NBADA

Non-blocking Algorithms and Data Structures Library

Reference Manual
version 0.1.0-pre0

Anders Gidenstam (andersg(at)mpi-inf.mpg.de)

Draft 16th September 2008

Saarbrücken, Germany, 2008

Chapter 1

Introduction

NBADA is a collection of non-blocking algorithms and concurrent data structures in a common infrastructure aiming to be accessible for use by non-expert programmers as well as providing support for implementation of further non-blocking algorithms. NBADA is implemented in ADA 95 and is distributed as free software under the terms of the GNU Public License (c.f. Appendix A).

1.1 Concurrent data access in shared memory systems

[1]

In a shared memory system the processes[2] have access to a set of shared memory locations which they may use to communicate. A process can *read* data from and *write* data to each shared memory location. The number of processes can be much larger than the number processors due to multiprogramming, which may interleave the execution of several processes on the same processor. The processes are often considered to be asynchronous, that is, their rate of execution might vary arbitrarily, because of the interleaving. This has certain implications for the possibilities for the synchronization and coordination of processes which we will discuss below.

1.1.1 Linearizability

We want the semantics of all operations on a shared data object to be the same as for the same operation on the corresponding sequential object. The consistency model that captures this is called *linearizability* and was introduced by Herlihy and Wing in [HW90]. Linearizability requires that for each operation, in a concurrent execution of operations on the shared data object, there is an atomic time instant that lies within its duration where the operation takes effect, in a way such that the outcome of the operation agrees with the object's sequential specification.

1.1.2 Lock-based synchronization

The traditional way to synchronize processes/threads accessing a shared data object in a concurrent program is to use *mutual exclusion*. Mutual exclusion is normally implemented using a lock, which is a shared variable together with routines to atomically *acquire* and *release* the lock. The atomicity of *acquire* and *release* guarantees that only one process can acquire and hold the lock at a time. The most common approach when synchronizing using locks is to use the lock to implement *critical sections*, that is, some pieces of code that can only be run by one process at the time. For a shared data object, it is common that the operations it supports are implemented as mutually exclusive critical sections.

---

[1] This introduction is based on the introduction in [Gid06]

[2] We will use the term process and thread interchangeably in the context of general shared memory synchronization. If we talk about threads and processes in the operating system sense it will be made clear from context.

The use of locks and the sequential nature of critical sections cause a number of drawbacks, namely:

- **Deadlock prone.** With locks it is not hard to create circular lock dependencies that cause two (or more processes) to get blocked by both trying to acquire a lock that is held by the other. Furthermore, a process that crashes while holding some lock(s) is also likely to block the progress of other processes.

- **Blocking.** The process that has acquired the lock will delay all other processes that also need that lock until it has finished executing inside the critical section. To make matters worse the process inside the critical section may itself be delayed by being preempted by the scheduler, suffer a page-fault, try to acquire another lock or wait for IO inside the critical section.

- **Priority inversion.** This is a pathological case that can occur when using a strict priority based scheduler, where a medium priority process can delay a high priority process, potentially indefinitely on a single processor system, by preempting a low priority process that has acquired a lock needed by the high priority process. This problem can be avoided by employing *priority inheritance protocols* as proposed by Sha et al. [SRL90].

1.1.3 Non-blocking synchronization

Non-blocking synchronization techniques avoid the use of locks by using cunning algorithms, which often but not always use hardware synchronization primitives, to create shared data objects that can be accessed simultaneously by several processes. By avoiding locks non-blocking synchronization does not exhibit the problems of deadlocks, blocking and priority inversion, which lock-based synchronization suffers from. Non-blocking shared data objects also have a higher degree of fault-tolerance than lock-based ones since they can tolerate any number of processes experiencing stop-failures.

There are two kinds of non-blocking synchronization, *lock-free* synchronization and the stronger *wait-free* synchronization.

Lock-free synchronization

A *lock-free* algorithm guarantees that regardless of the contention caused by concurrent operations and the interleaving of their steps, at each point in time there is at least one operation which is able to make progress. However, as there is no fairness guarantee, some operation could be starved and take unbounded time to finish.

The lack of fairness guarantee significantly simplifies the construction of lock-free algorithms compared to wait-free ones and leads to algorithms that are fast when there are no conflicts but cause slow down for all except one process involved in a conflict. Hence, lock-free synchronization is also known as *optimistic synchronization* [Rin99].

In [Her93] Herlihy described a general method for transforming any sequential data object implementation to a lock-free shared data object implementation. In short, the methodology is like this: The state of the shared data object is represented by a pointer to the current version; an operation on the shared data object first makes a new private copy of the current version, applies the sequential version of the desired operation on the private copy and thus creates a new prospective state of the shared object. Then it tries to install this prospective state as the new version of the shared object using an atomic synchronization primitive that will only succeed if the current version of the shared object is still the same as the one the new state was computed from. If the operation fails to install its new state, some other operation(s) have managed to install their new versions and this operation has to retry from the beginning.

This general methodology is often not very efficient because (i) the entire object is copied for each update (this can be optimized though) and (ii) the resulting lock-free shared object is

not *disjoint-access parallel*, that is, all concurrent operations on it cause conflicts even when the operations only access disjoint parts of the shared object.

For the above reasons, a significant research effort is being spent on the task of designing and developing efficient lock-free implementations of various data structures.

The use of lock-free instead of lock-based synchronization can give significant performance gains in parallel applications, as shown by Tsigas and Zhang in [TZ01a, TZ02], as well as in operating systems, for example as suggested by Greenwald and Cheriton in [GC96].

Wait-free synchronization

A *wait-free* algorithm is both lock-free and *fair*, it guarantees that every operation finishes in a bounded number of its own steps, regardless of the actions of other operations. This is a very strong property, as it decouples the processes using the same shared data object from each other. This makes wait-free shared data objects attractive to use, for example, in hard *real-time systems* where the worst-case execution time has to be known for every operation and where lock-based critical sections limit the schedulability of the system and complicate the schedulability analysis. A drawback, however, is that algorithms that are wait-free, are often also quite complex, in particular for non-trivial shared objects.

A common approach in implementing wait-free algorithms is the use of *helping schemes* [Her91]. In a helping scheme each operation first announces information about what it wants to do with the shared object in some global data structure, then it checks in the announce-structure to see if there are other operations that it needs to help before proceeding with its own.

Barnes presented a method similar to helping in [Bar93]. In his method each operation on the shared data object is divided into a sequence of *virtually atomic suboperations*, where each suboperation is constructed so that once it has begun, it is guaranteed to be performed fully, either by the initiating process or by being helped by another process.

In [Her91] Herlihy presented a universal method for constructing a wait-free algorithm for any shared data object. However, as for the general methodology for construction of lock-free algorithms, the universal construction for wait-free algorithms is not practical in all cases and therefore significant research efforts are being spent on developing efficient wait-free algorithms.

Chapter 2

Installation

Installation of NBAda is simple: just extract the distribution archive anywhere you want. To make it convenient to compile programs which use components from NBAda there is a utility, **nbada_config**, that outputs suitable command-line options for use with **gnatmake**.

The utility **nbada_config**, located in **src/util**, is written in Ada, so it needs to be compiled and the binary installed somewhere convenient (e.g. /usr/local/bin). Before compiling the path to the directory where the NBAda source code is install needs to be entered into **nbada_config.adb** by modifying the following line in **nbada_config.adb**:

```
--   NBAda source code base directory .
Install_Base : constant String :=
  "/usr/local/share/NBAda/src";   -- Change this line .
--   Default architecture .
Default_Architecture : constant Architecture := IA32;
```

For convenience the default architecture can also be changed there, the full list of supported architectures is shown in Table 2.1. The **nbada_config.adb** can be compiled, e.g. with the command '**gnatmake nbada_config.adb**'.

2.1   Using `nbada_config`

As mentioned above, **nbada_config** outputs command line options for use with **gnatmake**. A typical usage pattern would be:

```
% gnatmake myprogram.adb 'nbada_config LF_SETS'
```

where the shell replaces '**nbada_config LF_SETS**' with the output of **nbada_config LF_SETS**.

