Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 — Principles of Computer Systems

Handout 30 May 7, 1997

Concurrent Caching

This handout presents several specifications and implementations for caches in concurrent systems. We begin with a specification for `CoherentMemory`, the kind of memory we would really like to have; it is just a function from addresses to data values We also specify the `IncoherentMemory` that has fast implementations, but is not very nice to use. Then we show how to change `IncoherentMemory` so that it implements `CoherentMemory` with as little communication as possible. We describe various strategies, including invalidation-based and update-based strategies, and strategies using incoherent memory plus locking.

Since the various strategies used in practice have a lot in common, we unify the presentation using successive refinements. We start with an cache implementation `GlobalImpl` that clearly works, but is not practical to implement directly because it is extremely non-local. Then we refine `GlobalImpl` in stages to obtain (abstract versions of) practical implementations. First we show how to use reader/writer locks to get a practical version of `GlobalImpl` called a coherent cache. We do this in two stages, an ideal cache `CurrentCaches` and a concrete cache `ExclusiveLocks`.

The caches change the guards on internal actions of `IncoherentMemory` as well as on the external read and write actions, so they can't be implemented externally, simply by adding a test before each read or write of `IncoherentMemory`, but require changes to its insides. There is another way to use locks to get a different practical version of `GlobalImpl`, called `ExternalLocks`. The advantage of `ExternalLocks` is that the locking is decoupled from the internal actions of the memory system so that it can be implemented separately, and hence `ExternalLocks` can run entirely in software on top of a memory system that only implements `IncoherentMemory`. In other words, `ExternalLocks` is a practical way to program coherent memory on a machine whose hardware provides only incoherent memory.

There are many practical implementations of the methods that are described abstractly here. Most of them originated in the hardware of shared-memory multiprocessors.[1] It is also possible to implement shared memory in software, relying on some combination of page faults from the virtual memory and checks supplied by the compiler. This is called 'distributed shared memory'.[2] Many of the techniques have been re-invented for coherent distributed file systems.[3]

All our implementations make use of a global memory that is modeled as a function from addresses to data values; in other words, the specification for the global memory is simply `CoherentMemory`. This means that an actual implementation may have a recursive structure, in which the top-level implementation of `CoherentMemory` using one of our algorithms contains a global memory that is implemented with another algorithm and contains another global memory, etc. This recursion terminates only when we lose interest in another level of virtualization. For example,

a processor's memory may consist of a first level cache plus
a global memory made up of a second level cache plus
a global memory made up of a main memory plus
a global memory made up of a local swapping disk plus
a global memory made up of a file server ....

Specifications

First we recall the spec for ordinary coherent memory. Then we give the spec for efficient but ugly incoherent memory. Finally, we discuss an alternative, less intuitive way of writing these specs.

Coherent memory

The first specification is for the memory that we really want, which ensures that all memory operations appear atomic. It is essentially the same as the `SimpleMemory` specification from Handout 5, except that `m` is defined to be total. In the literature, this is sometimes called a 'linearizable' memory.

```
MODULE CoherentMemory [P, A, D] EXPORT Read, Write =
% Arguments are Processors, Addresses and Data

TYPE M          =  A -> D SUCHTHAT (\ f: A->D | (ALL a | f!a))
VAR  m

APROC Read(p) -> D = << RET m(a) >>
APROC Write(p, a, d) = << m := d >>

END CoherentMemory
```

From this point we drop the `a` argument and study a memory with just one location. Since everything about the specs and implementations holds independently for each address, we don't lose anything by doing this, and it reduces clutter. We also write the `p` argument as a subscript, again to make the specs easier to read. The previous spec becomes

```
MODULE CoherentMemory [P, A, D] EXPORT Read, Write =
% Arguments are Processors, Addresses and Data

TYPE M          =  D                                    % Memory
VAR  m

APROC Read_p -> D = << RET m >>
APROC Write_p(d) = << m := d >>

END CoherentMemory
```

Of course, some implementations have limits on the number of addresses in a cache, or other resource limitations that can only be expressed by considering all the addresses at once, but we will not study this kind of detail here.

