HPC-GAP --- Reference Manual

Release 4.15.1, 2025-10-18

The GAP Group
Reimer Behrends
Vladimir Janjic

The GAP Group Email: support@gap-system.org
Homepage: https://www.gap-system.org

Reimer Behrends Email: behrends@gmail.com

Vladimir Janjic Email: vj32@st-andrews.ac.uk

Copyright

Copyright © (1987-2025) for the core part of the GAP system by the GAP Group.

Most parts of this distribution, including the core part of the GAP system are distributed under the terms of the GNU General Public License, see `https://www.gnu.org/licenses/gpl.html` or the file `GPL` in the `etc` directory of the GAP installation.

More detailed information about copyright and licenses of parts of this distribution can be found in (**Reference: Copyright and License**).

GAP is developed over a long time and has many authors and contributors. More detailed information can be found in (**Reference: Authors and Maintainers**).

Contents

1 Tasks **5**
- 1.1 Overview . . . 5
- 1.2 Running tasks . . . 6
- 1.3 Information about tasks . . . 9
- 1.4 Cancelling tasks . . . 10
- 1.5 Conditions . . . 11
- 1.6 Milestones . . . 11

2 Variables in HPC-GAP **13**
- 2.1 Global variables . . . 13
- 2.2 Thread-local variables . . . 13

3 How HPC-GAP organizes shared memory: Regions **15**
- 3.1 Thread-local regions . . . 15
- 3.2 Shared regions . . . 15
- 3.3 Ordering of shared regions . . . 15
- 3.4 The public region . . . 16
- 3.5 The read-only region . . . 16
- 3.6 Migrating objects between regions . . . 16
- 3.7 Region names . . . 17
- 3.8 Controlling access to regions . . . 17
- 3.9 Functions relating to regions . . . 17
- 3.10 Atomic functions . . . 27
- 3.11 Write-once functionality . . . 27

4 Console User Interface **30**
- 4.1 Console UI commands . . . 30
- 4.2 GAP functions to access the Shell UI . . . 33

5 Atomic objects **35**
- 5.1 Atomic lists . . . 35
- 5.2 Atomic records and component objects . . . 37
- 5.3 Replacement policy functions . . . 38
- 5.4 Thread-local records . . . 39

6 Thread functions **41**
- 6.1 Thread functions . . . 41

7 Channels **44**
7.1 Channels 44

8 Semaphores **50**
8.1 Semaphores 50

9 Synchronization variables **52**
9.1 Synchronization variables 52

10 Serialization support **54**
10.1 Serialization support 54

11 Low-level functionality **57**
11.1 Explicit lock and unlock primitives 57
11.2 Hash locks 58
11.3 Migration to the public region 59
11.4 Memory barriers 59
11.5 Object manipulation 60

Index **62**

Chapter 1

Tasks

1.1 Overview

Tasks provide mid- to high-level functionality for programmers to describe asynchronous workflows. A task is an asynchronously or synchronously executing job; functions exist to create tasks that are executed concurrently, on demand, or in the current thread; to wait for their completion, check their status, and retrieve any results.

Here is a simple example of sorting a list in the background:

Example

```
gap> task := RunTask(x -> SortedList(x), [3,2,1]);;
gap> WaitTask(task);
gap> TaskResult(task);
[ 1, 2, 3 ]
```

`RunTask` (1.2.1) dispatches a task to run in the background; a task is described by a function and zero or more arguments that are passed to `RunTask` (1.2.1). `WaitTask` (1.2.9) waits for the task to complete; and `TaskResult` returns the result of the task.

`TaskResult` (1.2.11) does an implicit `WaitTask` (1.2.9), so the second line above can actually be omitted:

Example

```
gap> task := RunTask(x -> SortedList(x), [3,2,1]);;
gap> TaskResult(task);
[ 1, 2, 3 ]
```

It is simple to run two tasks in parallel. Let's compute the factorial of 10000 by splitting the work between two tasks:

Example

```
gap> task1 := RunTask(Product, [1..5000]);;
gap> task2 := RunTask(Product, [5001..10000]);;
gap> TaskResult(task1) * TaskResult(task2) = Factorial(10000);
true
```

You can use `DelayTask` (1.2.3) to delay executing the task until its result is actually needed.

Example

```
gap> task1 := DelayTask(Product, [1..5000]);;
gap> task2 := DelayTask(Product, [5001..10000]);;
gap> WaitTask(task1, task2);
```

```
gap> TaskResult(task1) * TaskResult(task2) = Factorial(10000);
true
```

Note that WaitTask (1.2.9) is used here to start execution of both tasks; otherwise, task2 would not be started until TaskResult(task1) has been evaluated.

To start execution of a delayed task, you can also use ExecuteTask. This has no effect if a task has already been running.

For convenience, you can also use ImmediateTask (1.2.7) to execute a task synchronously (i.e., the task is started immediately and the call does not return until the task has completed).

Example

```
gap> task := ImmediateTask(x -> SortedList(x), [3,2,1]);;
gap> TaskResult(task);
[ 1, 2, 3 ]
```

This is indistinguishable from calling the function directly, but provides the same interface as normal tasks.

If e.g. you want to call a function only for its side-effects, it can be useful to ignore the result of a task. RunAsyncTask (1.2.4) provides the necessary functionality. Such a task cannot be waited for and its result (if any) is ignored.

Example

```
gap> RunAsyncTask(function() Print("Hello, world!\n"); end);;
gap> !list
--- Thread 0 [0]
--- Thread 5 [5] (pending output)
gap> !5
--- Switching to thread 5
[5] Hello, world!
!0
--- Switching to thread 0
gap>
```

For more information on the multi-threaded user interface, see Chapter 4.

Task arguments are generally copied so that both the task that created them and the task that uses them can access the data concurrently without fear of race conditions. To avoid copying, arguments should be made shared or public (see the relevant parts of section 3.6 on migrating objects between regions); shared and public arguments will not be copied.

HPC-GAP currently has multiple implementations of the task API. To use an alternative implementation to the one documented here, set the environment variable GAP_WORKSTEALING to a non-empty value before starting GAP.

1.2 Running tasks

1.2.1 RunTask

▷ RunTask(*func[, arg1, ..., argn]*) (function)

RunTask prepares a task for execution and starts it. The task will call the function *func* with arguments *arg1* through *argn* (if provided). The return value of *func* is the result of the task. The RunTask call itself returns a task object that can be used by functions that expect a task argument.

1.2.2 ScheduleTask

▷ ScheduleTask(*condition, func[, arg1, ..., argn]*) (function)

ScheduleTask prepares a task for execution, but, unlike RunTask (1.2.1) does not start it until *condition* is met. See on how to construct conditions. Simple examples of conditions are individual tasks, where execution occurs after the task completes, or lists of tasks, where execution occurs after all tasks in the list complete.

Example

```
gap> t1 := RunTask(x->x*x, 3);;
gap> t2 := RunTask(x->x*x, 4);;
gap> t := ScheduleTask([t1, t2], function()
>           return TaskResult(t1) + TaskResult(t2);
>    end);;
gap> TaskResult(t);
25
```

While the above example could also be achieved with RunTask (1.2.1) in lieu of ScheduleTask, since TaskResult (1.2.11) would wait for *t1* and *t2* to complete, the above implementation does not actually start the final task until the others are complete, making it more efficient, since no additional worker thread needs to be occupied.

1.2.3 DelayTask

▷ DelayTask(*func[, arg1, ..., argn]*) (function)

DelayTask works as RunTask (1.2.1), but its start is delayed until it is being waited for (including implicitly by calling TaskResult (1.2.11)).

1.2.4 RunAsyncTask

▷ RunAsyncTask(*func[, arg1, ..., argn]*) (function)

RunAsyncTask creates an asynchronous task. It works like RunTask (1.2.1), except that its result will be ignored.

1.2.5 ScheduleAsyncTask

▷ ScheduleAsyncTask(*condition, func[, arg1, ..., argn]*) (function)

ScheduleAsyncTask creates an asynchronous task. It works like ScheduleTask (1.2.2), except that its result will be ignored.

1.2.6 MakeTaskAsync

▷ MakeTaskAsync(*task*) (function)

MakeTaskAsync turns a synchronous task into an asynchronous task that cannot be waited for and whose result will be ignored.

1.2.7 ImmediateTask

▷ ImmediateTask(*func[, arg1, ..., argn]*) (function)

ImmediateTask executes the task specified by its arguments synchronously, usually within the current thread.

1.2.8 ExecuteTask

▷ ExecuteTask(*task*) (function)

ExecuteTask starts *task* if it is not already running. It has only an effect if its argument is a task returned by DelayTask (1.2.3); otherwise, it is a no-op.

1.2.9 WaitTask

▷ WaitTask(*task1, ..., taskn*) (function)
▷ WaitTask(*condition*) (function)
▷ WaitTasks(*task1, ..., taskn*) (function)

WaitTask waits until *task1* through *taskn* have completed; after that, it returns. Alternatively, a condition can be passed to WaitTask in order to wait until a condition is met. See on how to construct conditions. WaitTasks is an alias for WaitTask.

1.2.10 WaitAnyTask

▷ WaitAnyTask(*task1, ..., taskn*) (function)

The WaitAnyTask function waits for any of its arguments to finish, then returns the number of that task.

Example

```
gap> task1 := DelayTask(x->SortedList(x), [3,2,1]);;
gap> task2 := DelayTask(x->SortedList(x), [6,5,4]);;
gap> which := WaitAnyTask(task1, task2);
2
gap> if which = 1 then
>      Display(TaskResult(task1));Display(TaskResult(task2));
>    else
>      Display(TaskResult(task2));Display(TaskResult(task1));
>    fi;
[ 4, 5, 6 ]
[ 1, 2, 3 ]
```

One can pass a list of tasks to WaitAnyTask as an argument; WaitAnyTask([task1, ..., taskn]) behaves identically to WaitAnyTask(task1, ..., taskn).

1.2.11 TaskResult

▷ TaskResult(*task*) (function)

The TaskResult function returns the result of a task. It implicitly calls WaitTask (1.2.9) if that is necessary. Multiple invocations of TaskResult with the same task argument will not do repeated waits and always return the same value.

If the function executed by *task* encounters an error, TaskResult returns fail. Whether *task* encountered an error can be checked via TaskSuccess (1.3.1). In case of an error, the error message can be retrieved via TaskError (1.3.2).

1.2.12 CullIdleTasks

▷ CullIdleTasks() (function)

This function terminates unused worker threads.

1.3 Information about tasks

1.3.1 TaskSuccess

▷ TaskSuccess(*task*) (function)

TaskSuccess waits for *task* and returns true if the it finished without encountering an error. Otherwise the function returns false.

1.3.2 TaskError

▷ TaskError(*task*) (function)

TaskError waits for *task* and returns its error message, if it encountered an error. If it did not encounter an error, the function returns fail.

1.3.3 CurrentTask

▷ CurrentTask() (function)

The CurrentTask returns the currently running task.

1.3.4 RunningTasks

▷ RunningTasks() (function)

This function returns the number of currently running tasks. Note that it is only an approximation and can change as new tasks are being started by other threads.

