# Requirements for Safety-Critical Java Virtual Machines.

James Baxter

This report contains a formal model in the *Circus* specification language of the services that must be provided by a Safety-Critical Java virtual machine (SCJVM). As shown in Figure 1, we view an SCJVM as being divided into two components: the core execution environment, which executes Java bytecode programs, and the virtual machine services, which provide the services required to support the core execution environment and the SCJ infrastructure, such as scheduling and memory management. These components may also make use of the services of an underlying operating system or hardware abstraction layer.

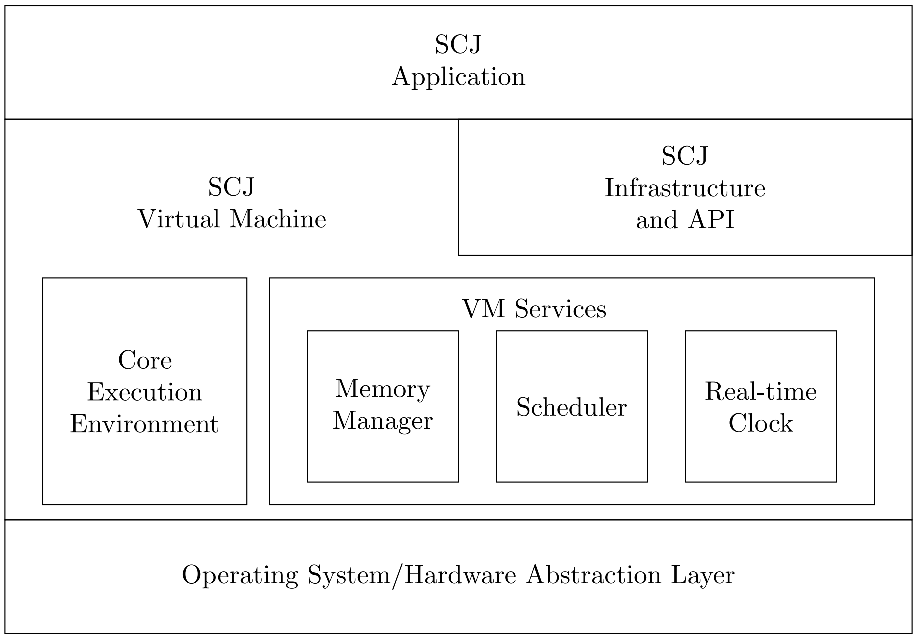
SCJ Application
SCJ Virtual Machine
SCJ Infrastructure and API
Core Execution Environment
VM Services
Memory Manager
Scheduler
Real-time Clock
Operating System/Hardware Abstraction Layer

Figure 1: A diagram showing the structure of the SCJ virtual machine and its relation to the SCJ infrastructure and the operating system/hardware abstraction layer.

The semantics of the core execution environment are similar to those of Java bytecode. In this report we instead focus on the virtual machine services, which we view as being divided into three areas:

- the memory manager, which manages backing stores for memory areas and allocation within them;
- the scheduler, which manages threads and interrupts, and allows for implementation of SCJ event handlers; and
- the real-time clock, which provides an interface to the system real-time clock.

Each of these services is used either by the core execution environment or by the SCJ infrastructure; some of the services also rely on each other. For example, the scheduler must update the allocation context in the memory manager when performing a thread switch.

In this report the formal model for each of the three areas identified is presented in a separate section: the memory manager in Section 1, the scheduler in Section 2, and the real-time clock in Section 3. The parts of the model are then combined to form a complete model of the virtual machine services in Section 4.

## 1 Memory Manager

**section** $\mathit{memorymanager}$ **parents** $\mathit{standard\_toolkit}$

SCJ deals with memory in terms of memory areas, which are Java objects with associated regions of memory as backing stores. The SCJVM deals with memory solely in terms of these backing stores, with the handling of memory areas as Java objects left to the SCJ infrastructure. Each backing store is identified by an implementation-defined backing store identifier, which may simply be a pointer to the backing store's location in memory.

$[\mathit{BackingStoreID}]$

The SCJVM memory manager must be capable of allocating, clearing and resizing backing stores as well as allocating memory within them. Backing stores may be created within other backing stores and can be arbitrarily nested. The memory allocated by the SCJVM is in the form of raw contiguous blocks of memory. There is no notion of objects as they only exist at the level of Java code and dealing solely with blocks of memory at the level of the VM allows for objects to be represented in a way appropriate to the form of the compiled code. Memory allocation by the SCJVM must be performed in constant time and must avoid memory fragmentation. The SCJVM must also be able to determine which backing store a given block of memory is in, for the purpose of implementing the `getMemoryArea` method of `MemoryArea`. Lastly, there must also be a means of checking the size and free space in a backing store. The SCJVM must also manage stacks, which are also referred to by unique identifiers. The SCJVM must be capable of creating and destroying stacks.

Memory addresses are modelled as natural numbers on the assumption that there are countably many memory addresses. We use the concept of natural number ranges to define the concept of a contiguous memory block, which forms the basis of specifying the requirement against fragmentation.

$$\begin{aligned}
&\mathit{MemoryAddress} == \mathbb{N}\\
&\mathit{ContiguousMemory} ==\\
&\quad \{\, m : \mathbb{P}\,\mathit{MemoryAddress} \mid \exists\, a, b : \mathit{MemoryAddress} \bullet m = a \,..\, b \,\}
\end{aligned}$$

In addition to managing backing stores, the memory manager must also manage stacks, which are also referred to by unique identifiers. The SCJVM must provide operations for creating and destroying stacks.

$[\mathit{StackID}]$

We declare channels for each of the operations of the memory manager. Operations that return a value have a separate channel to pass that value.

**channel** $\mathit{MMgetRootBackingStore}$
**channel** $\mathit{MMgetRootBackingStoreRet} : \mathit{BackingStoreID}$
**channel** $\mathit{MMgetCurrentAllocationContext}$
**channel** $\mathit{MMgetCurrentAllocationContextRet} : \mathit{BackingStoreID}$
**channel** $\mathit{MMsetCurrentAllocationContext} : \mathit{BackingStoreID}$
**channel** $\mathit{MMgetTotalSize}, \mathit{MMgetUsedSize}, \mathit{MMgetFreeSize} : \mathit{BackingStoreID}$
**channel** $\mathit{MMgetTotalSizeRet}, \mathit{MMgetUsedSizeRet}, \mathit{MMgetFreeSizeRet} : \mathbb{N}$
**channel** $\mathit{MMfindBackingStore} : \mathit{MemoryAddress}$
**channel** $\mathit{MMfindBackingStoreRet} : \mathit{BackingStoreID}$
**channel** $\mathit{MMallocateMemory} : \mathbb{N};\ \mathit{MMallocateMemoryRet} : \mathit{MemoryAddress}$
**channel** $\mathit{MMmakeBackingStore} : \mathbb{N};\ \mathit{MMmakeBackingStoreRet} : \mathit{BackingStoreID}$
**channel** $\mathit{MMclearCurrentAllocationContext}$
**channel** $\mathit{MMresizeBackingStore} : \mathit{BackingStoreID} \times \mathbb{N}$
**channel** $\mathit{MMresizeBackingStoreRet} : \mathit{BackingStoreID}$
**channel** $\mathit{MMcreateStack} : \mathbb{N};\ \mathit{MMcreateStackRet} : \mathit{StackID}$
**channel** $\mathit{MMdestroyStack} : \mathit{StackID}$

Each memory manager function reports a value signalling whether an error occurred and, if so, what error. These error values are of the type *MMReport*, defined below and are reported over the channel *MMreport*.

$$\begin{aligned}&MMReport ::= MMokay \mid MMoutOfMemory \mid MMnotEmpty \mid \\ &\quad MMnonexistentAllocation \mid MMsizeTooSmall \mid MMnonexistentBS \mid MMstoreInUse \mid \\ &\quad MMrootBSResize \mid MMnotOnlyChild \mid MMunknownAddress \mid MMnonexistentStack \mid \\ &\quad MMcannotShrink \mid MMinvalidThreadAC \mid MMfragmentation\end{aligned}$$

$\textbf{channel}\ MMreport : MMReport$

The memory manager also needs to interact with the scheduler regarding threads in order to determine the current allocation context. We provide here the channels used for communication with the scheduler as well as the type of thread identifiers.

$[ThreadID]$

$\textbf{channel}\ ScurrentThread : ThreadID$
$\textbf{channel}\ MMaddThread : ThreadID \times BackingStoreID$
$\textbf{channel}\ MMremoveThread : ThreadID$

Lastly, we declare a channel through which the memory manager's initialisation information can be supplied.

$\textbf{channel}\ MMinit : ContiguousMemory \times ContiguousMemory$

Having declared the channels, we begin the process declaration.

$\textbf{process}\ MemoryManager \mathrel{\widehat=} \textbf{begin}$

This specification allows a certain amount of memory overhead to be included in allocated blocks of memory and backing stores to allow for implementation of some memory management algorithm. Memory allocation operations must ensure that there is enough memory available for both the requested amount of memory and the additional overhead. The overhead values must be constant but may be zero for some memory management algorithms.

$| \quad allocationOverhead, backingStoreOverhead : \mathbb{N}$

## 1.1 Memory Blocks

Memory is allocated within memory blocks that keep a record of the amount of used and free memory. Memory blocks form the basis for both backing stores and stack allocation space. It is required that the free memory not be fragmented and the total memory must also not be fragmented in order to maintain this requirement. It is not required that the used memory be fragmentation free however. The used and free memory may not cover all of the memory in the memory block as there may be some overhead, as mentioned above. The used and free memory must be disjoint.

*MemoryBlock*

$$\begin{aligned}&free, total : ContiguousMemory\\&used : \mathbb{P}\ MemoryAddress\\ \hline &used \cup free \subseteq total\\&used \cap free = \varnothing\end{aligned}$$

A memory block must be initialised with the total memory covered by the block, including any overhead, and initially has no used memory.

$$\begin{array}{l}
\textit{MemoryBlockInit} \\
\hline
\textit{MemoryBlock}' \\
\textit{addresses?} : \textit{ContiguousMemory} \\
\hline
\textit{total}' = \textit{addresses?} \\
\textit{free}' \subseteq \textit{addresses?} \\
\textit{used}' = \varnothing \\
\hline
\end{array}$$

It must be possible to allocate memory within memory blocks. This removes a contiguous block of addresses of a given size from the free memory and adds them to the used memory, returning the allocated block. There must be sufficient free memory for the requested allocation size.

$$\begin{array}{l}
\textit{MBAllocate} \\
\hline
\Delta \textit{MemoryBlock} \\
\textit{size?} : \mathbb{N} \\
\textit{allocated!} : \textit{ContiguousMemory} \\
\hline
\textit{size?} \leq \# \textit{free} \\
\# \textit{allocated!} = \textit{size?} \\
\textit{allocated!} \subseteq \textit{free} \\
\textit{used}' = \textit{used} \cup \textit{allocated!} \\
\textit{free}' = \textit{free} \setminus \textit{allocated!} \\
\textit{total}' = \textit{total} \\
\hline
\end{array}$$

It must be possible to clear a memory block, making all its used memory free.

$$\begin{array}{l}
\textit{MBClear} \\
\hline
\Delta \textit{MemoryBlock} \\
\hline
\textit{total}' = \textit{total} \\
\textit{free}' = \textit{used} \cup \textit{free} \\
\textit{used}' = \varnothing \\
\hline
\end{array}$$

It must be possible to resize a memory block. The operation of resizing here just sets the memory block to use a new set of contiguous addresses. The new set must have room for the implicit overhead (the difference between the size of *total* and the size of *free*), the size of which is preserved by this operation. To avoid fragmentation, this is only possible when the memory block is empty. The old set of addresses used by the memory block is returned.

$$\begin{array}{l}
\textit{MBResize} \\
\hline
\Delta \textit{MemoryBlock} \\
\textit{newAddresses?} : \textit{ContiguousMemory} \\
\textit{oldAddresses!} : \textit{ContiguousMemory} \\
\hline
\# \textit{newAddresses?} \geq \# \textit{total} - \# \textit{free} \\
\textit{used} = \varnothing \\
\textit{total}' = \textit{newAddresses?} \\
\textit{used}' = \textit{used} \\
\textit{free}' \subseteq \textit{newAddresses?} \\
\# \textit{free}' = \# \textit{newAddresses?} - (\# \textit{total} - \# \textit{free}) \\
\textit{oldAddresses!} = \textit{total} \\
\hline
\end{array}$$

It must also be possible to resize all or part of the used memory within a memory block. This simply frees the old part of the memory and allocates a new part of the given size, keeping free memory contiguous. The old addresses must actually be in use and there must be sufficient free space for this to work.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBContentsResize}} \\ \hline
\Delta \mathit{MemoryBlock} \\
\mathit{newSize?} : \mathbb{N} \\
\mathit{oldAddresses?} : \mathit{ContiguousMemory} \\
\mathit{newAddresses!} : \mathit{ContiguousMemory} \\ \hline
\mathit{oldAddresses?} \subseteq \mathit{used} \\
\mathit{newSize?} - \# \mathit{oldAddresses?} \leq \# \mathit{free} \\
\mathit{newAddresses!} \subseteq \mathit{free} \cup \mathit{oldAddresses?} \\
\# \mathit{newAddresses!} = \mathit{newSize?} \\
\mathit{used}' = (\mathit{used} \setminus \mathit{oldAddresses?}) \cup \mathit{newAddresses!} \\
\mathit{free}' = (\mathit{free} \cup \mathit{oldAddresses?}) \setminus \mathit{newAddresses!} \\
\mathit{total}' = \mathit{total} \\ \hline
\end{array}$$

It is also necessary to be able to read the total, used and free sizes of a memory block.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBGetTotalSize}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{size!} : \mathbb{N} \\ \hline
\mathit{size!} = \# \mathit{total} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBGetUsedSize}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{size!} : \mathbb{N} \\ \hline
\mathit{size!} = \# \mathit{used} \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBGetFreeSize}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{size!} : \mathbb{N} \\ \hline
\mathit{size!} = \# \mathit{free} \\ \hline
\end{array}$$

The operations on the memory manager must then be made into robust operations by adding error reporting. Errors are reported by returning a value to indicate the type of error, taken from the *MMReport* type declared earlier.

The successful completion of an operation is indicated by returning *MMokay*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Success}} \\ \hline
\mathit{report!} : \mathit{MMReport} \\ \hline
\mathit{report!} = \mathit{MMokay} \\ \hline
\end{array}$$

It must be reported if a memory block has insufficient free memory to fulfil an allocation request.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{MBOutOfMemory}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{size?} : \mathbb{N} \\
\mathit{report!} : \mathit{MMReport} \\ \hline
\mathit{size?} > \# \mathit{free} \\
\mathit{report!} = \mathit{MMoutOfMemory} \\ \hline
\end{array}$$

It must be reported if an attempt is made to resize a nonempty memory block.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MBNotEmpty}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{report!} : \mathit{MMReport} \\ \hline
\mathit{used} \neq \varnothing \\
\mathit{report!} = \mathit{MMnotEmpty} \\ \hline
\end{array}$$

It must be reported if the new set of addresses is insufficiently large to contain the overhead of the memory block in a resizing.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MBResizeBellowOverhead}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{newAddresses?} : \mathit{ContiguousMemory} \\
\mathit{report!} : \mathit{MMReport} \\ \hline
\#\, \mathit{total} - \#\, \mathit{free} > \#\, \mathit{newAddresses?} \\
\mathit{report!} = \mathit{MMcannotShrink} \\ \hline
\end{array}$$

It must be reported if an attempt is made to resize memory outside the allocated memory in a memory block.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MBNonexistentAllocation}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{oldAddresses?} : \mathit{ContiguousMemory} \\
\mathit{report!} : \mathit{MMReport} \\ \hline
\neg\, (\mathit{oldAddresses?} \subseteq \mathit{used}) \\
\mathit{report!} = \mathit{MMnonexistentAllocation} \\ \hline
\end{array}$$

It must be reported if there is insufficient memory to fulfil a requested resizing.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{MBInsufficientFreeSize}} \\ \hline
\Xi \mathit{MemoryBlock} \\
\mathit{oldAddresses?} : \mathit{ContiguousMemory} \\
\mathit{newSize?} : \mathbb{N} \\
\mathit{report!} : \mathit{MMReport} \\ \hline
\mathit{newSize?} > \#(\mathit{free} \cup \mathit{oldAddresses?}) \\
\mathit{report!} = \mathit{MMoutOfMemory} \\ \hline
\end{array}$$

The operations on memory blocks can then be lifted to robust versions.

$$\begin{array}{l}
\mathit{RMBAllocate} \;==\; (\mathit{MBAllocate} \land \mathit{Success}) \lor \mathit{MBOutOfMemory} \\
\mathit{RMBClear} \;==\; \mathit{MBClear} \land \mathit{Success} \\
\mathit{RMBResize} \;==\; (\mathit{MBResize} \land \mathit{Success}) \lor \mathit{MBNotEmpty} \lor \mathit{MBResizeBellowOverhead} \\
\mathit{RMBContentsResize} \;== \\
\qquad (\mathit{MBContentsResize} \land \mathit{Success}) \lor \mathit{MBNonexistentAllocation} \lor \mathit{MBInsufficientFreeSize} \\
\mathit{RMBGetTotalSize} \;==\; \mathit{MBGetTotalSize} \land \mathit{Success} \\
\mathit{RMBGetUsedSize} \;==\; \mathit{MBGetUsedSize} \land \mathit{Success} \\
\mathit{RMBGetFreeSize} \;==\; \mathit{MBGetFreeSize} \land \mathit{Success}
\end{array}$$

## 1.2 Backing Stores

The memory manager deals with memory in the form of backing stores, which are memory blocks that may contain other backing stores. In this model the child backing stores of a backing store are represented by a finite set of identifiers. The relationship between the backing stores can only be properly specified in the global memory manager, which is presented later in this report. Each backing store in this model also stores its own identifier for the purpose of defining an injectivity condition later on. A backing store is required to not be a child of itself. At this stage the implicit overhead present in memory blocks is replaced with the correct overhead for backing stores.

$$
\begin{array}{|l}
\textit{BackingStore} \\
\hline
\textit{MemoryBlock} \\
\textit{children} : \mathbb{F}\, \textit{BackingStoreID} \\
\textit{self} : \textit{BackingStoreID} \\
\hline
\textit{self} \notin \textit{children} \\
\#\, \textit{used} + \#\, \textit{free} + \textit{backingStoreOverhead} = \#\, \textit{total} \\
\hline
\end{array}
$$

Backing stores are initialised as for memory blocks, with the additional requirement that they initially have no child backing stores. The initial size of the backing store must be large enough to account for the backing store overhead.

$$
\begin{array}{|l}
\textit{BackingStoreInit} \\
\hline
\textit{BackingStore}' \\
\textit{MemoryBlockInit} \\
\hline
\#\, \textit{addresses}? \geq \textit{backingStoreOverhead} \\
\#\, \textit{free}' = \#\, \textit{addresses}? - \textit{backingStoreOverhead} \\
\textit{children}' = \varnothing \\
\hline
\end{array}
$$

It must be possible to allocate a new child backing store within a backing store. This is the same as a memory block allocation operation except that the newly created child's identifier is added to the backing store's set of children. The size input to this operation is assumed to include the child's backing store overhead, though that cannot be checked at this stage of the model. The new child's identifier must not be the identifier of an existing child or the identifier of the parent.

$$
\begin{array}{|l}
\textit{BSAllocateChild} \\
\hline
\Delta \textit{BackingStore} \\
\textit{MBAllocate} \\
\textit{childID}! : \textit{BackingStoreID} \\
\hline
\textit{childID}! \notin \textit{children} \land \textit{childID}! \neq \textit{self} \\
\textit{children}' = \textit{children} \cup \{\textit{childID}!\} \\
\textit{self}' = \textit{self} \\
\hline
\end{array}
$$

Clearing a backing store must also be possible. This is the same as clearing a memory block but with the removal of the backing store's children.

$$
\begin{array}{|l}
\textit{BSClear} \\
\hline
\Delta \textit{BackingStore} \\
\textit{MBClear} \\
\hline
\textit{children}' = \varnothing \\
\textit{self}' = \textit{self} \\
\hline
\end{array}
$$

The resizing of a child backing store must be allowed for. This is the same as resizing the contents of a memory block but the identifier of the child may change. The old identifier must be the identifier of a child backing store and the new identifier must not be the identifier of any other backing store (though it may be the same as the old identifier).