Incoherent memory

The next spec describes the minimum guarantees made by hardware: there is a private cache for each processor, and internal actions that move data back and forth between caches and the main memory, and between different caches. The only guarantee is that data written to a cache is not overwritten in that cache by anyone else's data. However, there is no ordering on writes from the

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1996, chapter 8, pp 635-754.
[2] K. Li and P. Hudak, Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* **7**, 4 (Nov. 1989), pp 321-359. For recent work in this active field see any ISCA, ASPLOS, OSDI, or SOSP proceedings.
[3] M. Nelson et al., Caching in the Sprite network file system. *ACM Transactions on Computer Systems* **11**, 2 (Feb. 1993), pp 228-239. For recent work in this active field see any OSDI or SOSP proceedings.

cache to main memory. Since this is not enough to get any useful work done, we add a `Barrier` synchronization operation that forces data from the cache into memory. This can be used after a `Write` to ensure that an update has been written back to main memory, or before a `Read` to ensure that the data being read is current. `Barrier` was called `Sync` when we studied disks and file systems in handout 7.

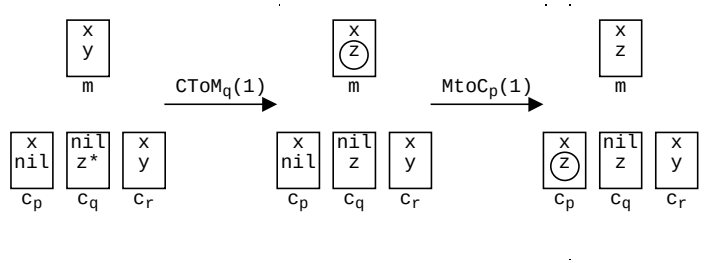
Note that `Read` has a guard `Live` that it makes no attempt to satisfy (hardware implementations usually have an explicit flag called `valid`). Instead, there is another action `MtoC` that makes `Live` true. In a real system an attempt to do a `Read` will trigger a `MtoC` so that the `Read` can go ahead, but in Spec we can omit the direct linkage between the two actions and let the non-determinism do the work. We use this coding trick repeatedly in this handout.

We adopt the convention that an invalid cache entry has the value `nil`.

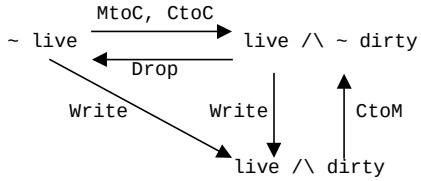
```
MODULE IncoherentMemory [P, A, D] EXPORT Read, Write, Barrier =

TYPE M          =  D                                  % Memory
     C          =  P -> (D + Null)                    % Cache

VAR  m          :  CoherentMemory.M                   % main memory
     c          := C{* -> nil}                        % local caches
     dirty      :  P -> Bool := {*->false}            % dirty flags

% INVARIANT Inv1: (ALL p | c!p)                       % every processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> c_p # nil)      % dirty data is in the cache

APROC Read_p -> D = << Live_p => RET c_p >>            % MtoC gets data into cache
APROC Write_p(d)  = << c_p := d; dirty_p := true >>

APROC Barrier_p = << ~ Live_p => SKIP >>               % wait until not in cache

FUNC Live_p -> Bool = RET (c_p # nil)

% Internal actions

THREAD Internal_p = DO MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p [] SKIP OD

APROC MtoC_p = << ~ dirty_p => c_p := m >>             % Copy memory to cache
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >>% Copy cache to memory
APROC CtoC_p,p' = << ~ dirty_p' /\ Live_p => c_p' := c_p >>  % Copy from cache p to p
APROC Drop_p = << ~ dirty_p => c_p := nil >>           % Drop clean data from cache

END IncoherentMemory
```

This memory is 'incoherent': different caches can have different data for the same address, so that reads and writes by different processors may see completely different data. Thus, it does not implement the `CoherentMemory` specification given earlier. However, after a $Barrier_p$, c_p is guaranteed to agree with `m` until the next time `m` changes.[4] There are commercial machines whose memory systems have essentially this spec.[5] Others have explored similar specs.[6]

Here is a simple example which shows the contents of two addresses `0` and `1` in `m` and in three processors `p`, `q`, and `r`. A dirty value is marked with a `*`, and circles mark values that have changed. Initially $Read_q(1)$ yields the dirty value `z`, $Read_r(1)$ yields `y`, and $Read_p(1)$ blocks because $c_p(1)$ is `nil`. After the $CtoM_q$ the global location `m(1)` has been updated with `z`. After the $MtoC_p$, $Read_p(1)$ yields `z`. One way to ensure that the $CtoM_q$ and $MtoC_p$ actions happen before the $Read_p(1)$ is to do $Barrier_q$ followed by $Barrier_p$ between the $Write_q(1)$ that makes `z` dirty in c_a and the $Read_p(1)$.