1.3.5 TaskStarted

▷ TaskStarted(*task*) (function)

This function returns true if the task has started executing (i.e., for any non-delayed task), false otherwise.

1.3.6 TaskFinished

▷ `TaskFinished(`*task*`)` (function)

This function returns true if the task has finished executing and its result is available, false otherwise.

1.3.7 TaskIsAsync

▷ `TaskIsAsync(`*task*`)` (function)

This function returns true if the task is asynchronous, true otherwise.

1.4 Cancelling tasks

HPC-GAP uses a cooperative model for task cancellation. A programmer can request the cancellation of another task, but it is up to that other task to actually terminate itself. The tasks library has functions to request cancellation, to test for the cancellation state of a task, and to perform actions in response to cancellation requests.

1.4.1 CancelTask

▷ `CancelTask(`*task*`)` (function)

`CancelTask` submits a request that *task* is to be cancelled.

1.4.2 TaskCancellationRequested

▷ `TaskCancellationRequested(`*task*`)` (function)

`TaskCancellationRequested` returns true if `CancelTask` (1.4.1) has been called for *task*, false otherwise.

1.4.3 OnTaskCancellation

▷ `OnTaskCancellation(`*exit_func*`)` (function)

`OnTaskCancellation` tests if cancellation for the current task has been requested. If so, then *exit_func* will be called (as a parameterless function) and the current task will be aborted. The result of the current task will be the value of *exit_func()*.

Example

```
gap> task := RunTask(function()
>      while true do
>        OnTaskCancellation(function() return 314; end);
>      od;
>    end);;
gap> CancelTask(task);
gap> TaskResult(task);
314
```

1.4.4 OnTaskCancellationReturn

▷ `OnTaskCancellationReturn(`*`value`*`)` (function)

`OnTaskCancellationReturn` is a convenience function that does the same as: `OnTaskCancellation(function() return value; end);`

1.5 Conditions

`ScheduleTask` (1.2.2) and `WaitTask` (1.2.9) can be made to wait on more complex conditions than just tasks. A condition is either a milestone, a task, or a list of milestones and tasks. `ScheduleTask` (1.2.2) starts its task and `WaitTask` (1.2.9) returns when the condition has been met. A condition represented by a task is met when the task has completed. A condition represented by a milestone is met when the milestone has been achieved (see below). A condition represented by a list is met when all conditions in the list have been met.

1.6 Milestones

Milestones are a way to represent abstract conditions to which multiple tasks can contribute.

1.6.1 NewMilestone

▷ `NewMilestone(`*`[list]`*`)` (function)

The `NewMilestone` function creates a new milestone. Its argument is a list of targets, which must be a list of integers and/or strings. If omitted, the list defaults to `[0]`.

1.6.2 ContributeToMilestone

▷ `ContributeToMilestone(`*`milestone, target`*`)` (function)

The `ContributeToMilestone` milestone function contributes the specified target to the milestone. Once all targets have been contributed to a milestone, it has been achieved.

1.6.3 AchieveMilestone

▷ `AchieveMilestone(`*`milestone`*`)` (function)

The `AchieveMilestone` function allows a program to achieve a milestone in a single step without adding individual targets to it. This is most useful in conjunction with the default value for `NewMilestone` (1.6.1), e.g.

Example

```
gap> m := NewMilestone();;
gap> AchieveMilestone(m);
```

>

1.6.4 IsMilestoneAchieved

▷ IsMilestoneAchieved(*milestone*) (function)

IsMilestoneAchieved tests explicitly if a milestone has been achieved. It returns true on success, false otherwise.

Example

```
gap> m := NewMilestone([1,2]);;
gap> ContributeToMilestone(m, 1);
gap> IsMilestoneAchieved(m);
false
gap> ContributeToMilestone(m, 2);
gap> IsMilestoneAchieved(m);
true
```

Chapter 2

Variables in HPC-GAP

Variables with global scope have revised semantics in HPC-GAP in order to address concurrency issues. The normal semantics of global variables that are only accessed by a single thread remain unaltered.

2.1 Global variables

Global variables in HPC-GAP can be accessed by all threads concurrently without explicit synchronization. Concurrent access is safe, but it is not deterministic. If multiple threads attempt to modify the same global variable simultaneously, the resulting value of the variable is random; it will be one of the values assigned by a thread, but it is impossible to predict with certainty which specific one will be assigned.

2.2 Thread-local variables

HPC-GAP supports the notion of thread-local variables. Thread-local variables are (after being declared as such) accessed and modified like global variables. However, unlike global variables, each thread can assign a distinct value to a thread-local variable.

Example

```
gap> MakeThreadLocal("x");
gap> x := 1;;
gap> WaitTask(RunTask(function() x := 2; end));
gap> x;
1
```

As can be seen here, the assignment to `x` in a separate thread does not overwrite the value of `x` in the main thread.

2.2.1 MakeThreadLocal

▷ `MakeThreadLocal(`*name*`)` (function)

`MakeThreadLocal` makes the variable described by the string *name* a thread-local variable. It normally does not give it an initial value; either explicit per-thread assignment or a call to

BindThreadLocal (2.2.2) or BindThreadLocalConstructor (2.2.3) to provide a default value is necessary.

If a global variable with the same name exists and is bound at the time of the call, its value will be used as the default value as though BindThreadLocal (2.2.2) had been called with that value as its second argument.

2.2.2 BindThreadLocal

▷ BindThreadLocal(*name*, *obj*) (function)

BindThreadLocal gives the thread-local variable described by the string *name* the default value *obj*. The first time the thread-local variable is accessed in a thread thereafter, it will yield *obj* as its value if it hasn't been assigned a specific value yet.

2.2.3 BindThreadLocalConstructor

▷ BindThreadLocalConstructor(*name*, *func*) (function)

BindThreadLocal (2.2.2) gives the thread-local variable described by the string *name* the constructor *func*. The first time the thread-local variable is accessed in a thread thereafter, it will yield *func()* as its value if it hasn't been assigned a specific value yet.

2.2.4 ThreadVar

▷ ThreadVar (global variable)

All thread-local variables are stored in the thread-local record ThreadVar. Thus, if x is a thread-local variable, using ThreadVar.x is the same as using x.

Chapter 3

How HPC-GAP organizes shared memory: Regions

HPC-GAP allows multiple threads to access data shared between them; to avoid common concurrency errors, such as race conditions, it partitions GAP objects into regions. Access to regions is regulated so that no two threads can modify objects in the same region at the same time and so that objects that are being read by one thread cannot concurrently be modified by another.

3.1 Thread-local regions

Each thread has an associated thread-local region. When a thread implicitly or explicitly creates a new object, that object initially belongs to the thread's thread-local region.

Only the thread can read or modify objects in its thread-local region. For other threads to access an object, that object has to be migrated into a different region first.

3.2 Shared regions

Shared regions are explicitly created through the `ShareObj` (3.9.9) and `ShareSingleObj` (3.9.15) primitives (see below). Multiple threads can access them concurrently, but accessing them requires that a thread uses an `atomic` statement to acquire a read or write lock beforehand.

See the section on `atomic` statements (3.9.43) for details.

3.3 Ordering of shared regions

Shared regions are by default ordered; each shared region has an associated numeric precedence level. Regions can generally only be locked in order of descending precedence. The purpose of this mechanism is to avoid accidental deadlocks.

The ordering requirement can be overridden in two ways: regions with a negative precedence are excluded from it. This exception should be used with care, as it can lead to deadlocks.

Alternatively, two or more regions can be locked simultaneously via the `atomic` statement. In this case, the ordering of these regions relative to each other can be arbitrary.

3.4 The public region

A special public region contains objects that only permit atomic operations. These include, in particular, all immutable objects (immutable in the sense that their in-memory representation cannot change).

All threads can access objects in the public region at all times without needing to acquire a read- or write-lock beforehand.

3.5 The read-only region

The read-only region is another special region that contains objects that are only meant to be read; attempting to modify an object in that region will result in a runtime error. To obtain a modifiable copy of such an object, the `CopyRegion` (3.9.29) primitive can be used.

3.6 Migrating objects between regions

Objects can be migrated between regions using a number of functions. In order to migrate an object, the current thread must have exclusive access to that object; the object must be in its thread-local region or it must be in a shared region for which the current thread holds a write lock.

The `ShareObj` (3.9.9) and `ShareSingleObj` (3.9.15) functions create a new shared region and migrate their respective argument to that region; `ShareObj` will also migrate all subobjects that are within the same region, while `ShareSingleObj` will leave the subobjects unaffected.

The `MigrateObj` (3.9.21) and `MigrateSingleObj` (3.9.22) functions migrate objects to an existing region. The first argument of either function is the object to be migrated; the second is either a region (as returned by the `RegionOf` (3.9.7) function) or an object whose containing region the first argument is to be migrated to.

The current thread needs exclusive access to the target region (denoted by the second argument) for the operation to succeed. If successful, the first argument will be in the same region as the second argument afterwards. In the case of `MigrateObj` (3.9.21), all subobjects within the same region as the first argument will also be migrated to the target region.

Finally, `AdoptObj` (3.9.26) and `AdoptSingleObj` (3.9.27) are special cases of `MigrateObj` (3.9.21) and `MigrateSingleObj` (3.9.22), where the target region is the thread-local region of the current thread.

To migrate objects to the read-only region, one can use `MakeReadOnlyObj` (3.9.35) and `MakeReadOnlySingleObj` (3.9.36). The first migrates its argument and all its subjobjects that are within the same region to the read-only region; the second migrates only the argument itself, but not its subobjects.

It is generally not possible to migrate objects explicitly to the public region; only objects with purely atomic operations can be made public and that is done automatically when they are created.

The exception are immutable objects. When `MakeImmutable` (**Reference: MakeImmutable**) is used, its argument is automatically moved to the public region.

Example

```
gap> RegionOf(MakeImmutable([1,2,3]));
<public region>
```

3.7 Region names

Regions can be given names, either explicitly via SetRegionName (3.9.38) or when they are created via ShareObj (3.9.9) and ShareSingleObj (3.9.15). Thread-local regions, the public, and the read-only region are given names by default.

Multiple regions can have the same name.

3.8 Controlling access to regions

If either GAP code or a kernel primitive attempts to access an object that it is not allowed to access according to these semantics, either a "write guard error" (for a failed write access) or a "read guard error" (for a failed read access) will be raised. The global variable LastInaccessible will contain the object that caused such an error.

One exception is that threads can modify objects in regions that they have only read access (but not write access) to using write-once functions - see section 3.11.

To inspect objects whose contents lie in other regions (and therefore cannot be displayed by PrintObj (**Reference: PrintObj**) or ViewObj (**Reference: ViewObj**), the functions ViewShared (3.9.41) and UNSAFE_VIEW (3.9.42) can be used.

3.9 Functions relating to regions

3.9.1 NewRegion

▷ NewRegion(*[name,]prec*) (function)

The function NewRegion creates a new shared region. If the optional argument *name* is provided, then the name of the new region will be set to *name*.

Example

```
gap> NewRegion("example region");
<region: example region>
```

NewRegion will create a region with a high precedence level. It is intended to be called by user code. The exact precedence level can be adjusted with *prec*, which must be an integer in the range [-1000..1000]; *prec* will be added to the normal precedence level.