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BSResizeChild}} \\
\Delta \mathit{BackingStore} \\
\mathit{MBContentsResize} \\
\mathit{oldID}?, \mathit{newID}! : \mathit{BackingStoreID} \\
\hline
\mathit{oldID}? \in \mathit{children} \\
\mathit{newID}! \notin \mathit{children} \lor \mathit{newID}! = \mathit{oldID}? \\
\mathit{newID}! \neq \mathit{self} \\
\mathit{children}' = (\mathit{children} \setminus \{\mathit{oldID}?\}) \cup \{\mathit{newID}!\} \\
\mathit{self}' = \mathit{self}
\end{array}$$

Allocating within a backing store is the same as allocating within a memory block but must take into account the allocation overhead. This operation does not affect the children of the backing store.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BSAllocate}} \\
\Delta \mathit{BackingStore} \\
\mathit{size}? : \mathbb{N} \\
\mathit{allocated}! : \mathit{ContiguousMemory} \\
\hline
\exists\, \mathit{actualSize} : \mathbb{N} \mid \mathit{actualSize} = \mathit{size}? + \mathit{allocationOverhead} \bullet \\
\qquad \mathit{MBAllocate}[\mathit{actualSize}/\mathit{size}?] \\
\mathit{children}' = \mathit{children} \\
\mathit{self}' = \mathit{self}
\end{array}$$

It must be possible to resize a backing store, requiring that it has no children in order to ensure it is empty. The new addresses supplied must be able to contain the backing store overhead.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BSResize}} \\
\Delta \mathit{BackingStore} \\
\mathit{MBResize} \\
\hline
\#\, \mathit{newAddresses}? \geq \mathit{backingStoreOverhead} \\
\mathit{children}' = \mathit{children} = \varnothing \\
\mathit{self}' = \mathit{self}
\end{array}$$

The other operations on backing stores can be defined by promoting the memory block operations to operate on backing stores, keeping the set of children and the self identifier the same.

$$\begin{array}{lcl}
\mathit{BSGetTotalSize} & == & \mathit{MBGetTotalSize} \land \Xi \mathit{BackingStore} \\
\mathit{BSGetUsedSize} & == & \mathit{MBGetUsedSize} \land \Xi \mathit{BackingStore} \\
\mathit{BSGetFreeSize} & == & \mathit{MBGetFreeSize} \land \Xi \mathit{BackingStore}
\end{array}$$

These operations must then be made into robust operations that report error values if their preconditions are not met. Some of the error reporting schemas for memory blocks can be reused but there are new preconditions that must be accounted for, which require additional schemas.

It must be reported if a backing store has a nonempty set of children when it is required to be empty.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BSNotEmpty}} \\
\Xi \mathit{BackingStore} \\
\mathit{report!} : \mathit{MMReport} \\
\hline
\mathit{children} \neq \emptyset \\
\mathit{report!} = \mathit{MMnotEmpty}
\end{array}$$

It must be reported if the supplied old identifier for a child resize is not actually the identifier of a child.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BSNonexistentAllocation}} \\
\Xi \mathit{BackingStore} \\
\mathit{MBNonexistentAllocation} \\
\mathit{oldID?} : \mathit{BackingStoreID} \\
\mathit{report!} : \mathit{MMReport} \\
\hline
\mathit{oldID?} \notin \mathit{children} \\
\mathit{report!} = \mathit{MMnonexistentAllocation}
\end{array}$$

It must be reported if the supplied addresses do not contain enough space to store the backing store overhead.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{BSSizeTooSmall}} \\
\Xi \mathit{BackingStore} \\
\mathit{addresses?} : \mathit{ContiguousMemory} \\
\mathit{report!} : \mathit{MMReport} \\
\hline
\# \mathit{addresses?} < \mathit{backingStoreOverhead} \\
\mathit{report!} = \mathit{MMsizeTooSmall}
\end{array}$$

Then the operations on backing stores can be made into robust operations.

$$\begin{array}{l}
\mathit{RBackingStoreInit} \mathrel{==} \\
\quad (\mathit{BackingStoreInit} \land \mathit{Success}) \lor (\exists \mathit{BackingStore} \bullet \mathit{BSSizeTooSmall}) \\
\mathit{RBSAllocateChild} \mathrel{==} \\
\quad (\mathit{BSAllocateChild} \land \mathit{Success}) \lor (\mathit{MBOutOfMemory} \land \Xi \mathit{BackingStore}) \\
\mathit{RBSClear} \mathrel{==} \mathit{BSClear} \land \mathit{Success} \\
\mathit{RBSResizeChild} \mathrel{==} \\
\quad (\mathit{BSResizeChild} \land \mathit{Success}) \lor (\mathit{MBNonexistentAllocation} \land \Xi \mathit{BackingStore}) \lor \\
\quad (\mathit{MBInsufficientFreeSize} \land \Xi \mathit{BackingStore}) \lor \mathit{BSNonexistentAllocation} \\
\mathit{RBSAllocate} \mathrel{==} (\mathit{BSAllocate} \land \mathit{Success}) \lor (\mathit{MBOutOfMemory} \land \Xi \mathit{BackingStore}) \\
\mathit{RBSResize} \mathrel{==} \\
\quad (\mathit{BSResize} \land \mathit{Success}) \lor (\mathit{MBNotEmpty} \land \Xi \mathit{BackingStore}) \lor \\
\quad \mathit{BSNotEmpty} \lor \mathit{BSSizeTooSmall}[\mathit{newAddresses?} / \mathit{addresses?}] \\
\mathit{RBSGetTotalSize} \mathrel{==} \mathit{BSGetTotalSize} \land \mathit{Success} \\
\mathit{RBSGetUsedSize} \mathrel{==} \mathit{BSGetUsedSize} \land \mathit{Success} \\
\mathit{RBSGetFreeSize} \mathrel{==} \mathit{BSGetFreeSize} \land \mathit{Success}
\end{array}$$

## 1.3 Global Memory Manager

The memory manager must hold information on all backing stores and the identifier of the one that is the root backing store. The root backing store covers the whole of the memory available for allocation by the SCJVM and may be used as the backing store for `ImmortalMemory`. The root backing store cannot be resized. All backing stores must be, directly or indirectly, a child of the root backing store and no

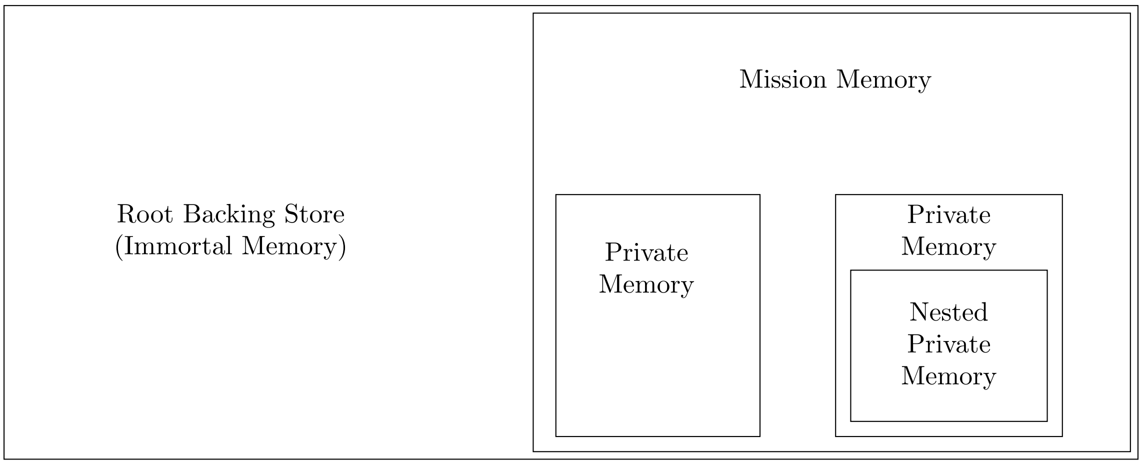

Figure 2: An illustration of an example memory layout

backing store may be a child of itself. An example of a layout of memory is shown in Figure 2. It is required that the root backing store's identifier be the identifier of some backing store. It is required that each backing store's identifier point to a backing store that has that identifier. It is required that the memory of a backing store's children be disjoint and within the set of used memory.

Many of the memory manager operations operate on a backing store that is designated as the current allocation context, which may be different for each thread. The memory manager stores the current allocation context for each thread so that, given the current thread identifier by the scheduler, the correct current allocation context can be determined. It is required that all thread allocation contexts be valid backing store identifiers.

There is a relation that relates each backing store identifier to the identifiers of the immediate children of that backing store. This is used to require that all backing stores must be either the root backing store or a child (direct or indirect) of it, and also that no backing store can be a child (direct or indirect) of itself.

*GlobalMemoryManager*

$$
\begin{array}{l}
\mathit{stores} : \mathit{BackingStoreID} \pfun \mathit{BackingStore} \\
\mathit{rootBackingStore} : \mathit{BackingStoreID} \\
\mathit{threadACs} : \mathit{ThreadID} \pfun \mathit{BackingStoreID} \\
\mathit{childRelation} : \mathit{BackingStoreID} \leftrightarrow \mathit{BackingStoreID}
\end{array}
$$

$$
\begin{array}{l}
\mathit{rootBackingStore} \in \operatorname{dom} \mathit{stores} \\
\forall\, b : \operatorname{dom} \mathit{stores} \bullet (\mathit{stores}\ b).\mathit{self} = b \\
\forall\, b : \operatorname{ran} \mathit{stores} \bullet \exists\, m : \mathbb{P}\, b.\mathit{used} \bullet \\
\qquad (\lambda\, x : b.\mathit{children} \bullet (\mathit{stores}\ x).\mathit{total}) \text{ partition } m \\
\operatorname{ran} \mathit{threadACs} \subset \operatorname{dom} \mathit{stores} \\
\mathit{childRelation} = \bigcup\{i : \operatorname{dom} \mathit{stores} \bullet \{j : (\mathit{stores}\ i).\mathit{children} \bullet (i, j)\}\} \\
\operatorname{dom} \mathit{stores} = (\mathit{childRelation}^{*}) (\!| \{\mathit{rootBackingStore}\} |\!) \\
\forall\, s : \operatorname{dom} \mathit{stores} \bullet s \notin \mathit{childRelation}^{+} (\!| \{s\} |\!)
\end{array}
$$

Initially there must be one backing store provided, which is the root backing store. The memory manager must be initialised with the size of the root backing store. The child relation is initially empty because there is only one backing store that initially has no children. There are initially no thread allocation contexts registered with the memory manager; the scheduler must be initialised after the memory manager so that it can register the allocation context for the first thread.

$$
\begin{array}{l}
\textit{GlobalMemoryManagerInit} \\
\hline
\textit{GlobalMemoryManager}\,' \\
\textit{addresses}? : \textit{ContiguousMemory} \\
\textit{report}! : \textit{MMReport} \\
\hline
\exists \textit{BackingStore}\,' \mid \textit{RBackingStoreInit} \bullet \\
\qquad \textit{stores}' = \{\textit{rootBackingStore}' \mapsto \theta\, \textit{BackingStore}\,'\} \\
\textit{childRelation}' = \emptyset \\
\textit{threadACs}' = \emptyset
\end{array}
$$

The operations on the global memory manager promote the backing store operations and update the *stores* map. Many of the operations take an input indicating which backing store is the allocation context to be acted on, this will be instantiated according to which thread is running when the operations are promoted to take a thread argument later in the model.

Many of the memory manager operations only affect *stores* (with *childRelation* being implicitly updated) so for brevity we define a schema that preserves the other variables of the global memory manager.

$$
\textit{GlobalFixedVars} \;==\; \Xi \textit{GlobalMemoryManager} \setminus (\textit{stores}, \textit{childRelation})
$$

It must be possible to make a new backing store inside a given backing store, using *RBSAllocateChild* to update the parent backing store, which is the given allocation context, and using *RBackingStoreInit* to initialise the newly created child backing store. The size input to this operation does not include the backing store overhead, as the API and program should not be aware of the memory manager's internal overhead, so the backing store overhead is added to the input size to get the size used for allocating the new backing store. The error reports from the two promoted operations are captured and required to be reports of success. The child identifier output from *RBSAllocateChild* is required to not already be a known backing store and must be used for the identifier of the newly created backing store.

$$
\begin{array}{l}
\textit{GlobalMakeBS} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
\textit{size}? : \mathbb{N} \\
\textit{ac}? : \textit{BackingStoreID} \\
\textit{childID}! : \textit{BackingStoreID} \\
\hline
\textit{ac}? \in \operatorname{dom} \textit{stores} \\
\exists \textit{actualSize} : \mathbb{N} \mid \textit{actualSize} = \textit{size}? + \textit{backingStoreOverhead} \bullet \\
\exists \textit{allocated}! : \textit{ContiguousMemory} \bullet \\
\exists \textit{Parent}, \textit{child} : \textit{BackingStore} \bullet \\
\qquad (\exists \Delta \textit{BackingStore};\ \textit{report}! : \textit{MMReport} \mid \\
\qquad\qquad \textit{RBSAllocateChild}[\textit{actualSize}/\textit{size}?] \bullet \\
\qquad\qquad \theta\, \textit{BackingStore} = \textit{stores}\ \textit{ac}? \land \\
\qquad\qquad \textit{Parent} = \theta\, \textit{BackingStore}\,' \land \\
\qquad\qquad \textit{childID}! \notin \operatorname{dom} \textit{stores} \land \\
\qquad\qquad \textit{report}! = \textit{MMokay}) \land \\
\qquad (\exists \textit{BackingStore}\,';\ \textit{report}! : \textit{MMReport} \mid \\
\qquad\qquad \textit{RBackingStoreInit}[\textit{allocated}!/\textit{addresses}?] \bullet \\
\qquad\qquad \textit{self}' = \textit{childID}! \land \\
\qquad\qquad \textit{child} = \theta\, \textit{BackingStore}\,' \land \\
\qquad\qquad \textit{report}! = \textit{MMokay}) \land \\
\qquad \textit{stores}' = \textit{stores} \oplus \{\textit{ac}? \mapsto \textit{Parent}, \textit{childID}! \mapsto \textit{child}\} \\
\textit{GlobalFixedVars}
\end{array}
$$

It must be possible to clear a backing store given as allocation context, removing the transitive closure of its children from the global memory manager. It is required that none of the backing stores removed are being used as thread allocation contexts.

$$
\begin{array}{l}
\textit{GlobalClearBS} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
ac? : \textit{BackingStoreID} \\
\hline
ac? \in \operatorname{dom} \textit{stores} \\
\exists \Delta \textit{BackingStore};\ \textit{report}! : \textit{MMReport} \mid \textit{RBSClear} \bullet \\
\quad \theta\, \textit{BackingStore} = \textit{stores}(ac?) \land \\
\quad \textit{report}! = \textit{MMokay} \land \\
\quad \exists\, \textit{reachable} : \mathbb{F}\, \textit{BackingStoreID} \mid \\
\quad\quad \textit{reachable} = \textit{childRelation}^{+} (\!| \{ac?\} |\!) \bullet \\
\quad\quad \textit{stores}' = (\textit{reachable} \ndres \textit{stores}) \oplus \{ac? \mapsto \theta\, \textit{BackingStore}'\} \land \\
\quad\quad \textit{reachable} \cap \operatorname{ran} \textit{threadACs} = \varnothing \\
\textit{GlobalFixedVars} \\
\hline
\end{array}
$$

It must be possible to resize a given backing store. The backing store given must not be the root backing store and must exist. To avoid fragmentation the backing store being resized cannot be allowed to have siblings. The resizing operation may change the identifier of the backing store (particularly if the identifiers are pointers) so the new identifier is given as output. The parent backing store must also be updated.

