RT++ — Higher Order Threads for C++ Tutorial and Reference Manual (Version 1.0)*

Wolfgang Schreiner

November 6, 1996

Abstract

This report presents RT++, a software package that provides higher-order threads in the programming language C++. Its features include a type-safe functional thread interface, lazy thread creation, garbage-collected types (lists, arrays, pointer structures) and controlled non-determinism (thread bags). Threads are first-order objects that can be used like any other objects and that are automatically reclaimed when they are not referenced any more. The package has been ported to numerous types of mono-processors and shared memory multiprocessors and can be easily embedded into existing application frameworks.

*Supported by the Austrian Science Foundation (FWF) grant S5302-PHY "Parallel Symbolic Computation" and by the Advanced Information Technology Program (AITP) of Information-technology Promotion Agency(IPA), Japan, in the frame of the project "Coordination Programming in Open Computing Environment". The online version of this document and the source code of the described system are available at `http://www.risc.uni-linz.ac.at/software/rt++`.

Contents

1 Introduction

This report describes RT++[1], a software package that provides higher-order threads for the programming language C++. "Yet another thread package" you may think but this is only partially true.

Of course RT++ performs what can be expected from any decent thread package. It gives the programmer the possibility to create threads, i.e. concurrent units of control or light-weight processes, and thus to write parallel (or at least pseudo-parallel) programs. The RT++ runtime system schedules these threads on one or several Unix processes which in turn are scheduled by the operating system. If the underlying hardware is a multi-processor, processes and thus threads are executed truly in parallel.

There exists a variety of thread packages that do the same (e.g. PRESTO [BLLW88], FastThreads [And90], uC++ [BDS92], Pthreads [Mue93], Chant [HCM94], Solaris threads [Sun96], see also the "Threads Page" [Thr96]). Still it is surprisingly difficult to find a package that has been ported to a larger portion of the of workstation and multi-processor brands that are in use today.

One of the most portable systems is QuickThreads [Kep93] which is actually a toolkit for building thread packages. QuickThreads provides for a variety of processor types the core routines for thread initialization and context switching and thus frees the poor thread package developer from the nasty task of assembler programming. RT++ is based on these routines and can thus be easily ported to any machine with a processor type supported by QuickThreads[2].

The main goal of RT++ however is to provide a programming interface which is considerably higher-level than that of comparable packages. RT++ considers threads as an *abstract datatype* with a *functional interface* i.e. threads communicate only via arguments (which are provided when the thread is defined) and results (which are delivered when the thread has terminated). Threads can be used like objects of any other type i.e. they can be stored in data structures, passed as thread arguments, and returned as thread results. Functional programming languages are called *higher-order* if they treat functions as first-order objects; we call RT++ higher-order because it treats threads in this way.

However, an abstract thread interface only makes sense if programmers need not be aware of such low-level issues as allocating and freeing thread resources. In most packages, the resources for a thread are allocated when the thread is created and are freed when the thread result is delivered to another (typically the parent) thread. This is a very low-level view of threads because (among other issues) it does not allow two retrieve twice the result of a thread (because the first result delivery destroys the thread) and, if a thread is forgotten, its resources remain allocated forever.

RT++ provides a thread/memory management system where programmers need not care about these low-level issues. Thread resources are implicitly allocated as late as possible and implicitly freed as early as possible regardless whether or how often the thread result is retrieved. Threads are subject to

[1] What the acronym stands for? Of course for "RISC-Linz Threads for C++"!

[2] However, not all of these ports have been done yet, see below.

garbage collection and automatically reclaimed when not referenced any more. RT++ also provides *type constructors* for arrays, lists, and general pointer structures whose objects can be passed by reference among threads and that are subject to automatic garbage collection. A high-level notion of non-determinism is supported via *thread bags* which retrieve their result in the order in which they become available and thus allow to write more abstract (and sometimes more efficient) parallel programs.

RT++ has been developed as a successor of the PACLIB kernel [HNS95] which served as the C-based runtime system of a parallel computer algebra package [SH93] and of a parallel functional programming language [Sch96]. RT++ owes to PACLIB its central ideas of thread scheduling [Sch94] which are based on "virtual threads" [KW92] respectively "lazy threads" [MKHJ91]. PACLIB was built upon the μSystem package [BS90]; the contents of this thread package taught us the essential concepts of multi-threading. The RT++ assembly code for mutual exclusion was mainly taken from the μSystem respectively its successor μC++ [BDS92].

RT++ considerably owes to the development group of the symbolic computation package STURM [HNS94] for communicating their experience with C++, in particular with the features and deficiencies of the "template" construct on which RT++ is heavily based. STURM uses reference counting for memory management and provides a parallel programming interface in the spirit of PACLIB. RT++ emerged as an attempt to provide a small and self-contained parallel programming package which seamlessly and efficiently integrates multi-threading and garbage collection within a type-safe C++ framework.

RT++ has been written and tested using the GNU g++ compiler (versions 2.6.3 and 2.7.2). While no GNU-specific extensions have been used, we have not verified that the system works with other C++ compilers. RT++ has been ported to and tested on the following architectures:

- PCs (Linux, Intel 386 family),
- Sequent Symmetry (Dynix/ptx, Intel 386 family),
- Silicon Graphics (Irix, MIPS family),
- Sun (SunOs and Solaris, SPARC family),
- Meiko CS-2 (Solaris, SPARC family),
- Hewlett Packard (HP-UX, PA-RISC family),
- Convex SPP (HP-UX, PA-RISC family).

Ports to other architectures using processors supported by the QuickThreads package (in particular the DEC AXP (Alpha) family) should be easy. However, in particular the IBM PowerRISC line is not yet supported.

The package is available under the GNU GPL on the World Wide Web at

`http://www.risc.uni-linz.ac.at/software/rt++`

Comments and bug reports can be sent to the the author under the email address `Wolfgang.Schreiner@risc.uni-linz.ac.at`.

```
// ------------------------------------------------------------------------
// Hello, World!
// ------------------------------------------------------------------------

#include <rt++.h>     // the RT++ header file
#include <iostream.h> // the C++ header files

// ------------------------------------------------------------------------
// the RT++ main program
// ------------------------------------------------------------------------
int hello(int argc, char *argv[])
{
  cout << "Hello, World!\n";
  return(0);
}

// ------------------------------------------------------------------------
// the C++ main program
// ------------------------------------------------------------------------
int main(int argc, char *argv[])
{
  Rt_Argument rtarg;                             // RT++ runtime parameters
  rtarg.proc = 1;                                // 1 process
  rtarg.memory = 1000000;                        // 1 MB shared memory
  rtarg.heap = 100000;                           // 100 KB initial heap
  rtarg.blocks = 4;                              // 4 heap blocks per process
  rtarg.stack = 15000;                           // 15 KB thread stack
  rtarg.verbose = 1;                             // print statistics
  exit(rt_main(hello, argc, argv, rtarg, 0)); // pass control to RT++ and exit
}
```

Figure 1: Hello, World!

2 Tutorial

This section demonstrates by a few simple examples the application of RT++ and the use of its main features. For comprehensive information on the individual constructs, please see the RT++ reference manual (Section 3).

2.1 Hello, World!

To start with, we present the simplest RT++ program that one can write. The program does not yet show how to utilize parallelism but only demonstrates the overall structure of an RT++ program, its compilation and execution. Figure 1 demonstrates the RT++ version of "Hello, World!".

Every RT++ program includes the header file *rt++.h* (among all other C++ header files that the program requires). The RT++ program is represented by a user-defined function with an arbitrary name *prog* and prototype

`int` *prog(int argc*`, char *`*argv*`[])`

The parameters *argc* and *argv* have the same interpretation as in the C++ *main* function (but it is up to the programmer to provide the argument values, see below). The return value of *prog* is an integer that denotes the success respectively failure of program execution.

In order to start up the RT++ runtime environment, the programmer has to explicitly pass control from the sequential C++ program to the RT++ main function. This is done by a call of `rt_main` (Section 3.1.8) which takes as arguments the function *prog* and the *argc* and *argv* passed to *prog*.

When the RT++ program terminates normally (e.g. by returning from *prog*) or abnormally (e.g. by a runtime error), control is returned to the caller of `rt_main`. The C++ program may then continue execution or start another RT++ program.

The argument *rtarg* is not passed to *prog* but used by RT++ itself to set up the runtime environment (Section 3.1). The programmer here specifies the number of processes, the maximum amount of shared memory, the initial size of the garbage collected heap, the number of heap blocks per process, the size of the stack of each concurrent thread and whether the output of statistical information is wished or not.

The last argument of `rt_main` can be used to retrieve the statistical information in a program variable for those cases where the programmer wants to process this information on her own. Otherwise, we just pass a 0 (null) pointer instead.

That's all there is! We can now compile and link the program like any other C++ program with a command line similar to the following:

```
g++ -O -Wall prog.cc -lrt++ -lqt -o prog
```

The program uses the RT++ library *librt++.a* which in turn is based (in its current implementation) on the QuickThreads library *libqt.a*. If these libraries and the associated header files are not installed in standard places, one has to provide additional `-I` and `-K` arguments that tell the compiler/linker where to search for the corresponding files.

The resulting program can be called as

```
./prog
```

and produces the output depicted in Figure 2.

The statistical information printed by RT++ includes the return code of the RT++ program (*not* that of the C++ program!), the real time spent in the RT++ program, the time spent in the RT++ program by the initial process respectively by all processes, the number of garbage collections, the number of times the garbage-collected heap was increased, the amount of real time spent for synchronization before garbage collection was performed, the amount of real time actually spent in garbage collection, the number of heap blocks and the number of bytes reclaimed.

If the `verbose` flag in *rtarg* is set to 0, all RT++ output is switched off and the program output simply is

```
Hello, World!
```

```
------------------------------------------------------------------------------
RT++ 1.0 (C) 1996 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
proc: 1, mem: 1000000, heap: 100032, blocks: 4, stack: 15000
------------------------------------------------------------------------------
Hello, World!
------------------------------------------------------------------------------
RT++ execution
        Return code:    0
        Real time(ms):  10
        Proc time(ms):  10
        All procs(ms):  10
RT++ garbage collections
        Number:         0
        Increments:     0
        Waiting(ms):    0
        Collecting(ms): 0
        Blocks:         0
        Bytes:          0
------------------------------------------------------------------------------
```

Figure 2: Program Output

2.2 Quicksort

Figure 3 illustrates the skeleton of a parallel version of the Quicksort algorithm for the in-place sorting of arrays. The full program can be found in Appendix B.1.

This program operates on arrays of type `Array<float>`. `Array` is an RT++ type constructor (Section 3.4) whose objects can be used similar to C++ arrays but are automatically reclaimed when not referenced any more.

The core of the program is the function `qsort` which takes an array a that is to be sorted within the index range $[l \dots r[$. If this index range is smaller than some threshold, the array is sorted sequentially. Otherwise, the array is partitioned according to some pivot element e into three ranges $[l, m_0[$ (all elements less than e) $[m_0, m_1[$ (all elements equal to e) and $[m_0, r[$ (all elements greater than e). What remains to be done is to sort the array in the first and in the third subrange. This is the task of two concurrent *threads* (units of control) t_0 and t_1. When both threads have terminated, the whole array is sorted.

Let's take a closer look at such a thread declaration

```
Thread3<int, Array<float>, int, int> t0(qsort, a, l, m0);
```

which creates a thread t_0 that executes the function application `qsort(a, l, m0)`. The type of t_0 is `Thread3<`*R*, *A1*, *A2*, *A3*`>` where the type fields *R*, *A1*, *A2*, *A3* correspond to the function prototype

R `qsort`(*A1*, *A2*, *A3*)

and the number 3 in `Thread3` corresponds to the number of function arguments. Threads may be declared in principle with an arbitrary number of argument types (but there is an implementation-dependent upper bound). The call

```
// --------------------------------------------------------------------------
// Quicksort
// --------------------------------------------------------------------------
#include <rt++.h>

...

// --------------------------------------------------------------------------
// sort <a/l/r> in parallel
// --------------------------------------------------------------------------
int qsort(Array<float> a, int l, int r)
{
  if (r-l < MINSIZE) return(sort(a, l, r));

  int m0, m1;
  partition(a, l, r, a[(l+r)/2], &m0, &m1);

  Thread3<int, Array<float>, int, int> t0(qsort, a, l, m0);
  Thread3<int, Array<float>, int, int> t1(qsort, a, m1, r);

  Wait(t0); Wait(t1);
  return(0);
}

// --------------------------------------------------------------------------
// RT++ main function
// --------------------------------------------------------------------------
int qmain(int argc, char *argv[])
{
  while(...)
  {
    Array<float> a(S);
    ...
    qsort(a, 0, S);
  }
  return(0);
}

// --------------------------------------------------------------------------
// C++ main function
// --------------------------------------------------------------------------
int main(int argc, char *argv[])
{
  RT_Argument rtarg;
  ...
  rt_arg.proc = 3;
  rt_main(qmain, argc, argv, rtarg, 0);
  ...
}
```

Figure 3: Parallel Quicksort

```
Wait(t0)
```

waits until t_0 has terminated and returns the result of the thread (i.e. the return value of `qsort`). In above case, this return value is always 0 and thus ignored.

Above skeleton omits a few ugly technical details that are only necessary because the GNU g++ compiler does not implement the full ANSI C++ standard yet (Section 3.1.3). The program namely has to contain also the following top-level declarations

```
Atom(int);
Atom(float);
Ref(Array<float>);
ThreadArg3(A, int, Array<float>, int, int);
```

The `Atom` declarations tell the RT++ garbage collector that `int` and `float` are atomic types that are allocated on the stack and do not contain pointers to other objects. The `Ref` declaration says that `Array<float>` is a heap-allocated type. Declarations of this kind have to be provided for any type used by an RT++ program (actually only for those which are used in the context of an RT++ type constructor).

`ThreadArg3(A,...)` declares that threads of the corresponding prototype are used (where `A` is an arbitrary unused identifier that is needed internally). Declarations of this kind have to be given for any thread type used by an RT++ program.

