flag` is handy:

```
process_flag(priority, Pri)
```

It causes the priority of the executing process to become `Pri`, which—as mentioned—can be either `normal` or `low`. Since `process_flag` is used for other things as well, `priority` serves as a sort of switch here.

This is all we can say about process scheduling in Erlang.

5.2 Process Scheduling in PERC

Before we can actually start, we shall give an introduction to the PERC philosophy. We will learn about the PERC realtime executive and its model of execution, resource negotiation, additional syntax elements for `timed` and `atomic` statements, the important concept of execution-time analyzable code, etc. Space does not permit this introduction to be exhaustive, and the reader is strongly urged to refer to (Nilsen, 1998) for further details.

5.2.1 Introduction to PERC

General Issues

PERC is a superset of Java 1.1. It provides additional features for time- and memory related concepts to the programmer. On the other hand, PERC tasks are different from Java threads. It is, for example, not allowed to invoke static `java.lang.Thread` methods from within PERC realtime tasks—violations result in the exception `IllegalThreadStateException` being thrown. To avoid the reinvention of the wheel, however, PERC tasks are permitted to enter `synchronized` statements, and to `wait` and be `notify`'ed.

PERC code is translated to ordinary Java byte code by a compiler known as **percolate**, which also understands ordinary Java code. Additionally, it recognizes the special syntax elements that are part of PERC's realtime extensions. And it is **percolate** that performs certain analysis of the PERC code that helps the worst-case execution time analyzer identify sections of the code whose execution time must be analyzed. The abstract process is shown in Figure 5.1.

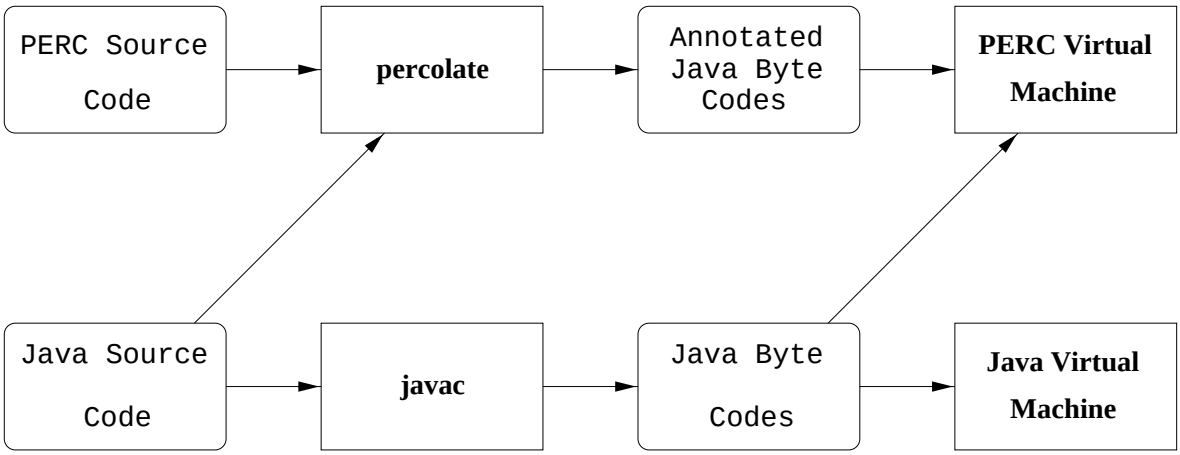

Figure 5.1: PERC Compilation Options

The Structure of Realtime Software Components

In PERC, realtime software is made up of *activities* and *tasks*. A realtime activity is organized as one or more realtime tasks. PERC offers classes that help implement these two abstractions, `Activity` and `RealTimeTask`, which we will look at in due course. An activity, prior to execution, must be *configured*. This consists of assessing how much CPU time and memory will be necessary to support the workload. Once we have ascertained that, we can express the resource requirements of the activity in terms of essential and desired resources. Figure 5.2 is an illustration.

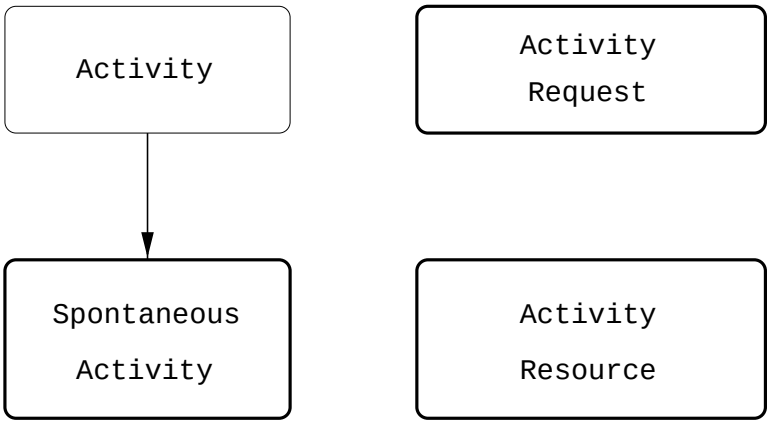

Figure 5.2: The class `Activity` and resource management classes

These results are communicated to the realtime executive which checks whether it can satisfy the request. The realtime executive offers a resource budget to the activity which contains the guaranteed and the expected resources. The guaranteed ones are always provided to the activity, and the realtime executive usually expects to provide the expected resources. The activity then either accepts or rejects the proposal. Not surprisingly, this process is called *resource negotiation*. If the activity approves the budget proposed, the system executive invokes the activity's `commit()` method, thereby including the activity into the current workload. This ultimately marks the beginning of the execution of the activity with the budgeted resources.

