Coordination Primitives for Thread-Safe Operations

Martin Lottermoser

Version 4.7 (2023-09-28)

Contents

1 Purpose

This document contains some personal notes concerning coordination primitives for multiple threads operating on shared memory regions.

The motivation for writing this up was that around 2015 I was trying to understand certain technical problems related to this topic. When looking for information on the Internet, I was surprised and severely disappointed by the quality of the free documentation I found; Wikipedia, in particular, once more proved its amateur status by providing isolated (and sometimes faulty) ramblings instead of anything even remotely resembling the concise systematic (and correct) coverage one would expect from an encyclopedia. I therefore collected what information I could and started thinking about the problems myself (which is more fun anyway, though not without drawbacks). These notes are the outcome of that process.

The reader should be aware that I have made no effort to systematically investigate the work of others or to attribute specific solutions to their authors. In particular, I do not even know myself what (if anything) of what I have described here is actually new, but that is not the point anyway: the goal of these notes was to improve my understanding, not to advance the field of computer science. In short, the reader should regard this document like lecture notes, not as a research paper; this includes the possibility of future changes to this text (if needed, use the version number on the title page for exact identification).

The result turned out to be reasonably useful for me, hence I hope that it might also be helpful to others.

2 Assumptions, Definitions and Conventions

These notes assume a computing environment with the following properties:

- There are a number of **processors**, each capable of sequentially fetching and independently executing a sequence of instructions. The process of executing such a sequence is called a **thread**.
- All the processors are under the control of the same **scheduler**.
- The scheduler supports a number of **tasks** running as threads on the processors. Apart from creation and termination of tasks, the scheduler provides the following functionality to each task:
  - The task owns one virtual processor (a possibly restricted variant of the underlying processor(s)) with associated state (e. g., the contents of a number of registers), i. e., constructing the sequence of instructions (**program**) to be executed by the task may ignore the actions of other tasks because they have no effect on this state. What is considered part of the state is a property of the scheduler (and obviously also depends on the processor); the extent must be sufficient to reliably write programs for this virtual processor even when suspensions occur (see below).
  - The task has private memory areas, accessible only to this task; this encompasses at least one stack. As with the processor state, modifications to these areas are entirely under the control of the task.[1]

  This permits an author of a program to ignore all other programs, at least while accessing only task-private storage.
- There are also memory areas which can be accessed by more than one task.
- There may be more tasks than processors, hence the scheduler has to **suspend** tasks at certain points during a program's execution, give the processor to another task, and later let the first task **resume** execution on some processor. The scheduler makes this temporary suspension invisible to the suspended task by saving and later restoring its processor state and possibly other resources ("context"). Note that at least with a preemptive scheduler the points in the program where a suspension might occur may lie within operations which look atomic at the source-code level (e. g., assigning an expression to a variable).

In general, names like "thread" and "task" have different meanings for different authors or in different contexts. That can lead to confusion, misunderstandings and thus wasted effort; for the purpose of these notes the description above defines the properties of threads and tasks. A key element of these definitions is that "task" effectively denotes a thread which can be suspended and resumed by a scheduler.

In addition, the following conventions apply:

[1] A scheduler may even support different kinds of tasks in parallel, distinguished by which resources beyond the processor's state (e. g., kinds of storage) are private to the task or a selected group of tasks. That, however, is irrelevant for the present discussion which is concerned with problems associated with non-private memory areas.

- These notes are about understanding the functionality, not about achieving optimal performance.
- For clarity, the code shown here uses simple loops to wait for certain conditions. A consideration of wait-free algorithms is beyond the scope of these notes.
- All source code examples are to be interpreted as C++ [3]. Choosing a real-world language for this purpose gives the notation sufficient precision so the logic can be more easily checked by the reader. It also helps me to guard against inadvertently assuming a *deus-ex-machina* mechanism to solve problems.

3 The Problems

Consider a region of memory accessible to several threads. The main cause for problems is that threads accessing that region could overlap in their attempts to do that: the scheduler might interrupt a task (typically between processor instructions) and give the processor to another task, leading to overlapping instruction sequences; on multi-processor systems, the threads may even be executing instructions affecting the memory region in parallel. All such almost simultaneous occurrences may lead to problems.

One kind of problem ("read problem") is that modifications to data in any region often temporarily result in an intermediate state where the data have unintended values or violate consistency constraints. An example of the former is a large field which is logically one value (e.g., an integer) but which a processor can overwrite only in several steps; an example of the latter is inserting an element in a doubly-linked list where forward and backward pointers between neighbouring elements must be consistent. A thread trying to read the data while they're being modified may then obtain invalid information.

This situation can be solved if the modifications can be made to look atomic to any thread reading the data: readers either get the old or the new value, but no intermediate state. This effectively means that read operations which need a consistent result must not overlap with modifications; instead they must be squeezed between any of those.

A second kind of problem ("write problem") arises if two threads overwrite the region (or parts of it) at roughly the same time. If the actions on memory are interleaving, this also can lead to unintended values or a violation of consistency constraints, but these would now be permanent instead of temporary.

The solution for this problem is similar as for the read problem: in order to result in a consistent state for the region, the two operations must act atomically and therefore must be serialized, with the later operation determining the final result.

Update operations, i.e., operations which read the content of the region and modify it based on what has been read, can be thought of as a combination of a consistent read of the entire region, some transformations on those data within the thread, and a subsequent write. They therefore inherit the requirements for both, read and write operations, with the additional requirement that no other thread may modify the region during the entire update interval.

All this together means that only read operations may overlap with each other; all other kinds of access must in general be strictly serialized. This requires a certain amount of coordination between threads. In case of updates to application-specific data structures and with general-purpose processors, there will be no ready-made support from the processor, hence the threads must cooperate. In order to do that, they have to exchange some information, i.e., one thread has to send some data to the other which has to read it, which, of course, requires some coordination between the threads, and now we have a chicken-egg problem.

This cycle can only be broken if the processor offers some elementary operations which perform their actions on memory independently of what other threads or the scheduler do at the same time. Such instructions are **inherently thread-safe** and can be used to build other operations which are thread-safe provided the threads cooperate.

4 What the Processor Might Offer

4.1 Guarding Against Change

All the problems above would be solveable if a processor had instructions for inhibiting change by other threads (or processors) to particular memory areas:

```
void prevent_change_by_others(const void *p_area, size_t size_of_area,
  bool will_not_modify = false);
void allow_change(const void *p_area, size_t size_of_area);
```

A thread calling *prevent_change_by_others()* can use the argument `will_not_modify` to give an assurance that the thread itself will not modify the region, thereby permitting other threads to still read that region.