$$
\begin{array}{l}
\textit{GlobalResizeBS} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
\textit{toResize}? : \textit{BackingStoreID} \\
\textit{newSize}? : \mathbb{N} \\
\textit{newID}! : \textit{BackingStoreID} \\
\hline
\textit{toResize}? \neq \textit{rootBackingStore} \\
\textit{toResize}? \in \operatorname{dom} \textit{stores} \\
\exists\, \textit{child}, \textit{Parent} : \textit{BackingStore};\ \textit{parentID} : \textit{BackingStoreID} \mid \\
\quad (\textit{stores}(\textit{parentID})).\textit{children} = \{\textit{toResize}?\} \bullet \\
\exists\, \textit{oldAddresses}?, \textit{newAddresses}! : \textit{ContiguousMemory} \bullet \\
\quad (\exists \Delta \textit{BackingStore};\ \textit{report}! : \textit{MMReport} \mid \\
\quad\quad \textit{RBSResize}[\textit{newAddresses}!/\textit{newAddresses}?, \\
\quad\quad\quad\quad \textit{oldAddresses}?/\textit{oldAddresses}!] \bullet \\
\quad\quad \theta\, \textit{BackingStore} = \textit{stores}(\textit{toResize}?) \land \\
\quad\quad \textit{report}! = \textit{MMokay} \land \\
\quad\quad \textit{child} = \theta\, \textit{BackingStore}') \land \\
\quad (\exists \Delta \textit{BackingStore};\ \textit{report}! : \textit{MMReport} \mid \\
\quad\quad \textit{RBSResizeChild}[\textit{toResize}?/\textit{oldID}?] \bullet \\
\quad\quad \theta\, \textit{BackingStore} = \textit{stores}(\textit{parentID}) \land \\
\quad\quad \textit{report}! = \textit{MMokay} \land \\
\quad\quad \textit{Parent} = \theta\, \textit{BackingStore}') \land \\
\quad (\textit{newID}! \notin \operatorname{dom} \textit{stores} \lor \textit{newID}! = \textit{toResize}?) \land \\
\quad \textit{stores}' = (\{\textit{toResize}?\} \ndres \textit{stores}) \\
\quad\quad \oplus \{\textit{parentID} \mapsto \textit{Parent}, \textit{newID}! \mapsto \textit{child}\} \\
\textit{GlobalFixedVars} \\
\hline
\end{array}
$$

It must be possible to determine which backing store a given memory address belongs to, in order to implement the `getMemoryArea` method of `MemoryArea`. The address belongs to the furthest descendant that contains it and must be in the allocatable memory (i.e. the area covered by the root backing store) for this to work.

$$\begin{array}{l}
\textit{GlobalFindAddress} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{address?} : \textit{MemoryAddress} \\
\textit{backingStore!} : \textit{BackingStoreID} \\
\hline
\textit{address?} \in (\textit{stores}\ \textit{rootBackingStore}).\textit{total} \\
\exists\, \textit{containingStores} : \mathbb{P}\, \textit{BackingStoreID} \mid \\
\quad \textit{containingStores} = \\
\quad\quad \{\, \textit{bsid} : \operatorname{dom} \textit{stores} \mid \textit{address?} \in (\textit{stores}\ \textit{bsid}).\textit{total} \,\} \bullet \\
\quad \textit{backingStore!} = \\
\quad\quad (\mu\, \textit{bsid} : \textit{containingStores} \mid \\
\quad\quad\quad \forall\, x : \textit{containingStores} \bullet (x, \textit{bsid}) \in \textit{childRelation}^{\,*}) \\
\hline
\end{array}$$

It must be possible to obtain the identifier of the root backing store from the global memory manager, though it is not possible to change the root backing store.

$$\begin{array}{l}
\textit{GlobalGetRootBackingStore} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{rbs!} : \textit{BackingStoreID} \\
\hline
\textit{rbs!} = \textit{rootBackingStore} \\
\hline
\end{array}$$

The remaining operations on the global memory manager are obtained by promoting the corresponding operations on backing stores to operate on a given allocation context. The operation of allocating memory is adjusted here to return the address of the start of the allocated block since the entire block of addresses is only required for specifying the *GlobalMakeBS* operation.

$$\begin{array}{l}
\textit{PromoteAC} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
\Delta \textit{BackingStore} \\
\textit{ac?} : \textit{BackingStoreID} \\
\hline
\textit{ac?} \in \operatorname{dom} \textit{stores} \\
\theta\, \textit{BackingStore} = \textit{stores}(\textit{ac?}) \\
\textit{stores}' = \textit{stores} \oplus \{\textit{ac?} \mapsto \theta\, \textit{BackingStore}\,'\} \\
\textit{GlobalFixedVars} \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{GlobalAllocateMemory} \ == \\
\quad \exists\, \Delta \textit{BackingStore};\ \textit{allocated!} : \textit{ContiguousMemory} \bullet \\
\quad\quad \textit{RBSAllocate} \wedge \textit{PromoteAC} \wedge \\
\quad\quad [\textit{allocated!} : \textit{ContiguousMemory};\ \textit{address!} : \textit{MemoryAddress} \mid \\
\quad\quad\quad \textit{address!} = \min \textit{allocated!}] \\
\textit{GlobalGetTotalSize} \ == \ \exists\, \Delta \textit{BackingStore} \bullet \textit{RBSGetTotalSize} \wedge \textit{PromoteAC} \\
\textit{GlobalGetUsedSize} \ == \ \exists\, \Delta \textit{BackingStore} \bullet \textit{RBSGetUsedSize} \wedge \textit{PromoteAC} \\
\textit{GlobalGetFreeSize} \ == \ \exists\, \Delta \textit{BackingStore} \bullet \textit{RBSGetFreeSize} \wedge \textit{PromoteAC}
\end{array}$$

These operations on the global memory manager must then be made into robust operations that report errors. The robust versions of the backing store operations have been used in specifying the global memory manager operations but some new error reporting schemas are required to handle errors that can occur at the level of the global memory manager and errors in the captured report variables must be handled.

It must be reported if the specified allocation context is not actually a valid backing store.

$$
\begin{array}{l}
\textit{GlobalNonexistentBS} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
ac? : \textit{BackingStoreID} \\
report! : \textit{MMReport} \\
\hline
ac? \notin \operatorname{dom} stores \\
report! = \textit{MMnonexistentBS} \\
\hline
\end{array}
$$

It must be reported if an attempt is made to clear a backing store when one of its children is the allocation context of a thread.

$$
\begin{array}{l}
\textit{GlobalStoreInUse} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
ac? : \textit{BackingStoreID} \\
report! : \textit{MMReport} \\
\hline
(\textit{childRelation}^{+} \llparenthesis \{ac?\} \rrparenthesis) \cap \operatorname{ran} \textit{threadACs} \neq \varnothing \\
report! = \textit{MMstoreInUse} \\
\hline
\end{array}
$$

It must be reported if an attempt is made to resize the root backing store.

$$
\begin{array}{l}
\textit{GlobalRootBSResize} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
toResize? : \textit{BackingStoreID} \\
report! : \textit{MMReport} \\
\hline
toResize? = \textit{rootBackingStore} \\
report! = \textit{MMrootBSResize} \\
\hline
\end{array}
$$

It must be reported if an attempt is made to resize a backing store that is not an only child.

$$
\begin{array}{l}
\textit{GlobalNotOnlyChild} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
toResize? : \textit{BackingStoreID} \\
report! : \textit{MMReport} \\
\hline
toResize? \neq \textit{rootBackingStore} \\
toResize? \in \operatorname{dom} stores \\
\forall bsid : \operatorname{dom} stores \bullet (stores(bsid)).children \neq \{toResize?\} \\
report! = \textit{MMnotOnlyChild} \\
\hline
\end{array}
$$

It must be reported if the attempt to allocate a new child backing store returned an error.

$$
\begin{array}{l}
\textit{GlobalBSAllocateChildError} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
ac? : \textit{BackingStoreID} \\
size? : \mathbb{N} \\
report! : \textit{MMReport} \\
\hline
ac? \in \operatorname{dom} stores \\
\exists childID! : \textit{BackingStoreID};\ allocated! : \textit{ContiguousMemory} \bullet \\
\exists actualSize : \mathbb{N} \mid actualSize = size? + \textit{backingStoreOverhead} \bullet \\
\exists \Delta \textit{BackingStore} \mid \textit{RBSAllocateChild}[actualSize/size?] \bullet \\
\qquad \theta\, \textit{BackingStore} = stores(ac?) \wedge \\
\qquad report! \neq \textit{MMokay} \\
\hline
\end{array}
$$

It must be reported if the attempt to resize a backing store returns an error.

$$
\begin{array}{l}
\textit{GlobalBSResizeError} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{toResize?} : \textit{BackingStoreID} \\
\textit{newSize?} : \mathbb{N} \\
\textit{report!} : \textit{MMReport} \\
\hline
\textit{toResize?} \neq \textit{rootBackingStore} \\
\textit{toResize?} \in \operatorname{dom} \textit{stores} \\
\exists \Delta \textit{BackingStore};\ \textit{newAddresses?}, \textit{oldAddresses!} : \mathbb{F}\,\textit{MemoryAddress} \mid \\
\quad \textit{RBSResize} \bullet \\
\quad \theta\,\textit{BackingStore} = \textit{stores}(\textit{toResize?}) \land \\
\quad \textit{report!} \neq \textit{MMokay}
\end{array}
$$

$$
\begin{array}{l}
\textit{GlobalBSResizeChildError} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{toResize?} : \textit{BackingStoreID} \\
\textit{newSize?} : \mathbb{N} \\
\textit{newID!} : \textit{BackingStoreID} \\
\textit{report!} : \textit{MMReport} \\
\hline
\textit{toResize?} \neq \textit{rootBackingStore} \\
\textit{toResize?} \in \operatorname{dom} \textit{stores} \\
\exists \textit{parentID} : \textit{BackingStoreID} \mid \\
\quad (\textit{stores}(\textit{parentID})).\textit{children} = \{\textit{toResize?}\} \bullet \\
\exists \textit{oldAddresses?}, \textit{newAddresses!} : \mathbb{F}\,\textit{MemoryAddress} \bullet \\
\exists \Delta \textit{BackingStore} \mid \textit{RBSResizeChild}[\textit{toResize?}/\textit{oldID?}] \bullet \\
\quad \theta\,\textit{BackingStore} = \textit{stores}(\textit{toResize?}) \land \\
\quad \textit{report!} \neq \textit{MMokay}
\end{array}
$$

$$
\begin{array}{l}
\textit{GlobalUnknownAddress} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{address?} : \textit{MemoryAddress} \\
\textit{report!} : \textit{MMReport} \\
\hline
\textit{address?} \notin (\textit{stores}\ \textit{rootBackingStore}).\textit{total} \\
\textit{report!} = \textit{MMunknownAddress}
\end{array}
$$

Then the operations on the global memory manager can be made into robust operations. The operations promoted with *PromoteAC* output the report of the promoted operation if no other error occurs so do not need to specify *Success*.

$$
\begin{array}{l}
RGlobalMemoryManagerInit == GlobalMemoryManagerInit \\
RGlobalMakeBS == (GlobalMakeBS \land Success) \lor GlobalStoreInUse \lor \\
\quad GlobalNonexistentBS \lor GlobalBSAllocateChildError \\
RGlobalClearBS == (GlobalClearBS \land Success) \lor GlobalNonexistentBS \lor GlobalStoreInUse \\
RGlobalResizeBS == (GlobalResizeBS \land Success) \lor \\
\quad GlobalNonexistentBS[toResize?/ac?] \lor GlobalRootBSResize \lor \\
\quad GlobalNotOnlyChild \lor GlobalBSResizeError \lor GlobalBSResizeChildError \\
RGlobalGetRootBackingStore == GlobalGetRootBackingStore \land Success \\
RGlobalFindAddress == (GlobalFindAddress \land Success) \lor GlobalUnknownAddress \\
RGlobalAllocateMemory == (GlobalAllocateMemory \land Success) \lor GlobalNonexistentBS \\
RGlobalGetTotalSize == (GlobalGetTotalSize \land Success) \lor GlobalNonexistentBS \\
RGlobalGetUsedSize == (GlobalGetUsedSize \land Success) \lor GlobalNonexistentBS \\
RGlobalGetFreeSize == (GlobalGetFreeSize \land Success) \lor GlobalNonexistentBS
\end{array}
$$

## 1.4 Thread Memory Manager

Operations that need to operate on the current allocation context must be lifted to determine the current allocation context from the current thread. Note that the operations to get the used, free and total sizes of a backing store work on a backing store identifier passed to the memory manager, not necessarily the current allocation context and so do not need to be lifted.

$$
\begin{array}{|l}
\multicolumn{1}{l}{PromoteThread} \\
\hline
\Delta GlobalMemoryManager \\
ac? : BackingStoreID \\
thread? : ThreadID \\
\hline
thread? \in \operatorname{dom} threadACs \\
ac? = threadACs\ thread? \\
\hline
\end{array}
$$

$$
\begin{array}{l}
ThreadMakeBS == \exists ac? : BackingStoreID \bullet RGlobalMakeBS \land PromoteThread \\
ThreadClearBS == \exists ac? : BackingStoreID \bullet RGlobalClearBS \land PromoteThread \\
ThreadAllocateMemory == \\
\quad \exists ac? : BackingStoreID \bullet RGlobalAllocateMemory \land PromoteThread
\end{array}
$$

It must also be possible to get and set the current allocation context for a given preexisting thread.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ThreadGetAC} \\
\hline
\Xi GlobalMemoryManager \\
thread? : ThreadID \\
ac! : BackingStoreID \\
\hline
thread? \in \operatorname{dom} threadACs \\
ac! = threadACs\ thread? \\
\hline
\end{array}
$$

$$\begin{array}{l}
\textit{ThreadSetAC} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
\textit{thread?} : \textit{ThreadID} \\
\textit{ac?} : \textit{BackingStoreID} \\
\hline
\textit{thread?} \in \operatorname{dom} \textit{threadACs} \\
\textit{threadACs}' = \textit{threadACs} \oplus \{\textit{thread?} \mapsto \textit{ac?}\} \\
\textit{stores}' = \textit{stores} \\
\textit{childRelation}' = \textit{childRelation} \\
\textit{rootBackingStore}' = \textit{rootBackingStore} \\
\hline
\end{array}$$

It must also be possible for the scheduler to add and remove threads in the *threadACs* map. These operations are used by the scheduler when threads are created or destroyed.

$$\begin{array}{l}
\textit{ThreadAdd} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
\textit{thread?} : \textit{ThreadID} \\
\textit{ac?} : \textit{BackingStoreID} \\
\hline
\textit{threadACs}' = \textit{threadACs} \oplus \{\textit{thread?} \mapsto \textit{ac?}\} \\
\textit{stores}' = \textit{stores} \\
\textit{childRelation}' = \textit{childRelation} \\
\textit{rootBackingStore}' = \textit{rootBackingStore} \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{ThreadRemove} \\
\hline
\Delta \textit{GlobalMemoryManager} \\
\textit{thread?} : \textit{ThreadID} \\
\hline
\textit{threadACs}' = \{\textit{thread?}\} \lhd \textit{threadACs} \\
\textit{stores}' = \textit{stores} \\
\textit{childRelation}' = \textit{childRelation} \\
\textit{rootBackingStore}' = \textit{rootBackingStore} \\
\hline
\end{array}$$

Operations that take a thread input must be made robust by accounting for whether or not the given thread is already in the map.

$$\begin{array}{l}
\textit{InvalidThreadAC} \\
\hline
\Xi \textit{GlobalMemoryManager} \\
\textit{thread?} : \textit{ThreadID} \\
\textit{report!} : \textit{MMReport} \\
\hline
\textit{thread?} \notin \operatorname{dom} \textit{threadACs} \\
\textit{report!} = \textit{MMinvalidThreadAC} \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{RThreadMakeBS} == \textit{ThreadMakeBS} \lor \textit{InvalidThreadAC} \\
\textit{RThreadClearBS} == \textit{ThreadClearBS} \lor \textit{InvalidThreadAC} \\
\textit{RThreadAllocateMemory} == \textit{ThreadAllocateMemory} \lor \textit{InvalidThreadAC} \\
\textit{RThreadGetAC} == (\textit{ThreadGetAC} \land \textit{Success}) \lor \textit{InvalidThreadAC} \\
\textit{RThreadSetAC} == (\textit{ThreadSetAC} \land \textit{Success}) \lor \textit{InvalidThreadAC}
\end{array}$$

The operations to add and remove threads do not report errors as they are only used internally by the SCJVM scheduler.

## 1.5 Stack Memory Manager

In addition to providing facilities for backing stores and memory allocation, the SCJVM must allow for allocating thread stacks. The stacks should be allocated in an area separate from the root backing store, set aside for the allocation of stacks when the SCJVM starts. The SCJVM memory management need only provide the memory for the stacks, management of the stack contents must be handled by the core execution environment.

The stack area is a memory block that holds additional information about allocated stacks so that they can be deallocated when the thread is removed. Thread stacks may have their own memory overhead associated with them.

$$stackOverhead : \mathbb{N}$$

The stack memory manager is a memory block with a function mapping stack identifiers to the memory of the associated stack. The memory allocated for stacks must partition the used stack memory.

$$\begin{array}{l}
\textit{StackMemoryManager} \\
\hline
MemoryBlock \\
stacks : StackID \pfun ContiguousMemory \\
\hline
stacks \text{ partition } used
\end{array}$$

The stack manager is initialised with a given area of memory for allocating stacks and initially has no stacks allocated.

$$\begin{array}{l}
\textit{StackMemoryManagerInit} \\
\hline
StackMemoryManager\,' \\
stackSpace? : ContiguousMemory \\
\hline
total' = stackSpace? \\
free' \subseteq total' \\
used' = \varnothing \\
stacks' = \varnothing
\end{array}$$

It must be possible to create a new stack of a given size. This operation behaves as *RMBAllocate* except that the stack overhead must be taken into account and the new stack must be stored with its identifier. The new stack's identifier must be one not already in use.

$$\begin{array}{l}
\textit{StackCreate} \\
\hline
\Delta StackMemoryManager \\
size? : \mathbb{N} \\
newStack! : StackID \\
\hline
newStack! \notin \operatorname{dom} stacks \\
\exists\, actualSize : \mathbb{N} \mid actualSize = size? + stackOverhead \bullet \\
\exists\, report! : MMReport;\ allocated! : ContiguousMemory \bullet \\
\quad RMBAllocate[actualSize / size?] \land report! = MMokay \land \\
\quad stacks' = stacks \oplus \{newStack! \mapsto allocated!\}
\end{array}$$

It must also be possible to delete a stack, freeing the memory used for it.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StackDelete}} \\ \hline
\Delta \mathit{StackMemoryManager} \\
\mathit{stack}? : \mathit{StackID} \\ \hline
\mathit{stack}? \in \operatorname{dom} \mathit{stacks} \\
\mathit{used}' = \mathit{used} \setminus \mathit{stacks}\ \mathit{stack}? \\
\mathit{free}' = \mathit{free} \cup \mathit{stacks}\ \mathit{stack}? \in \mathit{ContiguousMemory} \\
\mathit{stacks}' = \{\mathit{stack}?\} \ndres \mathit{stacks} \\
\mathit{total}' = \mathit{total} \\ \hline
\end{array}$$

The stack memory manager operations must then be made into robust operations. This requires some new error handling schemas to deal with additional errors that could occur.