The good message is that these declarations are only needed in *one* object file; if your program is composed of multiple source files that are individually compiled and finally linked together, all necessary declarations can be collected in a single source file of the application.

In case none of the object files contains the required declarations, the linker complains with error messages as the following:

```
qsort.o(.text+0xef8): undefined reference to `Type<int>::init'
qsort.o(.text+0xf0b): undefined reference to `Type<int>::scan'
qsort.o(.text+0x987): undefined reference to `Type<float>::init'
qsort.o(.text+0x9fe): undefined reference to `Type<float>::scan'
qsort.o(.text+0xee5): undefined reference to `Type<Array<float> >::scan'
qsort.o(.text+0xf39): undefined reference to `Type<Array<float> >::init'
qsort.o(.text+0xb6d): undefined reference to
  `Type<ArgumentThread3<int, Array<float>, int, int> >::scan'
```

When the program is compiled and executed, we get statistics output of the kind depicted in Figure 4 (for demonstration purposes, this is the output of a different RT++ program).

The `RT++ GC` messages are generated by the system whenever a garbage collection is performed (unless the verbose flag is not set). Such a message displays the garbage collection counter, the current time (from the start of the RT++ program), the number and total size of the reclaimed free heap blocks, the duration of garbage collection and the synchronization time needed before garbage

```
--------------------------------------------------------------------------------
RT++ 1.0 (C) 1996 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
proc: 3, mem: 1000000, heap: 100032, blocks: 12, stack: 5000
--------------------------------------------------------------------------------
RT++ GC 1 (1360 ms): 8 blocks with 42560 bytes in 30 (+ 0) ms.
RT++ GC 2 (1820 ms): 11 blocks with 64112 bytes in 20 (+ 0) ms.
RT++ GC 3 (2640 ms): 9 blocks with 53456 bytes in 20 (+ 0) ms.
RT++ GC 4 (3200 ms): 3 blocks with 15984 bytes in 30 (+ 0) ms.
RT++ GC: less than 25008 bytes heap (1 block per process) free.
RT++ GC: increase heap by 100032 bytes.
RT++ GC 5 (4670 ms): 7 blocks with 33520 bytes in 50 (+ 0) ms.
RT++ GC 6 (5120 ms): 16 blocks with 89952 bytes in 40 (+ 10) ms.
RT++ GC 7 (6350 ms): 14 blocks with 75008 bytes in 40 (+ 0) ms.
RT++ GC 8 (7140 ms): 14 blocks with 67344 bytes in 50 (+ 0) ms.
RT++ GC 9 (7890 ms): 18 blocks with 94864 bytes in 40 (+ 0) ms.
RT++ GC 10 (9170 ms): 4 blocks with 7664 bytes in 50 (+ 10) ms.
RT++ GC: less than 25008 bytes heap (1 block per process) free.
RT++ GC: increase heap by 100032 bytes.
RT++ GC 11 (10650 ms): 16 blocks with 74144 bytes in 60 (+ 10) ms.
RT++ GC 12 (11520 ms): 6 blocks with 17872 bytes in 70 (+ 0) ms.
RT++ GC: less than 25008 bytes heap (1 block per process) free.
RT++ GC: increase heap by 100032 bytes.
RT++ GC 13 (13170 ms): 41 blocks with 243984 bytes in 70 (+ 0) ms.
--------------------------------------------------------------------------------
RT++ execution
        Return code:    0
        Real time(ms):  16650
        Proc time(ms):  16620
        All procs(ms):  46900
RT++ garbage collections
        Number:         13
        Increments:     3
        Waiting(ms):    30
        Collecting(ms): 570
        Blocks:         167
        Bytes:          880464
--------------------------------------------------------------------------------
```

Figure 4: Parallel Program Output

collection could start (in parentheses). If the heap is increased due to insufficient free heap space after garbage collection, this is also shown.

A hint concerning the number of processes used for the execution of an RT++ program: if the total process time of all processes is drastically smaller than the real time of program execution multiplied with the number of processes, the machine is probably overloaded i.e. not every process ready for execution finds a free processor. Consequently, only as many processes should be used for program execution as processors are idle (keeping in mind that in a multi-user system also other activities may be going on!). Likewise, if synchronization times for garbage collection phases get large, this is a sign that some process is not executing while other processes are waiting for garbage collection to begin.

2.3 Long Integer Multiplication

Figure 5 depicts the skeleton of an RT++ program for the multiplication of two arbitrary precision integers (represented as arrays of digits). The full program can be found in Appendix B.2.

The core of this program is the function mult which implements a parallel version of the traditional school algorithm where a number is an object of type Array<Digit>:

1. In the first phase, the algorithm creates a thread for each multiplication of number a with some digit b_i. All threads are collected in a list l.
2. In the second phase, the algorithm creates a thread for the addition of each pair of thread results by removing the corresponding threads from l and inserting the new thread into l.

Eventually there is only thread left in l whose result denotes the overall computation result.

There are several issues remarkable about above implementation:

1. The program applies the RT++ type constructor List to construct element sequences of arbitrary length. New elements can be inserted at the head of a list and a list can be decomposed into its head element and the tail list (Section 3.3).
2. The thread list l has type List< Thread<Number> > i.e. each element of a list holds a thread returning a number. The type Thread<...> does (in contrast to the types Thread*n*<...>) *not* specify which argument types these threads have; correspondingly it can hold *any* kind of thread with the denoted result type (indeed both program phases insert threads of different argument types into l).
3. The program does *not* wait for the results of the first phase threads before creating the second phase threads. It simply extracts two appropriate threads from the list and creates a new thread that take the other threads

```
#include <rt++.h>

typedef char Digit;
typedef Array<Digit> Number;
...

// ---------------------------------------------------------------------------
// sum of two thread results
// ---------------------------------------------------------------------------
Number sum(Thread<Number> ta, Thread<Number> tb)
{
  Number a = Wait(ta); Number b = Wait(tb);
  ...
  return(s);
}

// ---------------------------------------------------------------------------
// product of number and digit
// ---------------------------------------------------------------------------
Number prod(Number a, Digit b)
{
  ...
  return(p);
}

// ---------------------------------------------------------------------------
// multiplication of two numbers
// ---------------------------------------------------------------------------
Number mult(Number a, Number b)
{
  int lb = Length(b);
  List< Thread<Number> > l;                 // l = Thread<Number>::Nil();
  for (int i = 0; i < lb; i++)
    {
      Thread2<Number, Number, Digit> t(prod, a, b[i]);
      l.cons(t, l);                         // l = Cons(t, l);
    }
  int n = 2*lb-1;
  for (int j = 0; j < n-2 ; j += 2)
    {
      Thread<Number> ta(Head(l)); l.tail(); // l = Tail(l);
      Thread<Number> tb(Head(l)); l.tail(); // l = Tail(l);
      Thread2<Number, Thread<Number>, Thread<Number> > t(sum, ta, tb);
      l.cons(t, l);                        // l = Cons(t, l);
    }
  return(Wait(Head(l)));
}

int multmain(int argc, char *argv[])
{
  ...
  Number a(la); Number b(lb);
  ...
  Number c = mult(a, b);
  ...
}
```

Figure 5: Long Integer Multiplication

as arguments. Threads are thus objects like any other kind of runtime data, in particular they can be stored in data structures (as shown above), passed as thread arguments and also returned as thread results.

4. The program has a design flaw by inserting the second phase threads at the *begin* of the thread list l (i.e. using l as a *stack*) and thus generating an unbalanced "thread tree" of depth $n-1$ (where n is the number of threads). It would be better to insert the second phase threads at the *end* of l (i.e. to use l as a *queue*) and thus generate a fully balanced tree of depth $\log n$. This can be efficiently done by making use of RT++ destructive list operations (Section 3.3.5).

The program makes heavy use of the RT++ type constructors and must provide the following top-level declarations for the garbage collector:

```
Atom(Digit);
Ref(Number);
Ref(Thread<Number>);

ThreadRes(Number);
ThreadArg2(A, Number, Number, Digit);
ThreadArg2(B, Number, Thread<Number>, Thread<Number>);
```

The first three declarations are necessary because digits, numbers, and threads are used as thread arguments and results and stored in lists and arrays, respectively. The remaining three declarations refer to the kinds of threads used in the program. The `ThreadRes<...>` declaration is required for every kind of `Thread<...>` type. If forgotten, the linker complains with error messages as the following

```
mult.o(.text+0xef8): undefined reference to
  `Type<ResultThread<Number> >::init'
mult.o(.text+0xf0b): undefined reference to
  `Type<ResultThread<Number> >::scan'
```

2.4 Long Integer Multiplication (Bag Version)

The previous program for long integer multiplication was a bit over-specified because it explicitly determined the order in which pairs of integers were added in the second phase of the algorithm; however *any* ordering would have been legal. In Figure 6 we give a variant of the program that copes without this over-specification. The full program can be found in Appendix B.3.

The program is based on the concept of *thread bags* (Section 3.6) that collect thread results and provide them in the order in which they have been computed (i.e. in the order in which the threads have terminated):

1. The declaration

```
ThreadBag<Number> bag(n);
```

```
// ----------------------------------------------------------------------------
// Long Integer Multiplications (Bag Version)
// ----------------------------------------------------------------------------
#include <rt++.h>

typedef char Digit;
typedef Array<Digit> Number;
...

// ----------------------------------------------------------------------------
// sum of two numbers
// ----------------------------------------------------------------------------
Number sum(Number a, Number b)
{
  ...
  return(s);
}

// ----------------------------------------------------------------------------
// product of number and digit
// ----------------------------------------------------------------------------
Number prod(Number a, Digit b)
{
  ...
  return(p);
}

// ----------------------------------------------------------------------------
// multiplication of two numbers
// ----------------------------------------------------------------------------
Number mult(Number a, Number b)
{
  int lb = Length(b);
  int n = 2*lb-1;
  ThreadBag<Number> bag(n);
  for (int i = 0; i < lb; i++)
    {
      ThreadBag2<Number, Number, Digit>(bag, prod, a, b[i]);
    }
  for (int j = 0; j < n-2 ; j += 2)
    {
      Number a = Wait(bag, j);
      Number b = Wait(bag, j+1);
      ThreadBag2<Number, Number, Number>(bag, sum, a, b);
    }
  return(Wait(bag, n-1));
}

...
```

Figure 6: Long Integer Multiplication (Bag Version)

declares a *bag* that can hold up to *n* threads of result type `Number`. It is also possible to declare bags without an explicit size specification but this is less efficient.

2. A statement

   ```
   ThreadBag2<Number, Number, Digit>(bag, prod, a, b[i]);
   ```

   creates a new thread computing `prod(a, b[i])`. The thread itself is *anonymous* but its result can be retrieved via *bag*.

3. The expression

   ```
   Wait(bag, j)
   ```

   returns the *j*-th thread result in *bag* where it is *undefined* by which thread this result was computed (but two calls of `Wait` with the same index in the same program run always return the same result).

Furthermore the program must provide the following top-level declarations for the garbage collector:

```
Atom(Digit);
Ref(Number);

ThreadBagRes(Number);
```

The first two declarations are required because digits and numbers are used as thread arguments and results, respectively. The third sort of declaration is required for any type of thread bags used by the program (the individual kinds of anonymous threads need not be declared).

Above solution has one disadvantage compared to the approach presented in the previous section. The main program waits for the first phase thread results in the bag before it starts the second phase threads. We can overcome this *over-synchronization* by passing the thread bag directly to the second phase threads and let these threads themselves synchronize with their required arguments. This idea is sketched in Figure 7 (the full program can be found in Appendix B.4).

In order to get an executable program, the following top-level declarations have to be provided to the garbage collector

```
Atom(Digit);
Ref(Number);
Atom(int);
Ref(ThreadBag<Number>);

ThreadBagRes(Number);
```

```
// ---------------------------------------------------------------------------
// Long Integer Multiplications (Bag Version)
// ---------------------------------------------------------------------------
#include <rt++.h>

typedef char Digit;
typedef Array<Digit> Number;
...

// ---------------------------------------------------------------------------
// sum of two thread bag results
// ---------------------------------------------------------------------------
Number sum(ThreadBag<Number> a, int j)
{
  Number a = Wait(bag, j);
  Number b = Wait(bag, j+1);
  ...
  return(s);
}

// ---------------------------------------------------------------------------
// product of number and digit
// ---------------------------------------------------------------------------
Number prod(Number a, Digit b)
{
  ...
  return(p);
}

// ---------------------------------------------------------------------------
// multiplication of two numbers
// ---------------------------------------------------------------------------
Number mult(Number a, Number b)
{
  int lb = Length(b);
  int n = 2*lb-1;
  ThreadBag<Number> bag(n);
  for (int i = 0; i < lb; i++)
    {
      ThreadBag2<Number, Number, Digit>(bag, prod, a, b[i]);
    }
  for (int j = 0; j < n-2 ; j += 2)
    {
      ThreadBag2<Number, ThreadBag<Number>, int>(bag, sum, bag, j);
    }
  return(Wait(bag, n-1));
}

...
```

Figure 7: Long Integer Multiplication (Second Bag Version)

The examples presented in this and the previous section outline the sort of programming style suggested by RT++: threads are entities that can be used and passed around as any other sort of runtime values. Consequently it is not necessary for the creator of a thread to resolve synchronization dependencies; this task can be as well performed by the threads themselves (self-synchronization). Furthermore, the program need not explicitly allocate or free resources; this task is performed by the underlying memory management system (garbage collection). The resulting programming style closely resembles the *dataflow* style of parallel computation.

3 Reference Manual

RT++ is a system which allows to write parallel programs in C++ that can be executed on mono-processor and shared memory multi-processor machines. It consists of a runtime system that manages concurrent threads of control and garbage-collected heap objects and of a set of abstract data types for parallel programming.

RT++ schedules threads among processes much like the underlying operating systems schedules these processes on hardware processors. If the underlying machine provides more than one processor, an RT++ program thus executes truly in parallel.

This section is a reference manual that describes the user-level view of RT++; it does not give more information on the implementation of the system as necessary for understanding (for a few implementation issues see Section A). On the other hand, it is also *not* a user manual that teaches the basics of RT++ programming (for a short tutorial see Section 2).