Although I don't know of any hardware processor offering such functionality, these operations do exist conceptually as part of particular machine instructions, but only for areas with certain restrictions on size and alignment.[2]

Note that an implementation of these operations in hardware would have to observe the following constraints between calls to *prevent_change_by_others()* and *allow_change()*:

- A thread executing on another processor in parallel and attempting to write to the memory region concerned must be delayed.
- A switch to another task on the caller's own processor can't be allowed because logically the reservation effected should be owned by the task (not by the processor), how a task is identified depends on the scheduler and the hardware should not depend on the scheduler's implementation. Hence the relation between task and processor must remain fixed during this interval.

Also, there are the obvious possibilities for deadlocks, e.g., when two threads request the same two memory areas but in different order. That should explain why the operations above are not individually available in hardware but exist only properly paired within certain processor instructions where the processor's designers can avoid these problems.

However, I should at least mention that for hardware environments with a single processor there is usually a kind of nuclear option available: masking the interrupt(s) the scheduler uses for preemption. This protects *all* of memory against change by other threads, though not by an interrupt handler for an unmasked interrupt or for non-maskable interrupts (like NMI on Intel-like processors). Besides, note that a switch to a different thread might also be initiated by certain system calls, hence this works reliably only for operating systems with well-documented behaviour and where the running thread is suitably careful with its calls. On modern systems with multiple processors, however, this is not a viable solution for preventing changes anyway.

4.2 Memory Barriers

Standards for programming languages like C++ typically grant compilers a certain latitude concerning the order of generated machine instructions. It is only at specific points in the source code, called "sequence points" in C/C++, that the standards effectively guarantee that all machine instructions needed to implement everything before that point have been issued and the operations needed for everything after it have not yet begun.

Modern processors have not been granted a similar freedom for ordering the effects of instructions operating on memory, but their designers have taken that freedom anyway in order to improve performance (e.g., by using speculative fetch, with occasionally disastrous consequences for security). Obviously this must be implemented such that any reordering is invisible to the thread executing the instructions and this therefore can be expected not to lead to inconsistencies in a single-thread-on-one-processor situation. It can be different if multiple processors with shared memory are involved, and unfortunately the situation usually *is* different.

This problem can be handled by issuing a command to set a "memory barrier", also called a "memory fence": this marks a point in the instruction sequence where the processor architecture guarantees that all operations of a certain class on memory by any processor will have completed before it or that all operations of a certain class in the current thread will not have started before it. This prevents the hardware from using out-of-order execution. Sometimes a processor architecture offers several kinds of such barriers.

Thread-safe operations on memory need such an assurance at the beginning: all relevant modifications by other threads must have been completed before any read access to the area begins in the current thread. Although the implementation of memory barriers is usually (always?) not specific to a particular memory area, logically, this can be formulated as requiring a function with the following signature:

```
void no_write_read_overlap(const void *p_area, size_t size_of_area);
```

[2]There will be more such unimplemented and possibly unimplementable pseudo functions in this section. They are used in these notes merely for describing the functionality of other operations which *do* exist.

The pseudo function *prevent_change_by_others()* should therefore be thought of as first blocking any future changes and then to contain a call to *no_write_read_overlap()* in order to let outstanding write operations take effect before reading from the area is allowed.

Beyond that, explicit calls to *no_write_read_overlap()* or to a function which implies that call are needed in a program if a thread accesses memory areas by operations which are not inherently thread-safe but which are thread-safe if suitably coordinated. An example can be found in section 6.2.

4.3 Read and Write

Assuming the identifier "`Value`" to denote a copyable[3] type, thread-safe read or write operations can be thought of as:

```
template <typename Value>
Value ts_read(const Value *p_var)
{
  prevent_change_by_others(p_var, sizeof(Value), true);
  Value value = *p_var;
  allow_change(p_var, sizeof(Value));

  return value;
}

template <typename Value>
void ts_write(Value *p_var, Value new_value)
{
  prevent_change_by_others(p_var, sizeof(Value));
  *p_var = new_value;
  allow_change(p_var, sizeof(Value));

  return;
}
```

A slightly extended variant of *ts_write()* will turn out to be much more useful:

```
template <typename Value>
Value ts_swap(Value *p_var, Value new_value)
{
  prevent_change_by_others(p_var, sizeof(Value));
  Value old_value = *p_var;
  *p_var = new_value;
  allow_change(p_var, sizeof(Value));

  return old_value;
}
```

Calls to it can obviously replace calls to *ts_write()* by simply ignoring the return value. In contrast, although the function looks as if it also contained the functionality of *ts_read()*, it can't act as a replacement for that function; section 4.4 will show what is necessary for that.

To which extent these operations are available in hardware typically depends on the size of the object being accessed but also on its alignment.

For example, in Intel's x86 instruction set architecture, certain ordinary read/write operations on memory are guaranteed to be "atomic". For all processors beginning with the Intel486, this includes [2]:

- Reading or writing a byte (octet)
- Reading or writing a word (2 bytes) aligned on a word boundary

[3] A copyable type supports copy construction and copy assignment. Newer editions of the C++ standard with formally defined template parameter requirements call these "`CopyConstructible`" and "`CopyAssignable`", respectively.

- Reading or writing a doubleword (4 bytes) aligned on a doubleword boundary

Examples of *ts_swap()* are the `XCHG` instruction in Intel's x86 ISA [1] and the `AMOSWAP` instruction in the RISC-V ISA [5].

4.4 Compare and Swap

Compare and Swap (CAS) conceptually operates as follows ("`Value`" is assumed to be a copyable type and to support comparison for equality[4]):

```
template <typename Value>
Value ts_cas(Value *p_var, Value expected_value, Value new_value)
{
  prevent_change_by_others(p_var, sizeof(Value));
  Value old_value = *p_var;
  if (old_value == expected_value) *p_var = new_value;
  allow_change(p_var, sizeof(Value));

  return old_value;
}
```

This looks similar to *ts_swap()* except that the memory location is only changed if it contains a specified value. By comparing the return value with `expected_value` the caller can determine whether the change was made.[5]

The functionality of *ts_read()* and *ts_swap()* can be implemented by CAS:

```
template <typename Value>
Value ts_read_impl(Value *p_var)
{
  static const Value some_value = Value();    // The actual value doesn't matter.
  return ts_cas<Value>(p_var, some_value, some_value);
}

template <typename Value>
Value ts_swap_impl(Value *p_var, Value new_value)
{
  Value old_value;
  do {
    old_value = ts_read_impl<Value>(p_var);
      // Need not be thread-safe if it at least sometimes gives the real value.
  } while (ts_cas<Value>(p_var, old_value, new_value) != old_value);

  return old_value;
}
```

CAS can even be used for operations which might fail, provided a failure can be recognized in advance:

[4] Newer versions of the C++ standard call the latter template parameter requirement "`EqualityComparable`".