3.9.2 NewLibraryRegion

▷ NewLibraryRegion(*[name,]prec*) (function)

NewLibraryRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is below that of NewRegion (3.9.1). It is intended to be used by user libraries and GAP packages.

3.9.3 NewSystemRegion

▷ NewSystemRegion(*[name,]prec*) (function)

NewSystemRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is below that of NewLibraryRegion (3.9.2). It is intended to be used by the standard GAP library.

3.9.4 NewKernelRegion

▷ NewKernelRegion(*[name,]prec*) (function)

NewKernelRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is below that of NewSystemRegion (3.9.3). It is intended to be used by the GAP kernel, and GAP library code that interacts closely with the kernel.

3.9.5 NewInternalRegion

▷ NewInternalRegion(*[name]*) (function)

NewInternalRegion functions like NewRegion (3.9.1), except that the precedence of the region it creates is the lowest available. It is intended to be used for regions that are self-contained; i.e. no function that uses such a region may lock another region while accessing it. The precedence level of an internal region cannot be adjusted.

3.9.6 NewSpecialRegion

▷ NewSpecialRegion(*[name]*) (function)

NewLibraryRegion (3.9.2) functions like NewRegion (3.9.1), except that the precedence of the region it creates is negative. It is thus exempt from normal ordering and deadlock checks.

3.9.7 RegionOf

▷ RegionOf(*obj*) (function)

Example

```
gap> RegionOf(1/2);
<public region>
gap> RegionOf([1,2,3]);
<region: thread region #0>
gap> RegionOf(ShareObj([1,2,3]));
<region 0x45deaa0>
gap> RegionOf(ShareObj([1,2,3]));
<region 0x45deaa0>
gap> RegionOf(ShareObj([1,2,3], "test region"));
<region: test region>
```

Note that the unique number that each region is identified with is system-specific and can change each time the code is being run. Region objects returned by RegionOf can be compared:

Example

```
gap> RegionOf([1,2,3]) = RegionOf([4,5,6]);
true
```

The result in this example is true because both lists are in the same thread-local region.

3.9.8 RegionPrecedence

▷ RegionPrecedence(*obj*) (function)

RegionPrecedence will return the precedence of the region of *obj*.

Example

```
gap> RegionPrecedence(NewRegion("Test"));
30000
gap> RegionPrecedence(NewRegion("Test2", 1));
30001
gap> RegionPrecedence(NewLibraryRegion("LibTest", -1));
19999
```

3.9.9 ShareObj

▷ ShareObj(*obj[[, name], prec]*) (function)

The ShareObj function creates a new shared region and migrates the object and all its subobjects to that region. If the optional argument *name* is provided, then the name of the new region is set to *name*.

ShareObj will create a region with a high precedence level. It is intended to be called by user code. The actual precedence level can be adjusted by the optional *prec* argument in the same way as for NewRegion (3.9.1).

3.9.10 ShareLibraryObj

▷ ShareLibraryObj(*obj[[, name], prec]*) (function)

ShareLibraryObj functions like ShareObj (3.9.9), except that the precedence of the region it creates is below that of ShareObj (3.9.9). It is intended to be used by user libraries and GAP packages.

3.9.11 ShareSystemObj

▷ ShareSystemObj(*obj[[, name], prec]*) (function)

ShareSystemObj functions like ShareObj (3.9.9), except that the precedence of the region it creates is below that of ShareLibraryObj (3.9.10). It is intended to be used by the standard GAP library.

3.9.12 ShareKernelObj

▷ ShareKernelObj(*obj[[, name], prec]*) (function)

ShareKernelObj functions like ShareObj (3.9.9), except that the precedence of the region it creates is below that of ShareSystemObj (3.9.11). It is intended to be used by the GAP kernel, and GAP library code that interacts closely with the kernel.

3.9.13 ShareInternalObj

▷ ShareInternalObj(*obj[, name]*) (function)

ShareInternalObj functions like ShareObj (3.9.9), except that the precedence of the region it creates is the lowest available. It is intended to be used for regions that are self-contained; i.e. no function that uses such a region may lock another region while accessing it.

3.9.14 ShareSpecialObj

▷ ShareSpecialObj(*obj[, name]*) (function)

ShareSpecialObj functions like ShareObj (3.9.9), except that the precedence of the region it creates is negative. It is thus exempt from normal ordering and deadlock checks.

3.9.15 ShareSingleObj

▷ ShareSingleObj(*obj[[, name], prec]*) (function)

The ShareSingleObj function creates a new shared region and migrates the object, but not its subobjects, to that region. If the optional argument *name* is provided, then the name of the new region is set to *name*.

Example

```
gap> m := [ [1, 2], [3, 4] ];;
gap> ShareSingleObj(m);;
gap> atomic readonly m do
>      Display([ IsShared(m), IsShared(m[1]), IsShared(m[2]) ]);
>    od;
[ true, false, false ]
```

ShareSingleObj will create a region with a high precedence level. It is intended to be called by user code. The actual precedence level can be adjusted by the optional *prec* argument in the same way as for NewRegion (3.9.1).

3.9.16 ShareSingleLibraryObj

▷ ShareSingleLibraryObj(*obj[[, name], prec]*) (function)

ShareSingleLibraryObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is below that of ShareSingleObj (3.9.15). It is intended to be used by user libraries and GAP packages.

3.9.17 ShareSingleSystemObj

▷ ShareSingleSystemObj(*obj[[, name], prec]*) (function)

ShareSingleSystemObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is below that of ShareSingleLibraryObj (3.9.16). It is intended to be used by the standard GAP library.

3.9.18 ShareSingleKernelObj

▷ ShareSingleKernelObj(*obj[[, name], prec]*) (function)

ShareSingleKernelObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is below that of ShareSingleSystemObj (3.9.17). It is intended to be used by the GAP kernel, and GAP library code that interacts closely with the kernel.

3.9.19 ShareSingleInternalObj

▷ ShareSingleInternalObj(*obj[, name]*) (function)

ShareSingleInternalObj functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is the lowest available. It is intended to be used for regions that are self-contained; i.e. no function that uses such a region may lock another region while accessing it.

3.9.20 ShareSingleSpecialObj

▷ ShareSingleSpecialObj(*obj[, name]*) (function)

ShareSingleLibraryObj (3.9.16) functions like ShareSingleObj (3.9.15), except that the precedence of the region it creates is negative. It is thus exempt from normal ordering and deadlock checks.

3.9.21 MigrateObj

▷ MigrateObj(*obj, target*) (function)

The MigrateObj function migrates *obj* (and all subobjects contained within the same region) to the region denoted by the *target* argument. Here, *target* can either be a region object returned by RegionOf (3.9.7) or a normal gap object. If *target* is a normal gap object, *obj* will be migrated to the region containing target.

For the operation to succeed, the current thread must have exclusive access to the target region and the object being migrated.

3.9.22 MigrateSingleObj

▷ MigrateSingleObj(*obj, target*) (function)

The MigrateSingleObj function works like MigrateObj (3.9.21), except that it does not migrate the subobjects of obj.

3.9.23 LockAndMigrateObj

▷ LockAndMigrateObj(*obj, target*) (function)

The LockAndMigrateObj function works like MigrateObj (3.9.21), except that it will automatically try to acquire a lock for the region containing *target* if it does not have one already.

3.9.24 IncorporateObj

▷ IncorporateObj(*target, index, value*) (function)

The IncorporateObj function allows convenient migration to a shared list or record. If *target* is a list, then IncorporateObj is equivalent to:

Example

```
IncorporateObj := function(target, index, value)
  atomic value do
    target[index] := MigrateObj(value, target)
  od;
end;
```

If *target* is a record, then it is equivalent to:

Example

```
IncorporateObj := function(target, index, value)
  atomic value do
    target.(index) := MigrateObj(value, target)
  od;
end;
```

The intended purpose is the population of a shared list or record with values after its creation. Example:

Example

```
gap> list := ShareObj([]);
gap> atomic list do
>      IncorporateObj(list, 1, [1,2,3]);
>      IncorporateObj(list, 2, [4,5,6]);
>      IncorporateObj(list, 3, [7,8,9]);
>    od;
gap> ViewShared(list);
[ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8, 9 ] ]
```

Using plain assignment would leave the newly created lists in the thread-local region.

3.9.25 AtomicIncorporateObj

▷ AtomicIncorporateObj(*target, index, value*) (function)

AtomicIncorporateObj extends IncorporateObj (3.9.24) by also locking the target. I.e., for a list, it is equivalent to:

Example

```
AtomicIncorporateObj := function(target, index, value)
  atomic target, value do
    target[index] := MigrateObj(value, target)
  od;
end;
```

If target is a record, then it is equivalent to:

Example

```
AtomicIncorporateObj := function(target, index, value)
  atomic value do
    target.(index) := MigrateObj(value, target)
  od;
end;
```

3.9.26 AdoptObj

▷ AdoptObj(*obj*) (function)

The AdoptObj function migrates *obj* (and all its subobjects contained within the same region) to the thread's current region. It requires exclusive access to *obj*.

Example

```
gap> l := ShareObj([1,2,3]);;
gap> IsThreadLocal(l);
false
gap> atomic l do AdoptObj(l); od;
gap> IsThreadLocal(l);
true
```

3.9.27 AdoptSingleObj

▷ AdoptSingleObj(*obj*) (function)

The AdoptSingleObj function works like AdoptObj (3.9.26), except that it does not migrate the subobjects of *obj*.

3.9.28 LockAndAdoptObj

▷ LockAndAdoptObj(*obj*) (function)

The LockAndAdoptObj function works like AdoptObj (3.9.26), except that it will attempt acquire an exclusive lock for the region containing *obj* if it does not have one already.

3.9.29 CopyRegion

▷ CopyRegion(*obj*) (function)

The CopyRegion function performs a structural copy of *obj*. The resulting objects will be located in the current thread's thread-local region. The function returns the copy as its result.

Example

```
gap> l := MakeReadOnlyObj([1,2,3]);
[ 1, 2, 3 ]
gap> l2 := CopyRegion(l);
[ 1, 2, 3 ]
gap> RegionOf(l) = RegionOf(l2);
false
```

```
gap> IsIdenticalObj(l, l2);
false
gap> l = l2;
true
```

3.9.30 IsPublic

▷ IsPublic(*obj*) (function)

The IsPublic function returns true if its argument is an object in the public region, false otherwise.

Example

```
gap> IsPublic(1/2);
true
gap> IsPublic([1,2,3]);
false
gap> IsPublic(ShareObj([1,2,3]));
false
gap> IsPublic(MakeImmutable([1,2,3]));
true
```

3.9.31 IsThreadLocal

▷ IsThreadLocal(*obj*) (function)

The IsThreadLocal function returns true if its argument is an object in the current thread's thread-local region, false otherwise.

Example

```
gap> IsThreadLocal([1,2,3]);
true
gap> IsThreadLocal(ShareObj([1,2,3]));
false
gap> IsThreadLocal(1/2);
false
gap> RegionOf(1/2);
<public region>
```

3.9.32 IsShared

▷ IsShared(*obj*) (function)

The IsShared function returns true if its argument is an object in a shared region. Note that if the current thread does not hold a lock on that shared region, another thread can migrate *obj* to a different region before the result is being evaluated; this can lead to race conditions. The function is intended primarily for debugging, not to build actual program logic around.