In the course of the events, it is likely that new activities are introduced into or others are removed from the system. The workload is thus possibly dynamic. Activities may wish to alter their resource needs. This gives rise to a revision of the system-wide resource allocation, or a *reconfiguration*. A reconfiguration comes about in a similar way as a (initial) configuration. PERC has provisions for this, too.

Realtime tasks in PERC come in four flavors, a graphical overview of which is given in Figure 5.3.

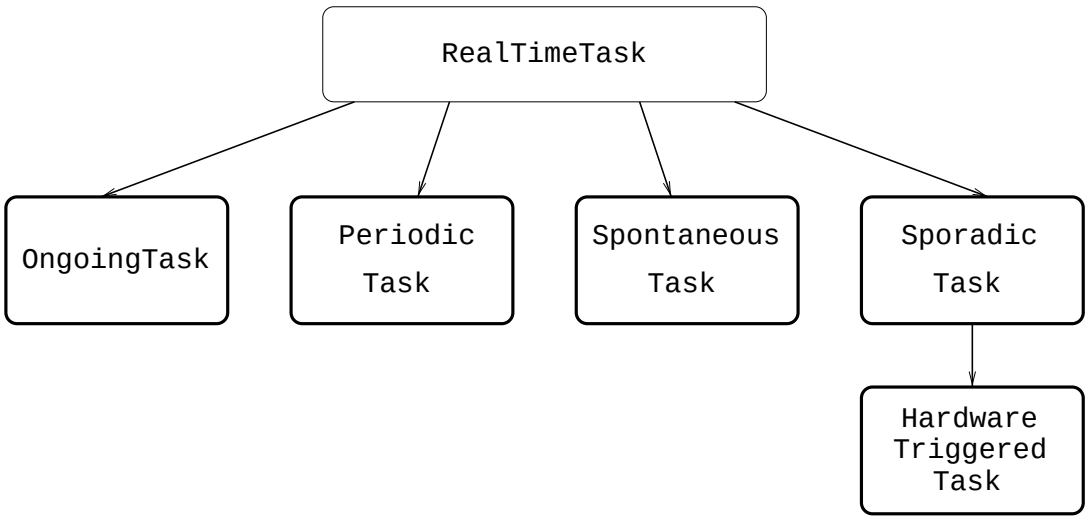

Figure 5.3: PERC tasks types

1. *Periodic tasks* are regularly scheduled once every period. The period is given when the task is being created.

2. *Sporadic tasks* owe their name from the fact that they are invoked by dynamic, or sporadic, conditions. Hardware interrupts spring to mind, for example. The maximum trigger frequency and other limits/bounds are determined in advance during the process of configuration/negotiation.

3. *Spontaneous tasks*, like sporadic ones, react upon dynamic conditions. This time, however, the trigger must stem from the software. Another difference between sporadic tasks—for which resources are allotted when the task is configured—and spontaneous tasks is that resources are set aside on the fly the moment the spontaneous task is actually triggered. This bears the risk that the task cannot obtain the resources necessary when it intends to run. Laxly speaking, we thus have: "If there are sufficient resources to allow this task to reliably execute within a particular period of time starting from now, schedule it; otherwise, simply report that there are not sufficient resources." (Nilsen, 1998, p. 4) Spontaneous tasks are an alternative to sporadic tasks, and should be used whenever the application is not willing to allot in advance the resources required for the regular scheduling of the task.

4. *Ongoing tasks*, unlike the three kinds of tasks described above—which run to completion each time they are triggered/are aborted if they fail to execute whatever they wanted to in their budgeted CPU time—, are simply de-scheduled at the end of each time allotment and are resumed the next time they are allowed to run. A number crunching task is a good case in point.

Further elaboration on the issue of spontaneous activities reveals the following:

The only way to execute a spontaneous task is to make it part of a spontaneous activity. In the PERC class hierarchy, `SpontaneousActivity` is a subclass of `Activity`. All of the tasks in a spontaneous activity must be spontaneous tasks, and spontaneous tasks are not allowed to appear in traditional (non-spontaneous) activities. In order to be able to trigger a spontaneous task for execution, the corresponding spontaneous activity must have previously been configured. During configuration, the spontaneous activity computes how much execution time is required for each of the spontaneous tasks and how much memory is required by the combined workload represented by all tasks. When the spontaneous activity is triggered, the precise semantics is that all of the resources required for execution of all the spontaneous tasks must be available or none of the tasks that comprise the spontaneous activity will be scheduled for execution." (Nilsen, 1998, p. 4)

Resource Budgeting and Execution Time Analysis

Given the very nature of realtime systems, we must be able to determine resource requirements in the local execution environment *beforehand*. PERC assists the programmer in this effort by providing means for both *analytical* and *empirical* determination of resource requirements. The analytical approach is to determine, through the analysis of existing code, the maximal amount of time and memory needed to execute the code. Empirical determination is just another term for running the program and timing its execution/ measuring the amount of memory that was consumed during execution. But it is a bit more intricate: "during [the] configuration of an activity, the system executive dedicates a particular realtime activity solely to the execution of the newly introduced activity's `configure()` method. Thus, the `configure()` method may safely assume that all resources consumed by the activity during its execution were dedicated to [the] execution of the `configure()` code."