Here are the possible transitions of `IncoherentMemory` for a given address.



This is the weakest shared-memory spec that seems likely to be useful in practice. But perhaps it is too weak. Why do we introduce this messy incoherent memory? Wouldn't we be much better off with the simple and familiar coherent memory? There are two reasons to prefer `IncoherentMemory`.

- An implementation of `IncoherentMemory` can run faster — there is more locality and less communication. As we will see later in `ERxternalLocks`, software can batch the communication that is needed to make a coherent memory out of `IncoherentMemory`.
- Even `CoherentMemory` is tricky to use when there are concurrent clients. Experience has shown that it's necessary to have wizards to package it so that ordinary programmers can use it safely. This packaging takes the form of rules for writing concurrent programs and procedures that encapsulate references to shared memory. The two most common examples are:

  Mutual exclusion / critical sections / monitors, which ensure that a number of references to shared memory can be done without interference, just as in a sequential program. Reader/writer locks are an important variation.

  Producer-consumer buffers.

For the ordinary programmer only the simplicity of the package is important, not the subtlety of its implementation. We need a smarter wizard to package `IncoherentMemory`, but the result is as simple to use as the packaged `CoherentMemory`.

[4] An alternative version of `Barrier` has the guard $\sim live_p \setminus/ (c_p = m)$; this is equivalent to the current $Barrier_p$ followed by an optional $MtoC_p$. You might think that it's better because it avoids a copy from `m` to c_p in case they already agree. But this is a spec, not an implementation, and the change doesn't affect its external behavior.

[5] Digital Equipment Corporation, *Alpha Architecture Handbook*, 1992.

[6] Gharachorloo, K., et al., Memory consistency and event ordering in scalable shared-memory multiprocessors, *Proc. 17th Symposium on Computer Architecture*, 1990, pp 15-26. Gibbons, P. and Merritt, M., Specifying nonblocking shared memories, *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp 158-168.

Specifying legal histories directly

It's common in the literature to write the specifications CoherentMemory and IncoherentMemory explicitly in terms of legal sequences of references in each processor, rather than as state machines (see the references in the previous section). We digress briefly to explain this approach.

For CoherentMemory, there must be a total ordering of all the Read_p and $\text{Write}_p(d)$ actions done by the processors (for all the addresses) that

- respects the order at each p, and
- such that for each Read and closest preceding Write(d), the Read returns d.

For IncoherentMemory, *for each address* there must be a total ordering of the Read_p, Write_p, and Barrier_p actions done by the processors that has the same properties. IncoherentMemory is weaker than CoherentMemory because it allows references to different addresses to be ordered differently. If there were only one address and no other communication (so that you couldn't see the relative ordering of the operations), you couldn't tell the difference between the two specs. A real barrier operation usually does a Barrier for every address, and thus forces all the references before it at a given processor to precede all the references after it.

It's not hard to show that these specs written in terms of ordering are almost equivalent to CoherentMemory and IncoherentMemory. Actually they are somewhat more permissive. For example, IncoherentMemory allows the following history

- Initially x=1, y=1.
- Processor p reads 4 from x, then writes 8 to y.
- Processor q reads 8 from y, then writes 4 to x.

For x we have the ordering $\text{Write}_q(4)$; Read_p, and for y the ordering $\text{Write}_p(8)$; Read_q.

We can rule out this kind of predicting the future by observing that the processors make their references in some total order in real time, and requiring that a suitable ordering exist for the references in each prefix of this real time order. With this restriction, the two versions of IncoherentMemory are equivalent.