[5] Some descriptions of CAS (e. g., the Wikipedia article "Compare-and-swap", revision of 2023-02-24) state that this operation merely returns a boolean value indicating whether the change occurred. That does not agree with existing implementations in hardware and substantially reduces the functionality of the operation.

```
int add_to_counter(int *p_var, int increment)
{
  int old_value;
  do {
    old_value = ts_read_impl<int>(p_var);
      // Not making this thread-safe could give spurious failures.
    if (0 <= increment && old_value > INT_MAX - increment
        || increment < 0 && old_value < INT_MIN - increment)
      return -ERANGE;      // Failure
  } while (ts_cas<int>(p_var, old_value, old_value + increment) != old_value);

  return 0;   // Success
}
```

> **Side remark:** I've chosen above und in what follows `int` as the return value type for functions indicating success or failure because I'm not certain how much failure information is needed in all cases. Mostly, though, I suspect this could be changed to `bool`, although I never found convincing reasons for which of the two values should then indicate failure (while for `int` anything but the UNIX convention of a negative value is just unnecessarily insular); my intuitive leaning would be to choose `true` for a failure (because this simplifies the notation for error handling branches), but Microsoft did it exactly in the opposite manner (see, e.g., *ReadFile()*). One might disregard this as Microsoft is anything but known for the high quality of its software design decisions, but programmers might have become accustomed to that interpretation and choosing the opposite one here could lead to confusion and hence unnecessary errors. And, after all, these notes are meant to improve understanding and not to provide a template for thoughtless implementation.

Note that these examples use loops which cannot be guaranteed to ultimately terminate, let alone to terminate at the latest after a number of iterations known in advance, because termination depends on the behaviour of other threads. It's far from being as bad as it could be, though, because termination does not rely on some other thread actually doing something but instead only on a situation occurring when no other thread modifies the variable at the same time.

In this context it should also be mentioned that, because of the limited range of the type `bool`, an implementation of *ts_swap()* for `bool` in terms of CAS is possible without a loop:

```
bool ts_swap_impl(bool *p_var, bool new_value)
{
  return ts_cas<bool>(p_var, !new_value, new_value);
}
```

Because *ts_write()* and *ts_swap()* overwrite the target location unconditionally, CAS cannot be implemented by the operations in section 4.3. Because the converse implementation by CAS *is* possible, CAS is a more powerful operation than any of those.

CAS has been available on some IBM mainframe systems since 1976. On Intel processors starting with the Intel486 (1989), `LOCK CMPXCHG` (Compare and Exchange) can be used (on bytes, words or doublewords; alignment is irrelevant). As CAS requires an instruction set architecture permitting 3 operands, CAS is not usually available in RISC instruction sets.

4.5 Load and Watch/Store Conditional

Some machine architectures provide a pair of instructions which together provide the functionality of CAS. The first instruction, variously called load-link(ed), load-reserve(d) or load-exclusive, is a command which returns the current value of a memory location, the second, usually called store-conditional or store-exclusive, stores a new value at the location only if the latter hasn't been modified since the call to the first command. I'll use the names "load and watch (LW)" and "store conditional (SC)" for these instructions.

Conceptually and in addition to *prevent_change_by_others()/allow_change()*, this requires pseudo functions

```
void mark_for_watching(const void *p_area, size_t size_of_area);
bool not_modified(const void *p_area, size_t size_of_area);
```

to exist in the processor which are used as follows ("`Value`" is again assumed to be a copyable type):

```
template <typename Value>
Value ts_load_and_watch(const Value *p_var)
{
  prevent_change_by_others(p_var, sizeof(Value), true);
  Value value = *p_var;
  mark_for_watching(p_var, sizeof(Value));
  allow_change(p_var, sizeof(Value));

  return value;
}

template <typename Value>
int ts_store_conditional(Value *p_var, Value value)
{
  prevent_change_by_others(p_var, sizeof(Value));
  bool ok = not_modified(p_var, sizeof(Value));
  if (ok) *p_var = value;
  allow_change(p_var, sizeof(Value));

  return ok ? 0 : -1;
}
```

These two functions can be considered as being constructed from parts of *ts_cas()* supplemented by additional infrastructure to allow the functions together to achieve the same functionality as CAS. A difference to CAS is that for SC the value in the memory location is irrelevant; even if another thread has written the already held value to it, this counts as a modification and the subsequent SC is guaranteed to fail.

Implementations of LW/SC vary in the behaviour of *mark_for_watching()* and *not_modified()*. The theoretical ideal is to maintain, for each task separately, a boolean variable "modified" for every byte of memory. A call to *mark_for_watching()* sets the variables for the specified memory region to false for the calling thread. A subsequent write operation to a memory cell by any thread will set the variables for that cell to true for every thread (or at least for all threads other than the writer). The function *not_modified()* than merely checks whether "modified" for the current thread is false for all bytes in the specified memory area.

This theoretical model implies that a thread does not need to always pair calls to LW and SC. While an unpaired SC is useless (effectively, just a thread-safe write which might succeed or not), an unpaired LW is just a thread-safe read operation.

The real world is different from this ideal. In particular, existing implementations of LW/SC can't track all changes to memory, let alone separate that by task, and therefore must occasionally declare a failure for SC although that would not be logically necessary. This seems to be because implementations typically keep track of write operations by checking whether there is an unmodified cache entry for this memory area. Any event which repurposes the cache line or empties the entire cache therefore causes a subsequent SC to fail. Using overlapping LW/SC pairs on locations which might end up in the same cache line is then also a really bad idea for a thread. Some implementations are said to even make SC fail if there was *any* modification to memory since the call to LW (stack!) or if there was an intervening LW call for another location.

However, if it works, LW/SC allows a functionally equivalent implementation of CAS:

```
template <typename Value>
Value ts_cas_impl(Value *p_var, Value expected_value, Value new_value)
{
  Value old_value;
  do {
    old_value = ts_load_and_watch<Value>(p_var);
  } while (old_value == expected_value
    && ts_store_conditional<Value>(p_var, new_value) < 0);

  return old_value;
}
```

The only problem is that when or even whether the loop can be expected to terminate depends on the implementation of LW/SC. The RISC-V ISA, for example, gives some useful assertions on eventual progress with LW/SC under certain conditions ("constrained LR/SC loops") [5].