The full set of options recognized by **nbada_config** is outlined below and explained in Table 2.1 and Table 2.2.

```
Usage: nbada_config [OPTIONS] [LIBRARIES]
Options:
        [--isa=<IA32|SPARCV8PLUS|SPARCV9|MIPSN32>]
        [--help]
Libraries:
        PRIMITIVES      (default)
        LF_POOLS
        EBMR
        HPMR
        PTB
        LFRC
        LFMR
        SW_LL_SC
        LF_STACKS_EBMR
```

Architecture	Description
IA32	32-bit Intel x86 Architecture (Intel Pentium and above).
SPARCV8PLUS	32-bit mode on SPARC v9 compatible processor.
SPARCV9	64-bit mode on SPARC v9 compatible processor.
MIPSN32	32-bit mode on 64-bit MIPS processor (e.g. R-10000)

Table 2.1: Supported instruction set architectures.

Option	Description	Chapter
PRIMITIVES	Hardware atomic primitives.	4.3
LF_POOLS	Lock-free storage pools.	4.2
EBMR	Epoch-based memory reclamation [Fra04].	4.1
HPMR	Hazard pointers memory reclamation [Mic04a, Mic02b].	4.1
PTB	Pass the buck memory reclamation [HLM02, HLMM05].	4.1
LFRC	Lock-free reference counting memory reclamation [HLMM05].	4.1
LFMR	Lock-free reference counting memory reclamation [GPST05, GPST08].	4.1
SW_LL_SC	Lock-free load-linked/store-conditional primitive [Mic04b].	3.1.3
LF_STACKS_EBMR	Lock-free dynamic stack [IBM83, Mic04a].	3.2.1
LF_STACKS_HPMR	Lock-free dynamic stack [IBM83, Mic04a].	3.2.1
LF_QUEUES_BOUNDED	Lock-free bounded queue [TZ01b].	3.2.2
LF_QUEUES_EBMR	Lock-free dynamic queue [MS96].	3.2.2
LF_QUEUES_HPMR	Lock-free dynamic queue [MS96].	3.2.2
LF_QUEUES_LFRC	Lock-free dynamic queue [HSS07].	3.2.2
LF_QUEUES_LFMR	Lock-free dynamic queue [HSS07].	3.2.2
LF_DEQUES_LFRC	Lock-free dynamic deque (a.k.a double ended queue) [ST04].	3.2.3
LF_DEQUES_LFMR	Lock-free dynamic deque (a.k.a double ended queue) [ST04].	3.2.3
LF_PRIORITY_QUEUES_EBMR	Lock-free dynamic priority queue.	3.2.4
LF_PRIORITY_QUEUES_HPMR	Lock-free dynamic priority queue.	3.2.4
LF_SETS_EBMR	Lock-free dynamic set [Mic02a].	3.2.5
LF_SETS_HPMR	Lock-free dynamic set [Mic02a].	3.2.5
LF_DICTIONARIES_EBMR	Lock-free dynamic dictionary [Mic02a].	3.2.5
LF_DICTIONARIES_HPMR	Lock-free dynamic dictionary [Mic02a].	3.2.5

Table 2.2: Include library options for `nbada_config`.

```
LF_STACKS_HPMR
LF_QUEUES_BOUNDED
LF_QUEUES_EBMR
LF_QUEUES_HPMR
LF_QUEUES_LFMR
LF_QUEUES_LFRC
LF_DEQUES_LFMR
LF_DEQUES_LFRC
LF_PRIORITY_QUEUES_EBMR
LF_PRIORITY_QUEUES_HPMR
LF_SETS_EBMR
LF_SETS_HPMR
LF_DICTIONARIES_EBMR
LF_DICTIONARIES_HPMR
```

Examples

To compile the NBAda `queue_test` micro-benchmark (`src/benchmarks/Queues`) using a lock-free queue algorithm with epoch-based memory reclamation the following command line could be used:

```
% gnatmake queue_test -ILock-Free_Queue `nbada_config LF_QUEUES_EBMR`
```

The argument `-ILock-Free_Queue` is used to select which queue implementation the benchmark will use as it can be compiled with several different ones. Here is the command line to build with the same lock-free queue algorithm but with the hazard pointers memory reclamation scheme:

```
% gnatmake queue_test -ILock-Free_Queue `nbada_config LF_QUEUES_HPMR`
```

This command line builds the `queue_test` benchmark with a bounded lock-free queue algorithm:

```
% gnatmake queue_test -ILock-Free_Bounded_Queue `nbada_config LF_QUEUES_BOUNDED LF_POOLS`
```

Chapter 3

Data structures

3.1 Atomic Objects

3.1.1 Large Register

An atomic register is a multi-word object that can be read and written with non-blocking atomic operations.

The package NBAda.Atomic_Single_Writer_Registers

NBAda provides two implementations of linearizable single writer multiple reader multi-word registers:

- Peterson's register algorithm [Pet83]; and

- the ReaderField algorithm by Larsson et al. [LGH+04].

The atomic register implementations in NBAda have the same public package specification, so an application can be compiled against any of them without source code changes.

```
   generic
2     type Element_Type is private;
   package NBAda.Atomic_Single_Writer_Registers is
4
      type Atomic_1_M_Register (No_Of_Readers : Positive) is limited private;
6
      type Reader_Id is private;
8
      procedure Write (Register : in out Atomic_1_M_Register;
10                     Value    : in     Element_Type);
      procedure Read  (Register : in out Atomic_1_M_Register;
12                     Reader   : in     Reader_Id;
                       Value    :    out Element_Type);
14
      function  Register_Reader (Register : in Atomic_1_M_Register)
16                               return Reader_Id;
      procedure Deregister_Reader (Register : in out Atomic_1_M_Register;
18                                  Reader   : in     Reader_Id);
20    Maximum_Number_Of_Readers_Exceeded : exception;
22 private
24    ...  --  Implementation details.
26 end NBAda.Atomic_Single_Writer_Registers;
```

Application constraints:

- Concurrent calls to Write on the same atomic register are forbidden.

- Concurrent calls to Read on the same atomic register with the same Reader_Id argument are forbidden.

- Reader_Id values should not be passed between tasks.

- Register_Reader/Deregister_Reader should be used as seldom as possible.

3.1.2   Linearizable Snapshots

A snapshot is a composite data structure consisting of a number of fields. Each field can be written separately and the entire state of the composite can be read atomically.

The package NBAda.Atomic_Multiwriter_Snapshots

The NBAda package NBAda.Atomic_Multiwriter_Snapshots implements the multiple writer per component multiple scanner lock-free linearizable snapshot algorithm by Jayanti [Jay05].

```
   generic
2     Max_Number_Of_Components : Natural;
      ——  Maximum number of components in the snapshot.
4     with package Process_Ids is
        new Process_Identification (<>);
6     ——  Process identification.
   package NBAda.Atomic_Multiwriter_Snapshots is
8
      type Snapshot (<>) is private;
10
      function Scan return Snapshot;
12
      Maximum_Number_Of_Components_Exceeded : exception;
14
      generic
16        ——  Use pragma Atomic and pragma Volatile for Element.
          ——  Element'Object_Size MUST be System.Word_Size.
18        type Element is private;
      package Element_Components is
20
          type Element_Component is private;
22
          function Create (Default_Value : in Element) return Element_Component;
24
          procedure Write (To    : in Element_Component;
26                          Value : in Element);
28        function Read (Component : in Element_Component;
                          From      : in Snapshot) return Element;
30
      private
32
          ...  ——  Implementation details.
34
      end Element_Components;
36
   private
38
      ...  ——  Implementation details.
40
   end NBAda.Atomic_Multiwriter_Snapshots;
```

Application constraints:

- Any task that calls an operation in NBAda.Atomic_Multiwriter_Snapshots must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

- All types used for components must have an Object_Size equal to System.Word_Size.

3.1.3 Software Load-Linked/Store-Conditional for multi-word Objects

The package NBAda.Large_Primitives

The package NBAda.Large_Primitives implements the lock-free load-linked store-conditional algorithm by Michael [Mic04b].