Implementations

We give a sequence of refinements that implement CoherentMemory and are successively more practical: GlobalImpl, Current Caches, and ExclusiveLocks. Then we give a different kind of implementation that is based on IncoherentMemory.

Global implementation

Now we give code that implements CoherentMemory. We obtain it simply by strengthening the guards on the operations of IncoherentMemory (omitting Barrier, which we don't need). This code is not practical, however, because the guards involve checking global state, not just the state of a single processor. This module, like later ones, maintains the invariant Inv3 that an address is dirty in at most one cache; this is necessary for the abstraction function to make sense. Note that the definition of Current says that the cache agrees with the abstract memory.

We show only the code that differs from IncoherentMemory, boxing the new parts.

```
MODULE GlobalImpl [P, A, D] EXPORT Read, Write =  % implements CoherentMemory

TYPE ...                                          % as in IncoherentMemory
VAR  ...

% ABSTRACTION: CoherentMemory.m = (Clean() => m [*] {p | dirty_p | c_p}.choose)

% INVARIANT Inv3:                                 % dirty in at most one cache
    (ALL p, p' | dirty_p /\ dirty_p' ==> p = p')

APROC Read_p -> D = << Current_p  => RET c_p >>    % Read only current data
APROC Write_p(d) =                                % Write maintains Inv3
    << Clean() \/ dirty_p  => c_p := d; dirty_p := true >>

FUNC Current_p =                                  % p's cache is current?
    RET c_p = (Clean() => m [*] {p | dirty_p | c_p}.choose)
FUNC Clean() = RET (ALL p | ~ dirty_p)            % all caches are clean?

% Same internal actions as IncoherentMemory.

END GlobalImpl
```

Notice that the guard on Read checks that the data in the processor's cache is current, that is, equals the value currently stored in the abstract memory. This requires finding the most recent value, which is either in the main memory (if no processor has a dirty value) or in some processor's cache (if a processor has a dirty value). The guard on Write ensures that a given address is dirty in at most one cache. These guards make it obvious that GlobalImpl implements CoherentMemory, but both require checking global state, so they are impractical to implement directly.

Code in which caches are always current

We can't implement the guards of GlobalImpl directly. In this section, we refine GlobalImpl a bit, replacing some (but not all) of the global tests. We carry this refinement further in the following sections. Our strategy for correctness is to always strengthen the guards in the actions, without changing the rest of the code. This makes it obvious that we simulate the previous module and that existing invariants hold. The only thing to check is that new invariants hold.

The main idea of CurrentCaches is to always keep the data in the caches current, so that we no longer need the Current guard on Read. In order to achieve this, we impose a guard on a write that allows it to happen only if no other processor has a cached copy. This is usually implemented by having a write invalidate other cached copies before writing; in our code Write waits for Drop actions at all the other caches that are live. Note that Only implies the guard of GlobalImpl.Write because of Inv2 and Inv3, and Live implies the guard of GlobalImpl.Read because of Inv4. This makes it obvious that CurrentCaches implements GlobalImpl. CurrentCaches uses the non-local functions Clean and Only, but it eliminates Current.

As usual, the parts not shown are the same as in the last module, GlobalImpl.

```
MODULE CurrentCaches ... =                          % implements GlobalImpl

TYPE ...                                            % as in IncoherentMemory
VAR  ...

% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.

% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)    % data in caches is current

...

FUNC Only_p -> Bool = RET (ALL p' | Live_p' ==> p' = p) % appears only in p's cache

APROC Read_p -> D = << Live_p => RET c_p >>          % read locally; OK by Inv4
APROC Write_p(d) =                                  % write locally the only copy
     << Only_p => c_p := d; dirty_p := true >>

...

APROC MtoC_p = << Clean() => c_p := m >>             guard maintains Inv4
...

END CurrentCaches
```