Analytical tools are conservative, empirical ones are probably overly optimistic in that they might not represent the worst case—we cannot cover all cases with testing. It is recommended to combine the two techniques.

Now, how do the analytical tools work? How is it possible for the PVM, the PERC Virtual Machine, to ascertain the worst-case execution time (WCET) of particular code segments? This is a crucial part of the whole PERC philosophy which is why the author thinks it only proper to quote verbatim from the PERC reference. The following is from (Nilsen, 1998, p. 9):

---

1. Straight-line code is analyzable.
2. An invocation of a `final` or `static` method whose implementation consists entirely of analyzable code is analyzable.
3. An `if-then-else` statement for which the control expression, the `then`-clause, and the `else`-clause are analyzable is analyzable.
4. A `switch` statement for which the selector expression and each of the `case` statements is analyzable is analyzable.

5. A `try-catch-finally` statement in which each of the clauses is execution-time analyzable is analyzable.
6. A `throw` statement is analyzable if the expression that computes the thrown object is analyzable.
7. A `for`-loop is analyzable if the initializer control expression assigns a known constant value to the iteration variable, the increment control expression increments the initializer by a known constant value, the test control expression compares the value of the iteration variable with a known constant value, the body of the `for` loop does not modify the iteration variable, and the body of the loop is itself execution time analyzable. `break` and `continue` statements are allowed.
8. A `break` or `continue` statement, including a labeled `break` or `continue` statement, is analyzable within analyzable loops.
9. A `timed` statement with a constant time bound is analyzable, regardless of the complexity of the code that comprises its body. There is no analysis required other than determining the value of the constant time bound.

Note that our restrictions on loops are more strict than is really necessary. Certainly, it would be possible to analytically determine WCET for more loops than satisfy our fairly restrictive criteria. Our main objective, however, is to provide reliable support for execution-time analysis of a restricted subset of the Java language, and we want to make sure that programmers can easily understand the rules (though not necessarily the implementation) that characterize this restricted subset.

percolate recognizes that certain source code contexts require code to be execution time analyzable, and **percolate** checks each such context, issuing appropriate error and warning messages whenever conflicts are encountered.

The PERC Extensions to Java

The set of Java 1.1 keywords is augmented by the following two elements:

`timed` identifies code whose execution time must be bounded, and

`atomic` identifies a segment of code whose execution must be all or nothing. Additionally, `atomic` implies `synchronized` in that at most one thread of control is allowed, at any time, to execute an `atomic` section.

The idea behind a `timed` statement is that the code is given a specified amount of time to execute. If this time is sufficient, then we are done. If not, the code is aborted. The PVM permits `timed` statements to be executed both by Java threads and PERC tasks. It is best to look at an example to see how it is done:

```
timed(Time.us(30)) {  // throws TimeOutException
   Arbitrary_Code();
}
```

This example requests that `Arbitrary_Code` execute no longer than 30 μs. The `TimeOutException` can be caught to interrogate the status of the execution of `Arbitrary_Code()`.

`timed` statements may be nested, both statically and dynamically.

"An `atomic` statement is a section of code that is executed either in its entirety or not at all.[25] `atomic` statements are similar to `synchronized` statements, and they provide a superset of the semantics of `synchronized` statements. In particular, only one thread at a time is allowed to be executing a particular object's `atomic` statements ... Because the blocking time associated with `synchronized` statements in Java is difficult to analyze, we disallow the use of both `synchronized` and `atomic` constructs except for the following special case. If all `atomic` constructs are nested entirely within a `synchronized` statement or method where the synchronization object is the same as the object to which the `atomic` lock would correspond, then it is acceptable to have both `atomic` and `synchronized` statements associated with a single object. Note in this case that the `atomic` statement will never need to obtain the lock because the surrounding synchronization context has already acquired it." (Nilsen, 1998, p. 16)

Both Java threads and PERC tasks can make use of `atomic` statements:

```
atomic {
   Bounded_Code();
}
```

`Bounded_Code()` is either executed entirely or not at all.

`atomic` statements may not be nested (neither statically nor dynamically). Otherwise, improper use on multiprocessor systems would lead to deadlock.

This concludes the subsection on the introduction to PERC. The following subsection is concerned with the provisions of the `RealTime` package and, finally, with the proper topic of this chapter: process scheduling.

5.2.2 Scheduling

Unfortunately, we cannot give an exhaustive listing of the `RealTime` package for there is none in (Nilsen, 1998). We will pick out the most important pieces and examine them. We start with some principles and concepts.

It has been mentioned already that programmers must organize realtime software in PERC as `RealTime.Activity` objects, with each `Activity` containing several realtime tasks. The process of resource negotiation and configuration is done at the level of `Activity` by means of the class `ActivityRequest`, which represents an `Activity`'s request for resources.

The four flavors of realtime tasks in PERC (see page 84) are implemented as classes in `RealTime`. They have a common ancestor, `RealTimeTask`, which is an abstract class that lays the foundation for `OngoingTask`, `PeriodicTask`, `SpontaneousTask`, and `SporadicTask`. Corresponding to them, respectively, we have the classes `OngoingTaskResource`,

[25] It should not be confused with atomic transactions in database processing. The notion of a rollback does not apply to `atomic` statements [cmnt. by author].

etc., which are used to represent resource requests and budget configurations; their superclass is `TaskResource`. The class `HardwareTriggeredTask`, which is derived from `SporadicTask`, can be used to implement an interrupt handler (as suggested in Chapter 4). See Figure 5.4.