The algorithm relies on lock-free memory reclamation and the implementation uses the NBAda.Hazard_Pointers memory reclamation algorithm. The include flag for `nbada_config` is SW_LL_SC.

```
   generic
2     Max_Number_Of_Links : Natural;
      --  Maximum number of simultaneous LL/SC per thread.
4     with package Process_Ids is
        new Process_Identification (<>);
6     --  Process identification.
   package NBAda.Large_Primitives is
8
      package MR is < Implementation defined >
10
      generic
12       type Element is private;
      package Load_Linked_Store_Conditional is
14
         type Shared_Element is limited private;
16
         function   Load_Linked (Target : in Shared_Element) return Element;
18       function   Load_Linked (Target : access Shared_Element) return Element;
20       function   Store_Conditional (Target : in     Shared_Element;
                                       Value  : in     Element) return Boolean;
22       function   Store_Conditional (Target : access Shared_Element;
                                       Value  : in     Element) return Boolean;
24
         procedure Store_Conditional (Target : in out Shared_Element;
26                                     Value  : in     Element);
         procedure Store_Conditional (Target : access Shared_Element;
28                                     Value  : in     Element);
30
         function   Verify_Link (Target : in Shared_Element) return Boolean;
32       function   Verify_Link (Target : access Shared_Element) return Boolean;
34
         procedure Initialize (Target : in out Shared_Element;
36                             Value  : in     Element);
         procedure Initialize (Target : access Shared_Element;
38                             Value  : in     Element);
         --  Note: Initialize is only safe to use when there are no
40       --        concurrent updates.
42    private
44       ...  --  Implementation specific
46    end Load_Linked_Store_Conditional;
48    procedure Print_Statistics;
50 end NBAda.Large_Primitives;
```

Application constraints:

- All objects of type Shared_Element must be initialized with the operation Initialize before use.

- Any task that calls an operation in NBAda.Large_Primitives must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

3.2   Containers

NBADA includes a number of lock-free concurrent container data structures.

3.2.1   Stacks

The package NBAda.Lock_Free_Stacks

The package NBAda.Lock_Free_Stacks implements a lock-free unbounded stack data structure using an old well-known algorithm [IBM83, Mic04a]. It can use either the NBAda.Hazard_Pointers (LF_STACKS_HPMR) or NBAda.Epoch_Based_Memory_Reclamation (LF_STACKS_EBMR) memory reclamation algorithms.

```
   generic
2     type Element_Type is private;

4     with package Process_Ids is
        new NBAda. Process_Identification (<>);
6     —— Process identification .
   package NBAda. Lock_Free_Stack is

8
     type Stack_Type is limited private;

10
     Stack_Empty : exception;

12
     procedure Push (On      : in out Stack_Type;
14                    Element : in      Element_Type);
     procedure Pop  (From    : in out Stack_Type;
16                    Element :     out Element_Type);
     function  Pop  (From    : access Stack_Type)
18                    return Element_Type;

20    function  Top  (From    : access Stack_Type)
                     return Element_Type;

22
   private

24
         ...  ——  Implementation specific

26
   end NBAda. Lock_Free_Stack ;
```

Application constraints:

- Any task that calls an operation in NBAda.Lock_Free_Stack must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

3.2.2   Queues

The package NBAda.Lock_Free_Bounded_Queues

NBADA contains a lock-free bounded size queue data structure based on the algorithm by Tsigas and Zhang [TZ01b].

The include flag for `nbada_config` is `LF_QUEUES_BOUNDED`.

```
   generic
2    type Element_Type is private;
     —— The Element_Type must be atomic and Element_Type'Object_Size must be
4    —— equal to System.Word_Size.
     Null_0 : Element_Type;
6    Null_1 : Element_Type;
     —— NOTE: These two values MUST be different and MUST NOT appear as
8    ——      data values in the queue.
   package NBAda.Lock_Free_Bounded_Queues is

10
     type Queue_Size is mod 2**32;

12
     type Lock_Free_Queue (Max_Size : Queue_Size) is limited private;

14
     procedure Enqueue (Queue   : in out Lock_Free_Queue;
16                        Element : in     Element_Type);

18   procedure Dequeue (Queue   : in out Lock_Free_Queue;
                        Element :    out Element_Type);
20
     function  Dequeue (Queue : access Lock_Free_Queue) return Element_Type;

22
     function Is_Empty (Queue : access Lock_Free_Queue) return Boolean;

24
     procedure Make_Empty (Queue : in out Lock_Free_Queue);
26   —— NOTE: Make_Empty SHOULD NOT be used when concurrent access is possible.

28   Queue_Full  : exception;
     Queue_Empty : exception;

30
   private

32
         ...  ——  Implementation specific

34
   end NBAda.Lock_Free_Bounded_Queues;
```

Application constraints:

- The type Element_Type must be atomic.

- Element_Type'Object_Size must be equal to System.Word_Size.

- The values passed as the two generic formal parameters Null_0 and Null_1 MUST be different and MUST NOT appear as data values in the queue.

- The operation Make_Empty SHOULD NOT be used when concurrent access to the queue object is possible.

The package NBAda.Lock_Free_Queues

NBAda contains two lock-free implementations of dynamic queues, one based on the algorithm by Michael [MS96] and one on the algorithm by Hoffman et al. [HSS07].

The include flag for `nbada_config` is for Michael's queue algorithm `LF_QUEUES_HPMR` or `LF_QUEUES_EBMR` and for Hoffman et al.'s queue algorithm `LF_QUEUES_LFMR` or `LF_QUEUES_LFRC`.

```
   generic
2    type Element_Type is private;

4    with package Process_Ids is
       new Process_Identification (<>);
6    —— Process identification.
   package NBAda.Lock_Free_Queues is

8
     type Queue_Type is limited private;
```

```
10
      Queue_Empty : exception ;
12
      procedure Init       (Queue : in out Queue_Type );
14    function   Dequeue (From : access Queue_Type) return Element_Type ;
      procedure Enqueue (On         : in out Queue_Type ;
16                         Element : in      Element_Type );

18 private

20      ...   --   Implementation  specific

22 end NBAda. Lock_Free_Queues ;
```

Application constraints:

- Any task that calls an operation in NBAda.Lock_Free_Queues must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

- The operation Init SHOULD NOT be used when concurrent access to the queue object is possible.

3.2.3   Deques

The package NBAda.Lock_Free_Deques

The package NBAda.Lock_Free_Deques implements a lock-free unbounded double ended queue data structure based on the algorithm by Sundell and Tsigas [ST04].

The include flag for `nbada_config` is `LF_DEQUES_LFMR` or `LF_DEQUES_LFRC`.

```
  generic
2    type Element_Type is private ;

4    with package Process_Ids is
       new Process_Identification (<>);
6    --   Process  identification .
  package NBAda. Lock_Free_Deques is
8
     type Deque_Type is limited private ;
10
     Deque_Empty : exception ;
12
     procedure Init       (Deque : in out Deque_Type );
14
     function   Pop_Right (Deque   : access Deque_Type) return Element_Type ;
16   procedure Push_Right (Deque   : in out Deque_Type ;
                            Element : in      Element_Type );
18
     function   Pop_Left  (Deque   : access Deque_Type) return Element_Type ;
20   procedure Push_Left  (Deque   : in out Deque_Type ;
                            Element : in      Element_Type );
22
  private
24
        ...   --   Implementation  specific
26
  end NBAda. Lock_Free_Deques ;
```

Application constraints:

- Any task that calls an operation in NBAda.Lock_Free_Deques must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

- The operation Init SHOULD NOT be used when concurrent access to the queue object is possible.

3.2.4 Priority Queues

The package NBAda.Lock_Free_Priority_Queues

NBAda contains a lock-free dynamic priority queue data structure based on my (unpublished) modification of Michael's list-based lock-free set algorithm [Mic02a].