It must be reported if an action is attempted on a nonexistent stack.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StackNonexistentStack}} \\ \hline
\Xi \mathit{StackMemoryManager} \\
\mathit{stack}? : \mathit{StackID} \\
\mathit{report}! : \mathit{MMReport} \\ \hline
\mathit{stack}? \notin \operatorname{dom} \mathit{stacks} \\
\mathit{report}! = \mathit{MMnonexistentStack} \\ \hline
\end{array}$$

It must be reported if an allocation attempt returns an error.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StackMBAllocateError}} \\ \hline
\Xi \mathit{StackMemoryManager} \\
\mathit{size}? : \mathbb{N} \\
\mathit{report}! : \mathit{MMReport} \\ \hline
\exists \mathit{actualSize} : \mathbb{N} \mid \mathit{actualSize} = \mathit{size}? + \mathit{stackOverhead} \bullet \\
\exists \mathit{allocated}! : \mathit{ContiguousMemory} \bullet \\
\qquad \mathit{RMBAllocate}[\mathit{actualSize}/\mathit{size}?] \land \mathit{report}! \neq \mathit{MMokay} \\ \hline
\end{array}$$

It must be reported if deallocation of a stack would cause fragmentation.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{StackFragmentation}} \\ \hline
\Xi \mathit{StackMemoryManager} \\
\mathit{stack}? : \mathit{StackID} \\
\mathit{report}! : \mathit{MMReport} \\ \hline
\mathit{stack}? \in \operatorname{dom} \mathit{stacks} \\
\mathit{free} \cup \mathit{stacks}\ \mathit{stack}? \notin \mathit{ContiguousMemory} \\
\mathit{report}! = \mathit{MMfragmentation} \\ \hline
\end{array}$$

The stack memory manager operations can then be made into robust operations.

$$\mathit{RStackCreate} \ == \ (\mathit{StackCreate} \land \mathit{Success}) \lor \mathit{StackMBAllocateError}$$

$$\mathit{RStackDelete} \ == \ (\mathit{StackDelete} \land \mathit{Success}) \lor \mathit{StackNonexistentStack} \lor \mathit{StackFragmentation}$$

| Operation | Inputs | Outputs | Error Conditions |
|---|---|---|---|
| `getRootBackingStore` | (none) | backing store identifier | (none) |
| `getCurrentAllocationContext` | (none) | backing store identifier | no current thread allocation context |
| `setCurrentAllocationContext` | backing store identifier | (none) | invalid identifier<br>no current thread allocation context |
| `getTotalSize` | backing store identifier | size in bytes | invalid identifier |
| `getUsedSize` | backing store identifier | size in bytes | invalid identifier |
| `getFreeSize` | backing store identifier | size in bytes | invalid identifier |
| `findBackingStore` | memory pointer | backing store identifier | no backing store found |
| `allocateMemory` | size in bytes | memory pointer | insufficient free memory<br>no current thread allocation context |
| `makeBackingStore` | size in bytes | backing store identifier | insufficient free memory<br>no current thread allocation context |
| `clearCurrentAllocationContext` | (none) | (none) | nested backing store in use<br>no current thread allocation context |
| `resizeBackingStore` | backing store identifier<br>size in bytes | (none) | invalid identifier<br>backing store is root<br>backing store not empty<br>backing store not only child<br>insufficient free space<br>no space for memory overhead |
| `createStack` | size in bytes | stack identifier | insufficient free space |
| `destroyStack` | stack identifier | (none) | invalid identifier<br>stack space fragmentation |

Table 1: The operations of the SCJVM memory manager

## 1.6 Memory Manager Operations

The model of the memory manager offers the services detailed in this section. Not all of the services are made available to the user but are instead used by other parts of the SCJVM. The operations that are made available to the user are summarised in Table 1.

The SCJVM has little error checking — it is the responsibility of the infrastructure to ensure that the correct allocation context is set. It is the responsibility of the compiler or interpreter to ensure that allocated memory is used in a safe way. However, each memory management function must output an implementation-defined error code to indicate whether the operation could not be performed or whether it completed successfully. This error code could be implemented as a global flag rather than being passed as a return from the function. The conditions that cause an error to be reported are listed in the final column of Table 1. Operations used internally do not report error values.

The state of the memory manager process is made up of both the global memory manager and the stack memory manager.

$$\textbf{state}\ \mathit{GlobalMemoryManager} \land \mathit{StackMemoryManager}$$

The memory manager is initialised by using the root backing store and stack space as inputs and applying the initialisation schemas for both the global memory manager and the stack memory manager. The error value from the global memory manager initialisation is reported.

$$\begin{aligned}
&\mathit{Init} \mathrel{\widehat=} \textbf{var}\ \mathit{report} : \mathit{MMReport} \bullet \\
&\quad \mathit{MMinit}?\mathit{addresses}?\mathit{stackSpace} \longrightarrow \\
&\qquad \big(\mathit{RGlobalMemoryManagerInit} \land \mathit{StackMemoryManagerInit}\big); \\
&\quad \mathit{MMreport}!\mathit{report} \longrightarrow \textbf{Skip}
\end{aligned}$$

The operation `getRootBackingStore` allows for getting the identifier of the root backing store. It takes no inputs and gives the identifier as its output. This operation always succeeds.

$$\begin{array}{l} GetRootBackingStore \mathrel{\widehat=} \mathbf{var}\ report : MMReport;\ rbs : BackingStoreID \bullet \\ \quad MMgetRootBackingStore \longrightarrow \big(RGlobalGetRootBackingStore\big); \\ \quad MMgetRootBackingStoreRet!rbs \longrightarrow MMreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `getCurrentAllocationContext` allows for getting the identifier of the current allocation context of the current thread. The current thread is obtained from the processor and passed into the operation. This operation takes no inputs and outputs the allocation context identifier. This operation fails if no allocation context is set for the current thread but that should not happen if the scheduler behaves correctly.

$$\begin{array}{l} GetCurrentAllocationContext \mathrel{\widehat=} \mathbf{var}\ ac : BackingStoreID;\ report : MMReport \bullet \\ \quad MMgetCurrentAllocationContext \longrightarrow ScurrentThread?thread \longrightarrow \big(RThreadGetAC\big); \\ \quad MMgetCurrentAllocationContextRet!ac \longrightarrow MMreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `setCurrentAllocationContext` allows for setting the allocation context of the current thread. It takes as an input the identifier of the new allocation context and does not output anything. This operation fails if the current thread has not been registered with the memory manager but that should not happen if the scheduler behaves correctly. The operation can also fail if the provided backing store identifier is not a valid backing store identifier.

$$\begin{array}{l} SetCurrentAllocationContext \mathrel{\widehat=} \mathbf{var}\ report : MMReport \bullet \\ \quad MMsetCurrentAllocationContext?ac \longrightarrow ScurrentThread?thread \longrightarrow \big(RThreadSetAC\big); \\ \quad MMreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `getTotalSize` allows for obtaining the total size of a given backing store. It takes the backing store's identifier as an input and outputs its total size in bytes. This operation fails if the given identifier does not identify a valid backing store.

$$\begin{array}{l} GetTotalSize \mathrel{\widehat=} \mathbf{var}\ report : MMReport;\ size : \mathbb{N} \bullet \\ \quad MMgetTotalSize?bs \longrightarrow \big(RGlobalGetTotalSize[bs/ac?]\big); \\ \quad MMgetTotalSizeRet!size \longrightarrow MMreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `getUsedSize` allows for obtaining the used size of a given backing store. It takes the backing store's identifier as an input and outputs its used size in bytes. This operation fails if the given identifier does not identify a valid backing store.

$$\begin{array}{l} GetUsedSize \mathrel{\widehat=} \mathbf{var}\ report : MMReport;\ size : \mathbb{N} \bullet \\ \quad MMgetUsedSize?bs \longrightarrow \big(RGlobalGetUsedSize[bs/ac?]\big); \\ \quad MMgetUsedSizeRet!size \longrightarrow MMreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `getFreeSize` allows for obtaining the free size of a given backing store. It takes the backing store's identifier as an input and outputs its free size in bytes. This operation fails if the given identifier does not identify a valid backing store.

$$\begin{array}{l} GetFreeSize \mathrel{\widehat=} \mathbf{var}\ report : MMReport;\ size : \mathbb{N} \bullet \\ \quad MMgetFreeSize?bs \longrightarrow \big(RGlobalGetFreeSize[bs/ac?]\big); \\ \quad MMgetFreeSizeRet!size \longrightarrow MMreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `findBackingStore` allows for finding the backing store in which a given memory address is located, for implementing the `getMemoryArea` method of `MemoryArea`. It takes as input the memory address in question and outputs the identifier of the innermost nested backing store that contains the address. This operation fails if there is no backing store that contains the given address (i.e. if the address is outside the root backing store).

$$\begin{array}{l} \mathit{FindBackingStore} \mathrel{\widehat=} \mathbf{var}\ \mathit{backingStore} : \mathit{BackingStoreID};\ \mathit{report} : \mathit{MMReport} \bullet \\ \quad \mathit{MMfindBackingStore}?\mathit{address} \longrightarrow \big(\mathit{RGlobalFindAddress}\big); \\ \quad \mathit{MMfindBackingStoreRet}!\mathit{backingStore} \longrightarrow \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The operation `allocateMemory` allows for allocating memory in the allocation context of the current thread in the manner of the `new` bytecode instruction. It takes as input the required size in bytes of the memory allocation and outputs the address of the start of the allocated memory plus any overhead. This operation fails if there is insufficient free memory to meet the allocation request. It may also fail if the current thread has not been registered with the memory manager but that should not happen if the scheduler behaves correctly.

$$\begin{array}{l} \mathit{AllocateMemory} \mathrel{\widehat=} \mathbf{var}\ \mathit{address} : \mathit{MemoryAddress};\ \mathit{report} : \mathit{MMReport} \bullet \\ \quad \mathit{MMallocateMemory}?\mathit{size} \longrightarrow \mathit{ScurrentThread}?\mathit{thread} \longrightarrow \big(\mathit{RThreadAllocateMemory}\big); \\ \quad \mathit{MMallocateMemoryRet}!\mathit{address} \longrightarrow \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The operation `makeBackingStore` allows for the creation of a new backing store nested within the allocation context of the current thread. It takes as input the required size of the backing store in bytes and outputs the identifier of the newly created backing store. This operation fails if there is insufficient free memory to create the new backing store. It may also fail if the current thread has not been registered with the memory manager but that should not happen if the scheduler behaves correctly.

$$\begin{array}{l} \mathit{MakeBackingStore} \mathrel{\widehat=} \mathbf{var}\ \mathit{childID} : \mathit{BackingStoreID};\ \mathit{report} : \mathit{MMReport} \bullet \\ \quad \mathit{MMmakeBackingStore}?\mathit{size} \longrightarrow \mathit{ScurrentThread}?\mathit{thread} \longrightarrow \big(\mathit{RThreadMakeBS}\big); \\ \quad \mathit{MMmakeBackingStoreRet}!\mathit{childID} \longrightarrow \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The operation `clearCurrentAllocationContext` allows for clearing the allocation context of the current thread, removing all backing stores and memory allocations from it. It takes no inputs and gives no outputs. This operation fails if any children (direct or indirect) of the backing store to be cleared are in use by a thread but this should not happen if the infrastructure is behaving correctly. It may also fail if the current thread has not been registered with the memory manager but that should not happen if the scheduler behaves correctly.

$$\begin{array}{l} \mathit{ClearCurrentAllocationContext} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{MMReport} \bullet \\ \quad \mathit{MMclearCurrentAllocationContext} \longrightarrow \mathit{ScurrentThread}?\mathit{thread} \longrightarrow \big(\mathit{RThreadClearBS}\big); \\ \quad \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The operation `resizeBackingStore` allows for resizing of a given backing store to a new size. It takes as input the identifier of the backing store to be resized and the requested new size in bytes, and outputs the new identifier of the backing store (since it may have changed in resizing). This operation fails if

- the given identifier is not a valid backing store identifier,
- the backing store to be resized is the root backing store,
- the backing store to be resized is not empty,
- the backing store to be resized is not the only backing store nested within its parent
- there is insufficient free space to accommodate the new size, or
- the new size is too small to fit the backing store overhead.

Most of these requirements should be met if the infrastructure is behaving properly as the only times backing stores need resizing is the resizing of mission memory prior to starting a new mission and resizing nested private memory prior to reentering it.

$$
\begin{array}{l}
\mathit{ResizeBackingStore} \mathrel{\widehat=} \mathbf{var}\ \mathit{newID} : \mathit{BackingStoreID};\ \mathit{report} : \mathit{MMReport} \bullet \\
\quad \mathit{MMresizeBackingStore}?\mathit{toResize}?\mathit{newSize} \longrightarrow \left(\mathit{RGlobalResizeBS}\right); \\
\quad \mathit{MMresizeBackingStoreRet}!\mathit{newID} \longrightarrow \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip}
\end{array}
$$

The operation `createStack` allows for allocating memory for stacks. It takes as input the required stack size in bytes and outputs the identifier of the stack. This operation fails if there is insufficient space in the stack area to fulfil the requested allocation.

$$
\begin{array}{l}
\mathit{CreateStack} \mathrel{\widehat=} \mathbf{var}\ \mathit{newStack} : \mathit{StackID};\ \mathit{report} : \mathit{MMReport} \bullet \\
\quad \mathit{MMcreateStack}?\mathit{size} \longrightarrow \left(\mathit{RStackCreate}\right); \\
\quad \mathit{MMcreateStackRet}!\mathit{newStack} \longrightarrow \mathit{MMreport}!\mathit{report} \longrightarrow \mathbf{Skip}
\end{array}
$$

The operation `destroyStack` removes the stack with the given identifier, deallocating the space for it. It takes as input the identifier of the stack to be destroyed and gives no output. This operation fails if the removal of the stack would cause the stack space to become fragmented but that should not happen since threads are created and destroyed all at once when a mission starts and ends, and missions starting and ending do not overlap (at level 2 a nested mission finishes before its parent mission). The operation can also fail if the given identifier is not a valid stack identifier.

$$
\begin{array}{l}
\mathit{DestroyStack} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{MMReport} \bullet \\
\quad \mathit{MMdestroyStack}?\mathit{stack} \longrightarrow \left(\mathit{RStackDelete}\right); \\
\quad \mathit{MMreport}!\mathit{report} \longrightarrow \mathit{Skip}
\end{array}
$$

The operation `addThread` is used by the scheduler to register a thread with the memory manager. It takes as input the identifier of the thread to be added and the initial allocation context of the thread, and gives no output. As this is an internal operation it does not fail or report any errors; a thread's allocation context is silently replaced if it is already registered with the memory manger but that should not happen if the scheduler is behaving correctly.

$$
\begin{array}{l}
\mathit{AddThread} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{MMReport} \bullet \\
\quad \mathit{MMaddThread}?\mathit{thread}?\mathit{ac} \longrightarrow \left(\mathit{ThreadAdd}\right)
\end{array}
$$

The operation `removeThread` is used by the scheduler to deregister a thread from the memory manager. It takes as input the identifier of the thread to be removed and gives no output. As this is an internal operation it does not fail or report any errors; this operation does nothing if the given thread identifier is not registered with the memory manager.

$$
\begin{array}{l}
\mathit{RemoveThread} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{MMReport} \bullet \\
\quad \mathit{MMremoveThread}?\mathit{thread} \longrightarrow \left(\mathit{ThreadRemove}\right)
\end{array}
$$

The memory manager must be initialised before any operations can be used. After initialisation any operation can be used once the previous operation has completed.

$$
\begin{array}{l}
\mathit{Loop} \mathrel{\widehat=} \mathit{GetRootBackingStore} \\
\quad \Box\, \mathit{GetCurrentAllocationContext} \Box \mathit{SetCurrentAllocationContext} \\
\quad \Box\, \mathit{GetTotalSize} \Box \mathit{GetUsedSize} \Box \mathit{GetFreeSize} \\
\quad \Box\, \mathit{FindBackingStore} \Box \mathit{AllocateMemory} \Box \mathit{MakeBackingStore} \\
\quad \Box\, \mathit{ClearCurrentAllocationContext} \Box \mathit{ResizeBackingStore} \\
\quad \Box\, \mathit{CreateStack} \Box \mathit{DestroyStack} \Box \mathit{AddThread} \Box \mathit{RemoveThread}; \\
\quad \mathit{Loop}
\end{array}
$$

$$\bullet\ \mathit{Init}\ ;\ \mathit{Loop}$$

$$\mathbf{end}$$

# 2 Scheduler

$\textbf{section } \mathit{scheduler} \textbf{ parents } \mathit{memorymanager}$

The SCJVM schedulers must manage separate threads of execution, which are identified by unique implementation-defined thread identifiers of the *ThreadID* type, which was defined in the last section.

Threads are scheduled according to their priorities. Priorities are divided into hardware priorities, which are used for interrupt handlers, and software priorities, which are used for threads. There must be support for at least 28 priorities, with hardware priorities being higher than software priorities. One software priority must be designated as the normal priority.

$$
\begin{array}{|l}
minHwPriority, maxHwPriority : \mathbb{N} \\
minSwPriority, maxSwPriority : \mathbb{N} \\
normSwPriority : \mathbb{N} \\
\hline
(maxHwPriority - minHwPriority) + (maxSwPriority - minSwPriority) \geq 28 \\
minSwPriority < maxSwPriority < minHwPriority < maxHwPriority \\
minSwPriority \leq normSwPriority \leq maxSwPriority
\end{array}
$$

$$
\begin{aligned}
&ThreadPriority == minSwPriority \mathrel{..} maxSwPriority \\
&InterruptPriority == minHwPriority \mathrel{..} maxHwPriority \\
&Priority == ThreadPriority \cup InterruptPriority
\end{aligned}
$$

The threads represent threads of execution of Java bytecode programs in the core execution environment. While the scheduler is mostly agnostic to how the programs are run, the scheduler must store some program counter value, which is of some type *ProgramAddress*.