4.6 Test and (Un-)Set

The operation often called "Test and Set" operates conceptually as follows:

```
bool ts_test_and_set(bool *p_var)
{
  prevent_change_by_others(p_var, sizeof(bool));
  bool old_value = *p_var;
  *p_var = true;
  allow_change(p_var, sizeof(bool));

  return old_value;
}
```

Note that while *ts_test_and_set()* can obviously be used as a thread-safe write for the value "`true`", this can't be done for the value "`false`". A companion operation

```
bool ts_test_and_unset(bool *p_var)
{
  prevent_change_by_others(p_var, sizeof(bool));
  bool old_value = *p_var;
  *p_var = false;
  allow_change(p_var, sizeof(bool));

  return old_value;
}
```

is usually provided for that case. If we compare these two functions with the pseudocode for *ts_swap()* in section 4.3, we see that they are in combination actually equivalent to the latter for the type `bool` in the sense that each side can be implemented with the other:

```
bool ts_test_and_set_impl(bool *p_var)
{
  return ts_swap<bool>(true);
}


bool ts_test_and_unset_impl(bool *p_var)
{
  return ts_swap<bool>(false);
}



bool ts_swap_impl(bool *p_var, bool new_value)
{
  return new_value ? ts_test_and_set(p_var) : ts_test_and_unset(p_var);
}
```

Test and (Un-)Set therefore offers nothing fundamentally new.

In Intel's x86 instruction set architecture, *test_and_set()* is available via `LOCK BTS` (Bit Test and Set) and *test_and_unset()* via `LOCK BTR` (Bit Test and Reset) [1].

4.7 Other Fetch, Operate and Store Primitives

Some processors may offer a whole family of operations of the following kind:

```
template <typename Value, Value (*op)(Value, Value)>
Value ts_fetch_op_store(Value *p_var, Value operand)
{
  prevent_change_by_others(p_var, sizeof(Value));
  Value old_value = *p_var;
  *p_var = (*op)(old_value, operand);
  allow_change(p_var, sizeof(Value));

  return old_value;
}
```

These are only available for functions *op()* which perform some binary operation with their two arguments and do not access other variable data. They also are only useful for transformations where the result is always a meaningful value for all threads involved, i.e., the caller does not need to take corrective action afterwards depending on the result (e.g., for handling an overflow) and a *ts_read()* call for the variable *always* returns everything another thread needs to know about the current value.

An example of such operations is "Fetch and Add" for unsigned integer types:

```
template <typename UnsignedInteger>
UnsignedInteger op_add(UnsignedInteger old_value, UnsignedInteger operand)
{
  // May wrap:
  return old_value + operand;
}

template <typename UnsignedInteger>
UnsignedInteger ts_fetch_and_add(UnsignedInteger *p_var, UnsignedInteger operand)
{
  return ts_fetch_op_store<UnsignedInteger, op_add<UnsignedInteger> >(p_var, operand);
}
```

Functional specializations of Fetch and Add like

```
template <typename UnsignedInteger>
UnsignedInteger ts_increment(UnsignedInteger *p_var)
// This is (*p_var)++, not ++(*p_var).
{
  return ts_fetch_and_add<UnsignedInteger>(p_var, 1);
}
```

may also exist separately but obviously offer nothing new in functionality.

All such operations can be implemented with CAS:

```
template <typename Value, Value (*op)(Value, Value)>
Value ts_fetch_op_store_impl(Value *p_var, Value operand)
{
  Value old_value;
  do {
    static const Value some_value = Value();
    old_value = ts_cas<Value>(p_var, some_value, some_value);
      // This is just a ts_read().
  } while (ts_cas<Value>(p_var, old_value, (*op)(old_value, operand)) != old_value);

  return old_value;
}
```

In contrast, a CAS implementation is not possible with any of these instructions because the target variable is *always* overwritten and the available variants of *op()* do not offer a decision to possibly return the `old_value` based on an additional hidden parameter for the expected old value which would be needed for CAS.

An interesting point is that the read and write functions in section 4.3 can be considered special cases of this family: a *ts_read()* can be obtained with an *op()* simply returning its first argument (`old_value`) and ignoring the second parameter (`operand`), while an *op()* which simply returns its second argument results in an implementation of *ts_swap()*. That explains why the title of this section starts with "Other".

An implementation of *ts_read()* can also be obtained from any *ts_fetch_op_store()* variant where *op()* supports a special value for `operand` which leads to the function returning the value of its first argument (e.g., zero in the cases of addition or a bit-wise OR).

4.8 Conclusion

The instruction sets of processors offer a variety of thread-safe operations, some of which can be implemented in terms of the others. The functionally most powerful are *ts_cas()* or its substitute *ts_load_and_watch()*/ *ts_store_conditional()* because they can be used to implement *all* of the others. This does not preclude using others (in particular when that improves portability or eliminates loops), but as there seems to be no stronger primitive in existing processors, algorithms which require something more powerful may be unimplementable.

Besides functionality, another aspect to consider is necessity. If we look at the instruction set of a specific processor, which thread-safe primitives do we actually need in order to implement something useful and what is possible with it? Some answers to that question can be found in section 6; briefly, the really interesting subset of thread-safe processor instructions seems to be this:

ts_read()
ts_swap()
ts_increment()
Either *ts_cas()* or *ts_load_and_watch()*/*ts_store_conditional()*

The first two I consider essential, the third adds significant functionality, and the instructions in the last line give us everything we can expect.

5 Sharing C++ Objects Between Threads

Besides pointers or references, objects in C++ may also have hidden data in their memory representation which are only valid in the address space as seen when the constructor was run (vtable pointer, virtual inheritance infrastructure). Such objects are unsuitable to be placed in memory regions which will be shared between threads with different address space layouts. While the requirement may be relaxed by suitably restricting the set of threads which may share a particular object, a C++ type is **fully shareable**[6] between threads only if:

- The type is not a pointer, reference, or function type.
- If the type is an array type, the element type must be fully shareable.
- If the type is a class or union type:
  - The type has no non-const static data members.
  - The type has no virtual functions.
  - The type has no virtual base classes.
  - If the type has base classes or data members, the types for those must be fully shareable.

Of course, all programs involved must also have seen a delaration of the shareable type which is translated by their respective compilers to the same memory layout for the type. (Sorry; just thought I'ld mention it for

[6] One has to avoid to talk of "shared objects" in this context because historically that expression is used on UNIX systems for shared object *code* (i.e., shared libraries, implemented by position-independent code). That is a related but orthogonal concept; here we are interested in shared *data* with some programming-language-dependent peculiarities.

completeness because independently developed programs might mean different compilers or at least different compiler settings.)