Code using exclusive locks

The next code refines `CurrentCaches` by introducing an exclusive (write) lock with a `Free` test and `Acquire` and `Release` actions. A writer must hold the lock on an object while it writes, but a reader need not hold any lock (`Live` acts as a read lock according to `Inv6`). Thus, multiple readers can read in parallel, but only one writer can write at a time, and only if there are no concurrent readers. The code ensures that while a processor holds a lock, no other cache has a copy of the locked object. It uses the non-local functions `Clean` and `Free`, but everything else is local. Again, the guards are stronger than those in `CurrentCaches`, so it's obvious that `ExclusiveLocks0` implements `CurrentCaches`. We show the changes from `CurrentCaches`.

```
MODULE ExclusiveLocks0 ... =                        % implements CurrentCaches

TYPE ...                                            % as in IncoherentMemory
VAR  ...
     lock        :  P -> Bool := {*->false}         % p has lock on cache?

% ABSTRACTION to CurrentCaches: Identity on m, c, and dirty.

% INVARIANT Inv5:                                   % lock is exclusive
     (ALL p, p' | lock_p /\ lock_p' ==> p = p')
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)       % locked data is only copy

...

APROC Write_p(d) =                                  % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>
...

FUNC Free() -> Bool = RET (ALL p | ~ lock_p)        % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p  [] SKIP OD

APROC MtoC_p =                                      % guard maintains Inv4, Inv6
     << Clean() /\  (lock_p \/ Free()) => c_p := m >>
APROC CtoC_p,p' =                                   % guard maintains Inv6
     << Free()  /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>

APROC Acquire_p = << Free() /\ Only_p => lock_p:=true >> % exclusive lock is on cache
APROC Release_p = << lock_p := false >>              % release at any time

...

END ExclusiveLocks0
```

Note that this all works even in the presence of cache-to-cache copying of dirty data; a cache can be dirty without being locked. A strategy that allows such copying is called *update-based*.

The remaining global tests are the `Only` test in `Acquire`, the `Clean` test in `MtoC`, and the `Free` tests in `Acquire`, `MtoC`, and `CtoC`. In hardware these are most commonly implemented by snooping on a bus. A processor can broadcast on the bus to check that:

- No one else has a copy (`Only`).
- No one has a dirty copy (`Clean`).
- No one has a lock (`Free`).

For this to work, another processor that sees the test must either abandon its copy or lock, or signal `false`. The `false` signals are usually generated at exactly the same time by all the processors and combined by a simple 'or' operation. The processor can also request that the others relinquish their locks or copies; this is called 'invalidating'. Relinquishing a dirty copy means first writing it back to memory, whereas relinquishing a non-dirty copy means just dropping it from the cache. Sometimes the same broadcast is used to invalidate the old copies and update the caches with new copies, although our code breaks this down into separate `Drop`, `Write`, and `CtoC` actions.

Keeping dirty data locked

In the next module, we eliminate the cache-to-cache copying of dirty data. We modify `ExclusiveLocks` so that locks are held longer, until data is no longer dirty. Besides the delayed lock release, the only significant change is in the guard of `MtoC`. Now data can only be loaded into a cache `p` if it is not dirty in `p` and is not locked elsewhere; together, these facts imply that the data item is clean, so we no longer need the global `Clean` test.

```
MODULE ExclusiveLocks ... =                         % implements ExclusiveLocks0

TYPE ...                                            % as in ExclusiveLocks0
VAR  ...

% ABSTRACTION to ExclusiveLocks0: Identity on m, c, dirty, and lock.

% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)      % dirty data is locked

...

APROC MtoC_p =                                      % guard ensures Clean()
     << ~  dirty_p /\ (lock_p \/ Free()) => c_p := m >>
APROC Release_p = << ~  dirty_p => lock_p := false >>  % Don't release if dirty
...

END ExclusiveLocks
```