3.9.33 HaveReadAccess

▷ HaveReadAccess(*obj*) (function)

The HaveReadAccess function returns true if the current thread has read access to *obj*.

Example

```
gap> HaveReadAccess([1,2,3]);
true
gap> l := ShareObj([1,2,3]);;
gap> HaveReadAccess(l);
false
gap> atomic readonly l do t := HaveReadAccess(l); od;; t;
true
```

3.9.34 HaveWriteAccess

▷ HaveWriteAccess(*obj*) (function)

The HaveWriteAccess function returns true if the current thread has write access to *obj*.

Example

```
gap> HaveWriteAccess([1,2,3]);
true
gap> l := ShareObj([1,2,3]);;
gap> HaveWriteAccess(l);
false
gap> atomic readwrite l do t := HaveWriteAccess(l); od;; t;
true
```

3.9.35 MakeReadOnlyObj

▷ MakeReadOnlyObj(*obj*) (function)

The MakeReadOnlyObj function migrates *obj* and all its subobjects that are within the same region as *obj* to the read-only region. It returns *obj*.

3.9.36 MakeReadOnlySingleObj

▷ MakeReadOnlySingleObj(*obj*) (function)

The MakeReadOnlySingleObj function migrates *obj*, but not any of its subobjects, to the read-only region. It returns *obj*.

3.9.37 IsReadOnlyObj

▷ IsReadOnlyObj(*obj*) (function)

The IsReadOnlyObj function returns true if *obj* is in the read-only region, false otherwise.

Example

```
gap> IsReadOnlyObj([1,2,3]);
false
gap> IsReadOnlyObj(MakeImmutable([1,2,3]));
false
gap> IsReadOnlyObj(MakeReadOnlyObj([1,2,3]));
true
```

3.9.38 SetRegionName

▷ SetRegionName(*obj, name*) (function)

The SetRegionName function sets the name of the region of *obj* to *name*.

3.9.39 ClearRegionName

▷ ClearRegionName(*obj*) (function)

The ClearRegionName function clears the name of the region of *obj* to *name*.

3.9.40 RegionName

▷ RegionName(*obj*) (function)

The RegionName function returns the name of the region of *obj*. If that region does not have a name, fail will be returned.

3.9.41 ViewShared

▷ ViewShared(*obj*) (function)

The ViewShared function allows the inspection of objects in shared regions. It will try to lock the region and then call ViewObj(obj). If it cannot acquire a lock for the region, it will simply display the normal description of the object.

3.9.42 UNSAFE_VIEW

▷ UNSAFE_VIEW(*obj*) (function)

The UNSAFE_VIEW (3.9.42) function allows the inspection of any object in the system, regardless of whether the current thread has access to the region containing it. It should be used with care: If the object inspected is being modified by another thread concurrently, the resulting behavior is undefined.

Moreover, the function works by temporarily disabling read and write guards for regions, so other threads may corrupt memory rather than producing errors.

It is generally safe to use if all threads but the current one are paused.

3.9.43 The `atomic` statement.

The `atomic` statement ensures exclusive or read-only access to one or more shared regions for statements within its scope. It has the following syntax:

Example

```
atomic ([readwrite|readonly] expr (, expr)* )* do
  statements
od;
```

Each expression is evaluated and the region containing the resulting object is locked with either a read-write or read-only lock, depending on the keyword preceding the expression. If neither the `readwrite` nor the `readonly` keyword was provided, read-write locks are used by default. Examples:

Example

```
gap> l := ShareObj([1,2,3]);;
gap> atomic readwrite l do l[3] := 9; od;
gap> atomic l do l[2] := 4; od;
gap> atomic readonly l do Display(l); od;
[ 1, 4, 9 ]
```

Example

```
gap> l := ShareObj([1,2,3,4,5]);;
gap> l2 := ShareObj([6,7,8]);;
gap> atomic readwrite l, readonly l2 do
>      for i in [1..3] do l[i] := l2[i]; od;
>      l3 := AdoptObj(l);
>    od;
gap> l3;
[ 6, 7, 8, 4, 5 ]
```

Atomic statements must observe region ordering. That means that the highest precedence level of a region locked by an atomic statement must be less than the lowest precedene level of a region that is locked by the same thread at the time the atomic statement is executed.

3.10 Atomic functions

Instead of atomic regions, entire functions can be declared to be atomic. This has the same effect as though the function's body were enclosed in an atomic statement. Function arguments can be declared either `readwrite` or `readonly`; they will be locked in the same way as for a lock statement. If a function argument is preceded by neither `readwrite` nor `readonly`, the corresponding object will not be locked. Example:

Example

```
gap> AddAtomic := atomic function(readwrite list, readonly item)
>      Add(list, item);
>    end;
```

3.11 Write-once functionality

There is an exception to the rule that objects can only be modified if a thread has write access to a region. A limited sets of objects can be modified using the "bind once" family of functions. These allow the modifications of objects to which a thread has read access in a limited fashion.

For reasons of implementation symmetry, these functions can also be used on the atomic versions of these objects.

Implementation note: The functionality is not currently available for component objects.

3.11.1 BindOnce

▷ `BindOnce(`*`obj, index, value`*`)` (function)

`BindOnce` modifies *`obj`*, which can be a positional object, atomic positional object, component object, or atomic component object. It inspects `obj![index]` for the positional versions or `obj!.(index)` for the component versions. If the respective element is not yet bound, *`value`* is assigned to that element. Otherwise, no modification happens. The test and modification occur as one atomic step. The function returns the value of the element; i.e. the old value if the element was bound and *`value`* if it was unbound.

The intent of this function is to allow concurrent initialization of objects, where multiple threads may attempt to set a value concurrently. Only one will succeed; all threads can then use the return value of `BindOnce` as the definitive value of the element. It also allows for the lazy initialization of objects in the read-only region.

The current thread needs to have at least read access to *`obj`*, but does not require write access.

3.11.2 TestBindOnce

▷ `TestBindOnce(`*`obj, index, value`*`)` (function)

`TestBindOnce` works like `BindOnce` (3.11.1), except that it returns `true` if the value could be bound and `false` otherwise.

3.11.3 BindOnceExpr

▷ `BindOnceExpr(`*`obj, index, expr`*`)` (function)

`BindOnceExpr` works like `BindOnce` (3.11.1), except that it evaluates the parameterless function *`expr`* to determine the value. It will only evaluate *`expr`* if the element is not bound.

For positional objects, the implementation works as follows:

Example

```
BindOnceExprPosObj := function(obj, index, expr)
  if not IsBound(obj![index]) then
    return BindOnce(obj, index, expr());
  else
    return obj![index];
  fi;
end;
```

The implementation for component objects works analogously.

The intent is to avoid unnecessary computations if the value is already bound. Note that this cannot be avoided entirely, because `obj![index]` or `obj!.(index)` can be bound while *`expr`* is evaluated, but it can minimize such occurrences.

3.11.4 TestBindOnceExpr

▷ `TestBindOnceExpr(`*obj, index, expr*`)` (function)

`TestBindOnceExpr` works like `BindOnceExpr` (3.11.3), except that it returns `true` if the value could be bound and `false` otherwise.

3.11.5 StrictBindOnce

▷ `StrictBindOnce(`*obj, index, expr*`)` (function)

`StrictBindOnce` works like `BindOnce` (3.11.1), except that it raises an error if the element is already bound. This is intended for cases where a read-only object is initialized, but where another thread trying to initialize it concurrently would be an error.

Chapter 4

Console User Interface

HPC-GAP has a multi-threaded user interface to assist with the development and debugging of concurrent programs. This user interface is enabled by default; to disable it, and use the single-threaded interface, GAP has to be started with the `-S` option.

4.1 Console UI commands

The console user interface provides the user with the option to control threads by commands prefixed with an exclamation mark ("!"). Those commands are listed below.

For ease of use, users only need to type as many letters of each commands so that it can be unambiguously selected. Thus, the shell will recognize `!l` as an abbreviation for `!list`.

4.1.1 !shell [name]

Starts a new shell thread and switches to it. Optionally, a name for the thread can be provided.

Example

```
gap> !shell
--- Switching to thread 4
[4] gap>
```

4.1.2 !fork [name]

Starts a new background shell thread. Optionally, a name for the thread can be provided.

Example

```
gap> !fork
--- Created new thread 5
```

4.1.3 !list

List all current threads that are interacting with the user. This does not list threads created with `CreateThread()` that have not entered a break loop.

Example

```
gap> !list
--- Thread 0 [0]
--- Thread 4 [4]
--- Thread 5 [5] (pending output)
```

4.1.4 !kill id

Terminates the specified thread.

4.1.5 !break id

Makes the specified thread enter a break loop.

4.1.6 !name [id] name

Give the thread with the numerical identifier or name `id` the name `name`.

Example

```
gap> !name 5 test
gap> !list
--- Thread 0 [0]
--- Thread 4 [4]
--- Thread test [5] (pending output)
```

4.1.7 !info id

Provide information about the thread with the numerical identifier or name `id`. *Not yet implemented.*

4.1.8 !hide [id|*]

Hide output from the thread with the numerical identifier or name `id` when it is not the foreground thread. If no thread is specified, make this the default behavior for future threads.

4.1.9 !watch [id|*]

Show output from the thread with the numerical identifier or name `id` even when it is not the foreground thread. If no thread is specified, make this the default behavior for future threads.

4.1.10 !keep num

Keep `num` lines of output from each thread.

4.1.11 !prompt (id|*) string

Set the prompt for the specified thread (or for all newly created threads if `*` was specified) to be `string`. If the string contains the pattern `id`, it is replaced with the numerical id of the thread; if it contains the pattern `name`, it is replaced with the name of the thread; if the thread has no name, the numerical id is displayed instead.

4.1.12 !prefix (id|*) string

Prefix the output from the specified thread (or for all newly created threads if `*` was specified) with `string`. The same substitution rules as for the `!prompt` command apply.

4.1.13 !select id

Make the specified thread the foreground thread.

Example

```
gap> !select 4
gap> !select 4
--- Switching to thread 4
[4] gap>
```

4.1.14 !next

Make the next thread in numerical order the foreground thread.

4.1.15 !previous

Make the previous thread in numerical order the foreground thread.

4.1.16 !replay num [id]

Display the last `num` lines of output of the specified thread. If no thread was specified, display the last `num` lines of the current foreground thread.

4.1.17 !id

`!id` is a shortcut for `!select id`.

4.1.18 !source file

Read commands from file `file`.

4.1.19 !alias shortcut expansion

Create an alias. After defining the alias, `!shortcut 'rest of line'` will be replaced with `!expansion 'rest of line'`.

4.1.20 !unalias shortcut

Removes the specified alias.

4.1.21 !eval expr

Evaluates `expr` as a command.

4.1.22 !run function string

Calls the function with name `function`, passing it the single argument `string` as a GAP string.

4.2 GAP functions to access the Shell UI

There are several functions to access the basic functionality of the shell user interface. Other than `TextUIRegisterCommand` (4.2.1), they can only be called from within a registered command.