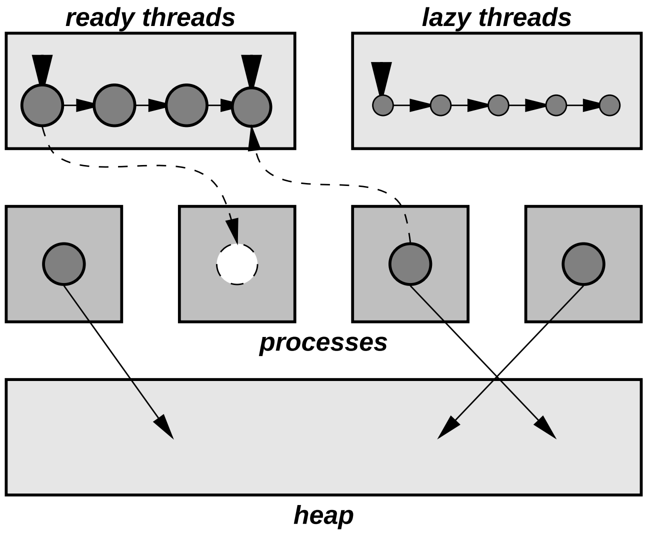

Figure 8: Execution Model

3.1 Program Control

An RT++ program is embedded into a conventional C++ program by the function `rt_main` (Section 3.1.8) which passes control to the main function of the RT++ program. After termination of the RT++ program, control is returned to the caller of `rt_main`.

3.1.1 Execution Model

The RT++ model of program execution can be sketched as follows (see Figure 8):

- A number of *processes* executes threads from the ready queue. Thread execution is preemptive i.e. after a certain time slice has elapsed, each process puts the currently executed thread back into the ready queue and selects a new thread for execution.
- A *ready queue* contains all active threads i.e. all threads that are immediately ready for execution (and that may have been previously under execution). Active threads are fairly scheduled for execution on a first in/first out basis.
- A *lazy stack* contains all threads that are in principle ready for execution but that still need to be activated before they can be actually executed. Lazy threads are activated on a last in/first out basis only if an idle process finds the ready queue empty.
- A *heap* contains all data structures on which the threads operate. Each thread can access each entity in the heap regardless by which process it is currently executed.

The representation of threads (which are heap objects themselves) is described in Section 3.2.1. An *automatic garbage collector* (Section A.2) reclaims heap objects (and threads) when they are not (directly or indirectly) referenced by the main thread (Section 3.1.8).

3.1.2 Program Structure

The basic structure of an RT++ program is as follows:

```
#include <rt++.h>

...

int my_main(int argc, char *argv[])
{
  ...
}

int main(int argc, char *argv[])
{
  ...
  RT_Argument rtarg;
  RT_Result rtres;
  ...
  int code = rt_main(my_main, argc, argv, rtarg, &rtres);
  ...
}
```

The call of `rt_main` (Section 3.1.8) transfers control from the C++ program to a thread that executes the function denoted by *my_main* (the *main thread*). The argument *rtarg* is used to set up the RT++ runtime environment while *rtres* delivers statistical information after termination of the RT++ program. *code* contains the return code of the RT++ program.

An RT++ program is basically an extension of a C++ program. However, there are several restrictions that deal with the fact that the RT++ program threads are generally executed by different processes (Section 3.1.6).

3.1.3 RT++ Type Declarations

All RT++ objects (threads (Section 3.2), lists (Section 3.3), arrays (Section 3.4), pointers (Section 3.5), thread bags (Section 3.6)) are represented as data structures in the heap. For the purpose of automatic garbage collection, the programmer has to provide some extra information about those types that are used as base types of RT++ type constructors or as argument types or result types of RT++ threads.

Type Constructors We distinguish two classes of types that may be used as base types of RT++ type constructors:

- **Atomic Types.** These are types that do not make use of any RT++ types. Examples of such types are C++ integers, floating points, and also classes provided that they do *not* contain RT++ types as components. For each such type T, the following global declaration must be provided:

  ```
  Atom(T);
  ```

  `Atom` is a macro whose application declares some externally visible symbols in the corresponding object file. For each type T, `Atom(`T`)` must therefore appear in *one* C++ source file only.

- **Reference Types.** These are C++ classes that contain at least one RT++ type as component. For each such type T, two public member functions have to be defined:

  - `init()`. This function must call the `init()` function of each RT++ type component.
  - `scan()`. This function must call the `scan()` function of each RT++ type component.

  Furthermore, in *one* C++ source file the following global declaration must be provided:

  ```
  Ref(T);
  ```

Implementation Note: On compilers that support static member initializers for template classes (which is required by the ANSI C++ standard but not yet supported by the GNU g++ compiler), `Ref` becomes superfluous by uncommenting a few lines in the RT++ header file *rt++/rt_type.h*. This makes virtually all of the following declarations redundant.

Example As an example, take a program that uses lists of arbitrary precision integers represented as follows:

```
class Int
{
  int sign;
  Array<unsigned long> digits;
  public:
  ...
  void init(void)
  {
    digits.init();
  }
  void scan(void)
```

```
  {
    digits.scan();
  }
}
```

Since the `digits` component uses the RT++ type `Array`, the member functions `init` and `scan` must call the corresponding counterparts of `digits`.

In order to use a variable

```
List<Int> l;
```

we have to provide in one C++ source code file the global declarations

```
Atom(unsigned long);
Ref(Int);
```

(since `unsigned long` is used in `Int` as the base type of `Array` and `Int` is used as the base type of `List`).

If none of the C++ object files contains an `Atom` or `Int` declaration for a type T which is used as the base type of an RT++ type constructor, the linker complains with an error message similar to the following:

```
prog.o(.text+0x7b8): undefined reference to 'Type<T>::init'
prog.o(.text+0x834): undefined reference to 'Type<T>::scan'
```

Template Types The fact that `Atom` and `Ref` are macros creates a syntactical problem with template types. It is e.g. not possible to declare

```
Atom(TemplateType<ArgType1,ArgType2>)
```

since the comma is interpreted by the preprocessor as the separator for arguments of `Atom`, *not* of `TemplateType`. However, this problem may be overcome by declaring

```
typedef TemplateType<ArgType1,ArgType2> TA;
Atom(TA);
```

(where `TA` is an arbitrary identifier not used elsewhere) or by manually inserting the macro definitions provided in the RT++ header file *rt++/rt_type.h*.

Threads In a similar fashion, also those types have to be declared on which threads and thread bags are based. If we want to create a thread (Section 3.2.2)

$\boldsymbol{Thread}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle\ t(f,\ a_1,\ \ldots,\ a_n);$

we have to provide the corresponding global declaration

`ThreadArg`$n(Id,\ \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n);$

in one C++ source file. *Id* is an arbitrary identifier not in use anywhere else (it is internally used by the macro because of the previously explained problem with template types).

Likewise, if we want to define a thread handle (Section 3.2.2)

$\boldsymbol{Thread}\langle \boldsymbol{R}\rangle\ t;$

we have to provide the global declaration

`ThreadRes`$(\boldsymbol{R});$

If one forgets some of the required declarations, the linker complains with error messages as that shown above (mentioning the internal types `ArgumentThread` and `ResultThread`, respectively).

Thread Bags If we want to use a thread bag

$\boldsymbol{ThreadBag}\langle \boldsymbol{T}\rangle\ bag;$

we have to provide a global declaration

`ThreadBagRes`$(\boldsymbol{T});$

If one forgets the required declaration, the linker will complain with error messages as that shown above (mentioning the internal RT++ type `ThreadBagBase`).

3.1.4 Compilation and Execution

An RT++ program may be compiled with the GNU C++ compiler[3] like any other C++ program with the single extension that the RT++ library *librt++.a* and the QuickThreads library *libqt.a* have to be linked to the program.

A typical compilation and linking sequence therefore is

```
g++ -Wall -O prog.c -o prog.o
g++ prog.o -lrt++ -lqt -o prog
```

(provided that all libraries are installed in standard paths). On some systems, you might have to link additional libraries (see the RT++ installation description).

[3] Actually it should also compile with other C++ compilers, but this has not been verified yet.

3.1.5 Preprocessor Constants

Various RT++ mechanisms can be modified by defining one or more of the following preprocessor constants before the RT++ header file *rt++.h* is included into a source file:

```
#define RT_PARAMETER
#include <rt++.h>
```

- `RT_STACK_PUSH`. If this preprocessor constant is defined, each declaration of an RT++ object checks whether the stack of the current thread has overflown and aborts the program if this is the case. This may be useful for debugging RT++ programs but considerably slows down execution.

- `RT_THREAD_NOTPAR`. If this preprocessor constant is defined, the code of each thread is executed when the thread is defined (Section 3.2.2) i.e. the program is sequentially executed by a single thread. This may be useful for debugging an RT++ program. *Defining this constant may change the termination semantics of a program (if a thread does not terminate).*

- `RT_THREAD_NOTLAZY`. If this preprocessor constant is defined, the thread definition (Section 3.2.2)

  $\boldsymbol{Threadn}\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle\ t(f,\ a_1,\ \ldots,\ a_n)$

  is equivalent to

  $\boldsymbol{Threadn}\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle\ t$
  $t.\texttt{start}(f_1,\ a_1,\ \ldots,\ a_n)$

- `RT_TBAG_NOTPAR`. If this preprocessor constant is defined, the code of each *bag thread* is executed when the bag thread is created (Section 3.6.3) i.e. the program is sequentially executed by a single thread. This may be useful for debugging an RT++ program. *Defining this constant may change the termination semantics of a program (if a thread does not terminate).*

- `RT_TBAG_NOTLAZY`. If this preprocessor constant is defined, the following operation for adding a thread to a thread bag (Section 3.6.3)

  $\boldsymbol{ThreadBagn}\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle(b,\ f,\ a_1,\ \ldots,\ a_n)$

  is equivalent to

  $\boldsymbol{ThreadBagn}\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle::\texttt{start}(b,\ f,\ a_1,\ \ldots,\ a_n)$

3.1.6 Restrictions

An RT++ program generates a set of threads that may be executed by any process of the RT++ runtime environment (Section 3.1.1). Normally the location of a thread is transparent i.e. it does not matter on *which* process a thread is executed.

However, there are a few exceptions all of which are related to the fact that the *data segment of a program is local to each process*. As a consequence, if data in this space are accessed, the location of a thread *does* matter.

The function `rt_bind` (Section 3.1.9) may be used to bind threads to specific processes in those cases, where the location of a thread is of concern.

As a simplification, the RT++ main thread (Section 3.1.8) *is bound to the initial process* such that it can do anything that can be done in a C++ program.

Standard Input The memory buffers associated to the standard input stream are allocated in the data segment of a process. When an RT++ program is started, the standard input stream is bound to the initial process (process 0). Therefore a thread may read from the standard input stream only if it is executed on that process (but it is illegal if multiple threads simultaneously read from standard input).

Only the main thread (which is bound to the initial process) may read from standard input without special provision. All other threads must protect reading sequences as follows:

```
rt_bind(0);
cin >> data;
rt_unbind();
```

Standard Output The memory buffers associated to the standard output stream are allocated in the data segment of a process. Unlike reading from the standard input stream (Section 3.1.6), the standard output stream may be written by any process, since the buffers of all processes are flushed to the same output device.

If only one thread writes to standard output, it suffices to bind the thread to any process (Section 3.1.9)

```
rt_bind();
cout << "data " << data << "\n";
rt_unbind();
```

The main thread (which is bound to the initial process) need not take any provisions at all.

However, if multiple threads write to standard output, it does not suffice to bind a thread to a specific process. We also have to guarantee that the process is not *interrupted* while performing the output operation, because otherwise another thread might interfere with its own output. Thus every thread (*also the main thread*) has to protect its output operations as follows (Section 3.1.9)

```
rt_lock();
cout << "data " << data;
rt_unlock();
```

The output sequences generated by threads on different processors are non-deterministically merged on the output device.

C++ Memory Allocation The C++ `new` operation (and likewise the C `malloc`) return memory that was allocated in the data segment of the current process. In case you would like to use these operations for the temporal allocation of dynamic memory, you have to bind the thread to a specific process during the whole use of this memory e.g. (Section 3.1.9)

```
rt_bind();
T *t = new T;
use(t);
rt_unbind();
```

Global Variables Globally declared variables are allocated in the data segment of a process; updates performed by a thread on one process are therefore not visible on other processes. However, any global variables declared and initialized by the C++ program before the start of the RT++ program are visible in the same way to all RT++ threads provided that these variables are not modified during the execution of the RT++ program.

3.1.7 Types

`RT_Argument`: **Runtime Environment**

`class` ***RT_Argument***
{
  `public:`
  int *proc*
  int *memory*
  int *heap*
  int *blocks*
  int *stack*
  int *verbose*
}
arg.`init`()
arg.`print`()
  RT_Argument *arg*

Specification: `init` fills *arg* with information about the runtime environment of the currently executing RT++ program. `print` writes the contents of *arg* to the standard output stream.

The runtime environment is defined by a structure of the same type passed to `rt_main` (Section 3.1.8).

The individual components have the following interpretation:

- *proc* is the number of processes used by the program.
- *memory* is the maximum amount of shared memory used by the program (in bytes).

- *heap* is (a lower bound for) the initial size of the garbage-collected heap (in bytes).
- *blocks* is the number of blocks into which the heap is partitioned *per processor* (i.e. the total block number is *blocks*∗*proc*).
- *stack* is the size of the stack used for the execution of each thread (in bytes).
- *verbose* is 1 if statistics output is generated during program execution and 0 else.

Note: The initial size of the heap is actually the smallest n with $n = m * proc * blocks \geq heap$ for some block size m. This block size m is an upper bound for the size of contiguous blocks that can be allocated in shared memory.

Implementation: During program execution, the heap may be gradually extended in multiples of the initial heap size. This takes place, if after garbage collection

1. less than $m * proc$ bytes heap memory are free, or
2. the largest contiguous memory block is not able to satisfy the allocation request that triggered the garbage collection.