The include flag for `nbada_config` is `LF_PRIORITY_QUEUES_EBMR` or `LF_PRIORITY_QUEUES_HPMR`.

```
generic

    type Element_Type is private;

    with function "<" (Left, Right : Element_Type) return Boolean is <>;
    --  Note: Element_Type must be totally ordered.

    with package Process_Ids is
      new Process_Identification (<>);
    --  Process identification.

package NBAda.Lock_Free_Priority_Queues is

    type Priority_Queue_Type is limited private;

    Queue_Empty      : exception;
    Already_Present  : exception;

    procedure Initialize (Queue : in out Priority_Queue_Type);

    procedure Insert    (Into     : in out Priority_Queue_Type;
                         Element : in      Element_Type);

    procedure Delete_Min (From     : in out Priority_Queue_Type;
                          Element :      out Element_Type);
    function   Delete_Min (From : in Priority_Queue_Type)
                          return Element_Type;
    function   Delete_Min (From : access Priority_Queue_Type)
                          return Element_Type;

private

        ...  --  Implementation specific

end NBAda.Lock_Free_Priority_Queues;
```

Application constraints:

- Any task that calls an operation in NBAda.Lock_Free_Priority_Queues must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

- The function "<" on Element_Type MUST define a total order.

- The operation Initialize SHOULD NOT be used when concurrent access to the priority queue object is possible.

3.2.5 Dictionaries and Sets

The package NBAda.Lock_Free_Sets

NBAda contains a lock-free dynamic set data structure based on the list-based lock-free set algorithm by Michael [Mic02a].

The include flag for `nbada_config` is `LF_SETS_EBMR` or `LF_SETS_HPMR`.

```ada
   generic

     type Value_Type is private;
     type Key_Type is private;

     with function "<" (Left, Right : Key_Type) return Boolean is <>;
     --   Note: Key_Type must be totally ordered.

     with package Process_Ids is
       new Process_Identification (<>);
     --   Process identification.

   package NBAda.Lock_Free_Sets is
     type Set_Type is limited private;

     Not_Found        : exception;
     Already_Present  : exception;

     procedure Init     (Set : in out Set_Type);

     procedure Insert   (Into  : in out Set_Type;
                         Key   : in      Key_Type;
                         Value : in      Value_Type);

     procedure Delete   (From : in out Set_Type;
                         Key  : in      Key_Type);

     function  Find     (In_Set : in Set_Type;
                         Key    : in Key_Type) return Value_Type;

   private

         ...   --   Implementation specific

   end NBAda.Lock_Free_Sets;
```

Application constraints:

- Any task that calls an operation in NBAda.Lock_Free_Sets must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

- The function "<" on Element_Type MUST define a total order.

- The operation Init SHOULD NOT be used when concurrent access to the set object is possible.

The package NBAda.Lock_Free_Dictionaries

NBAda contains a lock-free dynamic dictionary data structure based on the lock-free hash table and set algorithms by Michael [Mic02a].

The include flag for `nbada_config` is `LF_DICTIONARIES_EBMR` or `LF_DICTIONARIES_HPMR`.

```ada
   generic

     type Value_Type is private;
     type Key_Type is private;

     with function Hash (Key        : Key_Type;
                         Table_Size : Positive) return Natural;

     with function "<" (Left, Right : Key_Type) return Boolean is <>;
     --   Note: Key_Type must be totally ordered.

     with package Process_Ids is
       new NBAda.Process_Identification (<>);
```

```
14      --   Process identification .

16  package NBAda.Lock_Free_Dictionaries is

18      type Dictionary_Type (No_Buckets : Natural) is limited private ;

20      Not_Found        : exception ;
        Already_Present : exception ;
22
        procedure Init     (Dictionary : in out Dictionary_Type );
24
        procedure Insert  (Into  : in out Dictionary_Type ;
26                          Key   : in     Key_Type ;
                            Value : in     Value_Type );
28
        procedure Delete  (From  : in out Dictionary_Type ;
30                          Key   : in     Key_Type );

32      function  Lookup  (From  : in Dictionary_Type ;
                           Key   : in Key_Type)
34                         return Value_Type ;

36  private

38          ...   --   Implementation specific

40  end NBAda.Lock_Free_Dictionaries ;
```

Application constraints:

- Any task that calls an operation in NBAda.Lock_Free_Dictionaries must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

- The function "<" on Element_Type MUST define a total order.

- The function Hash MUST return a value in the range 0 .. Table_Size for every value of Key_Type.

- The operation Init SHOULD NOT be used when concurrent access to the set object is possible.

Chapter 4

Support Packages

4.1 Memory Reclamation Algorithms

In a concurrent program it is often not obvious when it is safe to free a dynamically allocated block of memory (consider e.g. the case when another thread holds a local pointer to the object). In the absence of a concurrency safe (and lock-free) general garbage collector there are efficient lock-free memory reclamation algorithms that can solve this problem, provided that the application or data structure use them to manage dynamically allocated nodes and the references to them.

The memory reclamation algorithms distinguish the managed nodes into *live* nodes that are part of the logical state of the user data structure and *logically deleted* nodes that are not part of the logical state of the user data structure. In some memory reclamation algorithms the user data structure is expected to notify the memory reclamation algorithm when a node changes state to logically deleted, in others in can be deduced from reachability. The memory reclamation algorithm will delay the actual reclamation of a logically deleted until there cannot be any potentially accesses to the node form any thread (using the memory reclamation API).

There are two different levels of service or "protection" offered by memory reclamation algorithms, I define them as follows:

- **Reclamation safe private references**. The memory reclamation algorithm only safeguards nodes referenced by private (task local) references, i.e. does not safeguard shared references. The application needs to take care that the shared references it uses cannot reference logically deleted nodes. E.g. applications can usually only follow (dereference) references in nodes it *knows* are alive.

- **Reclamation safe private and shared references**. The memory reclamation algorithm safeguards all private and shared references. The application can safely dereference any shared reference.

See [GPST08] for a more thorough treatment of lock-free memory reclamation algorithms and their properties.

NBAda includes implementations of several memory reclamation algorithms of both service levels.

4.1.1 Reclamation safe private references

NBAda includes implementations of the Hazard Pointers lock-free memory reclamation algorithm by Michael [Mic02b, Mic04a] (NBAda.Hazard_Pointers) and the epoch based concurrent memory reclamation algorithm described in [Fra04, Har05] (NBAda.Epoch_Based_Memory_Reclamation).

The intention is that the two packages should be API compatible.

The package NBAda.Hazard_Pointers and
the package NBAda.Epoch_Based_Memory_Reclamation

Application constraints:

- Any task that calls an memory reclamation operation must have registered an identity by
  calling the operation Register of the appropriate instance of NBAda.Process_Identification.

```
   generic
 2     Max_Number_Of_Dereferences : Natural;
       ---  Maximum number of simultaneously dereferenced links per thread.
 4     with package Process_Ids is
         new Process_Identification (<>);
 6     ---  Process identification.

 8     Integrity_Checking : Boolean := False;
       ---  Enable strong integrity checking.
10     Verbose_Debug : Boolean := False;
       ---  Enable verbose debug output.
12 package NBAda.Hazard_Pointers is

14     type Managed_Node_Base is abstract tagged limited private;
       ---  Inherit from this base type to create your own managed types.
16
       procedure Free (Object : access Managed_Node_Base) is abstract;
18
       generic
20        type Managed_Node is new Managed_Node_Base with private;
       package Operations is
22
         type Shared_Reference is limited private;
24        ---  Note: All shared variables of type Shared_Reference MUST be
          ---         declared atomic by 'pragma Atomic (Variable_Name);' .
26
         type Node_Access is access all Managed_Node;
28        ---  Note: There SHOULD NOT be any shared variables of type
          ---         Node_Access.
30
         function  Dereference (Shared : access Shared_Reference)
32                              return Node_Access;
          ---  Note:
34
         procedure Release    (Local  : in Node_Access);
36        ---  Note: Each dereferenced shared pointer MUST be released
          ---         eventually.
38
         procedure Delete     (Local  : in Node_Access);
40        ---  Note: Delete may only be called when the caller can
          ---         guarantee that there are NO and WILL NOT BE any more shared
42        ---         references to the node. The memory management scheme makes
          ---         sure the node is not freed until all local references have
44        ---         been released.
46        function  Boolean_Compare_And_Swap (Shared    : access Shared_Reference;
                                              Old_Value : in      Node_Access;
48                                            New_Value : in      Node_Access)
                                              return Boolean;
50
         procedure Value_Compare_And_Swap    (Shared    : access Shared_Reference;
52                                            Old_Value : in      Node_Access;
                                              New_Value : in out Node_Access);
54
         procedure Void_Compare_And_Swap     (Shared    : access Shared_Reference;
56                                            Old_Value : in      Node_Access;
                                              New_Value : in      Node_Access);
58
```

```
60      procedure Initialize (Shared    : access Shared_Reference;
                             New_Value : in      Node_Access);
62      —— Note: Initialize is only safe to use when there are no
        ——       concurrent updates.
64
    private
66
        type Shared_Reference is new Node_Access;
68      —— Note: All shared variables of type Shared_Reference MUST be
        ——       declared atomic by 'pragma Atomic (Variable_Name);' .
70
    end Operations;
72

74  type Shared_Reference_Base is limited private;
    —— For type separation between shared references to different
76  —— managed types derive your own shared reference types from
    —— Shared_Reference_Base and instantiate the memory management
78  —— operation package below for each of them.

80  generic

82      type Managed_Node is
          new Managed_Node_Base with private;
84
        type Shared_Reference is new Shared_Reference_Base;
86      —— All shared variables of type Shared_Reference MUST be declared
        —— atomic by 'pragma Atomic (Variable_Name);' .
88
    package Reference_Operations is
90
        type Node_Access is access all Managed_Node;
92      —— Note: There SHOULD NOT be any shared variables of type
        ——       Node_Access.
94
        type Private_Reference is private;
96      —— Note: There SHOULD NOT be any shared variables of type
        ——       Private_Reference.
98  Null_Reference : constant Private_Reference;
    —— Note: A marked null reference is not equal to Null_Reference.
100
        function  Dereference (Link : access Shared_Reference)
102                         return Private_Reference;

104     procedure Release (Node : in Private_Reference);

106     function  "+"      (Node : in Private_Reference)
                    return Node_Access;
108     function  Deref    (Node : in Private_Reference)
                    return Node_Access;
110
        function  Boolean_Compare_And_Swap (Link      : access Shared_Reference;
112                                          Old_Value : in Private_Reference;
                                             New_Value : in Private_Reference)
114                                      return Boolean;

116     procedure Void_Compare_And_Swap   (Link      : access Shared_Reference;
                                           Old_Value : in Private_Reference;
118                                         New_Value : in Private_Reference);

120     procedure Delete   (Node : in Private_Reference);

122     procedure Store    (Link : access Shared_Reference;
                          Node : in Private_Reference);
124     —— Note: Store is only safe to use when there cannot be any
        ——       concurrent updates to Link.
126
```

```
          generic
128          type User_Node_Access is access Managed_Node;
             --   Select an appropriate (preferably non-blocking) storage
130          --   pool by the "for User_Node_Access'Storage_Pool use ..."
             --   construct.
132          --   Note: The nodes allocated in this way must have an
             --         implementation of Free that use the same storage pool.
134      function Create return Private_Reference;
         --   Creates a new User_Node and returns a safe reference to it.
136
         procedure Mark       (Node : in out Private_Reference);
138      function   Mark       (Node : in      Private_Reference)
                                return Private_Reference;
140      procedure Unmark      (Node : in out Private_Reference);
         function   Unmark      (Node : in      Private_Reference)
142                             return Private_Reference;
         function   Is_Marked (Node : in      Private_Reference)
144                             return Boolean;

146      function   Is_Marked (Node : in      Shared_Reference)
                                return Boolean;
148
         function "=" (Link : in      Shared_Reference;
150                    Ref  : in      Private_Reference) return Boolean;
         function "=" (Ref  : in      Private_Reference;
152                    Link : in      Shared_Reference) return Boolean;

154   private

156       ...   --   Implementation details.

158   end Reference_Operations;

160   procedure Print_Statistics;

162 private

164   ...   --   Implementation details.

166 end NBAda.Hazard_Pointers;
```

4.1.2   Reclamation safe private and shared references

NBAda contains implementations of two memory reclamation algorithms that safeguards all private and shared references. The two algorithms are the lock-free reference counting algorithm SL-FRC by Herlihy et al. [HLM02, HLMM02, HLMM05] (NBAda.Lock_Free_Reference_Counting) and the lock-free reclamation algorithm Beware & Cleanup by Gidenstam et al. [GPST05] (NBAda.Lock_Free_Memory_Recla

**The package NBAda.Lock_Free_Reference_Counting and
the package NBAda.Lock_Free_Memory_Reclamation**

Application constraints:

- Any task that calls an memory reclamation operation must have registered an identity by calling the operation Register of the appropriate instance of NBAda.Process_Identification.

```
  generic
2
    Max_Number_Of_Dereferences : Natural;
4   --   Maximum number of simultaneously dereferenced links per thread.

6   Max_Number_Of_Links_Per_Node : Natural;
    --   Maximum number of links in a shared node.
8
```

```ada
      with package Process_Ids is
10       new NBAda.Process_Identification (<>);
      --  Process identification.

12
      Max_Delete_List_Size       : Natural :=
14       Process_Ids.Max_Number_Of_Processes ** 2 *
          (Max_Number_Of_Dereferences + Max_Number_Of_Links_Per_Node +
16         Max_Number_Of_Links_Per_Node + 1);

18    Clean_Up_Threshold          : Natural := Max_Delete_List_Size;
      --  The threshold on the delete list size for Clean_Up to be done.

20
      Scan_Threshold              : Natural := Clean_Up_Threshold;
22    --  The threshold on the delete list size for Scan to be done.

24    Collect_Statistics          : Boolean := True;
      --  Enable some statics gathering.

26
   package NBAda.Lock_Free_Memory_Reclamation is

28
      type Managed_Node_Base is abstract tagged limited private;
30    --  Inherit from this base type to create your own managed types.

32    procedure Dispose   (Node      : access Managed_Node_Base;
                           Concurrent : in     Boolean) is abstract;
34    --  Dispose should set all shared references inside the node to null.

36    procedure Clean_Up (Node : access Managed_Node_Base) is abstract;
      --  Clean_Up should make sure that none of the shared references
38    --  inside the node points to a node that was deleted at the point
      --  in time when Clean_Up was called.

40
      function Is_Deleted (Node : access Managed_Node_Base)
42                         return Boolean;
      --  Returns true if Delete (see below) has been called on the node.

44
      procedure Free (Object : access Managed_Node_Base) is abstract;
46    --  Note: Due to some peculiarities of the Ada storage pool
      --          management managed nodes need to have a dispatching primitive
48    --          operation that calls the instance of Unchecked_Deallocation
      --          appropriate for the specific node type at hand. Without
50    --          this the wrong instance of Unchecked_Deallocation might get
      --          called - often with disastrous consequences as it tries return
52    --          the memory to the wrong storage pool.

54    type Shared_Reference_Base is limited private;
      --  For type separation between shared references to different
56    --  managed types derive your own shared reference types from
      --  Shared_Reference_Base and instantiate the memory management
58    --  operation package below for each of them.

60    type Shared_Reference_Base_Access is access all Shared_Reference_Base;
      type Reference_Set is array (Integer range <>) of
62      Shared_Reference_Base_Access;
      --  These two types are defined for compatibility with the
64    --  Lock_Free_Reference_Counting package.

66    generic

68       type Managed_Node is
           new Managed_Node_Base with private;

70
         type Shared_Reference is new Shared_Reference_Base;
72       --  All shared variables of type Shared_Reference MUST be declared
         --  atomic by 'pragma Atomic (Variable_Name);' .

74
      package Operations is
```

```
76
        type Node_Access is access all Managed_Node;
78      --   Note: There SHOULD NOT be any shared variables of type
        --          Node_Access.
80
        type Private_Reference is private;
82      --   Note: There SHOULD NOT be any shared variables of type
        --          Private_Reference.
84      Null_Reference : constant Private_Reference;
        function Image (R : Private_Reference) return String;
86
        function   Dereference (Link : access Shared_Reference)
88                          return Private_Reference;

90      procedure Release (Node : in Private_Reference);

92      function   "+"      (Node : in Private_Reference)
                        return Node_Access;
94      function   Deref    (Node : in Private_Reference)
                        return Node_Access;
96
        function   Copy (Node : in Private_Reference) return Private_Reference;
98      --   Creates a new Private Reference to Node. Both will need to be
        --   released.
100
        function   Compare_And_Swap (Link      : access Shared_Reference;
102                                  Old_Value : in Private_Reference;
                                     New_Value : in Private_Reference)
104                                  return Boolean;

106     procedure Compare_And_Swap (Link      : access Shared_Reference;
                                    Old_Value : in      Private_Reference;
108                                 New_Value : in      Private_Reference);

110     procedure Delete   (Node : in Private_Reference);

112
        procedure Store    (Link : access Shared_Reference;
114                         Node : in Private_Reference);

116     generic
            type User_Node_Access is access Managed_Node;
118         --   Select an appropriate (preferably non-blocking) storage
            --   pool by the "for User_Node_Access'Storage_Pool use ..."
120         --   construct.
            --   Note: The nodes allocated in this way must have an
122         --          implementation of Free that use the same storage pool.
        function Create return Private_Reference;
124     --   Creates a new User_Node and returns a safe reference to it.

126     --   Private (and shared) references can be tagged with a mark.
        --   NOTE: A marked Null_Reference is not equal (=) to an unmarked.
128     procedure Mark       (Node : in out Private_Reference);

130     function   Mark      (Node : in      Private_Reference)
                        return Private_Reference;
132     procedure Unmark     (Node : in out Private_Reference);
        function   Unmark    (Node : in      Private_Reference)
134                     return Private_Reference;
        function   Is_Marked (Node : in      Private_Reference)
136                     return Boolean;

138     function   Is_Marked (Node : in      Shared_Reference)
                        return Boolean;
140
        function "=" (Left  : in      Private_Reference;
142              Right : in      Private_Reference) return Boolean;
```

```
       function "=" (Link : in       Shared_Reference;
144                   Ref  : in       Private_Reference) return Boolean;
       function "=" (Ref  : in       Private_Reference;
146                  Link : in       Shared_Reference) return Boolean;
       --    It is possible to compare a reference to the current value of a link.
148
       _____
150    --   Unsafe operations.
       --   These SHOULD only be use when the user algorithm guarantees
152    --   the absence of ABA-problems.
       --   In such algorithms the use of these operations in some particular
154    --   situations could allow some performance improving optimizations.
       _____
156
       type Unsafe_Reference_Value is private;
158    --   Note: An Unsafe_Reference_Value does not keep a claim to any
       --         node and can therefore only be used where ABA safety is
160    --         ensured by other means. It cannot be dereferenced.

162    function   Unsafe_Read (Link : access Shared_Reference)
                             return Unsafe_Reference_Value;
164
       function   Compare_And_Swap (Link      : access Shared_Reference;
166                                  Old_Value : in Unsafe_Reference_Value;
                                     New_Value : in Private_Reference)
168                                  return Boolean;
       function   Compare_And_Swap (Link      : access Shared_Reference;
170                                  Old_Value : in Unsafe_Reference_Value;
                                     New_Value : in Unsafe_Reference_Value)
172                                  return Boolean;
       procedure Compare_And_Swap (Link      : access Shared_Reference;
174                                 Old_Value : in       Unsafe_Reference_Value;
                                    New_Value : in       Private_Reference);
176    procedure Compare_And_Swap (Link      : access Shared_Reference;
                                    Old_Value : in       Unsafe_Reference_Value;
178                                 New_Value : in       Unsafe_Reference_Value);

180    function   Is_Marked (Node : in       Unsafe_Reference_Value)
                            return Boolean;
182
       function   Mark      (Node : in       Unsafe_Reference_Value)
184                         return Unsafe_Reference_Value;

186    function "=" (Val : in       Unsafe_Reference_Value;
                     Ref : in       Private_Reference) return Boolean;
188    function "=" (Ref : in       Private_Reference;
                     Val : in       Unsafe_Reference_Value) return Boolean;
190
       function "=" (Link : in       Shared_Reference;
192                  Ref  : in       Unsafe_Reference_Value) return Boolean;
       function "=" (Ref  : in       Unsafe_Reference_Value;
194                  Link : in       Shared_Reference) return Boolean;

196   private

198      ... --   Implementation details.

200   end Operations;

202   procedure Print_Statistics;

204 private

206    ... --    Implementation details.

208 end NBAda.Lock_Free_Memory_Reclamation;
```

4.2   Memory Allocation Pools

The package NBAda.Lock_Free_Fixed_Size_Storage_Pools

NBADA contains a generic fixed size lock-free storage pool based on the lock-free free-list algorithm in [IBM83].

Application constraints:

- A pool instance MUST NOT be used for object that have storage size larger than Block_Size.

```
   package NBAda.Lock_Free_Fixed_Size_Storage_Pools is
2
      type Block_Count is range 0 .. 2**16 − 1;
4
      type Lock_Free_Storage_Pool
6        (Pool_Size   : Block_Count;
          Block_Size  : System.Storage_Elements.Storage_Count) is
8        new System.Storage_Pools.Root_Storage_Pool with private;

10     procedure Allocate
         (Pool                   : in out Lock_Free_Storage_Pool;
12        Storage_Address        :    out System.Address;
          Size_In_Storage_Elements : in    System.Storage_Elements.Storage_Count;
14        Alignment              : in     System.Storage_Elements.Storage_Count);

16     procedure Deallocate
         (Pool                   : in out Lock_Free_Storage_Pool;
18        Storage_Address        : in     System.Address;
          Size_In_Storage_Elements : in    System.Storage_Elements.Storage_Count;
20        Alignment              : in     System.Storage_Elements.Storage_Count);

22     function Storage_Size (Pool : Lock_Free_Storage_Pool)
                             return System.Storage_Elements.Storage_Count;
24
       function Validate (Pool : Lock_Free_Storage_Pool)
26                      return Block_Count;

28     function Belongs_To (Pool            : Lock_Free_Storage_Pool;
                           Storage_Address : System.Address)
30                      return Boolean;

32     Storage_Exhausted    : exception;
       Implementation_Error : exception;
34
   private
36
       ...   −−   Implementation details.
38
   end NBAda.Lock_Free_Fixed_Size_Storage_Pools;
```

The package NBAda.Lock_Free_Growing_Storage_Pools

The growing storage pool in NBADA automatically grows in size when the memory demand warrants it. It never shirks, however.

Application constraints:

- A pool instance MUST NOT be used for object that have storage size larger than Block_Size.

```
   package NBAda.Lock_Free_Growing_Storage_Pools is
2
      type Lock_Free_Storage_Pool
```

```
4        ( Block_Size : System . Storage_Elements . Storage_Count) is
         new System . Storage_Pools . Root_Storage_Pool with private ;
6
     procedure Allocate
8       (Pool                        :  in out  Lock_Free_Storage_Pool ;
         Storage_Address             :      out System . Address ;
10       Size_In_Storage_Elements :  in       System . Storage_Elements . Storage_Count ;
         Alignment                   :  in       System . Storage_Elements . Storage_Count );
12
     procedure Deallocate
14      (Pool                        :  in out  Lock_Free_Storage_Pool ;
         Storage_Address             :  in       System . Address ;
16       Size_In_Storage_Elements :  in       System . Storage_Elements . Storage_Count ;
         Alignment                   :  in       System . Storage_Elements . Storage_Count );
18
     function Storage_Size (Pool : Lock_Free_Storage_Pool )
20                           return System . Storage_Elements . Storage_Count ;

22   function Validate (Pool : Lock_Free_Storage_Pool )
                          return Natural ;
24
     Storage_Exhausted : exception ;
26   Implementation_Error : exception ;

28 private

30   ...  --   Implementation  details .

32 end NBAda . Lock_Free_Growing_Storage_Pools ;
```