Threads can be specified either by their numerical identifier or by their name (as a string). The empty string can be used to specify the current foreground thread.

4.2.1 TextUIRegisterCommand

▷ `TextUIRegisterCommand(`*`name, func`*`)` (function)

Registers the command `!name` with the shell UI. It will call <func> with the rest of the command line passed as a string argument when typed.

4.2.2 TextUIForegroundThread

▷ `TextUIForegroundThread()` (function)

Returns the numerical identifier of the current foreground thread.

4.2.3 TextUIForegroundThreadName

▷ `TextUIForegroundThreadName()` (function)

Returns the name of the current foreground thread or `fail` if the current foreground thread has no name.

4.2.4 TextUISelectThread

▷ `TextUISelectThread(`*`id`*`)` (function)

Makes *`id`* the current foreground thread. Returns `true` or `false` to indicate success.

4.2.5 TextUIOutputHistory

▷ `TextUIOutputHistory(`*`id, count`*`)` (function)

Returns the last *`count`* lines of the thread specified by *`id`* (which can be a numerical identifier or a name). Returns `fail` if there is no such thread.

4.2.6 TextUISetOutputHistoryLength

▷ `TextUISetOutputHistoryLength(`*`length`*`)` (function)

By default, retain *`length`* lines of output history from each thread.

4.2.7 TextUINewSession

▷ `TextUINewSession(`*`foreground, name`*`)` (function)

Creates a new shell thread. Here, *`foreground`* is a boolean variable specifying whether it should be made the new foreground thread and *`name`* is the name of the thread. The empty string can be used to leave the thread without a name.

4.2.8 TextUIRunCommand

▷ `TextUIRunCommand(`*`command`*`)` (function)

Run the command denoted by *`command`* as though a user had typed it. The command must not contain a newline character.

4.2.9 TextUIWritePrompt

▷ `TextUIWritePrompt()` (function)

Display a prompt for the current thread.

Chapter 5

Atomic objects

HPC-GAP provides a number of atomic object types. These can be accessed by multiple threads concurrently without requiring explicit synchronization, but can have non-deterministic behavior for complex operations. Atomic lists are fixed-size lists; they can be assigned to and read from like normal plain lists. Atomic records are atomic versions of plain records. Unlike plain records, though, it is not possible to delete elements from an atomic record. The primary use of atomic lists and records is to facilitate storing the result of idempotent operations and to support certain low-level operations. Atomic lists and records can have three different replacement policies: write-once, strict write-once, and rewritable. The replacement policy determines whether an already assigned element can be changed. The write-once policy allows elements to be assigned only once, with subsequent assignments being ignored; the strict write-once policy allows elements also to be assigned only once, but subsequent assignments will raise an error; the rewritable policy allows elements to be assigned different values repeatedly. The default for new atomic objects is to be rewritable. Thread-local records are variants of plain records that are replicated on a per-thread basis.

5.1 Atomic lists

Atomic lists are created using the `AtomicList` or `FixedAtomicList` functions. After creation, they can be used exactly like any other list, except that atomic lists created with `FixedAtomicList` cannot be resized. Their contents can also be read as normal plain lists using `FromAtomicList`.

Example

```
gap> a := AtomicList([1,2,4]);
<atomic list of size 3>
gap> WaitTask(RunTask(function() a[1] := a[1] + a[2]; end));
gap> a[1];
3
gap> FromAtomicList(a);
[ 3, 2, 4 ]
```

Because multiple threads can read and write the list concurrently without synchronization, the results of modifying the list may be non-deterministic. It is faster to write to fixed atomic lists than to a resizable atomic list.

5.1.1 AtomicList

▷ AtomicList(*list*) (function)
▷ AtomicList(*count, obj*) (function)

AtomicList is used to create a new atomic list. It takes either a plain list as an argument, in which case it will create a new atomic list of the same size, populated by the same elements; or it takes a count and an object argument. In that case, it creates an atomic list with *count* elements, each set to the value of *obj*.

Example

```
gap> al := AtomicList([3, 1, 4]);
<atomic list of size 3>
gap> al[3];
4
gap> al := AtomicList(10, ‘"alpha");
<atomic list of size 10>
gap> al[3];
"alpha"
gap> WaitTask(RunTask(function() al[3] := ‘"beta"; end));
gap> al[3];
"beta"
```

5.1.2 FixedAtomicList

▷ FixedAtomicList(*list*) (function)
▷ FixedAtomicList(*count, obj*) (function)

FixedAtomicList works like AtomicList (5.1.1) except that the resulting list cannot be resized.

5.1.3 MakeFixedAtomicList

▷ MakeFixedAtomicList(*list*) (function)

MakeFixedAtomicList turns a resizable atomic list into a fixed atomic list.

Example

```
gap> a := AtomicList([99]);
<atomic list of size 1>
gap> a[2] := 100;
100
gap> MakeFixedAtomicList(a);
<fixed atomic list of size 2>
gap> a[3] := 101;
Error, Atomic List Element: <pos>=3 is an invalid index for <list>
```

5.1.4 FromAtomicList

▷ FromAtomicList(*atomic_list*) (function)

FromAtomicList returns a plain list containing the same elements as *atomic_list* at the time of the call. Because other threads can write concurrently to that list, the result is not guaranteed to be deterministic.

Example

```
gap> al := AtomicList([10, 20, 30]);;
gap> WaitTask(RunTask(function() al[2] := 40; end));
gap> FromAtomicList(al);
[ 10, 40, 30 ]
```

5.1.5 ATOMIC_ADDITION

▷ ATOMIC_ADDITION(*atomic_list, index, value*) (function)

ATOMIC_ADDITION (5.1.5) is a low-level operation that atomically adds *value* to *atomic_list[index]*. It returns the value of *atomic_list[index]* after the addition has been performed.

Example

```
gap> al := FixedAtomicList([4,5,6]);;
gap> ATOMIC_ADDITION(al, 2, 7);
12
gap> FromAtomicList(al);
[ 4, 12, 6 ]
```

5.1.6 COMPARE_AND_SWAP

▷ COMPARE_AND_SWAP(*atomic_list, index, old, new*) (function)

COMPARE_AND_SWAP (5.1.6) is an atomic operation. It atomically compares *atomic_list[index]* to *old* and, if they are identical, replaces the value (in the same atomic step) with *new*. It returns true if the replacement took place, false otherwise.

The primary use of COMPARE_AND_SWAP (5.1.6) is to implement certain concurrency primitives; most programmers will not need to use it.

5.2 Atomic records and component objects

Atomic records are atomic counterparts to plain records. They support assignment to individual record fields, and conversion to and from plain records.

Assignment semantics can be specified on a per-record basis if the assigned record field is already populated, allowing either an overwrite, keeping the existing value, or raising an error.

It is not possible to unbind atomic record elements.

Like plain records, atomic records can be converted to component objects using Objectify.

5.2.1 AtomicRecord

▷ AtomicRecord(*capacity*) (function)

▷ AtomicRecord(*record*) (function)

`AtomicRecord` is used to create a new atomic record. Its single optional argument is either a positive integer, denoting the intended capacity (i.e., number of elements to be held) of the record, in which case a new empty atomic record with that initial capacity will be created. Alternatively, the caller can provide a plain record with which to initially populate the atomic record.

Example

```
gap> r := AtomicRecord(rec( x := 2 ));
<atomic record 1/2 full>
gap> r.y := 3;
3
gap> TaskResult(RunTask(function() return r.x + r.y; end));
5
gap> [ r.x, r.y ];
[ 2, 3 ]
```

Any atomic record can later grow beyond its initial capacity. There is no limit to the number of elements it can hold other than available memory.

5.2.2 FromAtomicRecord

▷ `FromAtomicRecord(`*record*`)` (function)

`FromAtomicRecord` returns a plain record copy of the atomic record *record*. This copy is shallow; elements of *record* will not also be copied.

Example

```
gap> r := AtomicRecord();;
gap> r.x := 1;; r.y := 2;; r.z := 3;;
gap> FromAtomicRecord(r);
rec( x := 1, y := 2, z := 3 )
```

5.3 Replacement policy functions

There are three functions that set the replacement policy of an atomic object. All three can also be used with plain lists and records, in which case an atomic version of the list or record is first created. This allows programmers to elide `AtomicList` (5.1.1) and `AtomicRecord` (5.2.1) calls when the next step is to change their policy.

5.3.1 MakeWriteOnceAtomic

▷ `MakeWriteOnceAtomic(`*obj*`)` (function)

`MakeWriteOnceAtomic` takes a list, record, atomic list, atomic record, atomic positional object, or atomic component object as its argument. If the argument is a non-atomic list or record, then the function first creates an atomic copy of the argument. The function then changes the replacement policy of the object to write-once: if an element of the object is already bound, then further attempts to assign to it will be ignored.

5.3.2 MakeStrictWriteOnceAtomic

▷ MakeStrictWriteOnceAtomic(*obj*) (function)

MakeStrictWriteOnceAtomic works like MakeWriteOnceAtomic (5.3.1), except that the replacement policy is being changed to being strict write-once: if an element is already bound, then further attempts to assign to it will raise an error.

5.3.3 MakeReadWriteAtomic

▷ MakeReadWriteAtomic(*obj*) (function)

MakeReadWriteAtomic is the inverse of MakeWriteOnceAtomic (5.3.1) and MakeStrictWriteOnceAtomic (5.3.2) in that the replacement policy is being changed to being rewritable: Elements can be replaced even if they are already bound.

5.4 Thread-local records

Thread-local records allow an easy way to have a separate copy of a record for each individual thread that is accessed by the same name in each thread.

Example

```
gap> r := ThreadLocalRecord();; # create new thread-local record
gap> r.x := 99;;
gap> WaitThread( CreateThread( function()
>                              r.x := 100;
>                              Display(r.x);
>                              end ) );
100
gap> r.x;
99
```

As can be seen above, even though r.x is overwritten in the second thread, it does not affect the value of r.x| in the first thread

5.4.1 ThreadLocalRecord

▷ ThreadLocalRecord(*[defaults[, constructors]]*) (function)

ThreadLocalRecord creates a new thread-local record. It accepts up to two initial arguments. The *defaults* argument is a record of default values with which each thread-local copy is initially populated (this happens on demand, so values are not actually read until needed). The second argument is a record of constructors; parameterless functions that return an initial value for the respective element. Constructors are evaluated only once per thread and only if the respective element is accessed without having previously been assigned a value.

Example

```
gap> r := ThreadLocalRecord( rec(x := 99),
>      rec(y := function() return 101; end));;
gap> r.x;
99
```

```
gap> r.y;
101
gap> TaskResult(RunTask(function() return r.x; end));
99
gap> TaskResult(RunTask(function() return r.y; end));
101
```

5.4.2 SetTLDefault

▷ SetTLDefault(*record, name, value*) (function)

SetTLDefault can be used to set the default value of a record field after its creation. Here, *record* is a thread-local record, *name* is the string of the field name, and *value* is the value.

Example

```
gap> r := ThreadLocalRecord();;
gap> SetTLDefault(r, "x", 314);
gap> r.x;
314
gap> TaskResult(RunTask(function() return r.x; end));
314
```

5.4.3 SetTLConstructor

▷ SetTLConstructor(*record, name, func*) (function)

SetTLConstructor can be used to set the constructor of a thread-local record field after its creation, similar to SetTLDefault (5.4.2).