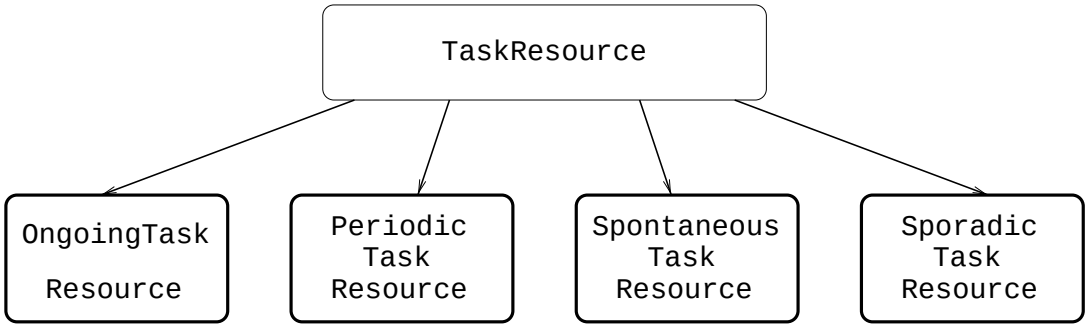

Figure 5.4: The resource classes

The tools for the self-analysis of resource requirements can be found in the class `ConfigurationManager`, and there is a `Time` class that provides handy methods for the handling of time.

To create a new `Activity`, developers are given the following constructors:

```
public Activity();

public Activity(String name);

public Activity(Bias requested_bias);

public Activity(String name, Bias requested_bias);
```

The `Bias` class "is used to describe systems-wide preferences regarding the relative importance and individual preferences regarding time-space tuning tradeoffs of currently loaded and running `RealTime.Activity` objects." (Nilsen, 1998, p. 18) Basically, it is just an ordered pair: $(Importance, Time\_Space\_Ratio)$.

The first constructor creates an `Activity` object with default name "`Activity-`" `+ number`, where `number` is the string representation of an integer, and default bias $(1.0, 1.0)$. The other three constructors allow for more customization.

The important methods of `Activity` are

```
public boolean add();

protected abstract void commit(ActivityResource proposed_guaranteed,
   ActivityResource proposed_expected);

protected abstract ActivityRequest configure();

protected abstract boolean negotiate(ActivityResource proposed_guar-
   anteed, ActivityResource proposed_expected);
```

```
public void remove();
```

which have been described, in principle, in Subsection 5.2.1, or whose semantics is apparent from their names.

Before we actually study the various tasks in PERC, let us have a quick look at the `ConfigurationManager` class and some of its tools:

```
long analyzeEET(String complete_method_name)
   throws MethodNotSupportedException;

long analyzeEET(Class C, String method_and_signature)
   throws MethodNotSupportedException;
```

"Given that `complete_method_name` or `c` combined with `method_and_signature` uniquely identify a particular analyzable method, return the expected execution time of this method, in nanoseconds, based on typical execution behavior of code on this machine. The expected execution time is calculated by summing the expected execution times of each instruction [on] the worst-case execution path. If the real-time executive determines that the method's code is not analyzable, `analyzeEET()` throws `MethodNotAnalyzableException`. If the virtual machine simply lacks the ability to analyze code, `analyzeEET()` throws `MethodNotSupportedException`." (Nilsen, 1998, p. 25)

For the analysis of the WCET, this class offers the method `analyzeWCET()`.

The class `RealTimeTask` is an abstract foundation for all PERC tasks, and it is high time we looked at the common features. The model of task execution in PERC is best described by the following three methods (all of which are defined in `RealTimeTask`):

```
protected void startup();
protected void work();
protected void finish();
```

It can be stated that these three methods are the essential parts of any PERC task. `startup()` is executed at the start of each execution period of the task while `finish()` is deemed to contain clean up code. Both methods must be execution time analyzable. The `work()` method, finally, is the place where the developer should put the code that is to be executed during each execution period, i.e., between `startup()` and `finish()`. Accompanying these three methods, we have `desiredWorkET()` and `essentialWorkET()`, `finishWCET()`, and `startupWCET()`. The first two are used to give an implementor of a realtime task the possibility to express the desired and minimum execution time for the `work()` method of the task in question. `startupWCET()` and `finishWCET()` are invoked during the resource configuration/negotiation phases to determine, respectively, the WCET of the `startup()` and `finish()` methods.

Other methods of `RealTimeTask` include `delay()`, which is an improved version of Java's `sleep()`,[26] `activity()`, which returns the corresponding `Activity` object, and methods that return the respective resource configuration objects and budgets.

[26] Remember that we are not allowed to call static `java.Thread` methods from within PERC tasks.

Another interesting method is `remainingTime()`, which plays a crucial rule in (the implementation) of `timed` and `atomic` statements. "This method returns, in nanoseconds, the amount of execution time remaining in the current time slice allotment (for this period). Note that, in case of a transient overload situation, it may be necessary for the system executive to adjust a task's CPU-time budget for a given period even while the task is currently executing within that period. Thus, the value returned by this method is only an estimate of how much additional time the task will be granted in this period." (Nilsen, 1998, p. 33)

Let us now examine the support of the specification of temporal scopes for the various flavors of tasks in PERC. We have said that PERC knows four basic kinds of tasks and that each kind is represented as a class. The constructors of the respective class can be used to create an object of the corresponding task. For example, to create an ongoing task, we use one of the following constructors:

```
public OngoingTask();

public OngoingTask(String name);
```

Using the first constructor, the task will be known as "`OngoingTask-`" `+ number`, where `number` is the string representation of an integer (see (JLS, 1996, 20.20.4)). The optional `name` allows for more customization.