`RT_Result`: Execution Statistics

`class` ***RT_Result***
{
  `public:`
  int *ret_code*
  int *time_real*
  int *time_proc*
  int *time_all*
  int *gc_number*
  int *gc_inc*
  int *gc_wait*
  int *gc_time*
  int *gc_blocks*
  int *gc_bytes*
}
res.`init`()
res.`print`()
  RT_Result *res*

Specification: `init` fills *res* with statistical information about the currently executing RT++ program. `print` writes the contents of *res* to the standard output stream.

A pointer to such a structure is used in `rt_main` (Section 3.1.8) to return statistical information about the execution of an RT++ program after the program has terminated.

The components of the structure have the following interpretation:

- If the structure is defined by `rt_main`, *ret_code* contains the return code r of the program as returned by the main function, respectively by a call of `rt_exit`(r) (Section 3.1.8). The return code is set to -1, if `rt_abort` (Section 3.1.8) was called during program execution.

  If the structure is defined by a call of `init`, *ret_code* is set to -1.

- *time_real* contains the real time spent in the RT++ program (in ms).

- *time_proc* contains the time that the current process spent in the RT++ program (the sum of user and system time in ms).

- *time_all* contains the time that the current process and its children processes spent in the RT++ program (the sum of all user and system times in ms).

- *gc_number* contains the number of garbage collections performed during program execution.

- *gc_inc* contains the number of increments of the heap.

- *gc_wait* contains the total amount of (real) time that passed from triggering a garbage collection to actually starting the garbage collection (in ms).

- *gc_time* contains the total amount of (real) time that passed from starting a garbage collection to completing the garbage collection (in ms).

- *gc_blocks* contains the total amount of contiguous memory blocks reclaimed during garbage collections.

- *gc_bytes* contains the total amount of bytes reclaimed during garbage collections.

3.1.8 Execution

`rt_main()`: Program Start

code = `rt_main`(*f*, *argc*, *argv*, *arg*, *res*)

    int *code*
    int (**f*)(***int***, ***char****[])
    int *argc*
    char **argv*[]
    RT_Argument *arg*
    RT_Result **res*

Specification: Executes the RT++ program $f(argc, argv)$ in the environment set up by the argument object *arg* (Section 3.1.7). The return value of f is returned as *code*. If *res* is not 0, some statistics information is returned in the result object referenced by *res* (Section 3.1.7).

If the RT++ program is terminated by `rt_exit`(r) (Section 3.1.8), *code* is r. If the RT++ program is aborted by `rt_abort`(s) (Section 3.1.8), *code* is -1. If during program execution the program anything goes wrong (i.e. the program crashes or runs out of memory), *code* is -1.

In any case, *code* is also stored in the *res* object.

Constraint: All components of the argument structure *arg* (Section 3.1.7) must be positive, except for the *verbose* flag which must be 0 or 1.

`rt_exit()`: Program Termination

`rt_exit`(n)
    int n

Specification: Terminates the RT++ program by passing control to the caller of `rt_main` (Section 3.1.8) with return code n.

Note: A return code of -1 is used to denote abnormal program abortion by `rt_abort` (Section 3.1.8).

`rt_abort()`: Program Abortion

`rt_abort`(s)
    char *s

Specification: Aborts the RT++ program by printing message s and returning control to the caller of `rt_main` (Section 3.1.8) with return code -1.

3.1.9 Miscellaneous

`rt_bind()`: Binding a Thread to a Processor

`rt_bind`()
`rt_bind`(n)
`rt_unbind`()
    int n

Specification: Binds the currently executing thread to the current process (first version) or to process n (second version). The third version resolves any previous binding of the currently executing thread.

A thread *bound* to a process is executed by that process only. Process 0 is the initial process on which execution of the RT++ program was triggered.

The main thread (Section 3.1.8)) is at program start bound to process 0.

Constraint: The process number n must range from 0 to $n-1$ where n is the number of processes (Section 3.1.9) in the runtime environment (Section 3.1.7).

Note: Applications of these operations are the protection of input operations, of output operations, and of C++ memory allocation operations (Section 3.1.6).

Implementation: A call of `rt_bind` in general involves a context switch such that the current thread can be scheduled for execution by another process.

`rt_yield()`: Descheduling a Thread

`rt_yield`()

Specification: Deschedules the current thread from execution by putting the thread back into the ready queue.

`rt_lock()`: Locking a Process

`rt_lock()`
`rt_unlock()`

Specification: The first form switches off preemption of the current process i.e. the process is going to execute no other thread than the current one. The second form switches on preemption again.

Constraint: No RT++ object may be constructed while preemption is switched off and no thread may be created during that time. Furthermore, the thread must neither terminate nor wait for the result of another thread while preemption is switched off.

Note: This operation may be used to switch off preemption during output operations (Section 3.1.6).

Implementation: Garbage collection cannot be performed while preemption is switched off on *any* process.

`rt_gc()`: Triggering Garbage Collection

`rt_gc()`

Specification: Triggers garbage collection.

Note: Garbage collection reclaims all threads which are not (directly or indirectly) referenced by the main thread. It is possible to prevent an unreferenced thread from being reclaimed by placing it into the root set of the garbage collector (Section 3.2.4).

Implementation: The current thread is descheduled and placed at the end of the ready queue. Therefore another thread may be scheduled for execution after garbage collection has been completed.

`rt_clock()`: Timing

time = `rt_clock()`
    int *time*

Specification: Returns the current (real) *time* in ms counted from an arbitrary (but fixed) point in the past.

Note: Clock resolution may be less than 1 ms, e.g. the clock value may be incremented by multiples of 10 ms only.

`rt_proc()`: Process Number

proc = `rt_proc()`
    int *proc*

Specification: Returns the number of processes *proc* used for executing the current RT++ program.

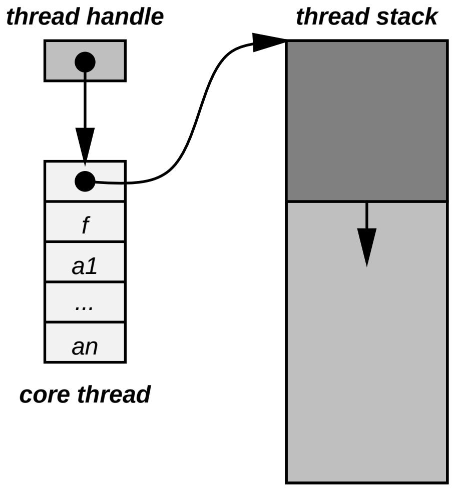

Figure 9: Thread Representation

3.2 Threads

A *thread* is the unit of concurrency. It executes some user-defined function independently of and (at least conceptually) simultaneously with all other threads.

3.2.1 Representation

A thread is represented by the following components (see Figure 9):

1. A *handle* by which the thread can be referenced. Multiple handles may reference the same thread.
2. The *core thread* which contains the information necessary for execution, in particular the executed function f and the function arguments a_i.
3. An optional *thread stack* which represents the space on which local variables are allocated during execution of the thread.

A core thread without stack is called *lazy* (or *virtual*). Before a lazy thread can be executed it has to be *activated* i.e. a stack has to be allocated on which the thread can operate. Since the thread stack is an order of magnitude larger than the core thread, threads are typically kept in a lazy state as long as possible in order to reduce the memory consumption of a program.

Note: The thread representation is a special case of the representation of pointer objects (Section 3.5).

3.2.2 Definition

$\textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ t$
$\textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ t$

Threadn$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ $t(f, a_1, \ldots, a_n)$
  R $(*f)(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n)$
  $\boldsymbol{A}_i$ a_i

Specification: Defines a thread handle t.

1. In the first form, t may be bound to any thread that returns a result of type ***R***. *t is not yet bound to any thread.*
2. In the second form, t may be bound to any thread that takes n arguments of types $\boldsymbol{A}_i$ and returns a result of type ***R***. *t is not yet bound to any thread.*
3. In the third form, t is bound to a new thread that returns $f(a_1, \ldots, a_n)$.

Note: A thread handle of type ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ is implicitly converted to a thread handle of type ***Thread***$\langle$***R***$\rangle$ whenever such a type is expected. Thus e.g. the following is legal:

> ***Thread***$\langle$***R***$\rangle$ s
> ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ $t(f, a_1, \ldots, a_n)$
> s = t

Restrictions: The number of arguments n ranges from 0 to an implementation-dependent constant (currently 4). This range may be extended by defining the corresponding template class.

Implementation: The third form is more efficient than the otherwise equivalent form

> ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ t
> t = ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$$(f_1, a_1, \ldots, a_n)$

Normally this form is also equivalent to to (Section 3.2.3)

> ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ t
> t.`create`$(f_1, a_1, \ldots, a_n)$

However, if the preprocessor constant `RT_THREAD_NOTLAZY` is defined before including the header file *rt++.h* (Section 3.1.5), the declaration is equivalent to (Section 3.2.3)

> ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ t
> t.`start`$(f_1, a_1, \ldots, a_n)$

3.2.3 Basics

`Create()`: Lazy Thread Creation

t = ***Threadn***$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$::`Create`$(f_1, a_1, \ldots, a_n)$
  Threadn$\langle$***R*, *A***$_1$, ..., ***A***$_n\rangle$ t

$\boldsymbol{R}$ $(*f)(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n)$
$\boldsymbol{A}_i$ a_i

Specification: Returns a thread computing $f(a_1, \ldots, a_n)$.

Implementation: The thread is not yet activated but is put into the lazy stack (Section 3.1.1).

Note: If the preprocessor constant `RT_THREAD_NOTLAZY` is *not* set, above form is equivalent to

$$t = \boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle(f_1, a_1, \ldots, a_n)$$

`Start()`: Eager Thread Creation

$t = \boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle$::`Start`$(f_1, a_1, \ldots, a_n)$
$\boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle$ t
$\boldsymbol{R}$ $(*f)(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n)$
$\boldsymbol{A}_i$ a_i

Specification: Returns a thread computing $f(a_1, \ldots, a_n)$.

Implementation: The thread is not yet activated but is put into the lazy stack (Section 3.1.1).

Note: If the preprocessor constant `RT_THREAD_NOTLAZY` *is* set, above form is equivalent to

$$t = \boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle(f_1, a_1, \ldots, a_n)$$

`create()`: Lazy Thread Creation (Binding Form)

t.`create`$(f_1, a_1, \ldots, a_n)$
$\boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle$ t
$\boldsymbol{R}$ $(*f)(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n)$
$\boldsymbol{A}_i$ a_i

Specification: Binds t to a thread returning $f(a_1, \ldots, a_n)$. This binding overrides any previous binding of t.

Implementation: The thread is not yet activated but is put into the lazy stack (Section 3.1.1).

Note: This operation is more efficient than the otherwise equivalent form (Section 3.2.3)

$$t = \boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle::\texttt{Create}(f_1, a_1, \ldots, a_n)$$

`start()`: Eager Thread Creation (Binding Form)

t.`start`$(f_1, a_1, \ldots, a_n)$
$\boldsymbol{Thread}n\langle \boldsymbol{R}, \boldsymbol{A}_1, \ldots, \boldsymbol{A}_n\rangle$ t
$\boldsymbol{R}$ $(*f)(\boldsymbol{A}_1, \ldots, \boldsymbol{A}_n)$
$\boldsymbol{A}_i$ a_i

Specification: Binds t to a thread returning $f(a_1, \ldots, a_n)$. This binding overrides any previous binding of t.

Implementation: The thread is activated by allocating its stack and putting it into the tail of the ready queue (Section 3.1.1).

Note: This operation is more efficient than the otherwise equivalent form (Section 3.2.3)

t = ***Thread**n*⟨***R***, ***A***$_1$, ..., ***A***$_n$⟩::`Start`(f_1, a_1, ..., a_n)

`Wait()`: Waiting for a Thread Result

R `Wait`(t)
  Thread⟨***R***⟩ t
  Thread**n*⟨R***, ***A***$_1$, ..., ***A***$_n$⟩ t

Specification: Returns the result of the thread to which the handle t is bound.

Constraints: It is illegal to apply this function, if t is not bound to any thread.

Implementation: If the thread is still executing, the current thread is blocked until the thread result becomes available.

3.2.4 Miscellaneous

`stop()`: Stopping a Thread

t`->stop()`
  Thread⟨***R***⟩ t
  Thread**n*⟨R***, ***A***$_1$, ..., ***A***$_n$⟩ t

Specification: Hints to the system that the result of the thread to which t is bound is not needed any more.

Constraints: It is illegal to call this procedure, if t is not bound to any thread. Any further application `Wait`(s) is illegal if s is bound to the same thread as t.

Complexity: $O(1)$.

Note: Since unreferenced threads are automatically reclaimed during garbage collection, it is not necessary to explicitly stop a thread. The only purpose of this facility is to guarantee that an unused and not yet activated thread is not activated before the next garbage collection takes place.

Implementation: If the thread is not yet activated, it is removed from the lazy thread stack. If the thread has been already activated, it will not be put back into the ready queue after the next preemption. If any other thread waits for the result of such a thread, it will be blocked forever (deadlock).

`root()`: Root Set of Garbage Collection

t`->root()`
t`->noroot()`
  Thread$\langle\boldsymbol{R}\rangle$ t
  Thread$n\langle\boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle$ t

Specification: The first form declares thread t to be in the root set of the garbage collector. A thread in the root set will be not reclaimed by the garbage collector even if it is not referenced any more by the main thread. The second form removes t from the root set.

Constraints: It is illegal to apply `root` to some t which is already in the root set of the garbage collector and to apply `noroot` to some t which is not in the root set.

Note: At program start, only the main thread is in the root set. A thread which is not (directly or indirectly) referenced by the main thread and which is also not in the root set of the garbage collector may be *terminated at any time* and may be even *not executed at all.*

3.2.5 General

`init()`: Unbinding a Threads

t`.init()`
  Thread$\langle\boldsymbol{R}\rangle$ t
  Thread$n\langle\boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle$ t

Specification: Resolves any binding of t.

Complexity: $O(1)$.