Example

```
gap> r := ThreadLocalRecord();;
gap> SetTLConstructor(r, "x", function() return 2718; end);
gap> r.x;
2718
gap> TaskResult(RunTask(function() return r.x; end));
2718
```

Chapter 6

Thread functions

HPC-GAP has low-level functionality to support explicit creation of threads. In practice, programmers should use higher-level functionality, such as tasks, to describe concurrency. The thread functions described here exist to facilitate the construction of higher level libraries and are not meant to be used directly.

6.1 Thread functions

6.1.1 CreateThread

▷ `CreateThread(`*`func[, arg1, ..., argn]`*`)` (function)

New threads are created with the function `CreateThread`. The thread takes at least one function as its argument that it will call in the newly created thread; it also accepts zero or more parameters that will be passed to that function.

The function returns a thread object describing the thread.

Only a finite number of threads can be active at a time (that limit is system-dependent). To reclaim the resources occupied by one thread, use the `WaitThread` (6.1.2) function.

6.1.2 WaitThread

▷ `WaitThread(`*`threadID`*`)` (function)

The `WaitThread` function waits for the thread identified by *`threadID`* to finish; it does not return any value. When it returns, it returns all resources occupied by the thread it waited for, such as thread-local memory and operating system structures, to the system.

6.1.3 CurrentThread

▷ `CurrentThread()` (function)

The `CurrentThread` function returns the thread object for the current thread.

6.1.4 ThreadID

▷ ThreadID(*thread*) (function)

The ThreadID function returns a numeric thread id for the given thread. The thread id of the main thread is always 0.

Example

```
gap> CurrentThread();
<thread #0: running>
gap> ThreadID(CurrentThread());
0
```

6.1.5 KillThread

▷ KillThread(*thread*) (function)

The KillThread function terminates the given thread. Any region locks that the thread currently holds will be unlocked. The thread can be specified as a thread object or via its numeric id.

The implementation for KillThread is dependent on the interpreter actually executing statements. Threads performing system calls, for example, will not be terminated until the system call returns. Similarly, long-running kernel functions will delay termination until the kernel function returns.

Use of CALL_WITH_CATCH will not prevent a thread from being terminated. If you wish to make sure that catch handlers will be visited, use InterruptThread (6.1.8) instead. KillThread should be used for threads that cannot be controlled anymore in any other way but still eat system resources.

6.1.6 PauseThread

▷ PauseThread(*thread*) (function)

The PauseThread function suspends execution for the given thread. The thread can be specified as a thread object or via its numeric id.

The implementation for PauseThread is dependent on the interpreter actually executing statements. Threads performing system calls, for example, will not pause until the system call returns. Similarly, long-running kernel functions will not pause until the kernel function returns.

While a thread is paused, the thread that initiated the pause can access the paused thread's thread-local region.

Example

```
gap> loop := function() while true do Sleep(1); od; end;;
gap> x := fail;;
gap> th := CreateThread(function() x := [1, 2, 3]; loop(); end);;
gap> PauseThread(th);
gap> x;
[ 1, 2, 3 ]
```

6.1.7 ResumeThread

▷ ResumeThread(*thread*) (function)

The ResumeThread function resumes execution for the given thread that was paused with PauseThread (6.1.6). The thread can be specified as a thread object or via its numeric id.

If the thread isn't paused, ResumeThread is a no-op.

6.1.8 InterruptThread

▷ InterruptThread(*thread, interrupt*) (function)

The InterruptThread function calls an interrupt handler for the given thread. The thread can be specified as a thread object or via its numeric id. The interrupt is specified as an integer between 0 and MAX_INTERRUPT (6.1.11).

An interrupt number of zero (or an interrupt number for which no interrupt handler has been set up with SetInterruptHandler (6.1.9) will cause the thread to enter a break loop. Otherwise, the respective interrupt handler that has been created with SetInterruptHandler (6.1.9) will be called.

The implementation for InterruptThread is dependent on the interpreter actually executing statements. Threads performing system calls, for example, will not call interrupt handlers until the system call returns. Similarly, long-running kernel functions will delay invocation of the interrupt handler until the kernel function returns.

6.1.9 SetInterruptHandler

▷ SetInterruptHandler(*interrupt, handler*) (function)

The SetInterruptHandler function allows the programmer to set up interrupt handlers for the current thread. The interrupt number must be in the range from 1 to MAX_INTERRUPT (6.1.11) (inclusive); the handler must be a parameterless function (or fail to remove a handler).

6.1.10 NewInterruptID

▷ NewInterruptID() (function)

The NewInterruptID function returns a previously unused number (starting at 1). These numbers can be used to globally coordinate interrupt numbers.

Example

```
gap> StopTaskInterrupt := NewInterruptID();
1
gap> SetInterruptHandler(StopTaskInterrupt, StopTaskHandler);
```

6.1.11 MAX_INTERRUPT

▷ MAX_INTERRUPT (global variable)

The global variable MAX_INTERRUPT (6.1.11) is an integer containing the maximum value for the interrupt arguments to InterruptThread (6.1.8) and SetInterruptHandler (6.1.9).

Chapter 7

Channels

7.1 Channels

Channels are FIFO queues that threads can use to coordinate their activities.

7.1.1 CreateChannel

▷ `CreateChannel(`*`[capacity]`*`)` (function)

`CreateChannel` returns a FIFO communication channel that can be used to exchange information between threads. Its optional argument is a capacity (positive integer). If insufficient resources are available to create a channel, it returns -1. If the capacity is not a positive integer, an error will be raised.

If a capacity is not provided, by default the channel can hold an indefinite number of objects. Otherwise, attempts to store objects in the channel beyond its capacity will block.

Example

```
gap> ch1:=CreateChannel();
<channel 0x460339c: 0 elements, 0 waiting>
gap> ch2:=CreateChannel(5);
<channel 0x460324c: 0/5 elements, 0 waiting>
```

7.1.2 SendChannel

▷ `SendChannel(`*`channel, obj`*`)` (function)

`SendChannel` accepts two arguments, a channel object returned by `CreateChannel` (7.1.1), and an arbitrary GAP object. It stores *`obj`* in *`channel`*. If *`channel`* has a finite capacity and is currently full, then `SendChannel` will block until at least one element has been removed from the channel, e.g. using `ReceiveChannel` (7.1.6).

`SendChannel` performs automatic region migration for thread-local objects. If *`obj`* is thread-local for the current thread, it will be migrated (along with all subobjects contained in the same region) to the receiving thread's thread-local data space. In between sending and receiving, *`obj`* cannot be accessed by either thread.

This example demonstrates sending messages across a channel.

Example

```
gap> ch1 := CreateChannel();;
gap> SendChannel(ch1,1);
gap> ch1;
<channel 0x460339c: 1 elements, 0 waiting>
gap> ReceiveChannel(ch1);
1
gap> ch1;
<channel 0x460339c: 0 elements, 0 waiting>
```

`Sleep` in the following example is used to demonstrate blocking.

Example

```
gap> ch2 := CreateChannel(5);;
gap> ch3 := CreateChannel();;
gap> for i in [1..5] do SendChannel(ch2,i); od;
gap> ch2;
<channel 0x460324c: 5/5 elements, 0 waiting>
gap> t:=CreateThread(
> function()
> local x;
> Sleep(10);
> x:=ReceiveChannel(ch2);
> Sleep(10);
> SendChannel(ch3,x);
> Print("Thread finished\n");
> end);;
> SendChannel(ch2,3); # this blocks until the thread reads from ch2
gap> ReceiveChannel(ch3); # the thread is blocked until we read from ch3
1
Thread finished
gap> WaitThread(t);
```

7.1.3 TransmitChannel

▷ `TransmitChannel(`*channel, obj*`)` (function)

`TransmitChannel` is identical to `SendChannel` (7.1.2), except that it does not perform automatic region migration of thread-local objects.

Example

```
gap> ch := CreateChannel(5);;
gap> l := [ 1, 2, 3];;
gap> original_region := RegionOf(l);;
gap> SendChannel(ch, l);
gap> WaitThread(CreateThread(function()
>      local ob; ob := ReceiveChannel(ch);
>      Display(RegionOf(ob) = original_region);
>    end));
false
gap> l := [ 1, 2, 3];;
gap> original_region := RegionOf(l);;
gap> TransmitChannel(ch, l);
```

```
gap> WaitThread(CreateThread(function()
>      local ob; ob := ReceiveChannel(ch);
>      Display(RegionOf(ob) = original_region);
>    end));
true
```

7.1.4 TrySendChannel

▷ TrySendChannel(*channel, obj*) (function)

TrySendChannel is identical to SendChannel (7.1.2), except that it returns if the channel is full instead of blocking. It returns true if the send was successful and false otherwise.

Example

```
gap> ch := CreateChannel(1);;
gap> TrySendChannel(ch, 99);
true
gap> TrySendChannel(ch, 99);
false
```

7.1.5 TryTransmitChannel

▷ TryTransmitChannel(*channel, obj*) (function)

TryTransmitChannel is identical to TrySendChannel (7.1.4), except that it does not perform automatic region migration of thread-local objects.

7.1.6 ReceiveChannel

▷ ReceiveChannel(*channel*) (function)

ReceiveChannel is used to retrieve elements from a channel. If *channel* is empty, the call will block until an element has been added to the channel via SendChannel (7.1.2) or a similar primitive.

See SendChannel (7.1.2) for an example.

7.1.7 TryReceiveChannel

▷ TryReceiveChannel(*channel, default*) (function)

TryReceiveChannel, like ReceiveChannel (7.1.6), attempts to retrieve an object from *channel*. If it does not succeed, however, it will return *default* rather than blocking.

Example

```
gap> ch := CreateChannel();;
gap> SendChannel(ch, 99);
gap> TryReceiveChannel(ch, fail);
99
gap> TryReceiveChannel(ch, fail);
fail
```

7.1.8 MultiSendChannel

▷ MultiSendChannel(*channel, list*) (function)

MultiSendChannel allows the sending of all the objects contained in the list *list* to *channel* as a single operation. The list must be dense and is not modified by the call. The function will send elements starting at index 1 until all elements have been sent. If a channel with finite capacity is full, then the operation will block until all elements can be sent.

The operation is designed to be more efficient than sending all elements individually via SendChannel (7.1.2) by minimizing potentially expensive concurrency operations.

See MultiReceiveChannel (7.1.10) for an example.

7.1.9 TryMultiSendChannel

▷ TryMultiSendChannel(*channel, list*) (function)

TryMultiSendChannel operates like MultiSendChannel (7.1.8), except that it returns rather than blocking if it cannot send any more elements if the channel is full. It returns the number of elements it has sent. If *channel* does not have finite capacity, TryMultiSendChannel will always send all elements in the list.

7.1.10 MultiReceiveChannel

▷ MultiReceiveChannel(*channel, amount*) (function)

MultiReceiveChannel is the receiving counterpart to MultiSendChannel (7.1.8).It will try to receive up to *amount* objects from *channel*. If the channel contains less than *amount* objects, it will return rather than blocking.

The function returns a list of all the objects received.