$[ProgramAddress]$

The scheduler must be able to communicate with the core execution environment to get and set the program counter value in order to handle thread switches. Similarly, the stack used by the core execution environment must be able to be set. These interactions with the core execution environment are handled by the following channels.

$$
\begin{aligned}
&\textbf{channel } CEEgetProgramCounter, CEEsetProgramCounter : ProgramAddress \\
&\textbf{channel } CEEsetStack : StackID
\end{aligned}
$$

Also, while the concept of objects is mainly handled by the core execution environment, the scheduler must have some notion of object identifiers in order to manage locks on objects. This is provided by the *ObjectID* type, which may simply represent an opaque pointer to the object with no checking that what is at the pointer is valid.

$[ObjectID]$

The SCJVM scheduler also manages interrupts, which also have unique identifiers. The precise set of identifiers will likely depend on what interrupt vectors the hardware offers.

$[InterruptID]$

The hardware interrupts, along with the interrupt identifier, are received through the *HWinterrupt* channel. The hardware is also required to offer channels for enabling and disabling interrupts.

$$
\begin{aligned}
&\textbf{channel } HWinterrupt : InterruptID \\
&\textbf{channel } HWenableInterrupts, HWdisableInterrupts
\end{aligned}
$$

Although it is mainly left implementation-defined as to what interrupts are offered, it is required in this model that there is a clock interrupt that is fired at regular intervals. This interrupt is not directly handled by the scheduler but is instead used by the real-time clock described in the next section.

$$\mid \quad \mathit{clockInterrupt} : \mathit{InterruptID}$$

When the real-time clock has an alarm trigger, it passes on the clock interrupt to the scheduler to run a handler for it. In this model that is represented by the *RTCclockInterrupt* channel.

$$\mathbf{channel}\ \mathit{RTCclockInterrupt}$$

The SCJVM scheduler offers services that, in this model, are accessed through the following channels.

$$
\begin{array}{l}
\mathbf{channel}\ \mathit{Sinit} : \mathit{StackID} \times \mathit{ThreadPriority} \times \mathit{ProgramAddress} \times \mathit{BackingStoreID} \\
\mathbf{channel}\ \mathit{SgetMaxSoftwarePriority} : \mathit{Priority} \\
\mathbf{channel}\ \mathit{SgetMinSoftwarePriority} : \mathit{Priority} \\
\mathbf{channel}\ \mathit{SgetNormSoftwarePriority} : \mathit{Priority} \\
\mathbf{channel}\ \mathit{SgetMaxHardwarePriority} : \mathit{Priority} \\
\mathbf{channel}\ \mathit{SgetMinHardwarePriority} : \mathit{Priority} \\
\mathbf{channel}\ \mathit{SgetMainThread} : \mathit{ThreadID} \\
\mathbf{channel}\ \mathit{SmakeThread} : \mathit{ProgramAddress} \times \mathit{ThreadPriority} \times \mathit{BackingStoreID} \times \mathit{StackID} \\
\mathbf{channel}\ \mathit{SmakeThreadRet} : \mathit{ThreadID} \\
\mathbf{channel}\ \mathit{SstartThread} : \mathit{ThreadID} \\
\mathbf{channel}\ \mathit{SgetCurrentThread} : \mathit{ThreadID} \\
\mathbf{channel}\ \mathit{SdestroyThread} : \mathit{ThreadID} \\
\mathbf{channel}\ \mathit{SsuspendThread} \\
\mathbf{channel}\ \mathit{SresumeThread} : \mathit{ThreadID} \\
\mathbf{channel}\ \mathit{SsetPriorityCeiling} : \mathit{ObjectID} \times \mathit{Priority} \\
\mathbf{channel}\ \mathit{StakeLock} : \mathit{ObjectID} \\
\mathbf{channel}\ \mathit{SreleaseLock} : \mathit{ObjectID} \\
\mathbf{channel}\ \mathit{SattachInterruptHandler} : \mathit{InterruptID} \times \mathit{ProgramAddress} \times \mathit{BackingStoreID} \\
\mathbf{channel}\ \mathit{SdetachInterruptHandler} : \mathit{InterruptID} \\
\mathbf{channel}\ \mathit{SgetInterruptPriority} : \mathit{InterruptID} \\
\mathbf{channel}\ \mathit{SgetInterruptPriorityRet} : \mathit{InterruptPriority} \\
\mathbf{channel}\ \mathit{SdisableInterrupts}, \mathit{SenableInterrupts} \\
\mathbf{channel}\ \mathit{SendInterrupt}
\end{array}
$$

As with the memory manager operations, each operation of the scheduler reports whether or not an error occurred and, if so, what error. These error values are of type *SReport* and are reported over the channel *Sreport*.

$$
\begin{array}{l}
\mathit{SReport} ::= \mathit{Sokay} \mid \mathit{SnonexistentThread} \mid \mathit{SthreadAlreadyStarted} \mid \\
\quad \mathit{SthreadNotBlocked} \mid \mathit{SthreadNotBlockable} \mid \mathit{SthreadNotDestroyable} \mid \\
\quad \mathit{SlockAlreadyTaken} \mid \mathit{SlockNotHeld} \mid \mathit{SthreadHoldingLocks} \mid \mathit{SnotInInterrupt}
\end{array}
$$

$$\mathbf{channel}\ \mathit{Sreport} : \mathit{SReport}$$

Finally, we require a boolean datatype for part of the scheduler specification (and for the real-time clock specification as well).

$$\mathit{Bool} ::= \mathit{True} \mid \mathit{False}$$

With the channels and datatypes declared, we begin the process declaration.

$$\mathbf{process}\ \mathit{Scheduler} \mathrel{\widehat=} \mathbf{begin}$$

## 2.1 Threads

The SCJVM scheduler schedules threads and stores information about them. The scheduler stores the stack identifier for each thread and the address of the program counter for that thread. Each thread also has a base and current priority (the current priority may change due to the priority ceiling emulation system described later). It is required that the current priority should not be less than the base priority as the priority can only be temporarily raised, not lowered. Additionally, each thread has a, possibly shared, allocation context but they are handled in the memory manager.

$$
\begin{array}{|l}
\textit{ThreadInfo} \\
\hline
\mathit{threadStack} : \mathit{ThreadID} \pfun \mathit{StackID} \\
\mathit{programCounter} : \mathit{ThreadID} \pfun \mathit{ProgramAddress} \\
\mathit{currentPriority} : \mathit{ThreadID} \pfun \mathit{Priority} \\
\mathit{basePriority} : \mathit{ThreadID} \pfun \mathit{Priority} \\
\hline
\operatorname{dom} \mathit{threadStack} = \operatorname{dom} \mathit{currentPriority} \\
\qquad = \operatorname{dom} \mathit{basePriority} = \operatorname{dom} \mathit{programCounter} \\
\forall t : \operatorname{dom} \mathit{currentPriority} \bullet \mathit{currentPriority}\ t \geq \mathit{basePriority}\ t \\
\hline
\end{array}
$$

An SCJVM thread may at any time be either waiting to be start, be started but not running (because a higher priority thread is running), be blocked or be the currently running thread. In addition to identifiers of threads in these states, there is also a set of free thread identifiers. There must also be an idle thread that is always available to run and separate from the other thread states (though it may be the current thread). One thread must be designated as the main thread, which cannot be freed or stopped, only blocked or preempted.

$$
\begin{array}{|l}
\textit{ThreadManager} \\
\hline
\mathit{free}, \mathit{created}, \mathit{started}, \mathit{blocked} : \mathbb{P}\, \mathit{ThreadID} \\
\mathit{current} : \mathit{ThreadID} \\
\mathit{main}, \mathit{idle} : \mathit{ThreadID} \\
\hline
\langle \mathit{free}, \mathit{created}, \mathit{started}, \mathit{blocked}, \{\mathit{current}, \mathit{idle}\} \rangle \text{ partition } \mathit{ThreadID} \\
\mathit{main} \in \mathit{started} \cup \mathit{blocked} \cup \{\mathit{current}\} \\
\mathit{main} \neq \mathit{idle} \\
\hline
\end{array}
$$

Initially all threads identifier are free, except the ones used for the idle and main threads. The main thread is initially running and there are no threads in any of the other states.

$$
\begin{array}{|l}
\textit{ThreadManagerInit} \\
\hline
\mathit{ThreadManager}' \\
\hline
\mathit{free}' = \mathit{ThreadID} \setminus \{\mathit{idle}', \mathit{main}'\} \\
\mathit{created}' = \mathit{started}' = \mathit{blocked}' = \varnothing \\
\mathit{current}' = \mathit{main}' \\
\mathit{main}' \neq \mathit{idle}' \\
\hline
\end{array}
$$

## 2.2 Priority Scheduler

The SCJVM scheduler is a preemptive priority scheduler that must behave as if there is a queue of threads for each priority implemented (the queues are conceptual and do not place any requirements on implementation). Only running threads must be in the priority queues (when a thread is blocked it is removed from the queues) and the idle thread is never in any of the queues. No thread may be in more than one queue. Some threads are the threads of interrupt handlers and the identifiers of such threads are recorded in the set *interruptThreads*. Interrupt threads cannot be blocked or waiting to start.

$$\begin{array}{l}
\textit{Scheduler} \\
\hline
\quad \textit{ThreadInfo} \\
\quad \textit{ThreadManager} \\
\quad \textit{priorityQueues} : \textit{Priority} \to \operatorname{iseq} \textit{ThreadID} \\
\quad \textit{interruptThreads} : \mathbb{P}\, \textit{ThreadID} \\
\hline
\quad \bigcup \{ q : \operatorname{ran} \textit{priorityQueues} \bullet \operatorname{ran} q \} = (\textit{started} \cup \{\textit{current}\}) \setminus \{\textit{idle}\} \\
\quad \operatorname{disjoint}(\lambda\, p : \textit{Priority} \bullet \operatorname{ran}(\textit{priorityQueues}\ p)) \\
\quad \operatorname{dom} \textit{currentPriority} = \textit{created} \cup \textit{started} \cup \textit{blocked} \cup \{\textit{current}, \textit{idle}\} \\
\quad \forall\, p : \textit{Priority} \bullet \forall\, t : \operatorname{ran}(\textit{priorityQueues}\ p) \bullet \textit{currentPriority}\ t = p \\
\quad \textit{interruptThreads} \subseteq \textit{started} \cup \{\textit{current}\} \\
\hline
\end{array}$$

Initially only the main and idle threads exist. The main thread is initialised with information supplied at SCJVM startup that points to a program that will set up the safelet and mission sequencer. The idle thread uses the same stack as the main thread and has the lowest possible priority. The program counter value for the idle thread should be some implementation-defined value that points to a program that does nothing (perhaps entering an infinite loop) and does not affect the stack. With the exception of the queue that contains the main thread, all priority queues are initially empty. The are initially no interrupt threads.

$$\begin{array}{l}
\textit{SchedulerInit} \\
\hline
\quad \textit{Scheduler}' \\
\quad \textit{ThreadManagerInit} \\
\quad \textit{mainStack}? : \textit{StackID} \\
\quad \textit{mainPriority}? : \textit{ThreadPriority} \\
\quad \textit{mainEntry}? : \textit{ProgramAddress} \\
\hline
\quad \textit{threadStack}' = \{\textit{main}' \mapsto \textit{mainStack}?, \textit{idle}' \mapsto \textit{mainStack}?\} \\
\quad \textit{currentPriority}' = \textit{basePriority}' = \{\textit{main}' \mapsto \textit{mainPriority}?, \textit{idle}' \mapsto \textit{minSwPriority}\} \\
\quad \exists\, \textit{idleEntry} : \textit{ProgramAddress} \bullet \\
\qquad \textit{programCounter}' = \{\textit{main}' \mapsto \textit{mainEntry}?, \textit{idle}' \mapsto \textit{idleEntry}\} \\
\quad \textit{priorityQueues}'\ \textit{mainPriority}? = \langle \textit{main}' \rangle \\
\quad \forall\, p : \textit{Priority} \mid p \neq \textit{mainPriority}? \bullet \textit{priorityQueues}'\ p = \langle\rangle \\
\quad \textit{interruptThreads}' = \varnothing \\
\hline
\end{array}$$

When a new thread is created, its entry point, priority and stack are stored. The new thread is given an identifier from the *free* identifier set and placed in the *created* set. The thread is not started immediately as threads within missions do not start until mission initialisation is finished.

$$\begin{array}{l}
\textit{ThreadCreate} \\
\hline
\quad \Delta \textit{Scheduler} \\
\quad \textit{entryPoint}? : \textit{ProgramAddress} \\
\quad \textit{priority}? : \textit{ThreadPriority} \\
\quad \textit{stack}? : \textit{StackID} \\
\quad \textit{newID}! : \textit{ThreadID} \\
\hline
\quad \textit{newID}! \in \textit{free} \\
\quad \textit{threadStack}' = \textit{threadStack} \oplus \{\textit{newID}! \mapsto \textit{stack}?\} \\
\quad \textit{currentPriority}' = \textit{currentPriority} \oplus \{\textit{newID}! \mapsto \textit{priority}?\} \\
\quad \textit{basePriority}' = \textit{basePriority} \oplus \{\textit{newID}! \mapsto \textit{priority}?\} \\
\quad \textit{programCounter}' = \textit{programCounter} \oplus \{\textit{newID}! \mapsto \textit{entryPoint}?\} \\
\quad \textit{free}' = \textit{free} \setminus \{\textit{newID}!\} \\
\quad \textit{created}' = \textit{created} \cup \{\textit{newID}!\} \\
\quad \textit{started}' = \textit{started} \land \textit{blocked}' = \textit{blocked} \land \textit{current}' = \textit{current} \\
\quad \textit{main}' = \textit{main} \land \textit{idle}' = \textit{idle} \\
\quad \textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}$$

When a thread is started, it is added to the back of the queue for its priority. Here we take the start of the sequence to be the back of a queue so that the distributed concatenation operator will place the threads near the front of the queue higher than those further back while also preserving priority ordering.

$$
\begin{array}{l}
\textit{ThreadStart} \\
\hline
\Delta \textit{Scheduler} \\
\Xi \textit{ThreadInfo} \\
\textit{thread}? : \textit{ThreadID} \\
\hline
\textit{thread}? \in \textit{created} \\
\textit{created}' = \textit{created} \setminus \{\textit{thread}?\} \\
\textit{started}' = \textit{started} \cup \{\textit{thread}?\} \\
\textit{free}' = \textit{free} \land \textit{blocked}' = \textit{blocked} \land \textit{current}' = \textit{current} \\
\textit{main}' = \textit{main} \land \textit{idle}' = \textit{idle} \\
\textit{priorityQueues}' = \textit{priorityQueues} \oplus \\
\qquad \{\textit{currentPriority thread}? \mapsto \langle \textit{thread}? \rangle \frown \textit{priorityQueues}\,(\textit{currentPriority thread}?)\} \\
\textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

When a new thread must be chosen to run, the thread at the front of the highest priority queue is chosen. If there are no threads in the priority queues then the idle thread runs. The old running thread is put back in the *started* thread set and the new running thread is removed. The idle thread is also removed from the started threads in case it was the old running thread. The program counter of the old thread is stored, and the program counter and stack for the new thread are output.

$$
\begin{array}{l}
\textit{RunThread} \\
\hline
\Delta \textit{Scheduler} \\
\textit{oldPC}?, \textit{newPC}! : \textit{ProgramAddress} \\
\textit{newStack}! : \textit{StackID} \\
\hline
\textit{current}' = \textit{last}\,(\langle \textit{idle} \rangle \frown \frown/\, \textit{priorityQueues}) \\
\textit{started}' = (\textit{started} \cup \{\textit{current}\}) \setminus \{\textit{current}', \textit{idle}\} \\
\textit{programCounter}' = \textit{programCounter} \oplus \{\textit{current} \mapsto \textit{oldPC}?\} \\
\textit{newPC}! = \textit{programCounter}'\ \textit{current}' \\
\textit{newStack}! = \textit{threadStack}\ \textit{current}' \\
\textit{free}' = \textit{free} \land \textit{created}' = \textit{created} \land \textit{blocked}' = \textit{blocked} \\
\textit{priorityQueues}' = \textit{priorityQueues} \\
\textit{currentPriority}' = \textit{currentPriority} \land \textit{basePriority}' = \textit{basePriority} \\
\textit{threadStack}' = \textit{threadStack} \\
\textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

A thread may be destroyed if it exists and is not the main thread, the idle thread, an interrupt thread or the current thread. Destroying the thread puts its identifier in the *free* set, removes its information from the scheduler, and removes it from the priority queues.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ThreadDestroy}} \\
\Delta \mathit{Scheduler} \\
\mathit{thread?} : \mathit{ThreadID} \\
\hline
\mathit{thread?} \notin \mathit{free} \\
\mathit{thread?} \neq \mathit{main} \land \mathit{thread?} \neq \mathit{idle} \land \mathit{thread?} \neq \mathit{current} \\
\mathit{thread?} \notin \mathit{interruptThreads} \\
\mathit{threadStack}' = \{\mathit{thread?}\} \ndres \mathit{threadStack} \\
\mathit{currentPriority}' = \{\mathit{thread?}\} \ndres \mathit{currentPriority} \\
\mathit{basePriority}' = \{\mathit{thread?}\} \ndres \mathit{basePriority} \\
\mathit{programCounter}' = \{\mathit{thread?}\} \ndres \mathit{programCounter} \\
\mathit{free}' = \mathit{free} \cup \{\mathit{thread?}\} \\
\mathit{created}' = \mathit{created} \setminus \{\mathit{thread?}\} \\
\mathit{started}' = \mathit{started} \setminus \{\mathit{thread?}\} \\
\mathit{blocked}' = \mathit{blocked} \setminus \{\mathit{thread?}\} \\
\mathit{priorityQueues}'\,(\mathit{currentPriority}\ \mathit{thread?}) = \\
\qquad \mathit{priorityQueues}\,(\mathit{currentPriority}\ \mathit{thread?}) \nrres \{\mathit{thread?}\} \\
\{\mathit{currentPriority}\ \mathit{thread?}\} \ndres \mathit{priorityQueues}' = \\
\qquad \{\mathit{currentPriority}\ \mathit{thread?}\} \ndres \mathit{priorityQueues} \\
\mathit{current}' = \mathit{current} \\
\mathit{main}' = \mathit{main} \land \mathit{idle}' = \mathit{idle} \\
\mathit{interruptThreads}' = \mathit{interruptThreads} \\
\hline
\end{array}
$$