`Null()`: Test for Unbound Thread

empty = `Null(`t`)`
  int *empty*
  Thread$\langle\boldsymbol{R}\rangle$ t
empty = `Null(`t`)`
  int *empty*
  Thread$n\langle\boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle$ t

Specification: Returns 1 if t is not bound to any thread and 1 else.

Complexity: $O(1)$.

`Thread()`: Duplication

Thread$\langle\boldsymbol{R}\rangle$ $t(s)$
    Thread$\langle\boldsymbol{T}\rangle$ s
Thread$\langle\boldsymbol{R}\rangle$ $t(s)$
  Thread$n\langle\boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n\rangle$ s

Specification: Defines a thread handle t bound to the same thread as s. If s is not bound to any thread, t is also unbound.

Note: This definition is more efficient than the otherwise equivalent form

$\textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ t = s$

=: Assignment

$t = s$

$\quad \textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ t$

$\quad \textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ s$

$t = s$

$\quad \textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ t$

$\quad \textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ s$

Specification: Binds a to the thread b (i.e. to the thread to which b is bound). This overrides any previous binding of a. If b is not bound to any thread, a becomes unbound.

Complexity: $O(1)$.

Note: By implicit conversion, the following assignment is legal:

$\textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ t$

$\textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ s$

$t = s$

==: Comparison

$equal = (t == s)$

$\quad \textbf{\textit{int}}\ equal$

$\quad \textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ t$

$\quad \textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ s$

$equal = (\textbf{\textit{int}}\ t == s)$

$\quad \textbf{\textit{int}}\ equal$

$\quad \textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ t$

$\quad \textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ s$

Specification: Returns 1 if a and b are bound to the same array and 0 else.

Complexity: $O(1)$.

Note: By implicit conversion, the following comparisons are legal:

$\textbf{\textit{Thread}}\langle \boldsymbol{R} \rangle\ t$

$\textbf{\textit{Thread}}n\langle \boldsymbol{R},\ \boldsymbol{A}_1,\ \ldots,\ \boldsymbol{A}_n \rangle\ s$

$t == s$

$s == t$

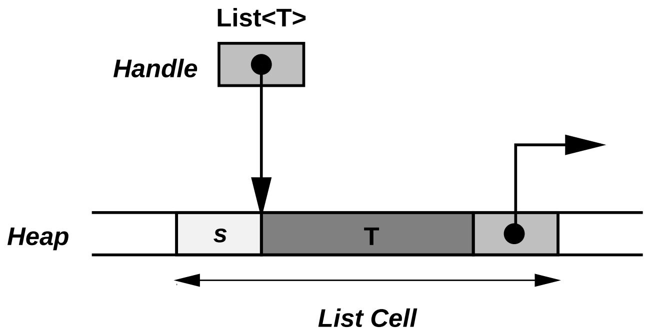

Figure 10: List Representation

3.3 Lists

A *list* of type ***List***$\langle$***T***$\rangle$ is either empty or consists of an *head* element h of type ***T*** and a tail list t of type ***List***$\langle$***T***$\rangle$. By repeated list construction, we can build sequences of ***T*** elements of arbitrary length.

3.3.1 Representation

A list of type ***List***$\langle$***T***$\rangle$ is represented as follows (see Figure 10):

- The *list handle* contains either the null pointer or a pointer to a list cell residing in the heap.
- A *list cell* consists of the head element (a ***T*** object) and a handle to the tail list.

Each list cell is prefixed by one word that holds the size of the cell. The size of s list cell is thus the sum of the sizes of an integer word, of ***T*** and of ***List***$\langle$***T***$\rangle$ where the size of ***List***$\langle$***T***$\rangle$ is the pointer size of the machine.

3.3.2 Definition

List$\langle$***T***$\rangle$ l
List$\langle$***T***$\rangle$ $l(h, t)$
    T h
    List$\langle$***T***$\rangle$ t

Specification: Defines a list handle l.

1. In the first form, l is bound to an empty list.
2. In the second form, l is bound to a new list cell with head element h and tail list t. This definition is equivalent to (Section 3.3.3)

> $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ l
> l.`cons`(h, t)

which is in turn equivalent to but more efficient than (Section 3.3.3)

> $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ l
> l = `Cons`(h, t)

3.3.3 Constructors

`Nil()`: Empty List

nil = $\boldsymbol{List}\langle\boldsymbol{T}\rangle$::`Nil`()
  $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ nil

Specification: Returns an empty list nil.

Complexity: $O(1)$.

`nil()`: Empty List (Binding Form)

l.`nil`()
  $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ l

Specification: Binds l to the empty list.

Complexity: $O(1)$.

Note: This operation is more efficient than the otherwise equivalent form (Section 3.3.3)

> l = $\boldsymbol{List}\langle\boldsymbol{T}\rangle$::`Nil`()

`Cons()`: List Construction

l = `Cons`(h, t)
  $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ l
  $\boldsymbol{T}$ h
  $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ t

Specification: Returns a list l with head element h and tail list t.

Complexity: $O(1)$.

`cons()`: List Construction (Binding Form)

l.`cons`(h, t)
  $\boldsymbol{T}$ h
  $\boldsymbol{List}\langle\boldsymbol{T}\rangle$ t

Specification: Binds l to a list with head element h and tail list t.

Complexity: $O(1)$

Note: This operation is more efficient than the otherwise equivalent form (Section 3.3.3)

> l = `Cons`(h, t)

3.3.4 Selectors

`Null()`: Test for Empty List

empty = `Null`(l)
    int *empty*
    List$\langle$***T***$\rangle$ l

Specification: Returns 1 if l is the empty list and 0 else.

Complexity: $O(1)$.

`Head()`: Head Element

h = `Head`(l)
    T h
    List$\langle$***T***$\rangle$ l

Specification: Returns the head element h of l.

Constraints: It is illegal to apply `Head` to an empty l.

Complexity: $O(1)$.

`Tail()`: Tail List

t = `Tail`(l)
    List$\langle$***T***$\rangle$ t
    List$\langle$***T***$\rangle$ l

Specification: Returns the tail list t of l.

Constraints: It is illegal to apply `Tail` to an empty l.

Complexity: $O(1)$.

`tail()`: Tail List (Binding Form)

l.`tail`()
    List$\langle$***T***$\rangle$ l

Specification: Binds l to the tail list of l.

Constraints: It is illegal to apply `tail` to an empty l.

Complexity: $O(1)$.

Note: This operation is more efficient than the otherwise equivalent form (Section 3.3.4).

    l = `Tail`(l)

3.3.5 Destructive Updating

`head()`: Update Head Element

l.`head`(h)
    List$\langle$***T***$\rangle$ l
    T h

Specification: Destructively updates the head element of l by e.

Constraints: It is illegal to apply `head` to an empty l.

Complexity: $O(1)$.

`tail()`: Update Tail List

l.`tail`(t)
    List$\langle$***T***$\rangle$ l
    List$\langle$***T***$\rangle$ t

Specification: Destructively updates the tail list of l by t.

Constraints: It is illegal to apply `tail` to an empty l.

Complexity: $O(1)$.

3.3.6 General

`init()`: Unbinding a List

l.`init`()
    List$\langle$***T***$\rangle$ l

Specification: Resolves any binding of l.

Complexity: $O(1)$.

Note: This operation is equivalent to (Section 3.3.3)

    l.`nil`()

`Null()`: Test for Unbound List

See `Null`(l) (Section 3.3.4).

`List()`: Duplication

List$\langle$***T***$\rangle$ $l(r)$
        List$\langle$***T***$\rangle$ r

Specification: Defines a list l bound to the same list as r. If r is not bound to any list, l is also unbound.

Note: This definition is more efficient than the otherwise equivalent form

    List$\langle$***T***$\rangle$ $l = r$

=: Assignment

$l = r$

  $\boldsymbol{List\langle T\rangle}\ l$
  $\boldsymbol{List\langle T\rangle}\ r$

Specification: Binds l to the list b (i.e. to the list to which b is bound). This overrides any previous binding of l.

Complexity: $O(1)$.

==: Comparison

$equal = (l == r)$

  $\boldsymbol{int}\ equal$
  $\boldsymbol{List\langle T\rangle}\ l$
  $\boldsymbol{List\langle T\rangle}\ r$

Specification: Returns 1 if l and r are bound to the same list and 0 else.

Complexity: $O(1)$.

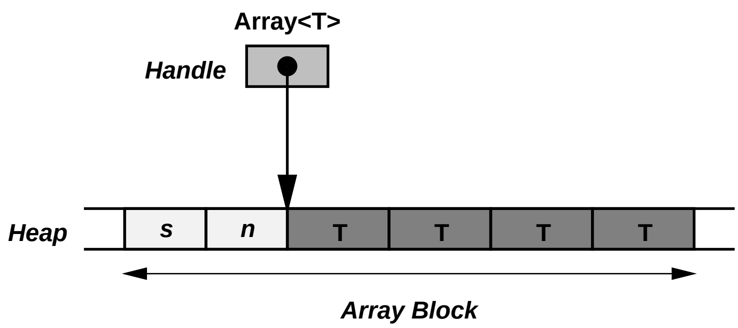

Figure 11: Array Representation

3.4 Arrays

An *array* of type $\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ consists of a sequence of $\boldsymbol{T}$ elements each of which can be accessed and updated in constant time. The *length* of the array (the number of elements) is arbitrary but fixed at definition time.

3.4.1 Representation

An array of type $\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ is represented as follows (see Figure 11):

- The *array handle* contains either the null pointer or a pointer to an array block residing in the heap.
- The *array block* consists of n slots each of which may hold a $\boldsymbol{T}$ object.

The array block is prefixed by two words that hold the size of the block and the number of array slots, respectively. The size of the array block is therefore the sum of the sizes of two integer words and of n $\boldsymbol{T}$ objects. The size of $\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ is the pointer size of the machine.

3.4.2 Definition

$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a
$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a(n)
    int n

Specification: Defines an array handle a.

1. In the first form, a is not yet bound to any array.
2. In the second form, a is bound to a new array with n slots. *The contents of the slots are undefined.*

Constraints: It is illegal to define a with $n < 0$.

Note: The second form is equivalent to (Section 3.4.3)

$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a
a.`alloc`(n)

which is in turn equivalent to but more efficient than (Section 3.4.3)

$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a
a = $\boldsymbol{Array}\langle \boldsymbol{T} \rangle$::`Alloc`($n$)

3.4.3 Allocation

`Alloc()`: Allocation

a = $\boldsymbol{Array}\langle \boldsymbol{T} \rangle$::`Alloc`($n$)
$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a
int n

Specification: Returns a new array a with n slots. *The contents of the slots are undefined.*

Constraints: It is illegal to call `Alloc` with $n < 0$.

`alloc()`: Allocation (Binding Form)

a.`alloc`(n)
$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a
int n

Specification: Binds a to a new array with n slots. *The contents of the slots are undefined.* This binding overrides any previous binding of a.

Constraints: It is illegal to call `alloc` with $n < 0$.

Note: This operation is more efficient than the otherwise equivalent form (Section 3.4.3)

a = $\boldsymbol{Array}\langle \boldsymbol{T} \rangle$::`Alloc`($n$)

3.4.4 Selection

`Length()`: Length

l = `Length`(a)
int l
$\boldsymbol{Array}\langle \boldsymbol{T} \rangle$ a

Specification: Returns the number of slots of array a.

Constraints: It is illegal to apply `Length` to an unbound a.

Complexity: $O(1)$.

`[]`: Indexing

e = *a*`[`*i*`]`
a`[`*i*`]` = *e*
    T *e*
    Array⟨***T***⟩ *a*
    int *i*

Specification: The first form returns the object *e* stored in slot *i* of array *a*. The second form destructively updates *a* by writing *e* into slot *i*. The first slot has index 0.

Constraints: It is illegal to apply this operator to an unbound *a* or to an index outside the range [0, `Length`(a)-1].

Complexity: $O(1)$.

3.4.5 General

`init()`: Unbinding an Array

a.`init`()
    Array⟨***T***⟩ *a*

Specification: Resolves any binding of *a*.

Complexity: $O(1)$.

`Null()`: Test for Unbound Array

unbound = `Null`(a)
    int *unbound*
    Array⟨***T***⟩ *a*

Specification: Returns 1 if a is not bound to any array and 1 else.

Complexity: $O(1)$.

`Array()`: Duplication

Array⟨***T***⟩ *a*(*b*)
        Array⟨***T***⟩ *b*

Specification: Defines an array a bound to the same array as *b*. If *b* is not bound to any array, a is also unbound.

Note: This definition is more efficient than the otherwise equivalent form

    Array⟨***T***⟩ a = *b*

=: Assignment

a = b
  Array$\langle$***T***$\rangle$ a
  Array$\langle$***T***$\rangle$ b

Specification: Binds a to the array b (i.e. to the array to which b is bound). This overrides any previous binding of a. If b is not bound to any array, a becomes unbound.

Complexity: $O(1)$.

==: Comparison

equal = (a == b)
  int *equal*
  Array$\langle$***T***$\rangle$ a
  Array$\langle$***T***$\rangle$ b

Specification: Returns 1 if a and b are bound to the same array and 0 else.

Complexity: $O(1)$.

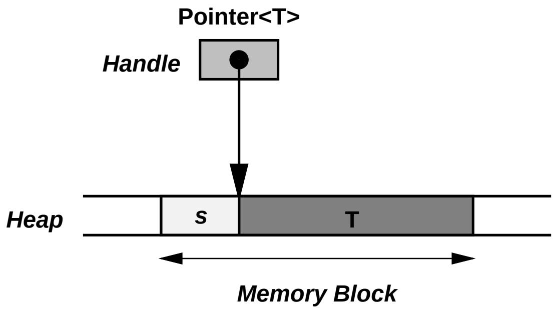

Figure 12: Pointer Representation

3.5 Pointers

A *pointer* of type ***Pointer***$\langle$***T***$\rangle$ is a handle that may refer to any object of type ***T*** stored in the heap. By this handle, the referenced object may be accessed and updated in constant time. Multiple pointers may refer to the same object.