But that only takes care of the data part of C++ types. Irrespective of multi-threading, all functions operating on instances of a type must also preserve the semantic constraints for the type, i.e., if an object was in a valid state before the function started it must be in a valid state afterwards. And there's more: a collection of objects of certain types might have (and will usually have) such constraints as well, this time on their internal relationships, therefore any function operating on parts of the collection must maintain those as well. This is "merely" a problem of correct programming, but in a multi-threaded environment these functions must in addition achieve that goal in view of other threads possibly calling functions operating on the same objects at the same time: the functions must therefore also be thread-safe.

Even if we have achieved thread safety, though, there is another problem: constraints only make sense during an object's lifetime (here counted from when the constructor finishes until the destructor starts). If a thread's existence starts before or ends after the lifetime of an object shared with other threads, additional measures are necessary to ensure that the thread uses the object only during the latter's lifetime. This is predominantly an initialization problem, because the destructor operates on a well-defined constructed object whereas the object's memory region might hold undefined content before construction. I see at least three solutions:

- Access to the memory region containing the object is only granted after the object has been constructed, e.g., via a function returning a pointer to the object only after it has been known to be constructed.

  This is not actually a solution, though, but merely a delegation of the problem to the entity making the object accessible to threads. Still, such a design removes the necessity for checking for construction from all the functions wishing to operate on the object: if the thread got a valid pointer, the object *has* been constructed.

- If the memory region used for sharing the object has to be allocated by an entity managing resources for the system, there might be an option for obtaining initialized memory. Obviously that will not usually mean that an application-specific constructor has been run, but in particular initialization with zero bytes is a demand which could be granted by a generic entity.

  In that case one can design the type of the object such that the initialized memory region becomes a valid state for the object, and the default constructor can then be implemented as empty. This might, of course, lead to an object initially only offering restricted functionality, which then has repercussions for the functions operating on the object.

  This solution is similar to the first in that it delegates the multi-thread construction problem, this time to a resource manager, but in contrast to the first solution this part can be implemented without the knowledge of application types.

- One of the threads starts before the others and is given the job to run a constructor. When it finishes, it gives permission to start other threads.

  As in the other cases, this solution frees the operating functions from having to check for a constructed object.

From this discussion it should be reasonably obvious that ordinary "worker" threads should only see objects shared with other threads via pointers or references, while construction or destruction will have to be handled by separate components, not least because these functions (unless empty) may only be executed once per object which is an additional requirement beyond thread safety.

This leads us to the remaining question of where the thread-safe functions operating on a C++ object shared between threads should be implemented. The principles of object-oriented programming tell us that it should be as member functions of the type of the object, but if that type has to be fully shareable we lose one of the most useful possibilities of object-oriented languages, namely the polymorphic behaviour of objects. The solution for that is simple: if `ST` is a fully shareable type, encapsulate the thread-local pointers to `ST` within instances of a new type `ST_Ptr` and implement the functions there:

```
class ST_Ptr {
  public:
    ST_Ptr(ST *p) : m_p(p) {}

    // ... Declare functions operating on "*m_p" here ...

  private:
    ST *m_p;
};
```

Objects of these types are not shared and therefore need not be shareable, hence their non-static member functions may be virtual. Logically, `ST` and `ST_Ptr` should collectively be considered one type, only split for implementation purposes. Such an outsourcing of functions might not be necessary and `ST_Ptr` could be replaced by `ST*` if virtual functions are not needed or the threads concerned have compatible address space layouts.

If there is a split, it is also possible to implement some (non-virtual) functions in `ST` and the others in `ST_Ptr`. This is primarily useful in situations where different shareable types logically offer the same interface; that interface can be declared in an abstract class and used as a base class for all relevant `*_Ptr` types where virtual functions map the interface to the method(s) of a shareable type.

6 Some Abstractions Built on the Processors' Functionality

Reuseability of source code is a highly desirable goal in software development. In particular, any dependency on properties of the underlying platform which are logically unrelated to the purpose of the software should be removed or at least contained to such an extent that it becomes reasonably simple to port the source code to another platform. That usually involves a certain measure of abstraction.

The present section therefore proposes a number of concepts which can be implemented in various ways by the synchronization primitives offered by processors. All the classes in this section are fully shareable.

The constructors are formulated as being assumed to be run only once and in a thread-safe environment before any of the other functions are called, but that is only because initialization is more efficiently described in that notation. Any other means of ensuring the same effect are also acceptable.

Similar considerations would apply to destructors, although with the explicit implementations given here they are empty anyway and their discussion would therefore be pointless.

6.1 Binary Switch

A "binary switch" is a two-state (up/down, left/right, on/off, 0/1, true/false, set/cleared, ...) variable with thread-safe operations to order a transition to a specific state.

```
class BinarySwitch {
  public:
    // Assumed to be executed once and in a thread-safe environment:
    BinarySwitch(bool is_up = false) : m_is_up(is_up) {}

    // These are thread-safe and return 0 on success (switch was moved) or a
    // negative value on failure (switch was already in that position).
    int set_to_up();
    int set_to_down();

  private:
    // Not implemented (delete in newer C++ versions):
    BinarySwitch(const BinarySwitch &);
    BinarySwitch &operator=(const BinarySwitch &);

    // Current state of the switch
    bool m_is_up;
};
```

An implementation is already possible with *ts_swap()*:

```
int BinarySwitch::set_to_up()
{
  return ts_swap<bool>(&m_is_up, true) ? -1 : 0;
}

int BinarySwitch::set_to_down()
{
  return ts_swap<bool>(&m_is_up, false) ? 0 : -1;
}
```

A binary switch can be employed as a **mutex** (from "mutual exclusion"), i.e., as a variable limiting the access to some resource to at most one thread. Threads using it have to agree which of the two positions gives the permission; only the thread successfully moving the switch to that position may proceed to the resource and only the thread which did that is allowed to move the switch back.

6.2 Bounded Counter

A "bounded counter" is an integer variable with a restricted range and thread-safe operations to increment or decrement the variable within that range. Let "`Integer`" be an integer type (as defined by [3]):

```
template <typename Integer>
class BoundedCounter {
  public:
    // Assumed to be executed once and in a thread-safe environment:
    BoundedCounter(Integer initial,
      Integer maximum = std::numeric_limits<Integer>::max(),
      Integer minimum = 0);

    // Thread-safe.
    Integer value();

    // Thread-safe; they return 0 on success and a negative value if the
    // counter has already reached its maximum or minimum, respectively.
    int increment();
    int decrement();

  private:
    // Unimplemented for simplicity:
    BoundedCounter(const BoundedCounter &);
    BoundedCounter &operator=(const BoundedCounter &);

    Integer m_counter;
    const Integer m_minimum;
    const Integer m_maximum;
};
```

The order of arguments to the constructor and their defaults have been chosen opportunistically to simplify what I consider the most common applications for such a counter.