An SCJVM thread may be suspended, causing it to pause running and block, moving to the set of blocked threads. A thread can only suspend itself. When a thread suspends itself, a new running thread is selected from the priority queues to replace it. A suspended thread remains suspended until it is signalled to resume, at which point it is placed at the back of the queue for its priority. It should be noted that SCJ programs do not have direct access to this functionality and it is provided for the infrastructure to implement things such as device access and mission initialisation.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ThreadSuspend}} \\
\Delta \mathit{Scheduler} \\
\mathit{oldPC?}, \mathit{newPC!} : \mathit{ProgramAddress} \\
\mathit{newStack!} : \mathit{StackID} \\
\hline
\mathit{current} \neq \mathit{idle} \\
\mathit{current} \notin \mathit{interruptThreads} \\
\mathit{blocked}' = \mathit{blocked} \cup \{\mathit{current}\} \\
\mathit{priorityQueues}'\,(\mathit{currentPriority}\ \mathit{current}) = \\
\qquad \mathit{squash}\,(\mathit{priorityQueues}\,(\mathit{currentPriority}\ \mathit{current}) \nrres \{\mathit{current}\}) \\
\{\mathit{currentPriority}\ \mathit{current}\} \ndres \mathit{priorityQueues}' = \\
\qquad \{\mathit{currentPriority}\ \mathit{current}\} \ndres \mathit{priorityQueues} \\
\mathit{current}' = \mathit{last}\,(\langle \mathit{idle} \rangle \frown {\frown}/\ \mathit{priorityQueues}) \\
\mathit{programCounter}' = \mathit{programCounter} \oplus \{\mathit{current} \mapsto \mathit{oldPC?}\} \\
\mathit{newPC!} = \mathit{programCounter}\ \mathit{current}' \\
\mathit{newStack!} = \mathit{threadStack}\ \mathit{current}' \\
\mathit{free}' = \mathit{free} \land \mathit{created}' = \mathit{created} \land \mathit{started}' = \mathit{started} \\
\mathit{main}' = \mathit{main} \land \mathit{idle}' = \mathit{idle} \\
\mathit{currentPriority}' = \mathit{currentPriority} \land \mathit{basePriority}' = \mathit{basePriority} \\
\mathit{threadStack}' = \mathit{threadStack} \\
\mathit{interruptThreads}' = \mathit{interruptThreads} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{ThreadResume} \\
\hline
\Delta Scheduler \\
\Xi ThreadInfo \\
thread? : ThreadID \\
\hline
thread? \in blocked \\
blocked' = blocked \setminus \{thread?\} \\
started' = started \cup \{thread?\} \\
free' = free \land created' = created \land current' = current \\
priorityQueues'\,(currentPriority\ thread?) = \\
\qquad \langle thread? \rangle \frown priorityQueues\,(currentPriority\ thread?) \\
\{currentPriority\ thread?\} \ntriangleleft priorityQueues' = \\
\qquad \{currentPriority\ thread?\} \ntriangleleft priorityQueues \\
\hline
\end{array}
$$

The operations must be made into robust operations by accounting for the error cases.

The successful completion of an operation is indicated by returning *Sokay*.

$$
\begin{array}{|l}
\textit{SSuccess} \\
\hline
report! : SReport \\
\hline
report! = Sokay \\
\hline
\end{array}
$$

It must be reported if an operation is requested to be performed on a nonexistent thread.

$$
\begin{array}{|l}
\textit{NonexistentThread} \\
\hline
\Xi Scheduler \\
thread? : ThreadID \\
report! : SReport \\
\hline
thread? \in free \\
report! = SnonexistentThread \\
\hline
\end{array}
$$

It must be reported if an attempt is made to start a thread that has already been started.

$$
\begin{array}{|l}
\textit{ThreadAlreadyStarted} \\
\hline
\Xi Scheduler \\
thread? : ThreadID \\
report! : SReport \\
\hline
thread? \in started \cup blocked \cup \{current, idle\} \\
report! = SthreadAlreadyStarted \\
\hline
\end{array}
$$

It must be reported if an attempt is made to destroy a thread that cannot be destroyed (i.e. the main thread, the idle thread, the current thread or an interrupt thread).

$$
\begin{array}{|l}
\textit{ThreadNotDestroyable} \\
\hline
\Xi Scheduler \\
thread? : ThreadID \\
report! : SReport \\
\hline
thread? \in \{main, idle, current\} \cup interruptThreads \\
report! = SthreadNotDestroyable \\
\hline
\end{array}
$$

It must be reported if an attempt is made to resume a thread that is not blocked.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ThreadNotBlocked}} \\
\Xi Scheduler \\
thread? : ThreadID \\
report! : SReport \\
\hline
thread? \in created \cup started \cup \{current, idle\} \\
report! = SthreadNotBlocked \\
\hline
\end{array}$$

It must be reported if an attempt is made to suspend a thread that cannot be blocked

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ThreadNotBlockable}} \\
\Xi Scheduler \\
report! : SReport \\
\hline
current \in \{idle\} \cup interruptThreads \\
report! = SthreadNotBlockable \\
\hline
\end{array}$$

The operations declared so far can then be lifted to robust operations. Note that *RunThread* is only used internally so it does not need to output an error value.

$$\begin{array}{l}
RThreadCreate == (ThreadCreate \land SSuccess) \\
RThreadStart == \\
\quad (ThreadStart \land SSuccess) \lor NonexistentThread \lor ThreadAlreadyStarted \\
RThreadDestroy == \\
\quad (ThreadDestroy \land SSuccess) \lor NonexistentThread \lor ThreadNotDestroyable \\
RThreadSuspend == \\
\quad (ThreadSuspend \land SSuccess) \lor ThreadNotBlockable \\
RThreadResume == \\
\quad (ThreadResume \land SSuccess) \lor NonexistentThread \lor ThreadNotBlocked
\end{array}$$

## 2.3 Priority Ceiling Emulation

The SCJVM must support priority ceiling emulation and locking of objects. This operates by assigning to each object identifier a priority ceiling representing the highest priority thread that can take the lock on that object. If an object is locked the scheduler tracks which thread holds the lock on that object and how many times the thread holds the lock (a thread may retake the lock on an object it has already locked, forming multiple nested locks). The scheduler also tracks what locks are held by a given thread. Only locks with a holder may have a nested lock count and only running threads (not blocked threads) may hold locks. The holder of a lock must have the locked object's identifier in its set of locks held. The current priority of a thread that holds locks must be the highest of the priority ceilings of the objects it holds locks on.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{PCEScheduler}} \\
Scheduler \\
priorityCeiling : ObjectID \rightarrow Priority \\
lockHolder : ObjectID \pfun ThreadID \\
lockCount : ObjectID \pfun \mathbb{N}_1 \\
locksHeld : ThreadID \pfun \mathbb{P}\, ObjectID \\
\hline
\mathrm{dom}\, lockCount = \mathrm{dom}\, lockHolder \\
\mathrm{ran}\, lockHolder \subseteq started \cup \{current\} \\
\mathrm{ran}\, lockHolder \subseteq \mathrm{dom}\, locksHeld \\
\forall o : \mathrm{dom}\, lockHolder \bullet o \in locksHeld\,(lockHolder\; o) \\
\forall t : \mathrm{dom}\, locksHeld \bullet \\
\qquad currentPriority\; t = max\,(\{ o : locksHeld\; t \bullet priorityCeiling\; o \} \cup \{basePriority\; t\}) \\
\hline
\end{array}$$

Initially all objects have the default priority ceiling of the maximum software priority and no locks are held.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PCESchedulerInit}} \\
\hline
\textit{PCEScheduler}' \\
\textit{SchedulerInit} \\
\hline
\forall x : \textit{ObjectID} \bullet \textit{priorityCeiling}'\ x = \textit{maxSwPriority} \\
\textit{lockCount}' = \varnothing \\
\textit{lockHolder}' = \varnothing \\
\textit{locksHeld}' = \varnothing \\
\hline
\end{array}
$$

A thread has its priority raised to the object's priority ceiling when it takes the lock on an object (moving it to the front of the appropriate priority queue). A thread with a priority above the priority ceiling of an object cannot take the lock on that object.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PCETakeLock}} \\
\hline
\Delta \textit{PCEScheduler} \\
\Xi \textit{ThreadManager} \\
\textit{object?} : \textit{ObjectID} \\
\hline
\textit{object?} \notin \operatorname{dom} \textit{lockHolder} \\
\textit{currentPriority}\ \textit{current} \leq \textit{priorityCeiling}\ \textit{object?} \\
\textit{lockHolder}' = \textit{lockHolder} \oplus \{\textit{object?} \mapsto \textit{current}\} \\
\textit{lockCount}' = \textit{lockCount} \oplus \{\textit{object?} \mapsto 1\} \\
\textit{locksHeld}' = \textit{locksHeld} \oplus \{\textit{current} \mapsto \textit{locksHeld}\ \textit{current} \cup \{\textit{object?}\}\} \\
\textit{currentPriority}' = \textit{currentPriority} \oplus \{\textit{current} \mapsto \textit{max} \\
\qquad (\{\, o : \textit{locksHeld}'\ \textit{current} \bullet \textit{priorityCeiling}\ o \,\} \cup \{\textit{basePriority}\ \textit{current}\})\} \\
\textit{priorityQueues}' = (\textit{priorityQueues} \oplus \\
\qquad \{\textit{currentPriority}\ \textit{current} \mapsto \\
\qquad\qquad \textit{squash}\ (\textit{priorityQueues}\ (\textit{currentPriority}\ \textit{current}) \unrhd \{\textit{current}\})\}) \oplus \\
\qquad \{\textit{currentPriority}'\ \textit{current} \mapsto \\
\qquad\qquad \textit{squash}\ (\textit{priorityQueues}\ (\textit{currentPriority}'\ \textit{current}) \unrhd \{\textit{current}\}) \frown \langle \textit{current} \rangle\} \\
\textit{priorityCeiling}' = \textit{priorityCeiling} \\
\textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

If a thread already holds the lock on an object then it may recurse on that lock, incrementing the count of how many times it has taken the lock.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PCERetakeLock}} \\
\hline
\Delta \textit{PCEScheduler} \\
\Xi \textit{ThreadManager} \\
\textit{object?} : \textit{ObjectID} \\
\hline
\textit{object?} \in \operatorname{dom} \textit{lockHolder} \\
\textit{lockHolder}\ \textit{object?} = \textit{current} \\
\textit{lockCount}'\ \textit{object?} = \textit{lockCount}\ \textit{object?} + 1 \\
\{\textit{object?}\} \unlhd \textit{lockCount}' = \{\textit{object?}\} \unlhd \textit{lockCount} \\
\textit{priorityCeiling}' = \textit{priorityCeiling} \\
\textit{lockHolder}' = \textit{lockHolder} \\
\textit{locksHeld}' = \textit{locksHeld} \\
\textit{priorityQueues}' = \textit{priorityQueues} \\
\textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

A thread returns to its previous active priority when it releases the lock (the thread may have other active locks, which still affect the thread's priority). The thread moves to the front of the appropriate priority queue for its new priority.

$$
\begin{array}{|l}
\textit{PCEReleaseLock} \\
\hline
\Delta \textit{PCEScheduler} \\
\Xi \textit{ThreadManager} \\
\textit{object?} : \textit{ObjectID} \\
\hline
\textit{object?} \in \textit{locksHeld}\ \textit{current} \\
\textit{lockCount}\ \textit{object?} = 1 \\
\textit{lockHolder}' = \{\textit{object?}\} \ndres \textit{lockHolder} \\
\textit{lockCount}' = \{\textit{object?}\} \ndres \textit{lockCount} \\
\textit{locksHeld}' = \textit{locksHeld} \oplus \{\textit{current} \mapsto \textit{locksHeld}\ \textit{current} \setminus \{\textit{object?}\}\} \\
\textit{currentPriority}' = \textit{currentPriority} \oplus \{\textit{current} \mapsto \textit{max} \\
\qquad (\{\, o : \textit{locksHeld}'\ \textit{current} \bullet \textit{priorityCeiling}\ o \,\} \cup \{\textit{basePriority}\ \textit{current}\})\} \\
\textit{priorityQueues}' = (\textit{priorityQueues} \oplus \\
\qquad \{\textit{currentPriority}\ \textit{current} \mapsto \\
\qquad\qquad \textit{squash}\ (\textit{priorityQueues}\ (\textit{currentPriority}\ \textit{current}) \rhd \{\textit{current}\})\}) \oplus \\
\qquad \{\textit{currentPriority}'\ \textit{current} \mapsto \\
\qquad\qquad \textit{squash}\ (\textit{priorityQueues}\ (\textit{currentPriority}'\ \textit{current}) \rhd \{\textit{current}\}) \frown \langle \textit{current} \rangle\} \\
\textit{priorityCeiling}' = \textit{priorityCeiling} \\
\textit{interruptThreads}' = \textit{interruptThreads} \\
\hline
\end{array}
$$

If the lock held has been taken more than once than the count merely decreases and the lock is still held.

$$
\begin{array}{|l}
\textit{PCEReleaseNestedLock} \\
\hline
\Delta \textit{PCEScheduler} \\
\Xi \textit{ThreadManager} \\
\textit{object?} : \textit{ObjectID} \\
\hline
\textit{object?} \in \textit{locksHeld}\ \textit{current} \\
\textit{lockCount}\ \textit{object?} > 1 \\
\textit{lockCount}'\ \textit{object?} = \textit{lockCount}\ \textit{object?} - 1 \\
\{\textit{object?}\} \ndres \textit{lockCount}' = \{\textit{object?}\} \ndres \textit{lockCount} \\
\textit{lockHolder}' = \textit{lockHolder} \\
\textit{locksHeld}' = \textit{locksHeld} \\
\textit{priorityCeiling}' = \textit{priorityCeiling} \\
\textit{priorityQueues}' = \textit{priorityQueues} \\
\hline
\end{array}
$$

The priority ceiling of an object can be set by the user.

$$
\begin{array}{|l}
\textit{PCESetPriorityCeiling} \\
\hline
\Delta \textit{PCEScheduler} \\
\Xi \textit{Scheduler} \\
\textit{object?} : \textit{ObjectID} \\
\textit{pc?} : \textit{Priority} \\
\hline
\textit{priorityCeiling}' = \textit{priorityCeiling} \oplus \{\textit{object?} \mapsto \textit{pc?}\} \\
\textit{lockCount}' = \textit{lockCount} \land \textit{lockHolder}' = \textit{lockHolder} \land \textit{locksHeld}' = \textit{locksHeld} \\
\hline
\end{array}
$$

In order to prevent deadlock, it is forbidden for a thread to suspend itself while holding a lock.

$$
\begin{array}{|l}
\textit{PCESuspend} \\
\hline
\Delta \textit{PCEScheduler} \\
\textit{ThreadSuspend} \\
\hline
\textit{current} \notin \operatorname{dom} \textit{locksHeld} \lor \textit{locksHeld}\ \textit{current} = \varnothing \\
\hline
\end{array}
$$

The error cases for these operations must then be accounted for.

It must be reported if a thread tries to take a lock held by another thread.

$$\begin{array}{|l}
\textit{LockAlreadyTaken} \\
\hline
\Xi PCEScheduler \\
object? : ObjectID \\
report! : SReport \\
\hline
object? \in \operatorname{dom} lockHolder \\
lockHolder~object? \neq current \\
report! = SlockAlreadyTaken \\
\hline
\end{array}$$

It must be reported if a thread tries to release a lock it doesn't hold.

$$\begin{array}{|l}
\textit{LockNotHeld} \\
\hline
\Xi PCEScheduler \\
object? : ObjectID \\
report! : SReport \\
\hline
object? \notin locksHeld~current \\
report! = SlockNotHeld \\
\hline
\end{array}$$

It must be reported if a thread tries to suspend itself while holding locks

$$\begin{array}{|l}
\textit{ThreadHoldingLocks} \\
\hline
\Xi PCEScheduler \\
report! : SReport \\
\hline
current \in \operatorname{dom} locksHeld \land locksHeld~current \neq \varnothing \\
report! = SthreadHoldingLocks \\
\hline
\end{array}$$

Then these operations can be lifted to robust operations.

$$\begin{array}{l}
RPCESetPriorityCeiling == PCESetPriorityCeiling \land SSuccess \\
RPCETakeLock == \\
\quad (PCETakeLock \land SSuccess) \lor (PCERetakeLock \land SSuccess) \lor LockAlreadyTaken \\
RPCEReleaseLock == \\
\quad (PCEReleaseLock \land SSuccess) \lor (PCEReleaseNestedLock \land SSuccess) \lor LockNotHeld \\
RPCESuspend == (PCESuspend \land SSuccess) \lor ThreadNotBlockable \lor ThreadHoldingLocks
\end{array}$$

## 2.4 Interrupt Handling

The SCJVM scheduler must manage interrupts, tracking the priority of each interrupt, the handler attached to it and the backing store to be used as the allocation context of the handler. The scheduler must track which interrupts are masked by interrupts already running and whether interrupts are enabled or not. In this model, interrupts are handled by creating a thread in which the interrupt handler runs, so there is a map from interrupts that have been fired to running handler threads, represented here as *interruptThreadMap*. It is required that every interrupt with a running handler be masked and that the handler threads associated with interrupts be exactly the interrupt threads described earlier in this specification. The base priority of the interrupt handler threads must be the priority of the interrupt they handle.