This pattern is repeated for all four task types in PERC, that is, each one has got two constructors which resemble the ones above.

To be able to state timing constraints applying to a particular task, a resource class is associated with each of the four task flavors, i.e., we have `OngoingTaskResource`, `PeriodicTaskResource`, `SpontaneousTaskResource`, and `SporadicTaskResource` (Figure 5.4). And it is these four classes which ultimately allow us to express deadlines. They are all derived from the abstract class `TaskResource`, which is described as follows in (Nilsen, 1998, p. 37):

"During resource configuration and negotiation, the real-time activity characterizes its CPU time requests in terms of `TaskResource` objects, with two `TaskResource` objects for each of the real-time tasks that comprise the activity's workload, one representing the task's absolute minimal CPU time requirements and the other representing the task's preferred or ideal CPU time requirements. The system executive characterizes its proposed resource budget by providing two `TaskResource` objects for each of the activity's real-time tasks, one representing the amount of CPU time that the executive promises it will always supply to the task, and the other representing the amount of CPU time that it expects to usually be able to supply to the task."

We begin with `OngoingTaskResource`—it offers the following constructors:

```
public OngoingTaskResource(OngoingTask task, long exec_time,
   long period, long max_jitter);
```

The ongoing task `task` shall have execution time `exec_time` nanoseconds, execution period `period` nanoseconds, and a maximum jitter given by `max_jitter` (in nanoseconds). The notion of a *jitter* is used to describe the softness of a realtime task. Soft realtime

tasks sometimes execute a bit ahead or a bit behind the desired execution times. Each realtime task in PERC may specify a jitter. A value of 0 means that the task is hard realtime and not willing to tolerate any deviation from its time constraints; a value of -1 indicates that the task is soft realtime and does not care about its softness. See (Nilsen, 1998, p. 20) for further details.

The second constructor is just like the first—except for that fact that it has a default jitter of -1, and the third one reads as follows:

```
public OngoingTaskResource(OngoingTask task, long exec_time,
   long period, long max_jitter, boolean refinable, float margin);
```

"This constructor specifies that the ongoing task represented by `task` has execution time `exec_time` nanoseconds, execution period of `period` nanoseconds, and has maximum allowed jitter as specified by the `max_jitter` argument, expressed in nanoseconds. The `refinable` argument, if true, authorizes the PERC executive to adjust (refine) the task's execution time value based on observed behavior of the corresponding task. If this resource represents an essential minimum execution time, the PERC executive may increase the execution time budget based on its measurement of the times required to execute the essential components of the task. If this resource represents an expected execution time, the PERC executive may increase or decrease the execution time budget based on measurements of the corresponding task's typical execution times. If `refinable`, `margin` specifies the safety margin for the PERC executive's adjusted execution time budgets. For example, if `margin` equals 0.20, the PERC executive's budget is 20% higher than measured execution times. For an essential execution time, `margin` is multiplied by the worst measured time for executing its `startup()` and `finish()` methods added to the worst measured time for aborting the task's `work()` component. For an expected execution time, `margin` is multiplied by the average measured execution time of running the complete task, including all of its optional and essential components." (Nilsen, 1998, p. 28)

A fourth constructor, which looks like the third one but has a default jitter of -1, is available.

The resource class for periodic tasks is `PeriodicTaskResource` and has the following four constructors:

```
public PeriodicTaskResource(PeriodicTask task, long exec_time,
   long period);

public PeriodicTaskResource(PeriodicTask task, long exec_time,
   long period, long max_jitter);

public PeriodicTaskResource(PeriodicTask task, long exec_time,
   long period, boolean refinable, float margin);

public PeriodicTaskResource(PeriodicTask task, long exec_time,
   long period, long max_jitter, boolean refinable, float margin);
```

Careful observation reveals that the pattern is different from that used for ongoing tasks (besides the obvious difference, i.e., we are talking about periodic tasks now). Nonetheless, the description of the constructors is similar and will, therefore, not be repeated here.

One common feature of ongoing tasks and periodic tasks is that they both have a period. Given an `OngoingTaskResource/PeriodicTaskResource` object, we can interrogate the period of the corresponding task by calling the function

```
public final long period();
```

which, as suggested, returns the period of execution, measured in nanoseconds.

`SpontaneousTaskResource` is the resource class that belongs to spontaneous tasks. To schedule their execution, we are given the following tools:

```
public SpontaneousTaskResource(SpontaneousTask task,
   long exec_time, long deadline);

public SpontaneousTaskResource(SpontaneousTask task,
   long exec_time, long deadline, long max_jitter);

public SpontaneousTaskResource(SpontaneousTask task,
   long exec_time, long deadline, boolean refinable, float margin);

public SpontaneousTaskResource(SpontaneousTask task,
   long exec_time, long deadline, long max_jitter,
   boolean refinable, float margin);
```

The only thing that is new here is the `deadline` parameter, which is used to indicate that the spontaneous task's deadline is `deadline`.