The constructor is functionally obvious:

```
template <typename Integer>
BoundedCounter<Integer>::BoundedCounter(
  Integer initial, Integer maximum, Integer minimum) :
  m_counter(initial), m_minimum(minimum), m_maximum(maximum)
{}
```

Of course, if we have to guard against caller errors, the condition

$$\texttt{minimum} \leq \texttt{initial} \leq \texttt{maximum}$$

needs to be checked and, on violation, either an exception has to be thrown or one could simply rearrange the three numbers such that the condition holds (by selecting `m_minimum` = min(`initial`, `maximum`, `minimum`), `m_maximum` = max(...), and `m_counter` to be the third value, whichever parameter holds it).[7]

If CAS is available for "`Integer`", the other methods can be implemented as follows:

```
template <typename Integer>
Integer BoundedCounter<Integer>::value()
{
  return ts_cas<Integer>(&m_counter, 0, 0);
  // Effectively ts_read<Integer>(&m_counter) which could be used instead.
}

template <typename Integer>
int BoundedCounter<Integer>::increment()
{
  Integer old_value;
  do {
    old_value = value();
    if (old_value >= m_maximum) return -1;
  } while (ts_cas<Integer>(&m_counter, old_value, old_value + 1) != old_value);

  return 0;
}

template <typename Integer>
int BoundedCounter<Integer>::decrement()
{
  Integer old_value;
  do {
    old_value = value();
    if (old_value <= m_minimum) return -1;
  } while (ts_cas<Integer>(&m_counter, old_value, old_value - 1) != old_value);

  return 0;
}
```

For an implementation with LW/SC, replace the *ts_cas()* call in *value()* by *ts_load_and_watch()* and the calls in the other two methods by *ts_store_conditional()* for the modified value.

If neither CAS nor LW/SC are available but *ts_swap()* is, one might consider an implementation for `Bounded-Counter` with a binary switch as a mutex in order to protect `m_counter` against access conflicts. Assuming an additional data member like

```
template <typename Integer>
class BoundedCounter {
  ...
  BinarySwitch m_switch;
};
```

(with the "up" state of the switch giving permission to access `m_counter`), we could for example write:

[7] In contrast, we don't need to guard against `minimum` being equal to `maximum` because the resulting counter would be merely useless instead of exhibiting faulty behaviour.

```
template <typename Integer>
int BoundedCounter<Integer>::increment()
{
  if (m_switch.set_to_up() != 0) return -EAGAIN;    // Access conflict

  no_write_read_overlap(&m_counter, sizeof(m_counter));
  Integer old_value = m_counter;
  int rv = -ERANGE;
  if (old_value < m_maximum) {
    ++m_counter;
    rv = 0;
  }
  (void)m_switch.set_to_down();

  return rv;
}
```

(The function *no_write_read_overlap()* is described in section 4.2.) This implementation, however, violates the interface specification given above where *increment()* is only allowed to fail when the counter has reached its maximum; now the caller would also have to be prepared to handle access conflicts. A similar problem arises when we try to implement *value()*: we either need *ts_read()* in addition (in which case we could eliminate the call to *no_write_read_overlap()* above but only if we're also protecting the increment operation below it) or have to change the signature of the function to also return a value indicating success or failure.

Both these problems could be resolved if we had a "waiting switch", i.e., a binary switch with a method

```
void WaitingSwitch::wait_for_up();
```

instead of *set_to_up()* or in addition to it; such a function would not return until the switch has been successfully moved to "up". Using an object of this type for `m_switch` we could eliminate the `EAGAIN` branch above and the error exit for *value()*. In contrast to thread safety, however, waiting belongs to a higher level of architecture depending also on the scheduler, and that introduces additional complications (in particular the possibility of deadlocks) which I prefer not to discuss at this point; presenting thread-safe operations and wait functionality together as a unit is in my opinion detrimental to understanding. Still, readers may wish to keep in mind that such an alternative implemention of bounded counters is possible.

Obviously the interface for `BoundedCounter` could be extended with more general operations, e.g.,

```
int add(Integer inc);
int assign(Integer value);
```

hence what we actually have here is an implementation of a thread-safe integer variable. However, as a thread cannot rely on the variable to preserve a stored result in the presence of other threads having also access to it, this functionality is not really helpful; in particular, the function *value()* is useless for this purpose. A bounded counter used by multiple threads should therefore rather be thought of as a set of increments: the elements in this set have no other property than their existence in the set, they have in particular no identity, and `value()` − `m_minimum` merely gives the number of elements currently in this set.

It should also be obvious that a bounded counter with a range (i.e., `m_maximum` − `m_minimum`) of 1 encompasses the same functionality as a binary switch, hence the latter is a reduced variant of the former. That is also apparent in C++: because `bool` is one of the integer types, instantiating `BoundedCounter<bool>` is permitted and the reader may wish to compare it with `BinarySwitch`.

Knowledgeable persons will also have realized that a bounded counter with a large (and therefore practically irrelevant) upper bound is close to an implementation of a single instance of Dijkstra's semaphores, except that the decrement function on semaphores will never fail but instead will wait if necessary until the operation becomes possible.

6.3 Number Dispenser

A "number dispenser"[8] is a take-a-number device for serializing requests. In what follows, "`NumberType`" is assumed to be an unsigned integer type:

```
template <typename NumberType>
class NumberDispenser {
  public:
    // Assumed to be executed once and in a thread-safe environment:
    NumberDispenser() : m_next_number(0), m_number_to_go(0) {}

    // Thread-safe methods
    NumberType take_a_number();
    bool may_proceed(NumberType number) const;
    bool may_proceed_immediately();    // implementation is optional
    void am_done();

  private:
    // Unimplemented for simplicity:
    NumberDispenser(const NumberDispenser &);
    NumberDispenser &operator=(const NumberDispenser &);

    // Requests are assigned sequential numbers (wrapping is permitted)
    NumberType m_next_number;
    // Number of the request which is currently permitted to proceed (wraps)
    NumberType m_number_to_go;
};
```

The function *may_proceed_immediately()* is a convenience function for clients which do not wish to bother with storing a request number. However, if that call fails the client has not obtained a place in the wait queue.

Some care must be taken to ensure that clients of this class perform calls to its methods in the correct order for every instance of the class:

- The client must *either*:
  - First call *take_a_number()* and store the return value.
  - Then call *may_proceed()* at suitable intervals with the stored number until the function returns `true`. (Such a call may also be repeated after it first returned `true`.)

  or (if supported):
  - Call *may_proceed_immediately()* and if it returns `false` leave this call sequence.
- When *may_proceed()* for the previously stored number or *may_proceed_immediately()* returned `true`, the client is permitted to do whatever is regulated by this object and must then call *am_done()*.