$$\begin{array}{|l}
\hline \mathit{InterruptScheduler} \\
\mathit{PCEScheduler} \\
\mathit{interruptPriority} : \mathit{InterruptID} \rightarrow \mathit{InterruptPriority} \\
\mathit{interruptHandler} : \mathit{InterruptID} \nrightarrow \mathit{ProgramAddress} \\
\mathit{interruptAC} : \mathit{InterruptID} \nrightarrow \mathit{BackingStoreID} \\
\mathit{maskedInterrupts} : \mathbb{P}\, \mathit{InterruptID} \\
\mathit{interruptsEnabled} : \mathit{Bool} \\
\mathit{interruptThreadMap} : \mathit{InterruptID} \nrightarrow \mathit{ThreadID} \\
\hline
\operatorname{dom} \mathit{interruptThreadMap} \subseteq \mathit{maskedInterrupts} \\
\operatorname{ran} \mathit{interruptThreadMap} = \mathit{interruptThreads} \\
\operatorname{dom} \mathit{interruptHandler} = \operatorname{dom} \mathit{interruptAC} \\
\mathit{interruptThreadMap} \fatsemi \mathit{basePriority} \subseteq \mathit{interruptPriority} \\
\hline
\end{array}$$

Initially, no interrupts have handlers attached and the priorities are set at implementation-defined values that cannot be changed. Interrupts are initially enabled.

$$\begin{array}{|l}
\hline \mathit{InterruptSchedulerInit} \\
\mathit{InterruptScheduler}' \\
\mathit{PCESchedulerInit} \\
\hline
\mathit{interruptHandler}' = \varnothing \\
\mathit{maskedInterrupts}' = \varnothing \\
\mathit{interruptThreadMap}' = \varnothing \\
\mathit{interruptsEnabled}' = \mathit{True} \\
\hline
\end{array}$$

It is possible to attach a handler to a specified interrupt, providing a program address for the handler and a backing store identifier for the allocation context. These values are stored in the appropriate maps and no other state is changed.

$$\begin{array}{|l}
\hline \mathit{InterruptAttachHandler} \\
\Delta \mathit{InterruptScheduler} \\
\Xi \mathit{PCEScheduler} \\
\mathit{interrupt?} : \mathit{InterruptID} \\
\mathit{handler?} : \mathit{ProgramAddress} \\
\mathit{allocationContext?} : \mathit{BackingStoreID} \\
\hline
\mathit{interruptHandler}' = \mathit{interruptHandler} \oplus \{\mathit{interrupt?} \mapsto \mathit{handler?}\} \\
\mathit{interruptAC}' = \mathit{interruptAC} \oplus \{\mathit{interrupt?} \mapsto \mathit{allocationContext?}\} \\
\mathit{interruptPriority}' = \mathit{interruptPriority} \\
\mathit{maskedInterrupts}' = \mathit{maskedInterrupts} \\
\mathit{interruptThreadMap}' = \mathit{interruptThreadMap} \\
\mathit{interruptsEnabled}' = \mathit{interruptsEnabled} \\
\hline
\end{array}$$

It is possible to detach an interrupt handler from a given interrupt, removing it from the domains of *interruptHandler* and *interruptAC*.

$$\begin{array}{|l}
\hline \mathit{InterruptDetachHandler} \\
\Delta \mathit{InterruptScheduler} \\
\Xi \mathit{PCEScheduler} \\
\mathit{interrupt?} : \mathit{InterruptID} \\
\hline
\mathit{interruptHandler}' = \{\mathit{interrupt?}\} \mathbin{⩤} \mathit{interruptHandler} \\
\mathit{interruptAC}' = \{\mathit{interrupt?}\} \mathbin{⩤} \mathit{interruptAC} \\
\mathit{interruptPriority}' = \mathit{interruptPriority} \\
\mathit{maskedInterrupts}' = \mathit{maskedInterrupts} \\
\mathit{interruptThreadMap}' = \mathit{interruptThreadMap} \\
\mathit{interruptsEnabled}' = \mathit{interruptsEnabled} \\
\hline
\end{array}$$

It is possible for the user to read the priority of a given interrupt, though there is no way to set the priority.

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{InterruptGetPriority}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Xi \mathit{InterruptScheduler} \\
\mathit{interrupt?} : \mathit{InterruptID} \\
\mathit{priority!} : \mathit{InterruptPriority} \\
\hline
\mathit{priority!} = \mathit{interruptPriority}\ \mathit{interrupt?} \\
\hline
\end{array}$$

It is possible to enable and disable interrupts.

$$\mathit{InterruptEnableFixedVars} == \Xi \mathit{InterruptScheduler} \setminus (\mathit{interruptsEnabled})$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{InterruptEnable}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{InterruptScheduler} \\
\Xi \mathit{PCEScheduler} \\
\hline
\mathit{interruptsEnabled}' = \mathit{True} \\
\mathit{InterruptEnableFixedVars} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{InterruptDisable}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{InterruptScheduler} \\
\Xi \mathit{PCEScheduler} \\
\hline
\mathit{interruptsEnabled}' = \mathit{False} \\
\mathit{InterruptEnableFixedVars} \\
\hline
\end{array}$$

Interrupts are handled, as already mentioned, by creating a thread for the interrupt handler. For the interrupt to be handled, interrupts must be enabled, the interrupt must not be masked and there must be a handler attached to it. The stack used by the handler thread is that of the current thread at the time the interrupt fires and the priority (base and current) is that of the interrupt. The program counter value and allocation context of the handler thread are those attached to the interrupt by the user. The thread is placed at the back of the appropriate priority queue for its priority. The interrupt and all interrupts of the same or lower priority are masked while the interrupt is being handled. The identifier of the handler thread and the allocation context of the thread are output so they can be registered with the memory manager.

$$
\begin{array}{l}
\textbf{HandleInterrupt} \\
\hline
\Delta InterruptScheduler \\
interrupt? : InterruptID \\
handler! : ThreadID \\
ac! : BackingStoreID \\
\hline
interruptsEnabled = True \\
interrupt? \notin maskedInterrupts \\
interrupt? \in \operatorname{dom} interruptHandler \\
handler! \in free \\
threadStack' = \\
\qquad threadStack \oplus \{handler! \mapsto threadStack\ current\} \\
currentPriority' = \\
\qquad currentPriority \oplus \{handler! \mapsto interruptPriority\ interrupt?\} \\
basePriority' = \\
\qquad basePriority \oplus \{handler! \mapsto interruptPriority\ interrupt?\} \\
programCounter' = \\
\qquad programCounter \oplus \{handler! \mapsto interruptHandler\ interrupt?\} \\
free' = free \setminus \{handler!\} \\
started' = started \cup \{handler!\} \\
priorityQueues'\ (interruptPriority\ interrupt?) = \\
\qquad \langle handler! \rangle \frown priorityQueues\ (interruptPriority\ interrupt?) \\
\{interruptPriority\ interrupt?\} \lhd priorityQueues' = \\
\qquad \{interruptPriority\ interrupt?\} \lhd priorityQueues \\
interruptThreads' = interruptThreads \cup \{handler!\} \\
ac! = interruptAC\ interrupt? \\
maskedInterrupts' = \\
\qquad \{\, i : InterruptID \mid interruptPriority\ i \leq interruptPriority\ interrupt? \,\} \\
interruptPriority' = interruptPriority \\
idle' = idle \land main' = main \land current' = current \\
created' = created \land blocked' = blocked
\end{array}
$$

If interrupts are disabled, the interrupt is masked or the interrupt has no handler attached then it is silently ignored.

$$
\begin{array}{l}
\textbf{IgnoreInterrupt} \\
\hline
\Xi InterruptScheduler \\
interrupt? : InterruptID \\
\hline
interruptsEnabled = False \lor \\
interrupt? \in maskedInterrupts \lor \\
interrupt? \notin \operatorname{dom} interruptHandler
\end{array}
$$

$$ProcessInterrupt == HandleInterrupt \lor IgnoreInterrupt$$

When an interrupt handler ends, the interrupt handler thread is destroyed, removing the information about it from the scheduler. All interrupts except those with priority less than or equal to the interrupts still being handled. The identifier of the interrupt handler thread is returned so that it can be deregistered from the memory manager.

$$
\begin{array}{l}
\textit{InterruptEnd} \\
\hline
\Delta \textit{InterruptScheduler} \\
\textit{handler}! : \textit{ThreadID} \\
\hline
\textit{current} \in \textit{interruptThreads} \\
\textit{threadStack}' = \{\textit{current}\} \ntriangleleft \textit{threadStack} \\
\textit{currentPriority}' = \{\textit{current}\} \ntriangleleft \textit{currentPriority} \\
\textit{basePriority}' = \{\textit{current}\} \ntriangleleft \textit{basePriority} \\
\textit{programCounter}' = \{\textit{current}\} \ntriangleleft \textit{programCounter} \\
\textit{priorityQueues}' \, (\textit{currentPriority} \; \textit{current}) = \\
\qquad \textit{squash} \, (\textit{priorityQueues} \, (\textit{currentPriority} \; \textit{current}) \ntriangleright \{\textit{current}\}) \\
\{\textit{currentPriority} \; \textit{current}\} \ntriangleleft \textit{priorityQueues}' = \\
\qquad \{\textit{currentPriority} \; \textit{current}\} \ntriangleleft \textit{priorityQueues} \\
\textit{current}' = \textit{last} \, (\langle \textit{idle} \rangle \frown \frown / \; \textit{priorityQueues}') \\
\textit{interruptThreads}' = \textit{interruptThreads} \setminus \{\textit{current}\} \\
\textit{maskedInterrupts}' = \\
\qquad \{ \, i : \textit{InterruptID} \mid \\
\qquad\qquad \exists \, j : \operatorname{dom} \textit{interruptThreadMap}' \bullet \\
\qquad\qquad\qquad \textit{interruptPriority} \; i \leq \textit{interruptPriority} \; j \} \\
\textit{interruptPriority}' = \textit{interruptPriority} \\
\textit{idle}' = \textit{idle} \land \textit{main}' = \textit{main} \\
\textit{created}' = \textit{created} \land \textit{blocked}' = \textit{blocked} \\
\textit{free}' = \textit{free} \land \textit{started}' = \textit{started} \\
\textit{handler}! = \textit{current}
\end{array}
$$

These must then be lifted to robust actions. The only error case that must be handled is the case of ending an interrupt when not in an interrupt.

$$
\begin{array}{l}
\textit{NotInInterrupt} \\
\hline
\Xi \textit{InterruptScheduler} \\
\textit{report}! : \textit{SReport} \\
\hline
\textit{current} \notin \textit{interruptThreads} \\
\textit{report}! = \textit{SnotInInterrupt}
\end{array}
$$

$$
\begin{array}{l}
\textit{RInterruptAttachHandler} == \textit{InterruptAttachHandler} \land \textit{SSuccess} \\
\textit{RInterruptDetachHandler} == \textit{InterruptDetachHandler} \land \textit{SSuccess} \\
\textit{RInterruptGetPriority} == \textit{InterruptGetPriority} \land \textit{SSuccess} \\
\textit{RInterruptEnable} == \textit{InterruptEnable} \land \textit{SSuccess} \\
\textit{RInterruptDisable} == \textit{InterruptDisable} \land \textit{SSuccess} \\
\textit{RInterruptEnd} == (\textit{InterruptEnd} \land \textit{SSuccess}) \lor \textit{NotInInterrupt}
\end{array}
$$

## 2.5 Scheduler Operations

The scheduler model offers the services detailed in this section. Not all of the services specified here are available to the user but instead some are for handling hardware interrupts, and communication with the memory manager and real-time clock. The operations that are made available to the user are summarised in Table 2

The state of the scheduler process is *InterruptScheduler*, which contains the state of the priority ceiling emulation manager and priority scheduler as well as the interrupt manager state.

**state** *InterruptScheduler*

| Operation | Inputs | Outputs | Error Conditions |
|---|---|---|---|
| `getMaxSoftwarePriority` | (none) | priority level | (none) |
| `getMinSoftwarePriority` | (none) | priority level | (none) |
| `getNormSoftwarePriority` | (none) | priority level | (none) |
| `getMaxHardwarePriority` | (none) | priority level | (none) |
| `getMinHardwarePriority` | (none) | priority level | (none) |
| `getMainThread` | (none) | thread identifier | (none) |
| `makeThread` | entry point<br>priority level<br>backing store identifier<br>stack identifier | thread identifier | (none) |
| `startThread` | thread identifier | (none) | invalid identifier<br>thread already started |
| `getCurrentThread` | (none) | thread identifier | (none) |
| `destroyThread` | thread identifier | (none) | invalid identifier<br>thread not destroyable |
| `suspendThread` | (none) | (none) | thread cannot be blocked<br>thread holds locks |
| `resumeThread` | thread identifier | (none) | invalid identifier<br>thread not blocked |
| `setPriorityCeiling` | pointer to object<br>priority level | (none) | invalid priority |
| `takeLock` | pointer to object | (none) | lock in use |
| `releaseLock` | pointer to object | (none) | lock not held |
| `attachInterruptHandler` | interrupt identifier<br>entry point | (none) | (none) |
| `detachInterruptHandler` | interrupt identifier | (none) | (none) |
| `getInterruptPriority` | interrupt identifier | priority level | (none) |
| `disableInterrupts` | (none) | (none) | (none) |
| `enableInterrupts` | (none) | (none) | (none) |

Table 2: The operations of the SCJVM scheduler

The scheduler is initialised with the main thread's stack, priority, entry point and initial allocation context. The stack, priority and entry point are stored by the scheduler as part of initialising its state. The allocation context for the main thread, along with main thread identifier from the state, is registered with the memory manager; this requires the memory manager to be initialised before the scheduler . Finally, the program counter and stack are set in the core execution environment using the provided values.

$$\begin{array}{l}
\mathit{Init} \mathrel{\widehat=} \mathbf{var}\ \mathit{mainAC} : \mathit{BackingStoreID};\ \mathit{mainEntry} : \mathit{ProgramAddress};\ \mathit{mainStack} : \mathit{StackID} \bullet \\
\quad \mathit{Sinit}?\mathit{mainStack}?\mathit{mainPriority}?\mathit{mainEntry}?\mathit{mainAC} \longrightarrow \big(\mathit{InterruptSchedulerInit}\big); \\
\quad \mathit{MMaddThread}!\mathit{main}!\mathit{mainAC} \\
\qquad \longrightarrow \mathit{CEEsetProgramCounter}!\mathit{mainEntry} \longrightarrow \mathit{CEEsetStack}!\mathit{mainStack} \longrightarrow \mathbf{Skip}
\end{array}$$

The SCJVM scheduler must provide functions to get the minimum, maximum and normal priorities. These operations always succeed as they simply return constants.

$$\begin{array}{l}
\mathit{GetMaxSoftwarePriority} \mathrel{\widehat=} \\
\quad \mathit{SgetMaxSoftwarePriority}!\mathit{maxSwPriority} \longrightarrow \mathit{Sreport}!\mathit{Sokay} \longrightarrow \mathbf{Skip} \\
\mathit{GetMinSoftwarePriority} \mathrel{\widehat=} \\
\quad \mathit{SgetMinSoftwarePriority}!\mathit{minSwPriority} \longrightarrow \mathit{Sreport}!\mathit{Sokay} \longrightarrow \mathbf{Skip} \\
\mathit{GetNormSoftwarePriority} \mathrel{\widehat=} \\
\quad \mathit{SgetNormSoftwarePriority}!\mathit{normSwPriority} \longrightarrow \mathit{Sreport}!\mathit{Sokay} \longrightarrow \mathbf{Skip} \\
\mathit{GetMaxHardwarePriority} \mathrel{\widehat=} \\
\quad \mathit{SgetMaxSoftwarePriority}!\mathit{maxHwPriority} \longrightarrow \mathit{Sreport}!\mathit{Sokay} \longrightarrow \mathbf{Skip} \\
\mathit{GetMinHardwarePriority} \mathrel{\widehat=} \\
\quad \mathit{SgetMinSoftwarePriority}!\mathit{minHwPriority} \longrightarrow \mathit{Sreport}!\mathit{Sokay} \longrightarrow \mathbf{Skip}
\end{array}$$

The operation `getMainThread` takes no inputs and outputs the identifier of the main thread. This operation always succeeds.

$$GetMainThread \mathrel{\widehat=} SgetMainThread!main \longrightarrow Sreport!Sokay \longrightarrow \mathbf{Skip}$$

The operation `makeThread` provides for creating a new thread. It takes as input the entry point, base priority, initial allocation context and stack for the thread and outputs the newly created thread's identifier. This operation also handles registering the new thread with the memory manager. The new thread created by this operation is initially not started. This operation always succeeds.

$$\begin{array}{l} MakeThread \mathrel{\widehat=} \mathbf{var}\ bsid : BackingStoreID;\ newID : ThreadID;\ report : SReport \bullet \\ \quad SmakeThread?entryPoint?priority?bsid?stack \longrightarrow \big(RThreadCreate\big); \\ \quad MMaddThread!newID!bsid \longrightarrow SmakeThreadRet!newID \longrightarrow Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

There are several operations that require a new thread to be selected and a thread switch to occur. The *Circus* action *Schedule* handles that and is used in some of the following operations. The thread switch involves getting the old thread's program counter value, selecting a new thread, storing the old thread's information, retrieving the new thread's information, and passing the new thread's program counter and stack to the core execution environment.

$$\begin{array}{l} Schedule \mathrel{\widehat=} \mathbf{var}\ newPC : ProgramAddress;\ newStack : StackID \bullet \\ \quad CEEgetProgramCounter?oldPC \longrightarrow \big(RunThread\big); \\ \quad CEEsetProgramCounter!newPC \longrightarrow CEEsetStack!newStack \longrightarrow \mathbf{Skip} \end{array}$$

The operation `startThread` allows for starting a thread that has been created but not yet started. It takes as input the identifier of the thread to be started and gives no output. The starting of a thread causes a thread switch as the newly started thread may preempt the currently running thread. This operation fails if the given thread does not exist or has already been started or if the given thread does not exist.

$$\begin{array}{l} StartThread \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet \\ \quad SstartThread?thread \longrightarrow \big(RThreadStart\big); \\ \quad Sreport!report \longrightarrow Schedule \end{array}$$

It must be possible to get the identifier of the current thread. There are two channels that provide access to the current thread's identifier: *ScurrentThread*, which is used by the memory manager, and *SgetCurrentThread*, which is made available to the user as the `getCurrentThread` operation. These operations always succeed but *ScurrentThread* does not communicate a report as the memory manager does not use the scheduler's reporting channel.

$$\begin{array}{l} GetCurrentThread \mathrel{\widehat=} SgetCurrentThread!current \longrightarrow Sreport!Sokay \longrightarrow \mathbf{Skip} \\ CurrentThread \mathrel{\widehat=} ScurrentThread!current \longrightarrow \mathbf{Skip} \end{array}$$