4.3   Hardware Abstraction Interface

The package NBAda.Primitives

```
  package NBAda . Primitives is
2
     Not_Implemented : exception ;
4
     procedure Membar ;
6
     type Standard_Unsigned is mod 2**System . Word_Size ;
8    pragma Atomic (Standard_Unsigned );

10   generic
        --   Element ' Object_Size  MUST be  System . Word_Size .
12      type Element is private ;
     function Standard_Atomic_Read (Target : access Element) return Element ;
14
     generic
16      --   Element ' Object_Size  MUST be  System . Word_Size .
        type Element is private ;
18   procedure Standard_Atomic_Write (Target : access Element ;
                                      Value : in       Element );
20
     generic
22      --   Element ' Object_Size  MUST be  System . Word_Size .
        type Element is private ;
24   procedure Standard_Compare_And_Swap (Target    : access Element ;
                                          Old_Value : in       Element ;
26                                         New_Value : in out Element );

28   generic
        --   Element ' Object_Size  MUST be  System . Word_Size .
30      type Element is private ;
     function Standard_Boolean_Compare_And_Swap (Target    : access Element ;
```

```
32                                                            Old_Value : in       Element;
                                                             New_Value : in       Element)
34                                                            return Boolean;

36   generic
        -- Element'Object_Size MUST be System.Word_Size.
38      type Element is private;
     procedure Standard_Void_Compare_And_Swap (Target    : access Element;
40                                              Old_Value : in       Element;
                                                New_Value : in       Element);

42
     procedure Fetch_And_Add (Target    : access Standard_Unsigned;
44                            Increment : in       Standard_Unsigned);

46   function   Fetch_And_Add (Target    : access Standard_Unsigned;
                               Increment : in       Standard_Unsigned)
48                            return Standard_Unsigned;


50
     type Unsigned_32 is mod 2**32;
52   pragma Atomic (Unsigned_32);

54   generic
        -- Element'Object_Size MUST be 32.
56      type Element is private;
     function Atomic_Read_32 (Target : access Element) return Element;

58
     generic
60      -- Element'Object_Size MUST be 32.
        type Element is private;
62   procedure Atomic_Write_32 (Target : access Element;
                                Value  : in       Element);

64
     generic
66      -- Element'Object_Size MUST be 32.
        type Element is private;
68   procedure Compare_And_Swap_32 (Target    : access Element;
                                    Old_Value : in       Element;
70                                   New_Value : in out Element);

72   generic
        -- Element'Object_Size MUST be 32.
74      type Element is private;
     function Boolean_Compare_And_Swap_32 (Target    : access Element;
76                                          Old_Value : in       Element;
                                            New_Value : in       Element)
78                                          return Boolean;

80   generic
        -- Element'Object_Size MUST be 32.
82      type Element is private;
     procedure Void_Compare_And_Swap_32 (Target    : access Element;
84                                        Old_Value : in       Element;
                                          New_Value : in       Element);
86
     procedure Fetch_And_Add_32 (Target    : access Unsigned_32;
88                               Increment : in       Unsigned_32);

90   function   Fetch_And_Add_32 (Target    : access Unsigned_32;
                                  Increment : in       Unsigned_32)
92                               return Unsigned_32;


94   type Unsigned_64 is mod 2**64;
     pragma Atomic (Unsigned_64);
96
     generic
98      -- Element'Object_Size MUST be 64.
```

```ada
          type Element is private;
100       function Atomic_Read_64 (Target : access Element) return Element;

102       generic
             --  Element'Object_Size MUST be 64.
104          type Element is private;
          procedure Atomic_Write_64 (Target : access Element;
106                                   Value  : in     Element);

108       generic
             --  Element'Object_Size MUST be 64.
110          type Element is private;
          procedure Compare_And_Swap_64 (Target    : access Element;
112                                       Old_Value : in     Element;
                                          New_Value : in out Element);

114
          generic
116          --  Element'Object_Size MUST be 64.
             type Element is private;
118       function Boolean_Compare_And_Swap_64 (Target    : access Element;
                                                Old_Value : in     Element;
120                                              New_Value : in     Element)
                                                return Boolean;

122
          generic
124          --  Element'Object_Size MUST be 64.
             type Element is private;
126       procedure Void_Compare_And_Swap_64 (Target    : access Element;
                                              Old_Value : in     Element;
128                                            New_Value : in     Element);

130       procedure Fetch_And_Add_64 (Target    : access Unsigned_64;
                                      Increment : in     Unsigned_64);
132
          function  Fetch_And_Add_64 (Target    : access Unsigned_64;
134                                    Increment : in     Unsigned_64)
                                      return Unsigned_64;
136
       end NBAda.Primitives;
```

The package NBAda.Process_Identification

```ada
   generic
2     Max_Number_Of_Processes : Natural;
   package NBAda.Process_Identification is
4
      type Process_ID_Type is new Natural range 1 .. Max_Number_Of_Processes;
6
      --  Register a process ID for this task.
8     procedure Register;

10    --  Returns the process ID of the calling task.
      function Process_ID return Process_ID_Type;
12
   end NBAda.Process_Identification;
```

Bibliography

[Bar93]   Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the 5th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.

[DG02]   Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *ISMM '02 Proceedings of the 3rd International Symposium on Memory Management*, ACM SIGPLAN Notices, pages 163–174. ACM Press, June 2002.

[Fra04]   Keir A. Fraser. *Practical lock-freedom*. PhD thesis, University of Cambridge, February 2004.

[GC96]   Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[Gid06]   Anders Gidenstam. *Algorithms for synchronization and consistency in concurrent system services*. PhD thesis, Chalmers University of Technology, 2006.

[GPST05]   Anders Gidenstam, Marina Papatriantafilou, Hkan Sundell, and Philippas Tsigas. Practical and efficient lock-free garbage collection based on reference counting. In *Proceedings of the 8th International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN)*, pages 202–207. IEEE Computer Society, December 2005.

[GPST08]   Anders Gidenstam, Marina Papatriantafilou, Håkan Sundell, and Philippas Tsigas. Efficient and reliable lock-free memory reclamation based on reference counting. *IEEE Transactions on Parallel and Distributed Systems*, August 2008. Preprint.

[GPT05]   Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. Allocating memory in a lock-free manner. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 329–242. Springer Verlag, October 2005.

[Har05]   Thomas E. Hart. Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures. Master's thesis, University of Toronto, 2005.

[Her91]   Maurice Herlihy. Wait-free synchronization. *ACM Transaction on Programming and Systems*, 11(1):124–149, January 1991.

[Her93]   Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[HLM02]   Maurice Herlihy, Victor Luchangco, and Mark Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of 16th International Symposium on Distributed Computing (DISC 2002)*, volume 2508 of *Lecture Notes in Computer Science*, pages 339–353. Springer Verlag, October 2002.

[HLMM02]   Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Dynamic-sized lock-free data structures. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pages 131–131. ACM Press, 2002.

[HLMM05]   Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Nonblocking memory management support for dynamic-sized data structures. *ACM Transactions on Computer Systems*, 23(2):146–196, 2005.

[HSS07]   Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *Proceedings of the 11th International Conference On the Principles Of Distributed Systems (OPODIS'07)*, volume 4878 of *Lecture Notes in Computer Science*, pages 401–414. Springer-Verlag, 2007.

[HW90]   Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IBM83]   IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.

[Jay05]   Prasad Jayanti. An optimal multi-writer snapshot algorithm. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing (STOC'05)*, pages 723–732. ACM Press, 2005.

[LGH⁺04]   Andreas Larsson, Anders Gidenstam, Phuong H Ha, Marina Papatriantafilou, and Philippas Tsigas. Multi-word atomic read/write registers on multiprocessor systems. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA'04) LNCS 3221*, pages 736–748. Springer-Verlag, September 2004.

[Mic02a]   Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA-02)*, pages 73–82. ACM Press, August 2002.

[Mic02b]   Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 21–30. ACM Press, 2002.

[Mic04a]   Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(8), August 2004.