3.5.1 Representation

A pointer of type ***Pointer***$\langle$***T***$\rangle$ is represented as follows (see Figure 12):

- The *handle*, i.e. the pointer itself, is either null or refers to a memory block in the heap.
- The *memory block* in the heap may hold any ***T*** object.

The memory block is prefixed by one word that holds the size of the block. The size of the memory block is thus the sum of the sizes of one integer word and of a ***T*** object. The size of ***Pointer***$\langle$***T***$\rangle$ is the pointer size of the machine.

3.5.2 Definition

Pointer$\langle$***T***$\rangle$ p
Pointer$\langle$***T***$\rangle$ $p(1)$

Specification: Defines a pointer p.

1. In the first form, p may refer to any object of type ***T***. *p does not yet refer to any object.*
2. In the second form, p is bound to a new memory block that may hold any ***T*** object. *The memory block does not yet hold any **T** object.*

Constraint: It is syntactically possible but illegal to define for $n \neq 0$

> ***Pointer***$\langle$***T***$\rangle$ $p(n)$

Note: The second form is equivalent to (Section 3.5.3)

> ***Pointer***$\langle$***T***$\rangle$ p
> p.`alloc`()

which is in turn equivalent to but more efficient than (Section 3.5.3)

> ***Pointer***$\langle$***T***$\rangle$ p
> p = ***Pointer***$\langle$***T***$\rangle$::`Alloc`()

3.5.3 Allocation

`Alloc()`: Allocation

p = ***Pointer***$\langle$***T***$\rangle$::`Alloc`()
  Pointer$\langle$***T***$\rangle$ p

Specification: Returns a reference p to a new memory block that may hold any ***T*** object. *The memory block does not yet hold any* ***T*** *object.*

`alloc()`: Allocation (Binding Form)

p.`alloc`()

Specification: Binds p to a new memory block that may hold any ***T*** object. *The memory block does not yet hold any* ***T*** *object.* This binding overrides any previous binding of p.

Note: This operation is more efficient than the otherwise equivalent form (Section 3.5.3)

> p = ***Pointer***$\langle$***T***$\rangle$::`Alloc`(n)

3.5.4 Selection

`Null()`: Test for Null Pointer

unbound = `Null`(p)
  int *unbound*
  Pointer$\langle$***T***$\rangle$ p

Specification: Returns 1 if p is not bound to any memory block and 0 else.

Complexity: $O(1)$.

***: Indirection**

e = *p
*p = e
  T e
  Pointer$\langle$***T***$\rangle$ p

Specification: The first form returns the object e stored in the memory block to which p refers. The second form destructively updates this memory block by overwriting its content with e.

Constraint: It is illegal to apply `*` to an unbound p.

Complexity: $O(1)$.

`->`: Dereferencing

e = p`->`a
p`->`a = e
    `struct` ***T*** $\{\ \ldots$ ***S*** `a`; $\ldots\}$
    Pointer$\langle$***T***$\rangle$ p
    S e

Specification: The first form returns into e the component a of the object stored in the memory block to which p refers. The second form destructively updates this component by overwriting its content with e.

Constraint: It is illegal to apply this operation to an unbound p.

Complexity: $O(1)$.

3.5.5 General

`init()`: Unbinding a Pointer

p.`init()`
    Pointer$\langle$***T***$\rangle$ p

Specification: Resolves any binding of p.

Complexity: $O(1)$.

`Null()`: Test for Unbound Pointer

unbound = `Null`(p)
    int *unbound*
    Pointer$\langle$***T***$\rangle$ p

Specification: Returns 1 if p is not bound to any object and 1 else.

Complexity: $O(1)$.

`Pointer()`: Duplication

Pointer$\langle$***T***$\rangle$ $p(q)$
        Pointer$\langle$***T***$\rangle$ q

Specification: Defines a pointer p bound to the same array as q. If q is not bound to any object, p is also unbound.

Note: This definition is more efficient than the otherwise equivalent form

  Pointer$\langle$***T***$\rangle$ p = q

=: Assignment

$p = q$
    Pointer$\langle$***T***$\rangle$ p
    Pointer$\langle$***T***$\rangle$ q

Specification: Binds p to the object q (i.e. to the object to which q is bound). This overrides any previous binding of p. If q is not bound to any object, p becomes unbound.

Complexity: $O(1)$.

==: Comparison

$equal = (p == q)$
    int *equal*
    Pointer$\langle$***T***$\rangle$ p
    Pointer$\langle$***T***$\rangle$ q

Specification: Returns 1 if p and q are bound to the same object and 0 else.

Complexity: $O(1)$.

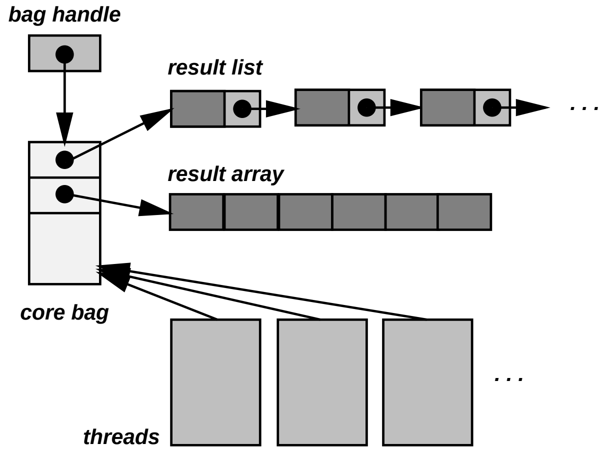

Figure 13: Thread Bag Representation

3.6 Thread Bags

A *thread bag* of type ***ThreadBag***$\langle$***T***$\rangle$ holds a collection of threads with result type ***T***. The order in which the bag delivers the thread results is unspecified apart from the constraint that the bag delivers the results of all terminating threads first (i.e. a thread waiting for a bag result may be permanently blocked only after it has received the results of all terminating threads). This constraint is implemented by returning those results first that are delivered first (non-determinism). This may improve the efficiency of parallel programs and is in most cases the actual reason why thread bags are applied.

3.6.1 Representation

A thread bag of type ***ThreadBag***$\langle$***T***$\rangle$ is represented as follows (see Figure 13):

- A *bag handle* that references the core bag.
- The *core bag* which contains one of the following:
  1. A *result list* collecting the thread results delivered before the bag is closed.
  2. A *result array* collecting the results of all threads.
- A set of *threads* delivering their results to the bag.

Initially a bag is in the *open* state. In this state, it is not known how many threads the bag will eventually hold; arbitrarily many threads can be added to the bag. When the bag is eventually *closed*, only a declared number of more threads can be added to the bag. During the open state, results delivered to the bag are collected in the result list. When the bag is closed (i.e. the final bag size is known) an array is allocated to receive all thread results, those that have been already collected in the result list, and those that still remain to be computed.

3.6.2 Definition

ThreadBag$\langle$***T***$\rangle$ b
ThreadBag$\langle$***T***$\rangle$ $b(n)$
    int n

Specification: Defines a thread bag handle b.

1. In the first form, b *is not yet bound to any bag.*
2. In the second form, b is bound to a new and empty bag which can hold n threads. *The bag is closed.*

Constraint: It is illegal to define a thread bag with $n<0$.

Note: The second form is equivalent to (Section 3.6.3)

> ***ThreadBag***$\langle$***T***$\rangle$ b
> b.`open`(n)

3.6.3 Constructors

`Open()`: Open a Bag

b = ***ThreadBag***$\langle$***T***$\rangle$::`Open`()
b = ***ThreadBag***$\langle$***T***$\rangle$::`Open`(n)
    ThreadBag$\langle$***T***$\rangle$ b
    int n

Specification: Returns a new and empty bag b.

1. In the first form, b is *open* and may collect an arbitrary number of threads.
2. In the second form, b is *closed* and at most n more threads may be added to the bag.

Constraint: It is illegal to apply `Open` to $n<0$.

Note: The second form is equivalent to (Section 3.6.5)

> ***ThreadBag***$\langle$***T***$\rangle$ b
> b = ***ThreadBag***$\langle$***T***$\rangle$::`Open`()
> b->`close`(n)

`open()`: Open a Bag (Binding Form)

b.`open`()
b.`open`(n)
    ThreadBag$\langle$***T***$\rangle$ b
    int n

Specification: Binds b to a new and empty bag. This overrides any previous binding of b.

1. In the first form, b is *open* and may collect an arbitrary number of threads.
2. In the second form, b is *closed* and at most n more threads may be added to the bag.

Constraint: It is illegal to apply `open` to $n<0$.

Note: The second operation is equivalent to (Section 3.6.5)

> b.`open`()
> b`->close`(n)

Both operations are more efficient than the otherwise equivalent forms (Section 3.6.3)

> b = ***ThreadBag***⟨***T***⟩::`Open`()
> b = ***ThreadBag***⟨***T***⟩::`Open`(n)

`ThreadBag`*n*: Adding a Thread

ThreadBag*n*⟨***R***, $\boldsymbol{A}_1$, ..., $\boldsymbol{A}_n$⟩(b, f, a_1, ..., a_n)
ThreadBag*n*⟨***R***, $\boldsymbol{A}_1$, ..., $\boldsymbol{A}_n$⟩::`create`(b, f, a_1, ..., a_n)
ThreadBag*n*⟨***R***, $\boldsymbol{A}_1$, ..., $\boldsymbol{A}_n$⟩::`start`(b, f, a_1, ..., a_n)
> ***ThreadBag***⟨***R***⟩ b
> ***R*** (*f)($\boldsymbol{A}_1$, ..., $\boldsymbol{A}_n$)
> $\boldsymbol{A}_i$ a_i

Specification: Adds to b a new thread computing $f(a_1, \ldots, a_n)$.

1. The first form is equivalent to the second or to the third form.
2. In the second form, the created thread is put into the lazy stack (Section 3.1.1).
3. In the third form, the created thread is activated and put into the ready queue (Section 3.1.1).

Constraints: It is illegal to apply this operation if b is unbound or if it is bound to a closed thread that holds already the number of declared threads.

Restrictions: The number of arguments n ranges from 0 to an implementation-dependent constant (currently 4). This range may be extended by defining the corresponding template class.

Note: Only the third form of thread creation by `start` guarantees the "true" semantics of thread bags, namely that in the ordering of delivered results (Section 3.6.4) non-terminating threads only appear after all terminating threads. However, if all threads are guaranteed to terminate, the second form is equivalent and in general much more efficient.

Implementation: Normally, the first form is equivalent to the second form. However, if the preprocessor constant `RT_TBAG_NOTLAZY` (Section 3.1.5) is defined before including the header file *rt++.h*, the first form is equivalent to the third form.

3.6.4 Selectors

`Wait()`: Returning the Bag Results

r = `Wait`(*b*, *i*)
r = `Wait`(*b*, *i*, &*d*)

 R r
 ThreadBag⟨***R***⟩ b
 int i
 int d

Specification: Returns the result *r* with index *i* from bag *b*. In the second form, *d* is set to 1 if an object with such an index exists and 0 else. The first result has index 0.

The index ordering of thread results is unspecified apart from the following constraint (n is the number of threads in b):

If `Wait`(*b*, *i*) blocks forever (because the corresponding thread t_i never terminates), then also each thread t_j with $i < j < n$ will never terminate provided that t_j has been started with `start` (Section 3.6.3).

Constraints: It is illegal to apply `Wait` to an unbound handle *b*. It is illegal to apply `Wait` to some *i* less than zero. In the first form, it is illegal to apply `Wait` to some *i* larger than `Size`(*b*)-1.

Complexity:

- $O(n)$ *before* the thread is closed.
- $O(1)$ *after* the thread has been closed.

Implementation: The thread result with index *i* is the (i+1)-th result that has been computed i.e. the index ordering reflects the order in which threads have terminated and delivered their results. A thread waiting on thread result *i* is therefore blocked until i+1 threads have terminated.

`Size()`: Size of a Thread Bag

size = `Size`(*b*)

 int *size*
 ThreadBag⟨***R***⟩ *b*

Specification: Returns the number of threads in the bag *after the bag has been closed.* This includes also all threads whose number has been declared in the `close` statement (Section 3.6.5) but that have actually not yet entered the bag.

Constraints: It is illegal to apply this operation if *b* is unbound.

Complexity: $O(1)$.

Implementation: If the bag is not yet closed, the current thread is blocked until another thread closes the bag. If the bag will never be closed, the current thread will be permanently blocked.

3.6.5 Miscellaneous

`close`: Closing a Bag

b`->close()`
b`->close(`n`)`
  ThreadBag$\langle$***T***$\rangle$ b

Specification: Closes the bag b.

1. In the first form, no more thread may be added to b.
2. In the second form, n more threads may be added to b.

Constraints: It is illegal to apply `close` if b is unbound or if b is bound to an already closed or stopped bag (Section 3.6.5). It is illegal to apply `close` to $n < 0$.

`stop`: Stopping a Bag

b`->stop()`
  ThreadBag$\langle$***T***$\rangle$ b

Specification: Stops the bag b i.e. ensures that the bag is not accessed any more.

Constraints: It is illegal to apply `close` if b is unbound or if b is bound to an already stopped bag. It is illegal to apply any operation to a closed b.

Implementation: All threads in the bag that have not yet completed execution are stopped (Section 3.2.4).

3.6.6 General

`init()`: Unbinding a Thread Bag

b`.init()`
  ThreadBag$\langle$***T***$\rangle$ b

Specification: Resolves any binding of b.

Complexity: $O(1)$.

`Null()`: Test for Unbound Thread Bag

unbound = `Null(`b`)`
  int *unbound*
  ThreadBag$\langle$***T***$\rangle$ b

Specification: Returns 1 if b is not bound to any thread bag and 1 else.

Complexity: $O(1)$.

`ThreadBag()`: Duplication

ThreadBag $\langle$***T***$\rangle$ $b(c)$
    ThreadBag $\langle$***T***$\rangle$ c

Specification: Defines a thread bag handle b bound to the same thread bag as c. If c is not bound to any bag, b is also unbound.