The fourth and last flavor of tasks in PERC is a sporadic task. An object of the corresponding resource class, `SporadicTaskResource`, can be created in one of the following four ways:

```
public SporadicTaskResource(SporadicTask task, long exec_time,
   long trigger_period, long deadline);

public SporadicTaskResource(SporadicTask task, long exec_time,
   long trigger_period, long deadline, long max_jitter);

public SporadicTaskResource(SporadicTask task, long exec_time,
   long trigger_period, long deadline,
   boolean refinable, float margin);

public SporadicTaskResource(SporadicTask task, long exec_time,
   long trigger_period, long deadline, long max_jitter,
   boolean refinable, float margin);
```

`trigger_period` is used to describe the maximum trigger frequency,[27] in nanoseconds, of the sporadic task `task`. Excessing this limit results in `FrequencyExcessException` being thrown.

Both spontaneous and sporadic tasks have a deadline—the function

```
public final long deadline();
```

can be used to find out what it is. For sporadic tasks, we have yet another criterion: the minimum time separation between successive triggers. It can be obtained by using

```
public final long period();
```

This completes the description of the PERC realtime tasks and their corresponding resource classes, which allow programmers to state the behavior in time, or—as we call it—temporal scopes, for the underlying tasks. It is the PERC Virtual Machine which takes care of these temporal scopes and which ensures that tasks can meet their deadlines. After the tasks and their temporal scopes have been created, the process of resource configuration/negotiation takes place. One basis of this process is the WCET analysis of code sections, which is performed at compile time by **percolate**. After a realtime activity has been proposed a budget and admitted to the workload, the PVM is able to schedule the tasks contained in an activity so that all deadlines can be met. The tasks are then automatically executed. A sporadic task is the only task whose execution can be triggered manually. For this purpose, the class `SporadicTask` offers the `trigger()` method. This has already been mentioned, as well as the fact that a subclass of `SporadicTask`—`HardwareTriggeredTask`—can be used to implement interrupt handlers, in Chapter 4.

For the sake of completeness, we shall mention that in the case of spontaneous activities, there is a special way to schedule the tasks of such an activity. It has been pointed out above that the only way to execute spontaneous tasks is to pack them into a `SpontaneousActivity`, which is an extension of `Activity` by the following method:

```
public boolean schedule(long max_release_time);
```

"Given that the target `SpontaneousActivity` has already been configured, schedule all of the spontaneous tasks associated with this `SpontaneousActivity` object for one-time execution with release sometime between now and `max_release_time` and individual task deadlines as represented by the activity's previously configured resource request representation if sufficient resources are available to do so. By design, `schedule()` does not trigger system-wide renegotiation for resources. The spontaneous workload is only accepted into the system if it can be handled without negatively impacting the current workload. `schedule()` returns `true` if the activity was effectively scheduled and `false` otherwise." (Nilsen, 1998, p. 34)

---

[27] It is a bit peculiar to refer to the maximum trigger frequency by `trigger_period`, and measure it in nanoseconds. Perhaps the expression $\frac{e}{tp}$ makes more sense, where e is the execution time and tp the trigger period.

PERC has additional facilities for memory budgeting and persistent memory. The issue of garbage collection is a subtle one in realtime systems and is discussed at length in the PERC manual. The inclined reader is strongly encouraged to consult this primer for further information.

5.2.3 An Example

After the presentation of the ideas relating to PERC's model of task scheduling (which was necessarily a trifle theoretic), let us now look at an example. Unfortunately, (Nilsen, 1998) does not contain any substantial examples, which is why we shall look at one we have already used in (Brömel, 1998, Chapter 6). Ultimately, it is derived from (Burns, 1997, p. 385). Timing requirements of an actor can roughly be grouped as follows:

- ◇ delay start by a known amount of time
- ◇ complete execution by a known deadline
- ◇ take no longer than a specified amount of time to undertake a computation

Consider the important realtime activity of making and drinking instant coffee:

```
Get_Cup
Put_Coffee_In_Cup
Boil_Water
Put_Water_In_Cup
Drink_Coffee
Replace_Cup
```

The act of making a cup of coffee should take no more than ten minutes; drinking is more complicated. A delay of three minutes should ensure that the mouth is not burnt; the cup itself should be emptied within 25 minutes (it would then be cold) or before 17:00 (in other words, 5 o'clock and time to go home). Two temporal scopes are required:

```
start elapse 10 do
   Get_Cup
   Put_Coffee_In_Cup
   Boil_Water
   Put_Water_In_Cup
end

start after 3 elapse 25 by 17:00 do
   Drink_Coffee
   Replace_Cup
end
```

For a temporal scope that is executed repetitively, a time loop construct is useful:

```
from <start> to <end> every <period>
```

For example, many software engineers require regular coffee throughout the working day:

```
from 9:00 to 16:15 every 45 do
   Make_And_Drink_Coffee
```

where `Make_And_Drink_Coffee` could be made up of the two temporal scopes given above (minus the `by` constraint on the drinking block). Note that if this were done, the maximum elapse time for each iteration of the loop would be 35 minutes; correctly less than the period of the loop.