[Mic04b]   Maged M. Michael. Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In *Proceedings of the 18th International Conference on Distributed Computing (DISC '04)*, October 2004.

[Mic04c]   Maged M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of SIGPLAN 2004 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices. ACM Press, June 2004.

[MS96]   Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275. ACM Press, 1996.

[Pet83]   Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[Rin99]   Martin C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, 1999.

[SAN06]   Scott Schneider, Christos Antonopoulos, and Dimitrios Nikolopoulos. Scalabel locality-conscious multithreaded memory allocation. In *Proceedings of the 2006 international symposium on Memory management (ISMM'06)*, pages 84–94. ACM, June 2006.

[SRL90]   Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[ST04]   Håkan Sundell and Philippas Tsigas. Lock-free and practical deques using single-word compare-and-swap. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS '04)*, volume 3544 of *Lecture Notes in Computer Science*. Springer Verlag, December 2004.

[TZ01a]   Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronisation on shared-memory multiprocessors. In *Proc. of the ACM SIGMETRICS 2001/Performance 2001*, pages 320–321. ACM press, June 2001.

[TZ01b]   Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143. ACM Press, 2001.

[TZ02]   Philippas Tsigas and Yi Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proc. of the 3rd ACM Workshop on Software and Performance (WOSP'02)*, pages 55–67. ACM press, July 2002.

Appendix A

GNU General Public Licence

```
    GNU GENERAL PUBLIC LICENSE
       Version 2, June 1991

 Copyright (C) 1989, 1991 Free Software Foundation, Inc.
  59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

    Preamble

  The licenses for most software are designed to take away your
freedom to share and change it.  By contrast, the GNU General Public
License is intended to guarantee your freedom to share and change free
software--to make sure the software is free for all its users.  This
General Public License applies to most of the Free Software
Foundation's software and to any other program whose authors commit to
using it.  (Some other Free Software Foundation software is covered by
the GNU Library General Public License instead.)  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
this service if you wish), that you receive source code or can get it
if you want it, that you can change the software or use pieces of it
in new free programs; and that you know you can do these things.

  To protect your rights, we need to make restrictions that forbid
anyone to deny you these rights or to ask you to surrender the rights.
These restrictions translate to certain responsibilities for you if you
distribute copies of the software, or if you modify it.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must give the recipients all the rights that
you have.  You must make sure that they, too, receive or can get the
source code.  And you must show them these terms so they know their
rights.

  We protect your rights with two steps: (1) copyright the software, and
(2) offer you this license which gives you legal permission to copy,
distribute and/or modify the software.
```

Also, for each author's protection and ours, we want to make certain
that everyone understands that there is no warranty for this free
software. If the software is modified by someone else and passed on, we
want its recipients to know that what they have is not the original, so
that any problems introduced by others will not reflect on the original
authors' reputations.

Finally, any free program is threatened constantly by software
patents. We wish to avoid the danger that redistributors of a free
program will individually obtain patent licenses, in effect making the
program proprietary. To prevent this, we have made it clear that any
patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and
modification follow.

    GNU GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains
a notice placed by the copyright holder saying it may be distributed
under the terms of this General Public License. The "Program", below,
refers to any such program or work, and a "work based on the Program"
means either the Program or any derivative work under copyright law:
that is to say, a work containing the Program or a portion of it,
either verbatim or with modifications and/or translated into another
language. (Hereinafter, translation is included without limitation in
the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope. The act of
running the Program is not restricted, and the output from the Program
is covered only if its contents constitute a work based on the
Program (independent of having been made by running the Program).
Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's
source code as you receive it, in any medium, provided that you
conspicuously and appropriately publish on each copy an appropriate
copyright notice and disclaimer of warranty; keep intact all the
notices that refer to this License and to the absence of any warranty;
and give any other recipients of the Program a copy of this License
along with the Program.

You may charge a fee for the physical act of transferring a copy, and
you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion
of it, thus forming a work based on the Program, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) You must cause the modified files to carry prominent notices
    stating that you changed the files and the date of any change.

    b) You must cause any work that you distribute or publish, that in
    whole or in part contains or is derived from the Program or any
    part thereof, to be licensed as a whole at no charge to all third

parties under the terms of this License.

c) If the modified program normally reads commands interactively
when run, you must cause it, when started running for such
interactive use in the most ordinary way, to print or display an
announcement including an appropriate copyright notice and a
notice that there is no warranty (or else, saying that you provide
a warranty) and that users may redistribute the program under
these conditions, and telling the user how to view a copy of this
License. (Exception: if the Program itself is interactive but
does not normally print such an announcement, your work based on
the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If
identifiable sections of that work are not derived from the Program,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works. But when you
distribute the same sections as part of a whole which is a work based
on the Program, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Program.

In addition, mere aggregation of another work not based on the Program
with the Program (or with a work based on the Program) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

  3. You may copy and distribute the Program (or a work based on it,
under Section 2) in object code or executable form under the terms of
Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable
source code, which must be distributed under the terms of Sections
1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three
years, to give any third party, for a charge no more than your
cost of physically performing source distribution, a complete
machine-readable copy of the corresponding source code, to be
distributed under the terms of Sections 1 and 2 above on a medium
customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer
to distribute corresponding source code. (This alternative is
allowed only for noncommercial distribution and only if you
received the program in object code or executable form with such
an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for
making modifications to it. For an executable work, complete source
code means all the source code for all modules it contains, plus any
associated interface definition files, plus the scripts used to

control compilation and installation of the executable. However, as a
special exception, the source code distributed need not include
anything that is normally distributed (in either source or binary
form) with the major components (compiler, kernel, and so on) of the
operating system on which the executable runs, unless that component
itself accompanies the executable.

If distribution of executable or object code is made by offering
access to copy from a designated place, then offering equivalent
access to copy the source code from the same place counts as
distribution of the source code, even though third parties are not
compelled to copy the source along with the object code.

   4. You may not copy, modify, sublicense, or distribute the Program
except as expressly provided under this License. Any attempt
otherwise to copy, modify, sublicense or distribute the Program is
void, and will automatically terminate your rights under this License.
However, parties who have received copies, or rights, from you under
this License will not have their licenses terminated so long as such
parties remain in full compliance.

   5. You are not required to accept this License, since you have not
signed it. However, nothing else grants you permission to modify or
distribute the Program or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by
modifying or distributing the Program (or any work based on the
Program), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Program or works based on it.

   6. Each time you redistribute the Program (or any work based on the
Program), the recipient automatically receives a license from the
original licensor to copy, distribute or modify the Program subject to
these terms and conditions. You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties to
this License.

   7. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Program at all. For example, if a patent
license would not permit royalty-free redistribution of the Program by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under
any particular circumstance, the balance of the section is intended to
apply and the section as a whole is intended to apply in other
circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any

such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system, which is
implemented by public license practices. Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

  8. If the distribution and/or use of the Program is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Program under this License
may add an explicit geographical distribution limitation excluding
those countries, so that distribution is permitted only in or among
countries not thus excluded. In such case, this License incorporates
the limitation as if written in the body of this License.

  9. The Free Software Foundation may publish revised and/or new versions
of the General Public License from time to time. Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

Each version is given a distinguishing version number. If the Program
specifies a version number of this License which applies to it and "any
later version", you have the option of following the terms and conditions
either of that version or of any later version published by the Free
Software Foundation. If the Program does not specify a version number of
this License, you may choose any version ever published by the Free Software
Foundation.

  10. If you wish to incorporate parts of the Program into other free
programs whose distribution conditions are different, write to the author
to ask for permission. For software which is copyrighted by the Free
Software Foundation, write to the Free Software Foundation; we sometimes
make exceptions for this. Our decision will be guided by the two goals
of preserving the free status of all derivatives of our free software and
of promoting the sharing and reuse of software generally.

			NO WARRANTY

  11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY
FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES
PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED
OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS
TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE
PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING,
REPAIR OR CORRECTION.

  12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR
REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES,
INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING
OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED

TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY
YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER
PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE
POSSIBILITY OF SUCH DAMAGES.

      END OF TERMS AND CONDITIONS