Note: This definition is more efficient than the otherwise equivalent form

  ThreadBag $\langle$***T***$\rangle$ c = b

=: Assignment

b = c
  ThreadBag $\langle$***T***$\rangle$ b
  ThreadBag $\langle$***T***$\rangle$ c

Specification: Binds b to the thread bag c (i.e. to the bag to which c is bound). This overrides any previous binding of b. If c is not bound to any bag, b becomes unbound.

Complexity: $O(1)$.

==: Comparison

equal = (b == c)
  int *equal*
  ThreadBag $\langle$***T***$\rangle$ b
  ThreadBag $\langle$***T***$\rangle$ c

Specification: Returns 1 if b and c are bound to the same thread bag and 0 else.

Complexity: $O(1)$.

References

[And90] Thomas E. Anderson. Fastthread's user's manual. Technical report, University of Washington, Seattle, Washington, 1990. Available as part of the Quartz distribution `ftp://ftp.cs.washington.edu /pub /Quartz1.0.tar.Z`.

[BDS92] Peter A. Buhr, Glen Ditchfield, and Richard A. Stroobosscher. μC++: Concurrency in the Object-Oriented Language C++. *Software — Practice and Experience*, 22(2):137–172, Feburary 1992. System available at `ftp://plg.uwaterloo.ca /pub /uSystem`.

[BLLW88] Brian N. Bershad, Edward D. Lazowska, Henry M. Levy, and David B. Wagner. PRESTO: A System for Object-Oriented Parallel Programming. *Software — Practice and Experience*, 18(8):713–732, 1988. System available at `ftp://ftp.cs.washington.edu /pub /presto1.0.tar.Z`.

[BS90] Peter A. Buhr and Richard A. Stroobosscher. The μSystem: Providing Light-Weight Concurrency on Shared-Memory Multiprocessor Computers Running UNIX. *Software — Practice and Experience*, 20(9):929–963, September 1990. System available at `ftp://plg.uwaterloo.ca /pub /uSystem /`.

[HCM94] Matthew Haines, David Cronk, and Piyush Mehrotra. The Chant User's Guide. Technical report, ICASE, NASA Langley Research Center, Hampton, VA, 1994. System available at `http://www.icase.edu /~haines /html /chant.html`.

[HNS94] Hoon Hong, Andreas Neubacher, and Volker Stahl. The STURM Library Manual — A C++ Library for Symbolic Computation. Technical Report 94-30, RISC-Linz, Johannes Kepler University, Linz, Austria, April 1994. Sturm home page `http://www.risc.uni-linz.ac.at /local /projects /sturm /index.html`.

[HNS95] Hoon Hong, Andreas Neubacher, and Wolfgang Schreiner. The Design of the SACLIB/PACLIB Kernels. *Journal of Symbolic Computation*, 19:111–132, 1995. `http://www.risc.uni-linz.ac.at /people /schreine /papers /jsc95.ps.gz`.

[Kep93] David Keppel. Tools and Techniques for Building Fast Portable Thread Packages. Technical Report UW-CSE 93-05-06, University of Washington, Seattle, Washington, 1993. Report available at `ftp://ftp.cs.washington.edu /tr /1993 /05 /UW-CSE-93-05-06.PS.Z`, System available at `ftp://ftp.cs.washington.edu /pub /qt-002.tar.gz`.

[KW92] Wolfgang Küchlin and Jeffrey Ward. Experiments with Virtual C Threads. In *4th IEEE Symposium on Parallel and Distributed Processing*, Arlington, Texas, December 1992. IEEE Press.

[MKHJ91] E. Mohr, D. A. Kranz, and R. H. Halstead Jr. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[Mue93] Frank Mueller. A Library Implementation of POSIX Threads under UNIX. In *1993 Winter USENIX Conference*, pages 29–41, San Diego, CA, January 25–29, 1993. Paper available at `ftp://ftp.cs.fsu.edu /pub /PART /publications /pthreads_usenix93.ps.gz`, System available at `ftp://ftp.cs.fsu.edu /pub /PART /PTHREADS /pthreads.tar.Z`.

[Sch94] Wolfgang Schreiner. Virtual Tasks for the PACLIB Kernel. In Bruno Buchberger and Jens Volkert, editors, *Parallel Processing: CONPAR 94 — VAPP VI International Conference on Parallel and Vector Processing*, volume 854 of *Lecture Notes in Computer Science*, pages 533–544, Linz, Austria, September 6–8, 1994. Springer, Berlin. Paper available at `http://www.risc.uni-linz.ac.at /people /schreine /papers /conpar94.ps.gz`.

[Sch96] Wolfgang Schreiner. A Para-Functional Programming Interface for a Parallel Computer Algebra Package. *Journal of Symbolic Computation*, 1996. To appear. Preprint available at `http://www.risc.uni-linz.ac.at /people /schreine /papers /jsc96.ps.gz`.

[SH93] Wolfgang Schreiner and Hoon Hong. A New Library for Parallel Algebraic Computation. In Richard F. Sincovec et al., editors, *Sixth SIAM Conference on Parallel Processing for Scientific Computing*, volume II, pages 776–783, Norfolk, Virginia, March 22–24, 1993. SIAM. Paper available at `http://www.risc.uni-linz.ac.at /people /schreine /papers /paclib.ps.gz`.

[Sun96] *Solaris 2.4 Multithreaded Programming Guide*, 1996. `http://www.sun.com /sunsoft /Developer-products /sig /threads /doc /MultithreadedProgrammingGuide_Solaris24.ps`.

[Thr96] *The Threads Page*, 1996. `http://www.sun.com /sunsoft /Developer-products /sig /threads /`.

A Implementation

In this section we give an overview on the implementation of the two main components of the RT++ runtime system, the scheduling of threads and the garbage collection of heap objects.

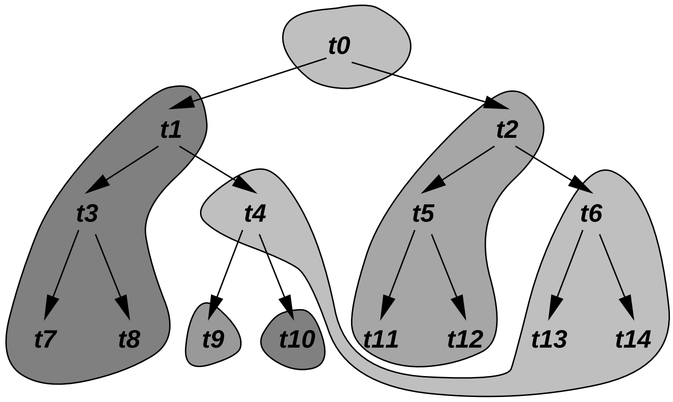

Figure 14: A Thread Tree

A.1 Thread Scheduling

By default, new threads are declared in a lazy state and pushed on a global stack. Initially, there is only one active thread, the program's main thread. A lazy thread in the global stack is executed on the following occasions only:

Idling An idle process finds no activated thread ready for execution. In this case, it takes a lazy thread from the global stack, allocates a thread space and switches control to the newly activated thread.

Waiting An active thread needs the result of a thread which is still in lazy state. In this case, the active threads executes the code of the lazy thread itself.

Terminating A thread is going to terminate but finds a lazy thread in the global stack. In this case, the thread continues execution with the code of the lazy thread.

Control is switched from one active thread to another only, if the time slice of a thread has ended or if the thread needs the result of another thread which has been already activated (by another process) but not yet terminated.

As an example, let us take a look at a program where the initial thread t_0 creates two threads and waits for their results in turn. Each new thread creates two other threads resulting in the tree of threads of depth 4 depicted in Figure 14.

Assume that there are three processes executing the program. A possible execution schedule is then described by the following table:

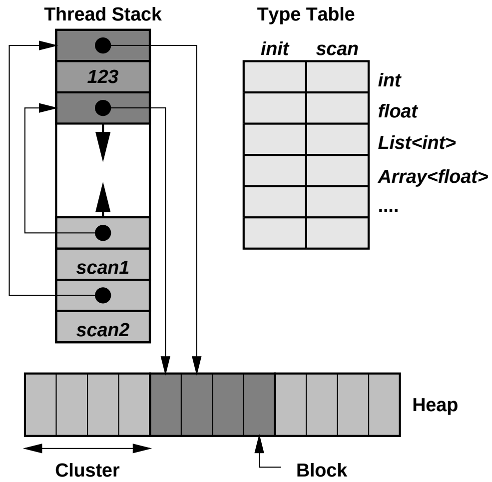

Figure 15: Heap Memory Management

Time	P_0	P_1	P_2	Task Stack
0	t_0			t_2, t_1
1		t_1	t_2	t_6, t_4, t_5, t_3
2	t_6	t_3	t_5	$t_{14}, t_{12}, t_8, t_{13}, t_{11}, t_7, t_4$
3	t_{13}	t_7	t_{11}	t_{14}, t_{12}, t_8, t_4
4	t_{14}	t_8	t_{12}	t_4
5	t_4			t_{10}, t_9
6		t_{10}	t_9	

At time 0, process P_0 executes thread t_0 which creates threads t_1 and t_2. t_1 is immediately taken by P_1 for execution as well as t_2 by P_2 such that P_0 becomes idle at time 1. However, t_1 and t_2 push t_3, t_5, t_4, t_6 on the thread stack such that at time 2 process P_0 starts execution of t_6. In the following time steps, all processes now create the corresponding thread subtrees and, by waiting for the respective results, execute the corresponding code. At time 5, process P_0 is the quickest to grab t_4 and is the only one to continue execution while the other processes idle. In the end P_1 and P_2 take the newly created t_{10} and t_9.

All in all, the 15 lazy threads of the tree are executed by only six real threads of which only four threads are active at the same time (including the one thread which is blocked waiting for the result). Figure 14 illustrates this fact by a “bubble” for each active thread (with the different shadings representing the different processes).

A.2 Garbage Collection

In Figure 15 the main components of the RT++ memory management system are outlined.

The heap is partitioned into *clusters* each of which is itself partitioned into an equal number of *blocks* of equal size. These blocks are the units of parallelism in the sweeping phase of the garbage collector (see below). Initially the heap consists of only one cluster. If after garbage collection too little heap space is free or the free heap space is too fragmented to satisfy a memory request, a new cluster is allocated unless the shared memory bound of the program is reached.

Each process has an associated list of free heap space. When a thread executed by the process allocates heap space, this space is taken from the local list (which can be accessed without synchronization with other processes). When the process runs out of free heap space, it requests a new free list from the global free list (which must be protected by a locking mechanism from simultaneous access by multiple processes). If also the global free list runs out of space, garbage collection is triggered.

Garbage collection is performed by a conventional mark and sweep scheme where both phases are parallelized. In the first phase, all heap entities reachable from the *root set* of pointers are marked. In the second phase, all unmarked heap entities are reclaimed into the global free list. By default, the root set consists just of the pointer to the main thread of the RT++ program. If during the marking phase a not yet marked active thread is encountered, its stack is put into a pool; all processes in parallel take stacks from this pool and scan the contained heap references. Likewise, in the sweep phase all processes in parallel collect free space from the individual heap blocks[4]. Threads are treated like all other heap objects; any thread not reachable by the root set is automatically reclaimed (together with its stack, if the thread has been already activated).

The marking phase of the garbage collector is based on the C++ type system such that he RT++ garbage collector "knows" the location and type of each heap reference and can thus safely mark all reachable heap objects:

When a C++ variable is declared within some function of an RT++ program, this variable is pushed on the stack of the current thread. If this variable is declared using an RT++ type constructor, the *address and type* of this variable are registered by pushing the corresponding information on another stack. This "shadow" stack is allocated at the opposite end of the space reserved for the actual thread stack and grows into the opposite direction.

The registered information is the address of the *scan function* corresponding to the type of the allocated variable. This function recursively scans and marks all heap blocks reachable via the variable by calling the scan functions of the corresponding subtypes. All scan functions are declared via `Ref` declarations (Section 3.1.3) using the C++ mechanism of static members of template classes. Alternatively `Atom` declarations register a null pointer for the scan function of a non-reference type (thus avoiding the overhead of the call of an empty scan function). The purpose of the *init* function also registered by `Ref/Atom` declarations is to set the reference(s) contained in a variable initially to null. Thus at any time all heap references in an RT++ program have a well-defined state.

B Example Programs

[4] This is possible because heap entities may not extend across block boundaries.