The operation `destroyThread` allows for an existing thread to be destroyed. It takes as input the identifier of the thread to be destroyed and gives no output. This operation fails if the given thread does not exist or is the current thread, the main thread, the idle thread or an interrupt thread. Because the current thread cannot be destroyed, this does not require a thread switch.

$$\begin{array}{l} DestroyThread \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet \\ \quad SdestroyThread?thread \longrightarrow \big(RThreadDestroy\big); \\ \quad Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `suspendThread` allows for the current thread to be suspended. It takes no input and gives no output. This operation fails if the current thread is the idle thread or an interrupt thread, or if the current thread is holding any locks on objects. The operation of suspending a thread removes it from being current and replaces it with a new current thread, so it performs a thread switch without the use of *Schedule*.

$$\begin{array}{l} SuspendThread \mathrel{\widehat=} \mathbf{var}\ newPC : ProgramAddress;\ newStack : StackID;\ report : SReport \bullet \\ \quad SsuspendThread \longrightarrow CEEgetProgramCounter?oldPC \longrightarrow \left(RPCESuspend\right); \\ \quad CEEsetProgramCounter!newPC \longrightarrow CEEsetStack!newStack \longrightarrow Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `resumeThread` allows for a thread that has been suspended to be resumed. It takes as input the identifier of the thread to be resumed and gives no output. This operation fails if the thread being resumed does not exist or is not blocked. As resuming a thread changes which threads are available to run, this operation performs a thread switch.

$$\begin{array}{l} ResumeThread \mathrel{\widehat=} \mathbf{var}\ thread : ThreadID;\ report : SReport \bullet \\ \quad SresumeThread?thread \longrightarrow \left(RThreadResume\right); \\ \quad Sreport!report \longrightarrow Schedule \end{array}$$

The operation `setPriorityCeiling` allows for the priority ceiling of an object to be set. It takes as input the identifier of the object and the value of the new priority ceiling, and gives no output. This operation always succeeds.

$$\begin{array}{l} SetPriorityCeiling \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet \\ \quad SsetPriorityCeiling?object?pc \longrightarrow \left(RPCESetPriorityCeiling\right); \\ \quad Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `takeLock` allows the current thread to take the lock on an object. It takes as input the identifier of the object to be locked and gives no output. This operation fails if the lock is already taken by another thread. A thread may lock the same object multiple times.

$$\begin{array}{l} TakeLock \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet \\ \quad StakeLock?object \longrightarrow \left(RPCETakeLock\right); \\ \quad Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `releaseLock` allows for the current thread to release a lock it has on a given object. It takes as input the identifier of the locked object and gives no output. This operation fails if the current thread does not hold the lock on the given object. If the object was locked more than once, this operation only releases one of the locks.

$$\begin{array}{l} ReleaseLock \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet \\ \quad SreleaseLock?object \longrightarrow \left(RPCEReleaseLock\right); \\ \quad Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `attachInterruptHandler` allows for an interrupt handler to be attached to a given interrupt. It takes as input the identifier of the interrupt, the entry point of the handler and the backing store to be used as the allocation context of the handler. This operation always succeeds.

$$\begin{array}{l} AttachInterruptHandler \mathrel{\widehat=} \mathbf{var}\ report : SReport \bullet \\ \quad SattachInterruptHandler?interrupt?handler?allocationContext \\ \qquad \longrightarrow \left(RInterruptAttachHandler\right); \\ \quad Sreport!report \longrightarrow \mathbf{Skip} \end{array}$$

The operation `detachHandler` allows for an interrupt handler to be detached from a given interrupt, leaving it unhandled. It takes as input the identifier of the interrupt and gives no output. This operation always succeeds.

$$\begin{array}{l} \mathit{DetachInterruptHandler} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{SReport} \bullet \\ \quad \mathit{SdetachInterruptHandler}?\mathit{interrupt} \longrightarrow \big(\mathit{RInterruptDetachHandler}\big); \\ \quad \mathit{Sreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The priority of an interrupt can be obtained using the `getInterruptPriority` operation. It takes as input the identifier of an interrupt and outputs the priority of that interrupt. The interrupt priorities are implementation-defined and cannot be set by the user. This operation always succeeds.

$$\begin{array}{l} \mathit{GetInterruptPriority} \mathrel{\widehat=} \mathbf{var}\ \mathit{priority} : \mathit{Priority};\ \mathit{report} : \mathit{SReport} \bullet \\ \quad \mathit{SgetInterruptPriority}?\mathit{interrupt} \longrightarrow \big(\mathit{RInterruptGetPriority}\big); \\ \quad \mathit{SgetInterruptPriorityRet}!\mathit{priority} \longrightarrow \mathit{Sreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The operations `enableInterrupts` and `disableInterrupts` allow for enabling and disabling interrupts respectively. They both take no inputs, give no outputs and always succeed. Enabling and disabling of interrupts is done at the hardware level through the *HWenableInterrupts* and *HWdisableInterrupts* channels in addition to adjusting the state of the scheduler.

$$\begin{array}{l} \mathit{EnableInterrupts} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{SReport} \bullet \\ \quad \mathit{SenableInterrupts} \longrightarrow \big(\mathit{RInterruptEnable}\big); \\ \quad \mathit{HWenableInterrupts} \longrightarrow \mathit{Sreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

$$\begin{array}{l} \mathit{DisableInterrupts} \mathrel{\widehat=} \mathbf{var}\ \mathit{report} : \mathit{SReport} \bullet \\ \quad \mathit{SdisableInterrupts} \longrightarrow \big(\mathit{RInterruptDisable}\big); \\ \mathit{HWdisableInterrupts} \longrightarrow \mathit{Sreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The SCJVM scheduler handles interrupts by creating a new thread to run the interrupt handler, registering it with the memory manager and then performing a thread switch. The scheduler does not handle the clock interrupt coming from hardware as that is handled by the real-time clock. The clock interrupt is only triggered when it passed from the real-time clock to the scheduler upon an alarm triggering.

$$\begin{array}{l} \mathit{Handle} \mathrel{\widehat=} \mathbf{val}\ \mathit{interrupt} : \mathit{InterruptID} \bullet \mathbf{var}\ \mathit{handler} : \mathit{ThreadID};\ \mathit{ac} : \mathit{BackingStoreID} \bullet \\ \quad \big(\mathit{ProcessInterrupt}\big)\ ;\ \ \mathit{MMaddThread}!\mathit{handler}!\mathit{ac} \longrightarrow \mathit{Schedule} \end{array}$$

$$\begin{array}{l} \mathit{HandleNonclockInterrupt} \mathrel{\widehat=} \\ \quad \mathit{HWinterrupt}?\mathit{interrupt} : (\mathit{interrupt} \neq \mathit{clockInterrupt}) \longrightarrow \mathit{Handle}(\mathit{interrupt}) \end{array}$$

$$\mathit{HandleClockInterrupt} \mathrel{\widehat=} \mathit{RTCclockInterrupt} \longrightarrow \mathit{Handle}(\mathit{clockInterrupt})$$

$$\begin{array}{l} \mathit{EndInterrupt} \mathrel{\widehat=} \mathbf{var}\ \mathit{handler} : \mathit{ThreadID};\ \mathit{report} : \mathit{SReport} \bullet \\ \quad \mathit{SendInterrupt} \longrightarrow \big(\mathit{RInterruptEnd}\big); \\ \quad \mathit{MMremoveThread}!\mathit{handler} \longrightarrow \mathit{Sreport}!\mathit{report} \longrightarrow \mathbf{Skip} \end{array}$$

The scheduler must be initialised before any operations can be used or interrupts can be handled. After initialisation any operation can be used once the previous operation has completed.

$$\begin{array}{l} \mathit{Loop} \mathrel{\widehat=} \mathit{GetMainThread} \mathrel{\Box} \mathit{MakeThread} \mathrel{\Box} \mathit{StartThread} \\ \quad \Box\, \mathit{GetCurrentThread} \mathrel{\Box} \mathit{DestroyThread} \\ \quad \Box\, \mathit{SuspendThread} \mathrel{\Box} \mathit{ResumeThread} \\ \quad \Box\, \mathit{SetPriorityCeiling} \mathrel{\Box} \mathit{TakeLock} \mathrel{\Box} \mathit{ReleaseLock} \\ \quad \Box\, \mathit{AttachInterruptHandler} \mathrel{\Box} \mathit{DetachInterruptHandler} \\ \quad \Box\, \mathit{GetInterruptPriority} \mathrel{\Box} \mathit{EnableInterrupts} \\ \quad \Box\, \mathit{DisableInterrupts} \mathrel{\Box} \mathit{HandleNonclockInterrupt} \\ \quad \Box\, \mathit{HandleClockInterrupt} \mathrel{\Box} \mathit{EndInterrupt}; \\ \quad \mathit{Loop} \end{array}$$

$\bullet\ \mathit{Init}\ ;\ \mathit{Loop}$

**end**

## 3 Real-time Clock

**section** *realtimeclock* **parents** *scheduler*

The SCJVM real-time clock provides an interface to a hardware real-time clock, which is used by the SCJ clock API. The periodic clock interrupt from the hardware is handled by the SCJVM clock and used to manage alarms that trigger when a certain time is reached. The interrupt is passed to the scheduler when an alarm triggers and the SCJ API implementation should attach an interrupt handler to it that simply calls the `triggerAlarm()` method of `Clock` for the real-time clock. The type used for interrupt identifiers is the same as that used by the scheduler.

Time is represented in this specification as a natural number of nanoseconds. The SCJ API has time as two numbers representing milliseconds and nanoseconds but it is easier for specification of the virtual machine to ignore that detail.

$$\mathit{Time} == \mathbb{N}$$

The clock must have a precision representing the number of nanoseconds between occurences of the hardware clock interrupt. The precision cannot be zero.

$$\begin{array}{|l} \mathit{precision} : \mathit{Time} \\ \hline \mathit{precision} > 0 \end{array}$$

The SCJVM real-time clock relies on the existence of a hardware real-time clock that must be capable of giving the current time in nanoseconds.

**channel** $\mathit{HWtime} : \mathit{Time}$

There are channels for each of the operations of the SCJVM real-time clock: getting the current time, getting the clock's precision, setting an alarm, and clearing the alarm. These operations are summarised in Table 3. Unlike the previous parts of this specification, the real-time clock specification makes less use of Z schemas and does not have a multi-part state so the operations are described along with their definitions.

**channel** $\mathit{RTCgetTime}, \mathit{RTCgetPrecision} : \mathit{Time}$
**channel** $\mathit{RTCsetAlarm} : \mathit{Time}$
**channel** $\mathit{RTCclearAlarm}$

| Operation | Inputs | Outputs | Error Conditions |
|---|---|---|---|
| getSystemTime | (none) | time | (none) |
| getSystemTimePrecision | (none) | time precision | (none) |
| setAlarm | time | (none) | time in past |
| clearAlarm | (none) | (none) | (none) |

Table 3: The operations of the SCJVM real-time clock

The SCJVM also uses the hardware interrupt channel, *HWinterrupt*, and has a channel to pass the clock interrupt on to the scheduler when appropriate, *RTCclockInterrupt*, which was declared earlier.

There is also a type and channel for reporting erroneous inputs to operations.

$$RTCReport ::= RTCokay \mid RTCtimeInPast$$

$$\mathbf{channel}\ report : RTCReport$$

Having defined the channels and types, the process definition begins.

$$\mathbf{process}\ RealtimeClock \mathrel{\widehat=} \mathbf{begin}$$

The real-time clock's state stores the current time of the clock (accurate to within the clock's precision). The state also stores the current alarm set (if any) as well as a boolean value indicating whether or not there is an alarm set. If an alarm is set then it must be in the future.

$$\begin{array}{|l}
\multicolumn{1}{l}{State} \\
currentTime : Time \\
currentAlarm : Time \\
alarmSet : Bool \\
\hline
alarmSet = True \Rightarrow currentAlarm \geq currentTime \\
\hline
\end{array}$$

$$\mathbf{state}\ State$$

Initialising the clock's state consists of getting the time from the hardware clock to initialise *currentTime*. Initially no alarm is set and so the time of the alarm is allowed to take any value since it is unused.

$$\begin{array}{|l}
\multicolumn{1}{l}{Init0} \\
State' \\
initTime? : Time \\
\hline
currentTime' = initTime? \\
alarmSet' = False \\
\hline
\end{array}$$

$$Init \mathrel{\widehat=} HWtime?initTime \longrightarrow \left(Init0\right)$$

The operation `getTime` allows for getting the current time from the clock's state. It only provides an output through the channel that corresponds to the operation and has no input. This operation always succeeds.

$$GetTime \mathrel{\widehat=} RTCgetTime!currentTime \longrightarrow report!RTCokay \longrightarrow \mathbf{Skip}$$

The operation `getPrecision` allows for getting the precision of the clock. This only involves ouputting a constant through the operation's channel and does not take any inputs. This operation always succeeds.

$$GetPrecision \mathrel{\widehat=} RTCgetPrecision!precision \longrightarrow report!RTCokay \longrightarrow \mathbf{Skip}$$

The operation `setAlarm` provides for setting a new alarm, which overwrites any existing alarm provided that the given time is in the future. If the given time is in the past, an error is reported and the new alarm is not set. This operation takes as an input the time of the new alarm to be set and does not produce an output.

$$\begin{array}{|l}
\hline
\mathit{SetAlarm0} \\
\Delta \mathit{State} \\
\mathit{alarmTime?} : \mathit{Time} \\
r! : \mathit{RTCReport} \\
\hline
\mathit{alarmTime?} \geq \mathit{currentTime} \\
\mathit{currentAlarm}' = \mathit{alarmTime?} \\
\mathit{alarmSet}' = \mathit{True} \\
\mathit{currentTime}' = \mathit{currentTime} \\
r! = \mathit{RTCokay} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\mathit{SetAlarm1} \\
\Xi \mathit{State} \\
\mathit{alarmTime?} : \mathit{Time} \\
r! : \mathit{RTCReport} \\
\hline
\mathit{alarmTime?} < \mathit{currentTime} \\
r! = \mathit{RTCtimeInPast} \\
\hline
\end{array}$$

$$\begin{array}{l}
\mathit{SetAlarm} \mathrel{\widehat=} \mathbf{var}\ r : \mathit{RTCReport} \bullet \\
\quad \mathit{RTCsetAlarm?alarmTime} \longrightarrow \big(\mathit{SetAlarm0} \lor \mathit{SetAlarm1}\big) \mathbin{;}\ \mathit{report!r} \longrightarrow \mathbf{Skip}
\end{array}$$

The `clearAlarm` operation allows for clearing any set alarm, which simply involves setting *alarmSet* to *False*. This operation takes no inputs, gives no outputs and always succeeds.

$$\mathit{ClearAlarm} \mathrel{\widehat=} \mathit{RTCclearAlarm} \longrightarrow \mathit{report!RTCokay} \longrightarrow \mathit{alarmSet} := \mathit{False}$$

When an alarm triggers, the clock interupt is sent to the scheduler. The current alarm is then cleared.

$$\mathit{TriggerAlarm} \mathrel{\widehat=} \mathit{RTCclockInterrupt} \longrightarrow \mathit{alarmSet} := \mathit{False}$$

Clock tick interupts come periodically from hardware, with a period equal to the clock's precision. The clock ticks are handled by checking if any alarm triggers and then incrementing *currentTime* by *precision*. Resolving alarms before updating *currentTime* is required to ensure the state invariant is maintained.

$$\begin{array}{l}
\mathit{Tick} \mathrel{\widehat=} \mathit{HWinterrupt?interrupt} : (\mathit{interrupt} = \mathit{clockInterrupt}) \longrightarrow \mathbf{if} \\
\quad \mathit{currentTime} + \mathit{precision} \geq \mathit{currentAlarm} \longrightarrow \mathit{TriggerAlarm} \\
\quad [\,]\ \mathit{currentTime} + \mathit{precision} < \mathit{currentAlarm} \longrightarrow \mathbf{Skip} \\
\mathbf{fi} \mathbin{;}\ \mathit{currentTime} := \mathit{currentTime} + \mathit{precision} \mathbin{;}\ \mathit{Tick}
\end{array}$$

Any of the actions available to the user may be chosen, and the process loops to allow another action to be taken. The process may handle an incoming clock tick instead of a user action. The process begins by performing the initialisation and then enters the loop.

$$\begin{array}{l}
\mathit{Loop} \mathrel{\widehat=} \\
\quad \mathit{SetAlarm} \Box \mathit{ClearAlarm} \Box \mathit{GetTime} \Box \mathit{GetPrecision} \Box \mathit{Tick} \mathbin{;}\ \mathit{Loop}
\end{array}$$

$$\bullet\ \mathit{Init} \mathbin{;}\ \mathit{Loop}$$

$$\mathbf{end}$$

## 4 Complete VM Services Model

$\textbf{section}\ \mathit{scjvmservices}\ \textbf{parents}\ \mathit{realtimeclock}$

With the three processes that model the three components of the VM services defined, we then compose them in parallel to form the complete model of the VM services. Certain channels are used to communicate between the different components and the components must synchronise on these channels. The channel set *MMSInterface* contains the channels used for communication between the memory manager and the scheduler. The channel set *RTCSInterface* contains the channel used to pass the clock interrupt from the real-time clock to the scheduler. The channels in *MMSInterface* and *RTCSInterface* are hidden as they are internal to the SCJVM.

$$\begin{array}{l}
\textbf{channelset}\ \mathit{MMSInterface} == \{\!|\ \mathit{MMaddThread}, \mathit{MMremoveThread}, \mathit{ScurrentThread}\ |\!\}\\
\textbf{channelset}\ \mathit{RTCSInterface} == \{\!|\ \mathit{RTCclockInterrupt}\ |\!\}\\
\textbf{channelset}\ \mathit{VMServicesInternals} == \mathit{MMSInterface} \cup \mathit{RTCSInterface}
\end{array}$$

$$\begin{array}{l}
\textbf{process}\ \mathit{VMServices} \mathrel{\widehat=} ((\mathit{MemoryManager}\ [\![\ \mathit{MMSInterface}\ ]\!]\ \mathit{Scheduler})\\
\qquad\qquad [\![\mathit{RTCSInterface}\ ]\!]\ \mathit{RealtimeClock})\\
\qquad\qquad\quad \setminus \mathit{VMServicesInternals}
\end{array}$$