Example

```
gap> ch:=CreateChannel();;
gap> MultiSendChannel(ch, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
gap> MultiReceiveChannel(ch,7);
[ 1, 2, 3, 4, 5, 6, 7 ]
gap> MultiReceiveChannel(ch,7);
[ 8, 9, 10 ]
gap> MultiReceiveChannel(ch,7);
[  ]
```

7.1.11 ReceiveAnyChannel

▷ ReceiveAnyChannel(*channel_1, ..., channel_n*) (function)
▷ ReceiveAnyChannel(*channel_list*) (function)

ReceiveAnyChannel is a multiplexing variant ofReceiveChannel (7.1.6). It blocks until at least one of the channels provided contains an object. It will then retrieve that object from the channel and return it.

Example

```
gap> ch1 := CreateChannel();;
gap> ch2 := CreateChannel();;
gap> SendChannel(ch2, [1, 2, 3]);;
gap> ReceiveAnyChannel(ch1, ch2);
[ 1, 2, 3 ]
```

7.1.12 ReceiveAnyChannelWithIndex

▷ ReceiveAnyChannelWithIndex(*channel_1, ..., channel_n*) (function)
▷ ReceiveAnyChannelWithIndex(*channel_list*) (function)

ReceiveAnyChannelWithIndex works like ReceiveAnyChannel (7.1.11), except that it returns a list with two elements, the first being the object being received, the second being the number of the channel from which the object has been retrieved.

Example

```
gap> ch1 := CreateChannel();;
gap> ch2 := CreateChannel();;
gap> SendChannel(ch2, [1, 2, 3]);;
gap> ReceiveAnyChannelWithIndex(ch1, ch2);
[ [ 1, 2, 3 ], 2 ]
```

7.1.13 TallyChannel

▷ TallyChannel(*channel*) (function)

TallyChannel returns the number of objects that a channel contains. This number can increase or decrease, as data is sent to or received from this channel. Send operations will only ever increase and receive operations will only ever decrease this count. Thus, if there is only one thread receiving data from the channel, it can use the result as a lower bound for the number of elements that will be available in the channel.

Example

```
gap> ch := CreateChannel();;
gap> SendChannel(ch, 2);
gap> SendChannel(ch, 3);
gap> SendChannel(ch, 5);
gap> TallyChannel(ch);
3
```

7.1.14 InspectChannel

▷ InspectChannel(*channel*) (function)

InspectChannel returns a list of the objects that a channel contains. Note that objects that are not in the shared, public, or read-only region will be temporarily stored in the so-called limbo region while in transit and will be inaccessible through normal means until they have been received.

Example

```
gap> ch := CreateChannel();;
gap> SendChannel(ch, 2);
```

```
gap> SendChannel(ch, 3);
gap> SendChannel(ch, 5);
gap> InspectChannel(ch);
[ 2, 3, 5 ]
```

This function is primarily intended for debugging purposes.

Chapter 8

Semaphores

8.1 Semaphores

Semaphores are synchronized counters; they can also be used to simulate locks.

8.1.1 CreateSemaphore

▷ `CreateSemaphore(`*`[value]`*`)` (function)

The function `CreateSemaphore` takes an optional argument, which defaults to zero. It is the counter with which the semaphore is initialized.

Example

```
gap> sem := CreateSemaphore(1);
<semaphore 0x1108e81c0: count = 1>
```

8.1.2 WaitSemaphore

▷ `WaitSemaphore(`*`sem`*`)` (function)

`WaitSemaphore` receives a previously created semaphore as its argument. If the semaphore's counter is greater than zero, it decrements the counter and returns; if the counter is zero, it waits until another thread increases it via `SignalSemaphore` (8.1.3), then decrements the counter and returns.

Example

```
gap> sem := CreateSemaphore(1);
<semaphore 0x1108e81c0: count = 1>
gap> WaitSemaphore(sem);
gap> sem;
<semaphore 0x1108e81c0: count = 0>
```

8.1.3 SignalSemaphore

▷ `SignalSemaphore(`*`sem`*`)` (function)

`SignalSemaphore` receives a previously created semaphore as its argument. It increments the semaphore's counter and returns.

Example

```
gap> sem := CreateSemaphore(1);
<semaphore 0x1108e81c0: count = 1>
gap> WaitSemaphore(sem);
gap> sem;
<semaphore 0x1108e81c0: count = 0>
gap> SignalSemaphore(sem);
gap> sem;
<semaphore 0x1108e81c0: count = 1>
```

8.1.4 Simulating locks

In order to use semaphores to simulate locks, create a semaphore with an initial value of 1. `WaitSemaphore` (8.1.2) is then equivalent to a lock operation, `SignalSemaphore` (8.1.3) is equivalent to an unlock operation.

Chapter 9

Synchronization variables

9.1 Synchronization variables

Synchronization variables (also often called dataflow variables in the literature) are variables that can be written only once; attempts to read the variable block until it has been written to.

Synchronization variables are created with `CreateSyncVar` (9.1.1), written with `SyncWrite` (9.1.2) and read with `SyncRead` (9.1.3).

Example

```
gap> sv := CreateSyncVar();;
gap> RunAsyncTask(function()
>      Sleep(10);
>      SyncWrite(sv, MakeImmutable([1, 2, 3]));
>    end);;
gap> SyncRead(sv);
[ 1, 2, 3 ]
```

9.1.1 CreateSyncVar

▷ `CreateSyncVar()` (function)

The function `CreateSyncVar` takes no arguments. It returns a new synchronization variable. There is no need to deallocate it; the garbage collector will free the memory and all related resources when it is no longer accessible.

9.1.2 SyncWrite

▷ `SyncWrite(`*`syncvar, obj`*`)` (function)

`SyncWrite` attempts to assign the value *`obj`* to *`syncvar`*. If *`syncvar`* has been previously assigned a value, the call will fail with a runtime error; otherwise, *`obj`* will be assigned to *`syncvar`*.

In order to make sure that the recipient can read the result, the *`obj`* argument should not be a thread-local object; it should be public, read-only, or shared.

9.1.3 SyncRead

▷ `SyncRead(`*`syncvar`*`)` (function)

`SyncRead` reads the value previously assigned to *`syncvar`* with `SyncWrite` (9.1.2). If no value has been assigned yet, it blocks. It returns the assigned value.

Chapter 10

Serialization support

10.1 Serialization support

HPC-GAP has support to serialize most GAP data. While functions in particular cannot be serialized, it is possible to serialize all primitive types (booleans, integers, cyclotomics, permutations, floats, etc.) as well as all lists and records.

Custom serialization support can be written for data objects, positional objects, and component objects; serialization of compressed vectors is already supported by the standard library.

10.1.1 SerializeToNativeString

▷ `SerializeToNativeString(`*obj*`)` (function)

`SerializeToNativeString` takes the object passed as an argument and turns it into a string, from which a copy of the original can be extracted using `DeserializeNativeString` (10.1.2).

10.1.2 DeserializeNativeString

▷ `DeserializeNativeString(`*str*`)` (function)

`DeserializeNativeString` reverts the serialization process.

Example:

Example

```
gap> DeserializeNativeString(SerializeToNativeString([1,2,3]));
[ 1, 2, 3 ]
```

10.1.3 InstallTypeSerializationTag

▷ `InstallTypeSerializationTag(`*type, tag*`)` (function)

`InstallTypeSerializationTag` allows the serialization of data objects, positional objects, and component objects. The value of *tag* must be unique for each type; it can be a string or integer. Non-negative integers are reserved for use by the standard library; users should use negative integers or strings instead.

Objects of such a type are serialized in a straightforward way: During serialization, data objects are converted into byte streams, positional objects into lists, and component objects into records. These objects are then serialized along with their tags; deserialization uses the type corresponding to the tag in conjunction with `Objectify` (**Reference: Objectify**) to reconstruct a copy of the original object.

Note that this functionality may be inadequate for objects that have complex data structures attached that are not meant to be replicated. The following alternative is meant for such objects.

10.1.4 InstallSerializer

▷ `InstallSerializer(`*`description, filters, method`*`)` (function)

The more general `InstallSerializer` allows for arbitrarily complex serialization code. It installs *`method`* as the method to serialize objects matching *`filters`*; *`description`* has the same role as for `InstallMethod` (**Reference: InstallMethod**).

The method must return a plain list matching a specific format. The first element must be a non-negative integer, the second must be a string descriptor that is unique to the serializer; these can then be followed by an arbitrary number of arguments.

As many of the arguments (starting with the third element of the list) as specified by the first element of the list will be converted from their object representation into a serializable representation. Data objects will be converted into untyped data objects, positional objects will be converted into plain lists, and component objects into records. Conversion will not modify the objects in place, but work on copies. The remaining arguments will remain untouched.

Upon deserialization, these arguments will be passed to a function specified by the second element of the list.

Example:

Example

```
InstallSerializer("8-bit vectors", [ Is8BitVectorRep ], function(obj)
  return [1, "Vec8Bit", obj, Q_VEC8BIT(obj), IS_MUTABLE_OBJ(obj)];
end);
```

Here, `obj` will be converted into its underlying representation, while the remaining arguments are left alone. `"Vec8Bit"` is the name that is used to look up the deserializer function.

10.1.5 InstallDeserializer

▷ `InstallDeserializer(`*`descriptor, func`*`)` (function)

The *`descriptor`* value must be the same as the second element of the list returned by the serializer; *`func`* must be a function that takes as many arguments as there were arguments after the second element of that list. For deserialization, this function is invoked and needs to return the deserialized object constructed from the arguments.

Example:

Example

```
InstallDeserializer("Vec8Bit", function(obj, q, mut)
  SET_TYPE_OBJ(obj, TYPE_VEC8BIT(q, mut));
  return obj;
end);
```

Here, the untyped `obj` that was passed to the deserializer needs to be given the correct type, which is calculated from `q` and `mut`.

Chapter 11

Low-level functionality

The functionality described in this section should only be used by experts, and even by those only with caution (especially the parts that relate to the memory model).

Not only is it possible to crash or hang the GAP kernel, it can happen in ways that are very difficult to reproduce, leading to software defects that are discovered only long after deployment of a package and then become difficult to correct.

The performance benefit of using these primitives is generally minimal; while concurrency can induce some overhead, the benefit from micromanaging concurrency in an interpreted language such as GAP is likely to be small.

These low-level primitives exist primarily for the benefit of kernel programmers; it allows them to prototype new kernel functionality in GAP before implementing it in C.

11.1 Explicit lock and unlock primitives

The `LOCK` (11.1.1) operation combined with `UNLOCK` (11.1.3) is a low-level interface for the functionality of the statement.

11.1.1 LOCK

▷ `LOCK(`*`[arg_1, ..., arg_n]`*`)` (function)

`LOCK` takes zero or more pairs of parameters, where each is either an object or a boolean value. If an argument is an object, the region containing it will be locked. If an argument is the boolean value `false`, all subsequent locks will be read locks; if it is the boolean value `true`, all subsequent locks will be write locks. If the first argument is not a boolean value, all locks until the first boolean value will be write locks.

Locks are managed internally as a stack of locked regions; `LOCK` returns an integer indicating a pointer to the top of the stack; this integer is used later by the `UNLOCK` (11.1.3) operation to unlock locks on the stack up to that position. If `LOCK` should fail for some reason, it will return `fail`.