No other call sequences involving *take_a_number()*, *may_proceed()*, *may_proceed_immediately()* or *am_done()* are permitted and any client which has started such a sequence is required to finish it.

A not-so-immediately-obvious consequence of these rules is that, unless external measures are in place to handle the situation, a thread is not allowed to terminate while within such a call sequence. (That requirement is not specific to `NumberDispenser` but occurs also with other synchronization operations; just consider a `BinarySwitch` used as a mutex.) Note that handling this requirement in the phase between a call to *take_a_number()* and the first successful *may_proceed()* call is particularly restrictive because there is no way to return an "unused" number. This also means that, while in such a sequence, a thread may not perform any operation which might cause the thread to crash.

In order to ensure that, at every moment, at most one client is between a successful *may_proceed()*/*may_proceed_immediately()* and the subsequent *am_done()* call (mutex functionality), each client presently within

[8] A more descriptive name would in my opinion be "sequencer", but that name already belongs to a variant of this type with functionality for waiting [4] instead of the `take_a_number()` and `may_proceed*()` functions.

a call sequence as defined above must be assigned a different request number. That is only possible if the cardinality of `NumberType` is at least as large as the number of unfinished call sequences existing at any moment; if we permit at most one call sequence per task at any time, it is simplest to choose a type sufficient for the maximal number of tasks supported by the scheduler.

Because the restrictions imposed here cannot be checked by looking only at one point in a client's code, a robust software design will encapsulate instances of this class in an environment where the possible call sequences can be determined and checked with reasonable effort.

The main methods of this class can be implemented with *ts_read()* and *ts_increment()*:

```
template <typename NumberType>
NumberType NumberDispenser<NumberType>::take_a_number()
{
  return ts_increment<NumberType>(&m_next_number);
}


template <typename NumberType>
bool NumberDispenser<NumberType>::may_proceed(NumberType number) const
{
  return ts_read<NumberType>(&m_number_to_go) == number;
}


template <typename NumberType>
void NumberDispenser<NumberType>::am_done()
// Assumed to be called only by clients which observe the call sequence rules.
{
  (void)ts_increment<NumberType>(&m_number_to_go);
}
```

An implementation of *may_proceed_immediately()* is possible if CAS is available:

```
template <typename NumberType>
bool NumberDispenser<NumberType>::may_proceed_immediately()
{
  NumberType to_go = ts_read<NumberType>(&m_number_to_go);
  return ts_cas<NumberType>(&m_next_number, to_go, to_go + 1) == to_go;
}
```

However, if CAS is available we might as well use its capabilities to fortify *take_a_number()* against a violation of the cardinality constraint for `NumberType`:

```
template <typename NumberType>
NumberType NumberDispenser<NumberType>::take_a_number()
{
  NumberType number;
  do {
    number = ts_read<NumberType>(&m_next_number);
    NumberType diff = ts_read<NumberType>(&m_number_to_go) - number;
    if (diff == 1)
      throw std::runtime_error("Too many active calls to NumberDispenser.");
  } while (ts_cas<NumberType>(&m_next_number, number, number + 1) != number);

  return number;
}
```

Note that the exception, if thrown, will occur before the caller has obtained a number and therefore does not require an addition to the call sequence rules.

6.4 Funnel

A `NumberDispenser` realizes what I like to call a "1-at-a-time funnel", a construct organizing a number of requests such that no more than one is proceeding at any time and the others are effectively organized in a FIFO queue at the funnel's mouth. Combining a `NumberDispenser` with a `BoundedCounter` we can generalize this to an "n-at-a-time funnel":

```
template <typename NumberType>
class Funnel {
  public:
    // Simplification (it's not necessary for these types to be the same):
    typedef NumberType CounterRange;

    // Assumed to be executed once and in a thread-safe environment:
    Funnel(CounterRange w) : m_available(w), m_width(w) {}

    // Thread-safe methods for clients of the funnel
    NumberType take_a_number();
    bool may_proceed(NumberType number);
    bool may_proceed_immediately();  // Optional
    void am_done();

    // Thread-safe method for everyone (optional)
    CounterRange width();

    // Thread-safe methods for service providers downstream from the funnel (optional)
    int widen();
    int narrow();

  private:
    // Not implemented:
    Funnel(const Funnel &);
    Funnel &operator=(const Funnel &);

    NumberDispenser<NumberType> m_dispenser;
    BoundedCounter<CounterRange> m_available;
    // Can be replaced by a constant if neither widen() nor narrow() are provided:
    BoundedCounter<CounterRange> m_width;
};
```

The argument for the constructor is the "width" n of the funnel, i.e., the maximal number of clients allowed to be in it at any time.

As with `NumberDispenser`, there are call sequence rules:

- A client must *either*
  - First call `take_a_number()` and store the result.
  - Then call `may_proceed()` in suitable intervals until it returns `true`.

  or (if supported):
  - Call *may_proceed_immediately()* and if it returns `false` leave this call sequence.
- Then the client may do whatever is regulated by the funnel and should at the end call `am_done()`.

As before, no other call sequences involving *take_a_number()*, *may_proceed()*, *may_proceed_immediately()* or *am_done()* are permitted, and a client which started such a sequence with *take_a_number()* must at least progress until a successful *may_proceed()* call. Every client should also finish the entire sequence, although in this case up to `width()` − 1 clients may choose not to make the final call without the funnel becoming blocked because of that.

The difference to `NumberDispenser` in these rules is that (a) *may_proceed()* may not be called again after it returned `true`, and (b) the final calls to `am_done()` by the clients need not be made in the order of the numbers obtained from *take_a_number()*.

Because of the use of `NumberDispenser` we have also inherited the rule that the cardinality of `NumberType` should or must be large enough to accommodate the maximal number of clients expected at any time, but in this case it means the number of clients between calls to *take_a_number()* and a successful *may_proceed()* call plus those currently in a *may_proceed_immediately()* call.