An implementation of this scheme is given below (in Ada):

```
task type Software_Engineer;

task body Software_Engineer is

   Now : constant Time := Clock;
   Time_To_Get_Started : constant Time := Time_Of(
     Year => Year(Now),
     Month => Month(Now),
     Day => Day(Now),
     Seconds => Day_Duration(9 * 3_600));  -- 9 a.m.
   Let_Us_Call_It_A_Day : constant Time := Time_Of(
     Year => Year(Now),
     Month => Month(Now),
     Day => Day(Now),
     Seconds => Day_Duration(17 * 3_600));  -- tea time
   Next_Time : Time;

begin
   delay until Time_To_Get_Started;
   select
      delay until Let_Us_Call_It_A_Day;
   then abort
      Next_Time := Clock + 45 * Minutes;
      loop
         Coffee_Ready := False;
         select
            delay 10 * Minutes;
         then abort
            Get_Cup;
            Put_Coffee_In_Cup;
            Boil_Water;
            Put_Water_In_Cup;
            Coffee_Ready := True;
         end select;

         if Coffee_Ready then
            delay 3 * Minutes;
```

```
            select -- nested ATC
               delay 25 * Minutes;
            then abort
               Drink_Coffee;
               Replace_Cup;
            end select;
         end if;

         delay until Next_Time;
         Next_Time := Next_Time + 45 * Minutes;
      end loop;
   end select;

end Software_Engineer;

My_Company : array(1 .. 10) of Software_Engineer;
```

Note the ease of writing temporal scopes at the beginning of our example and compare this with the effort needed to achieve the same expressive power in Ada. In PEARL, we could more or less write such code directly.

We see that the life of a software engineer is a simple one, things happen as predetermined, and there is not much fuss about anything. We can implement this life cycle as a periodic process which contains the two temporal scopes from above. The temporal scopes are implemented by means of `timed` statements. The minimum delay at the start of the second temporal scope must be achieved via an appropriate `delay()`. Note that PERC does not allow us to specify when a particular task is to be started. Thus, we have to wait explicitly until it is 9:00 as well. This, being a relative delay, is a bit more complicated than the (absolute) 3-minute delay at the beginning of the second temporal scope. Based on the current time, we have to compute an absolute value, which is then used in the `delay()`. This is not difficult, but it bears the risk of a race condition. What if the task is being de-scheduled between the computation of the delay value and the call to `delay()`? A relative delay, like Ada's `delay until`, would be most welcome here. Note that even the use of `atomic` is not really a solution since `atomic` does not necessarily imply that task dispatching has been disabled during the execution of the `atomic` code.[28] It is a workaround; the best we can get, and we will use it. Incidentally, the author has changed the example a little bit so as to distribute the scenario more evenly to the `startup()`, `work()`, and `finish()` methods, which are—as described earlier—the backbone of every PERC task.

```
class Software_Engineer_Task extends PeriodicTask {

   public Software_Engineer_Task(String Name) {
      super(Name);
   }

   protected void startup() {
      /* determine the time remaining until 9:00 today
         and wait for that amount of time, using an atomic
         statement as described above */
```

[28] although this is a possible implementation of `atomic` statements on a single processor

```
      atomic {
         Date Now = new Date();
         Date Time_To_Get_Started = new Date(Now.getYear(),
           Now.getMonth(), Now.getDate, 9, 0);
         delay(Time.ms(Time_To_Get_Started.getTime() -
           Now.getTime()));
      }
      timed(Time.m(10)) {  // 10 minutes
         Get_Cup();
         Put_Coffee_In_Cup();
         Boil_Water();
         Put_Water_In_Cup();
      }
   }

   protected void work() {
      delay(Time.m(3));  // so as to not burn the mouth
      timed(Time.m(25)) {
         Drink_Coffee();
      }
   }

   protected void finish() {
      Replace_Cup();
   }
}

class Life_Cycle extends Activity {

   /* tasks can only be created from within the
      configure() method of an activity */

   protected ActivityRequest configure() {
      Software_Engineer_Task Software_Engineer =
        new Software_Engineer_Task("Yossarian");

      /* note that most methods in the class Time require
         arguments of type long */
      final long Nine_To_Five = 8;
      final long Period = 45;

      /* finally, the specification of temporal scopes
         goes here */

      Software_Engineer_Resource = new PeriodicTaskResource(
         Software_Engineer, Time.h(Nine_To_Five), Time.m(Period));
   }
}
```

The example above creates a sporadic task which has an execution time of 8 hours and a period of 45 minutes. The idea is that following this, the process of resource negotiation takes place. `configure()` is invoked by the PVM and yields the activity's resource requirements. The PVM proposes a budget to the activity, and the programmer is given the opportunity to accept or reject the proposal. The method `protected boolean Activity.negotiate()` is reserved for this purpose, it should contain code that performs this kind of negotiation. If `negotiate()` "answers in the affirmative," the PVM invokes the `commit()` method of our sample activity, thereby admitting the granted resources to the current workload, which—in effect—means that the tasks are scheduled.

5.3 Summary and Comparison

Readers should consult (Brömel, 1998, Chapter 6) or at least the summary of that chapter. There, one can find the relevant details regarding Ada, CHILL, and Java. This chapter has concentrated on Erlang and PERC.

Even (especially?) from the small amount of text the Erlang manual has dedicated to process scheduling, it is obvious that Erlang does not support the specification of temporal scopes. To mimic them, workarounds must be used. In this light, Erlang joins Ada, CHILL, and Java. The principal building block for the workarounds is the timeout mechanism described in Subsection 1.3.3. Using this, we can simulate a `sleep()` (see Program 3.3), which, in turn, can serve as a building block for more sophisticated things like periodic execution and deadlines. The latter could be achieved by having a time server process that, having received the deadline of a client beforehand, simply lays dormant until the deadline has arrived, and notifies the client in time. The client, then, reacts by abandoning whatever it is doing at that time. We do not claim that this is an elegant solution. Ada and CHILL are better off in this respect since they both offer features to handle these cases at least locally within the process: ACT with a delay trigger, and time supervision.