B.1 Quicksort

```
// ---------------------------------------------------------------------------
//
// qsort.cc
// Quicksort
//
// $Id: app_qsort.tex,v 1.1 1996/04/01 07:24:21 schreine Exp $
// (C) 1996 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
//
// ---------------------------------------------------------------------------

#include <rt++.h>

Atom(int);
Atom(float);
Ref(Array<float>);
ThreadRes(int);
ThreadArg3(A, int, Array<float>, int, int);

const int MIN = 1000;

// ---------------------------------------------------------------------------
//
// swap two values
//
// ---------------------------------------------------------------------------
static inline void swap(Array<float> a, int x, int y)
{
  float z = a[x];
  a[x] = a[y];
  a[y] = z;
}

// ---------------------------------------------------------------------------
//
// split <a/l/r> by pivot <e> into <a/l/x0> and <a/y0/r>
//
// ---------------------------------------------------------------------------
void split(Array<float> a, int l, int r, float e, int *x0, int *y0)
{
  int x = l;
  int y = r-1;
  do
    {
      while (a[x] < e) x++;
      while (a[y] > e) y--;
      if (x < y)
        {
          swap(a, x, y);
          x++; y--;
        }
      else if (x == y)
        {
          break;
        }
    }
  while (x <= y);
  *x0 = x; *y0 = y+1;
}

// ---------------------------------------------------------------------------
//
```

```
// sort <a/l/r> sequentially
//
// ---------------------------------------------------------------------------
int qs0(Array<float> a, int l, int r)
{
  if (l < r)
    {
      float e = a[(l+r)/2];
      int m0, m1;
      split(a, l, r, e, &m0, &m1);
      qs0(a, l, m0);
      qs0(a, m1, r);
    }
  return(0);
}

// ---------------------------------------------------------------------------
//
// sort <a/l/r> in parallel
//
// ---------------------------------------------------------------------------
int qs(Array<float> a, int l, int r)
{
  if (r-l < MIN) return(qs0(a, l, r));
  float e = a[(l+r)/2];
  int m0, m1;
  split(a, l, r, e, &m0, &m1);
  Thread3 <int, Array<float>, int, int> t0(qs, a, l, m0);
  Thread3 <int, Array<float>, int, int> t1(qs, a, m1, r);
  Wait(t0);
  Wait(t1);
  return(0);
}

// ---------------------------------------------------------------------------
//
// sort <a>
//
// ---------------------------------------------------------------------------
void qsort(Array<float> a)
{
  int N = Length(a);
  qs(a, 0, N);
}

// ---------------------------------------------------------------------------
//
// check sorting of <a>
//
// ---------------------------------------------------------------------------
void check(Array<float> a)
{
  int N = Length(a);
  for (int i = 0; i < N-1; i++)
    {
      if (a[i] > a[i+1]) rt_abort("sorting error");
    }
}

// ---------------------------------------------------------------------------
//
// sorting main function
```

```
//
// ---------------------------------------------------------------------------
int qmain(int argc, char *argv[])
{
  int S = 1000;
  int N = 5;
  for (int i = 0; i < N; i++)
    {
      for (int k = 0; k < 5; k++)
        {
          Array<float> a(S);
          for (int j = 0; j < S; j++)
            {
              a[j] = rand();
            }
          qsort(a);
          check(a);
        }
      S *= 2;
    }
  return(0);
}

void init(RT_Argument *rt_arg, int argc, char *argv[]);

int main(int argc, char *argv[])
{
  RT_Argument rt_arg;
  init(&rt_arg, argc, argv);
  rt_arg.memory = 1000000;
  rt_arg.heap = 500000;
  rt_arg.blocks = 2;
  rt_arg.stack = 15000;
  exit(rt_main(qmain, 1, argv, rt_arg, 0));
}

// ---------------------------------------------------------------------------
// end of qsort.cc
// ---------------------------------------------------------------------------
// Local Variables: ***
// mode: C++ ***
// End: ***
```

B.2 Long Integer Multiplication

```
// ---------------------------------------------------------------------------
//
// mult.cc
// digit-wise integer multiplication with threads
//
// $Id: app_mult.tex,v 1.1 1996/04/01 07:24:21 schreine Exp $
// (C) 1996 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
//
// ---------------------------------------------------------------------------

#include <rt++.h>

typedef char Digit;
typedef Array<Digit> Number;
const int BASE = 10;

Atom(Digit);
Ref(Number);
Ref(Thread<Number>);

ThreadRes(Number);
ThreadArg2(A, Number, Number, Digit);
ThreadArg2(B, Number, Thread<Number>, Thread<Number>);

// ---------------------------------------------------------------------------
//
// check for overflow i.e. decompose <d> into <digit> and <carry>
//
// ---------------------------------------------------------------------------
static inline void over(Digit d, Digit *digit, Digit *carry)
{
  Digit c;
  if (d < BASE)
    {
      c = 0;
    }
  else
    {
      d -= BASE;
      c = 1;
    }
  *digit = d;
  *carry = c;
}

// ---------------------------------------------------------------------------
//
// sum of two numbers
//
// ---------------------------------------------------------------------------
Number sum(Thread<Number> ta, Thread<Number> tb)
{
  Number a = Wait(ta);
  Number b = Wait(tb);
  int la = Length(a);
  int lb = Length(b);
  int min = (la < lb) ? la : lb;
  int max = (la > lb) ? la : lb;
  Number c = (la > lb) ? a : b;
  Number s(max+1);
  Digit carry = 0;
```

```
  for (int i = 0; i < min; i++)
    {
      Digit digit, carry0;
      over(a[i]+b[i]+carry, &digit, &carry0);
      s[i] = digit;
      carry = carry0;
    }
  for (int j = min; j < max; j++)
    {
      Digit digit, carry0;
      over(c[j]+carry, &digit, &carry0);
      s[j] = digit;
      carry = carry0;
    }
  s[max] = carry;
  return(s);
}

// ---------------------------------------------------------------------------
//
// product of number and digit
//
// ---------------------------------------------------------------------------
Number prod(Number a, Digit b)
{
  int la = Length(a);
  Number p(la+1);
  Digit carry = 0;
  for (int i = 0; i < la; i++)
    {
      Digit digit, carry0;
      over(a[i]*b+carry, &digit, &carry0);
      p[i] = digit;
      carry = carry0;
    }
  p[la] = carry;
  return(p);
}

// ---------------------------------------------------------------------------
//
// multiplication of two numbers
//
// ---------------------------------------------------------------------------
Number mult(Number a, Number b)
{
  int lb = Length(b);
  int n = 2*lb-1;
  List< Thread<Number> > list;
  for (int i = 0; i < lb; i++)
    {
      Thread2<Number, Number, Digit> t(prod, a, b[i]);
      list.cons(t, list);
    }
  for (int j = 0; j < n-2 ; j += 2)
    {
      Thread<Number> ta(Head(list)); list.tail();
      Thread<Number> tb(Head(list)); list.tail();
      Thread2<Number, Thread<Number>, Thread<Number> > t(sum, ta, tb);
      list.cons(t, list);
    }
  return(Wait(Head(list)));
```

```
}

// ---------------------------------------------------------------------------
//
// main program
//
// ---------------------------------------------------------------------------
int multmain(int argc, char *argv[])
{
  for (int i = 0; i < 10; i++)
    {
      int la = 1+rand()%5000;
      int lb = 1+rand()%50;
      Number a(la);
      Number b(lb);
      for (int j = 0; j < la; j++)
        a[j] = rand()%BASE;
      for (int k = 0; k < lb; k++)
        b[k] = rand()%BASE;
      Number c = mult(a, b);
    }
  return(0);
}

void init(RT_Argument *rt_arg, int argc, char *argv[]);

int main(int argc, char *argv[])
{
  RT_Argument rt_arg;
  init(&rt_arg, argc, argv);
  rt_arg.stack = 15000;
  exit(rt_main(multmain, 1, argv, rt_arg, 0));
}

// ---------------------------------------------------------------------------
// end of mult.cc
// ---------------------------------------------------------------------------
// Local Variables: ***
// mode: C++ ***
// End: ***
```

B.3 Long Integer Multiplication (Bag Version)

```
// ---------------------------------------------------------------------------
//
// mult2.cc
// digit-wise integer multiplication with thread bags
//
// $Id: app_mult2.tex,v 1.1 1996/04/01 07:24:21 schreine Exp $
// (C) 1996 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
//
// ---------------------------------------------------------------------------

#include <rt++.h>

typedef char Digit;
typedef Array<Digit> Number;
const int BASE = 10;

Atom(Digit);
Ref(Number);
ThreadBagRes(Number);

// ---------------------------------------------------------------------------
//
// check for overflow i.e. decompose <d> into <digit> and <carry>
//
// ---------------------------------------------------------------------------
static inline void over(Digit d, Digit *digit, Digit *carry)
{
  Digit c;
  if (d < BASE)
    {
      c = 0;
    }
  else
    {
      d -= BASE;
      c = 1;
    }
  *digit = d;
  *carry = c;
}

// ---------------------------------------------------------------------------
//
// sum of two numbers
//
// ---------------------------------------------------------------------------
Number sum(Number a, Number b)
{
  int la = Length(a);
  int lb = Length(b);
  int min = (la < lb) ? la : lb;
  int max = (la > lb) ? la : lb;
  Number c = (la > lb) ? a : b;
  Number s(max+1);
  Digit carry = 0;
  for (int i = 0; i < min; i++)
    {
      Digit digit, carry0;
      over(a[i]+b[i]+carry, &digit, &carry0);
      s[i] = digit;
      carry = carry0;
```

```
    }
  for (int j = min; j < max; j++)
    {
      Digit digit, carry0;
      over(c[j]+carry, &digit, &carry0);
      s[j] = digit;
      carry = carry0;
    }
  s[max] = carry;
  return(s);
}

// ---------------------------------------------------------------------------
//
// product of number and digit
//
// ---------------------------------------------------------------------------
Number prod(Number a, Digit b)
{
  int la = Length(a);
  Number p(la+1);
  Digit carry = 0;
  for (int i = 0; i < la; i++)
    {
      Digit digit, carry0;
      over(a[i]*b+carry, &digit, &carry0);
      p[i] = digit;
      carry = carry0;
    }
  p[la] = carry;
  return(p);
}

// ---------------------------------------------------------------------------
//
// multiplication of two numbers
//
// ---------------------------------------------------------------------------
Number mult(Number a, Number b)
{
  int lb = Length(b);
  int n = 2*lb-1;
  ThreadBag<Number> bag(n);
  for (int i = 0; i < lb; i++)
    {
      ThreadBag2<Number, Number, Digit>(bag, prod, a, b[i]);
    }
  for (int j = 0; j < n-2 ; j += 2)
    {
      Number a = Wait(bag, j);
      Number b = Wait(bag, j+1);
      ThreadBag2<Number, Number, Number>(bag, sum, a, b);
    }
  return(Wait(bag, n-1));
}

// ---------------------------------------------------------------------------
//
// main program
//
// ---------------------------------------------------------------------------
int multmain(int argc, char *argv[])
```

```
{
  for (int i = 0; i < 10; i++)
    {
      int la = 1+rand()%5000;
      int lb = 1+rand()%50;
      Number a(la);
      Number b(lb);
      for (int j = 0; j < la; j++)
        a[j] = rand()%BASE;
      for (int k = 0; k < lb; k++)
        b[k] = rand()%BASE;
      Number c = mult(a, b);
    }
  return(0);
}

void init(RT_Argument *rt_arg, int argc, char *argv[]);

int main(int argc, char *argv[])
{
  RT_Argument rt_arg;
  init(&rt_arg, argc, argv);
  exit(rt_main(multmain, 1, argv, rt_arg, 0));
}

// ---------------------------------------------------------------------------
// end of mult2.cc
// ---------------------------------------------------------------------------
// Local Variables: ***
// mode: C++ ***
// End: ***
```

B.4 Long Integer Multiplication (Second Bag Version)

```
// ---------------------------------------------------------------------------
//
// mult3.cc
// digit-wise integer multiplication with bags and no over-synchronization
//
// $Id: app_mult3.tex,v 1.1 1996/04/01 07:24:21 schreine Exp $
// (C) 1996 Wolfgang Schreiner <Wolfgang.Schreiner@risc.uni-linz.ac.at>
//
// ---------------------------------------------------------------------------

#include <rt++.h>

typedef char Digit;
typedef Array<Digit> Number;
const int BASE = 10;

Atom(Digit);
Ref(Number);
Atom(int);
Ref(ThreadBag<Number>);
ThreadBagRes(Number);

// ---------------------------------------------------------------------------
//
// check for overflow i.e. decompose <d> into <digit> and <carry>
//
// ---------------------------------------------------------------------------
static inline void over(Digit d, Digit *digit, Digit *carry)
{
  Digit c;
  if (d < BASE)
    {
      c = 0;
    }
  else
    {
      d -= BASE;
      c = 1;
    }
  *digit = d;
  *carry = c;
}

// ---------------------------------------------------------------------------
//
// sum of two numbers
//
// ---------------------------------------------------------------------------
Number sum(ThreadBag<Number> bag, int k)
{
  Number a = Wait(bag, k);
  Number b = Wait(bag, k+1);
  int la = Length(a);
  int lb = Length(b);
  int min = (la < lb) ? la : lb;
  int max = (la > lb) ? la : lb;
  Number c = (la > lb) ? a : b;
  Number s(max+1);
  Digit carry = 0;
  for (int i = 0; i < min; i++)
    {
```

```
      Digit digit, carry0;
      over(a[i]+b[i]+carry, &digit, &carry0);
      s[i] = digit;
      carry = carry0;
    }
  for (int j = min; j < max; j++)
    {
      Digit digit, carry0;
      over(c[j]+carry, &digit, &carry0);
      s[j] = digit;
      carry = carry0;
    }
  s[max] = carry;
  return(s);
}

// ---------------------------------------------------------------------------
//
// product of number and digit
//
// ---------------------------------------------------------------------------
Number prod(Number a, Digit b)
{
  int la = Length(a);
  Number p(la+1);
  Digit carry = 0;
  for (int i = 0; i < la; i++)
    {
      Digit digit, carry0;
      over(a[i]*b+carry, &digit, &carry0);
      p[i] = digit;
      carry = carry0;
    }
  p[la] = carry;
  return(p);
}

// ---------------------------------------------------------------------------
//
// multiplication of two numbers
//
// ---------------------------------------------------------------------------
Number mult(Number a, Number b)
{
  int lb = Length(b);
  int n = 2*lb-1;
  ThreadBag<Number> bag(n);
  for (int i = 0; i < lb; i++)
    {
      ThreadBag2<Number, Number, Digit>(bag, prod, a, b[i]);
    }
  for (int j = 0; j < n-2 ; j += 2)
    {
      ThreadBag2<Number, ThreadBag<Number>, int>(bag, sum, bag, j);
    }
  return(Wait(bag, n-1));
}

// ---------------------------------------------------------------------------
//
// main program
//
```

```
// ---------------------------------------------------------------------------
int multmain(int argc, char *argv[])
{
  for (int i = 0; i < 10; i++)
    {
      int la = 1+rand()%5000;
      int lb = 1+rand()%50;
      Number a(la);
      Number b(lb);
      for (int j = 0; j < la; j++)
        a[j] = rand()%BASE;
      for (int k = 0; k < lb; k++)
        b[k] = rand()%BASE;
      Number c = mult(a, b);
    }
  return(0);
}

void init(RT_Argument *rt_arg, int argc, char *argv[]);

int main(int argc, char *argv[])
{
  RT_Argument rt_arg;
  init(&rt_arg, argc, argv);
  exit(rt_main(multmain, 1, argv, rt_arg, 0));
}

// ---------------------------------------------------------------------------
// end of mult3.cc
// ---------------------------------------------------------------------------
// Local Variables: ***
// mode: C++ ***
// End: ***
```