How to implement the functions intended for clients is fairly obvious:

```
template <typename NumberType>
NumberType Funnel<NumberType>::take_a_number()
{
  return m_dispenser.take_a_number();
}

template <typename NumberType>
bool Funnel<NumberType>::may_proceed(NumberType number)
{
  if (!m_dispenser.may_proceed(number) || m_available.decrement() != 0)
    return false;
  m_dispenser.am_done();

  return true;
}

template <typename NumberType>
bool Funnel<NumberType>::may_proceed_immediately()
{
  if (!m_dispenser.may_proceed_immediately()) return false;
  bool rv = m_available.decrement() == 0;
  m_dispenser.am_done();

  return rv;
}

template <typename NumberType>
void Funnel<NumberType>::am_done()
// Assumed to be called only by clients which observe the call sequence rules.
{
  (void)m_available.increment();
}
```

The function *width()* does not seem very useful but is trivial to implement:

```
template <typename NumberType>
typename Funnel<NumberType>::CounterRange Funnel<NumberType>::width()
{
  return m_width.value();
}
```

In contrast, a really interesting feature of this class is that it is possible to change the width of the funnel without interrupting its operation:

```
template <typename NumberType>
int Funnel<NumberType>::widen()
{
  int rv = -1;
  if (m_width.increment() == 0) {
    (void)m_available.increment();
      // Cannot fail because of the invariant m_available.value() <= m_width.value().
    rv = 0;
  }

  return rv;
}

template <typename NumberType>
int Funnel<NumberType>::narrow()
{
  int rv = -1;
  if (m_available.decrement() == 0) {
    (void)m_width.decrement();
    rv = 0;
  }

  return rv;
}
```

A call to *narrow()* will fail if and only if the number of clients currently in the funnel is equal to the current width (which could be zero).

6.5 Shared/Exclusive Locks

Section 3 explained that read operations on a shared memory region may overlap while write or update operations need exclusive access. The data structures and protocols used for coordination between threads in such a situation have been given various names (e. g., read/write locks or multi-reader/single-writer (MRSW) locks); I'm going to call them "shared/exclusive locks".

A simple implementation can be built with a binary switch and a bounded counter:

```
class SimpleSELock {
  public:
    // Assumed to be executed once and in a thread-safe environment:
    SimpleSELock() : m_shared(0) {}

    bool may_proceed_shared();
    bool may_proceed_exclusive();

    void am_done_shared();
    void am_done_exclusive();

  private:
    // Not implemented:
    SimpleSELock(const SimpleSELock &);
    SimpleSELock &operator=(const SimpleSELock &);

    BinarySwitch m_exclusive;
    BoundedCounter<unsigned int> m_shared;
};
```

As before, call sequence rules have to be obeyed: any client having successfully called one of the *may_proceed_*()* functions must later call the corresponding *am_done_*()* function and the latter may not be called without a preceding successful call to the former.

The implementation of the methods can be done as follows:

```
bool SimpleSELock::may_proceed_shared()
{
  if (m_exclusive.set_to_up() != 0) return false;
  int rv = m_shared.increment();
  (void)m_exclusive.set_to_down();
  return rv == 0;
}

bool SimpleSELock::may_proceed_exclusive()
{
  if (m_exclusive.set_to_up() != 0) return false;
  if (m_shared.value() != 0) {
    (void)m_exclusive.set_to_down();
    return false;
  }
  return true;
}

void SimpleSELock::am_done_shared()
// Assumed to be called only by clients which observe the call sequence rules.
{
  (void)m_shared.decrement();
}

void SimpleSELock::am_done_exclusive()
// Assumed to be called only by clients which observe the call sequence rules.
{
  (void)m_exclusive.set_to_down();
}
```

A problem with this implementation is that overlapping shared-access intervals may prevent a successful request for exclusive access indefinitely. I'm sure there are a number of ways to prevent that[9], but given our acquaintance with number dispensers only one solution is obvious, simple and fair:

```
template <typename NumberType>
class SELock {
  public:
    // Assumed to be executed once and in a thread-safe environment:
    SELock() {}

    NumberType take_a_number();
    bool may_proceed_shared(NumberType number);
    bool may_proceed_exclusive(NumberType number);

    void am_done_shared();
    void am_done_exclusive();

  private:
    // Not implemented:
    SELock(const SELock &);
    SELock &operator=(const SELock &);

    NumberDispenser<NumberType> m_dispenser;
    SimpleSELock m_lock;
};
```

[9]Years ago and before I knew of number dispensers, I once used a counter of pending exclusive requests to decide whether to grant new sharing requests.

Instead of using an instance of `SimpleSELock` we could insert its parts, but the choice presented here should make it clearer that this is a general design pattern which can be used to extend any request object with a FIFO queue in front of it; the reader may wish to compare the signatures of the member functions between `SimpleSELock` and `SELock`.

Again, there are requirements for the cardinality of `NumberType` and call sequence rules which I'll omit this time. The implementation is similarly boringly obvious:

```
template <typename NumberType>
NumberType SELock<NumberType>::take_a_number()
{
  return m_dispenser.take_a_number();
}

template <typename NumberType>
bool SELock<NumberType>::may_proceed_shared(NumberType number)
{
  if (!m_dispenser.may_proceed(number) || !m_lock.may_proceed_shared())
    return false;
  m_dispenser.am_done();
  return true;
}

template <typename NumberType>
bool SELock<NumberType>::may_proceed_exclusive(NumberType number)
{
  if (!m_dispenser.may_proceed(number) || !m_lock.may_proceed_exclusive())
    return false;
  m_dispenser.am_done();
  return true;
}

template <typename NumberType>
void SELock<NumberType>::am_done_shared()
// Assumed to be called only by clients which observe the call sequence rules.
{
  m_lock.am_done_shared();
}

template <typename NumberType>
void SELock<NumberType>::am_done_exclusive()
// Assumed to be called only by clients which observe the call sequence rules.
{
  m_lock.am_done_exclusive();
}
```

It's also obviously trivial to add *may_proceed_immediately_*()* functions to this class if that is desired.

Note that an `SELock` object provides a significant part of the functionality needed to implement something similar to the pseudo functions *prevent_change_by_others()/allow_change()* from section 4.1. In that sense we've now returned to the beginning.

Abbreviations

CAS	Compare and Swap
FIFO	First in, first out
ISA	Instruction Set Architecture
LW	Load and Watch
NMI	Non-maskable interrupt
RISC	Reduced Instruction Set Computer
SC	Store Conditional

References

[1] Intel Corporation. *Intel® 64 and IA-32 Architectures — Software Developer's Manual — Volume 2 (2A, 2B, 2C & 2D): Instruction Set Reference, A–Z*, June 2021. Order Number: 325383-075US.

[2] Intel Corporation. *Intel® 64 and IA-32 Architectures — Software Developer's Manual — Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, June 2021. Order Number: 325384-075US.

[3] International Organization for Standardization/International Electrotechnical Commission. *Programming languages — C++*. International Standard ISO/IEC 14882(E).

[4] David P. Reed, Rajendra K. Kanodia. Synchronization with Eventcounts and Sequencers. *Communications of the ACM*, 22(2), February 1979.

[5] RISC-V Foundation. *The RISC-V Instruction Set Manual — Volume I: Unprivileged ISA*, December 2019. Document version 20191213.

Legal Information

Greifswaldstrasse 28
38124 Braunschweig
Germany
`http://home.htp-tel.de/lottermose2`