PERC is undoubtedly outstanding. It is the only language in our quartette that allows us to specify temporal scopes. Corresponding to the four task types in PERC, ongoing, periodic, spontaneous, and sporadic ones, we can—during the process of creation/configuration—state the behavior in time for each task. Perhaps missing from the set of specifiable items is the one that permits us to say that an actor is to be started at a specified time. For example, if we want a particular task to execute at 06:00 each morning, then we have to fall back on workarounds.

However, if the PVM can guarantee that the deadlines can be met, then the tasks (or, to be more precise, the activity that contains the tasks) will be admitted to the workload. The tasks are then automatically scheduled and dispatched. The PERC model of execution, to which we have given a brief introduction, makes this possible. Resource configuration, negotiation, budgeting, and the important concept of execution-time analysis play a key role here.

Additionally, PERC offers two new concepts to the programmer: `atomic` and `timed` statements. While the latter can be easily achieved in Ada and CHILL (in Erlang, it is a bit clumsy, see above), the notion of an `atomic` statement is only found in PERC.

Last but not least, the author would like to remind readers of the provisions PEARL has for the specification of temporal scopes. The ease of associating them with processes, perhaps even more straightforward than in PERC, makes PEARL an outstanding language as well.

Bibliography

WHY MUST YOU DO A BIBLIOGRAPHY?
To give readers an opportunity to check out your sources for accuracy. An honest bibliography inspires reader confidence in your writing.
-- I. Lee, Chapter 11. Guidelines on Writing a Bibliography

(ARM, 1995) Taft, S. Tucker and Duff, Robert A.: *Ada 95 Reference Manual, Language and Standard Libraries*, International Standard ISO/IEC 8652:1995(E), Springer Verlag, 1995, Lecture Notes in Computer Science Vol. 1246.

(Armstrong, 1996) Armstrong, Joe et al.: *Concurrent Programming in Erlang*, Second Edition, Prentice Hall, 1996.

(Barnes, 1996) Barnes, John: *Programming in Ada 95*, Addison-Wesley, 1996.

(Bokhari, 1997) Bokhari, S. H.: *Multiprocessing the Sieve of Eratosthenes*, Computer 20:4, pp. 50–58, April 1997.

(Brömel, 1998) Brömel, Peter and Ecke, Frank: *Description and Comparison of the Concurrency Concepts of Ada, CHILL, and Java*, Project Alfa Core, Friedrich Schiller University Jena, Germany, 1998.

(Brömel, 1999) Brömel, Peter: (in German) *Vergleich der Nebenläufigkeitskonzepte von Ada, CHILL, Erlang und Java—Fallstudien-1*, Diploma Thesis, Friedrich Schiller University Jena, Germany, 1999.

(Brosgol, 1996) Brosgol, Benjamin M.: *The Dining Philosophers in Ada 95*, in *Reliable Software Technologies—Ada-Europe '96*, June 1996, Montreux, Switzerland, Springer Verlag, 1996, Lecture Notes in Computer Science Vol. 1088.

(Brosgol, 1998) Brosgol, Benjamin M.: *A Comparison of the Concurrency and Real-Time Features of Ada 95 and Java*, Ada UK'98 Conference, October 1998.

(Burns, 1997) Burns, Alan and Wellings, Andy: *Real-Time Systems and Programming Languages*, Second Edition, Addison-Wesley, 1997.

(Burns, 1998) Burns, Alan and Wellings, Andy: *Concurrency in Ada*, Second Edition, Cambridge University Press, 1998.

(CACM, 1998) Nilsen, Kelvin: *Adding Real-Time Capabilities to Java*; in *Communications of the ACM*, June 1998/Vol. 41, No. 6

(Dijkstra, 1971) Dijkstra, E.W.: *Hierarchical Ordering of Sequential Processes*, Acta Informatica, 1, 1971

(JLS, 1996) Gosling, James et al.: *The Java Language Specification*, First Printing, Addison-Wesley, August 1996.

(Knuth, 1984) Knuth, Donald Ervin: *The TEXbook (Computers and Typesetting; Volume A)*, 33rd Printing, American Mathematical Society and Addison-Wesley, 1984.

(Nilsen, 1998) Nilsen, Kelvin and Lee, Steve: *PERC Real-Time API*, (Draft 1.3, July 18, 1998), NewMonics Inc., `www.newmonics.com`, 1997.

(NODE, 1998) Pearsall, Judy: *The New Oxford Dictionary of English*, First Printing, Oxford University Press, 1998.

(Rat, 1995) Barnes, John: *Ada 95 Rationale. The Language and the Standard Libraries*, Springer Verlag, 1995, Lecture Notes in Computer Science Vol. 1247.

(Shapiro, 1995) Shapiro, E.: *Concurrent Logic-Programming Techniques*, tutorial presented at the *1985 International Conference on Parallel Processing*, St Charles, Illinois, August 23, 1995.

Selbständigkeitserklärung

Ich erkläre, daß ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Jena, 14. Juli 1999

Frank Ecke