Calling `LOCK` with no parameters returns the current lock stack pointer.

11.1.2 TRYLOCK

▷ `TRYLOCK(`*`[arg_1, ..., arg_n]`*`)` (function)

TRYLOCK works similarly to LOCK (11.1.1). If it cannot acquire all region locks, it returns fail and does not lock any regions. Otherwise, its semantics are identical to LOCK (11.1.1).

11.1.3 UNLOCK

▷ UNLOCK(*stackpos*) (function)

UNLOCK unlocks all regions on the stack at *stackpos* or higher and sets the stack pointer to *stackpos*.

Example

```
gap> l1 := ShareObj([1,2,3]);;
gap> l2 := ShareObj([4,5,6]);;
gap> p := LOCK(l1);
0
gap> LOCK(l2);
1
gap> UNLOCK(p); # unlock both RegionOf(l1) and RegionOf(l2)
gap> LOCK(); # current stack pointer
0
```

11.2 Hash locks

HPC-GAP supports *hash locks*; internally, the kernel maintains a fixed size array of locks; objects are mapped to a lock via hash function. The hash function is based on the object reference, not its contents (except for short integers and finite field elements).

Example

```
gap> l := [ 1, 2, 3];;
gap> f := l -> Sum(l);;
gap> HASH_LOCK(l);   # lock 'l'
gap> f(l);           # do something with 'l'
6
gap> HASH_UNLOCK(l); # unlock 'l'
```

Hash locks should only be used for very short operations, since there is a chance that two concurrently locked objects map to the same hash value, leading to unnecessary contention.

Hash locks are unrelated to the locks used by the atomic statements and the LOCK (11.1.1) and UNLOCK (11.1.3) primitives.

11.2.1 HASH_LOCK

▷ HASH_LOCK(*obj*) (function)

HASH_LOCK (11.2.1) obtains the read-write lock for the hash value associated with obj.

11.2.2 HASH_UNLOCK

▷ HASH_UNLOCK(*obj*) (function)

HASH_UNLOCK (11.2.2) releases the read-write lock for the hash value associated with obj.

11.2.3 HASH_LOCK_SHARED

▷ HASH_LOCK_SHARED(*obj*) (function)

HASH_LOCK_SHARED (11.2.3) obtains the read-only lock for the hash value associated with obj.

11.2.4 HASH_UNLOCK_SHARED

▷ HASH_UNLOCK_SHARED(*obj*) (function)

HASH_UNLOCK_SHARED (11.2.4) releases the read-only lock for the hash value associated with obj.

11.3 Migration to the public region

HPC-GAP allows migration of arbitrary objects to the public region. This functionality is potentially dangerous; for example, if two threads try resize a plain list simultaneously, this can result in memory corruption.

Accordingly, such data should never be accessed except through operations that protect accesses through locks, memory barriers, or other mechanisms.

11.3.1 MAKE_PUBLIC

▷ MAKE_PUBLIC(*obj*) (function)

MAKE_PUBLIC (11.3.1) makes *obj* and all its subobjects members of the public region.

11.3.2 MAKE_PUBLIC_NORECURSE

▷ MAKE_PUBLIC_NORECURSE(*obj*) (function)

MAKE_PUBLIC_NORECURSE (11.3.2) makes *obj*, but not any of its subobjects members of the public region.

11.4 Memory barriers

The memory models of some processors do no guarantee that read and writes reflect accesses to main memory in the same order in which the processor performed them; for example, code may write variable v1 first, and v2 second; but the cache line containing v2 is flushed to main memory first so that other processors see the change to v2 before the change to v1.

Memory barriers can be used to prevent such counter-intuitive reordering of memory accesses.

11.4.1 ORDERED_WRITE

▷ ORDERED_WRITE(*expr*) (function)

The ORDERED_WRITE (11.4.1) function guarantees that all writes that occur prior to its execution or during the evaluation of *expr* become visible to other processors before any of the code executed after.

Example:

Example

```
gap> y:=0;; f := function() y := 1; return 2; end;;
gap> x := ORDERED_WRITE(f());
2
```

Here, the write barrier ensure that the assignment to y that occurs during the call of f() becomes visible to other processors before or at the same time as the assignment to x.

This can also be done differently, with the same semantics:

Example

```
gap> t := f();; # temporary variable
gap> ORDERED_WRITE(0);; # dummy argument
gap> x := t;
2
```

11.4.2 ORDERED_READ

▷ ORDERED_READ(*expr*) (function)

Conversely, the ORDERED_READ (11.4.2) function ensures that reads that occur before its call or during the evaluation of *expr* are not reordered with respects to memory reads occurring after it.

11.5 Object manipulation

There are two new functions to exchange a pair of objects.

11.5.1 SWITCH_OBJ

▷ SWITCH_OBJ(*obj1, obj2*) (function)

SWITCH_OBJ (11.5.1) exchanges its two arguments. All variables currently referencing *obj1* will reference *obj2* instead after the operation completes, and vice versa. Both objects stay within their previous regions.

Example

```
gap> a := [ 1, 2, 3];;
gap> b := [ 4, 5, 6];;
gap> SWITCH_OBJ(a, b);
gap> a;
[ 4, 5, 6 ]
gap> b;
[ 1, 2, 3 ]
```

The function requires exclusive access to both objects, which may necessitate using an atomic statement, e.g.:

Example

```
gap> a := ShareObj([ 1, 2, 3]);;
gap> b := ShareObj([ 4, 5, 6]);;
gap> atomic a, b do SWITCH_OBJ(a, b); od;
gap> atomic readonly a do Display(a); od;
[ 4, 5, 6 ]
gap> atomic readonly b do Display(b); od;
[ 1, 2, 3 ]
```

11.5.2 FORCE_SWITCH_OBJ

▷ FORCE_SWITCH_OBJ(*obj1, obj2*) (function)

FORCE_SWITCH_OBJ (11.5.2) works like SWITCH_OBJ (11.5.1), except that it can also exchange objects in the public region:

Example

```
gap> a := ShareObj([ 1, 2, 3]);;
gap> b := MakeImmutable([ 4, 5, 6]);;
gap> atomic a do FORCE_SWITCH_OBJ(a, b); od;
gap> a;
[ 4, 5, 6 ]
```

This function should be used with extreme caution and only with public objects for which only the current thread has a reference. Otherwise, undefined behavior and crashes can result from other threads accessing the public object concurrently.

Index

AchieveMilestone, 11
AdoptObj, 23
AdoptSingleObj, 23
atomic
    no value, 27
AtomicIncorporateObj, 22
AtomicList, 36
    for a count and an object, 36
AtomicRecord, 37
    for a record, 37
ATOMIC_ADDITION, 37

BindOnce, 28
BindOnceExpr, 28
BindThreadLocal, 14
BindThreadLocalConstructor, 14

CancelTask, 10
ClearRegionName, 26
COMPARE_AND_SWAP, 37
ContributeToMilestone, 11
CopyRegion, 23
CreateChannel, 44
CreateSemaphore, 50
CreateSyncVar, 52
CreateThread, 41
CullIdleTasks, 9
CurrentTask, 9
CurrentThread, 41

DelayTask, 7
DeserializeNativeString, 54

ExecuteTask, 8

FixedAtomicList, 36
    for a count and an object, 36
FORCE_SWITCH_OBJ, 61
FromAtomicList, 36
FromAtomicRecord, 38

HASH_LOCK, 58
HASH_LOCK_SHARED, 59
HASH_UNLOCK, 58
HASH_UNLOCK_SHARED, 59
HaveReadAccess, 25
HaveWriteAccess, 25

ImmediateTask, 8
IncorporateObj, 22
InspectChannel, 48
InstallDeserializer, 55
InstallSerializer, 55
InstallTypeSerializationTag, 54
InterruptThread, 43
IsMilestoneAchieved, 12
IsPublic, 24
IsReadOnlyObj, 25
IsShared, 24
IsThreadLocal, 24

KillThread, 42

LOCK, 57
LockAndAdoptObj, 23
LockAndMigrateObj, 21

MakeFixedAtomicList, 36
MakeReadOnlyObj, 25
MakeReadOnlySingleObj, 25
MakeReadWriteAtomic, 39
MakeStrictWriteOnceAtomic, 39
MakeTaskAsync, 7
MakeThreadLocal, 13
MakeWriteOnceAtomic, 38
MAKE_PUBLIC, 59
MAKE_PUBLIC_NORECURSE, 59
MAX_INTERRUPT, 43
MigrateObj, 21
MigrateSingleObj, 21
MultiReceiveChannel, 47

MultiSendChannel, 47

NewInternalRegion, 18
NewInterruptID, 43
NewKernelRegion, 18
NewLibraryRegion, 17
NewMilestone, 11
NewRegion, 17
NewSpecialRegion, 18
NewSystemRegion, 17

OnTaskCancellation, 10
OnTaskCancellationReturn, 11
ORDERED_READ, 60
ORDERED_WRITE, 59

PauseThread, 42

ReceiveAnyChannel, 47
  for a list of channels, 47
ReceiveAnyChannelWithIndex, 48
  for a list of channels, 48
ReceiveChannel, 46
RegionName, 26
RegionOf, 18
RegionPrecedence, 19
ResumeThread, 42
RunAsyncTask, 7
RunningTasks, 9
RunTask, 6

ScheduleAsyncTask, 7
ScheduleTask, 7
SendChannel, 44
SerializeToNativeString, 54
SetInterruptHandler, 43
SetRegionName, 26
SetTLConstructor, 40
SetTLDefault, 40
ShareInternalObj, 20
ShareKernelObj, 19
ShareLibraryObj, 19
ShareObj, 19
ShareSingleInternalObj, 21
ShareSingleKernelObj, 21
ShareSingleLibraryObj, 20
ShareSingleObj, 20
ShareSingleSpecialObj, 21
ShareSingleSystemObj, 20
ShareSpecialObj, 20
ShareSystemObj, 19
SignalSemaphore, 50
StrictBindOnce, 29
SWITCH_OBJ, 60
SyncRead, 53
SyncWrite, 52

TallyChannel, 48
TaskCancellationRequested, 10
TaskError, 9
TaskFinished, 10
TaskIsAsync, 10
TaskResult, 8
TaskStarted, 9
TaskSuccess, 9
TestBindOnce, 28
TestBindOnceExpr, 29
TextUIForegroundThread, 33
TextUIForegroundThreadName, 33
TextUINewSession, 34
TextUIOutputHistory, 33
TextUIRegisterCommand, 33
TextUIRunCommand, 34
TextUISelectThread, 33
TextUISetOutputHistoryLength, 33
TextUIWritePrompt, 34
ThreadID, 42
ThreadLocalRecord, 39
ThreadVar, 14
TransmitChannel, 45
TRYLOCK, 57
TryMultiSendChannel, 47
TryReceiveChannel, 46
TrySendChannel, 46
TryTransmitChannel, 46

UNLOCK, 58
UNSAFE_VIEW, 26

ViewShared, 26

WaitAnyTask, 8
WaitSemaphore, 50
WaitTask, 8
  with a condition, 8
WaitTasks, 8

WaitThread, 41