For completeness, we give all the code for `ExclusiveLocks`, since there have been so many incremental changes.

```
MODULE ExclusiveLocks[P,A,D] EXPORT Read,Write = % implements CoherentMemory

TYPE M = D                                          % Memory
     C = P -> (D + Null)                            % Cache

VAR  m          :  CoherentMemory.M                 % main memory
     c          := C{* -> nil}                      % local caches
     dirty      :  P -> Bool := {*->false}          % dirty flags
     lock       :  P -> Bool := {*->false}          % p has lock on cache?

% ABSTRACTION to ExclusiveLocks: Identity on m, c, dirty, and lock.

% INVARIANT Inv1: (ALL p | c!p)                     % every processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)      % dirty data is in the cache
% INVARIANT Inv3:                                   % dirty in at most one cache
     (ALL p, p' | dirty_p /\ dirty_p' ==> p = p')
% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)    % data in caches is current
% INVARIANT Inv5:                                   % lock is exclusive
     (ALL p, p' | lock_p /\ lock_p' ==> p = p')
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)       % locked data is only copy
% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)      % dirty data is locked

APROC Read_p -> D = << Live_p => RET c_p >>          % read locally; OK by Inv4
APROC Write_p(d) =                                  % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>

FUNC Only_p -> Bool = RET (ALL p' | Live_p' ==> p' = p) % appears only in p's cache?
FUNC Free() -> Bool = RET (ALL p | ~ lock_p)        % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p [] SKIP OD

APROC MtoC_p =                                      % guard ensures Clean()
     << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >>% Copy cache to memory.
APROC CtoC_p,p' =                                   % guard maintains Inv6
     << Free() /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>
APROC Drop_p = << ~ dirty_p => c_p := nil >>         % Drop clean data from cache

APROC Acquire_p = << Free() /\ Only_p => lock_p:=true >> % exclusive lock is on cache
APROC Release_p = << ~ dirty_p => lock_p := false >>     % Don't release if dirty

END ExclusiveLocks
```

The remaining global tests are the `Only` test in the guard of `Acquire`, and the `Free` tests in the guards of `Acquire`, `MtoC` and `CtoC`. There are many ways to implement them: here are a few:

- Snooping on the bus, as described above. This is only practical when you have a cheap synchronous broadcast, that is, in a bus-based shared memory multiprocessor. The shared bus limits the maximum performance, so typically such systems are not built with more than about 8 processors.
- Directory-based: Keep a "directory" associated with main memory, containing information about where locks and copies are currently located. To check `Free`, a processor need only interact with the main memory. To check `Only`, the same strategy can be used; however, there is a difficulty if cache-to-cache copying is permitted—the main memory must be informed when such copying occurs. For this reason, directory-based code usually eliminates cache-to-cache copying entirely. To acquire a lock, the memory may need to communicate with other caches to get them to relinquish locks and copies. This technique extends to large-scale multiprocessor systems, distributed shared memory, and network file systems.
- Hierarchical: Partition the processors into sets, and maintain a directory for each set. The main directory attached to main memory keeps track of which processor sets have copies or locks; the directory for each set keeps track of which processors in the set have copies or locks. The hierarchy may have more levels, with the processor sets further subdivided.

Code based on `IncoherentMemory`

Next we consider a different kind of code for `CoherentMemory` that runs on top of `IncoherentMemory`. Coherence is guaranteed using an external read/write locking discipline. This is an example of an important general strategy—using weaker memory together with a programming discipline to guarantee strong coherence.

The code uses read/write locks, as defined earlier in the course, one per data item. There is a module $\texttt{ExternalLocks}_p$ for each processor `p`, which receives external `Read` and `Write` requests, obtains the needed locks, and invokes low-level `Read`, `Write`, and `Barrier` operations on the underlying `IncoherentMemory` memory. The composition of these pieces implements `CoherentMemory`. We give the code for $\texttt{ExternalLocks}_p$.

```
MODULE ExternalLocks_p [A, D] EXPORT Read, Write =   % implements CoherentMemory

% ReadAcquire_p acquires a read lock for processor p.
% Similarly for ReadRelease, WriteAcquire, WriteRelease

PROC Read_p =
     ReadAcquire_p; Barrier_p; VAR d| d:=IncoherentMemory.Read_p; ReadRelease_p; RET d

PROC Write_p(d) = WriteAcquire_p; IncoherentMemory.Write_p(d); Barrier_p; WriteRelease_p

END ExternalLocks_p
```

This code does not satisfy all the invariants of `CurrentCaches` and its implementations. In particular, the data in caches is not always current, as stated in `Inv4`. It is only guaranteed to be current if it is read-locked, or if it is write-locked and dirty.

Invariants `Inv1`, `Inv2`, and `Inv3` are still satisfied. Invariants `Inv5` and `Inv6` no longer apply because the lock discipline is completely different; in particular, a locked copy need not be the only copy of an item. Let `wLockPs` be the set of processors that have a write-lock, and `rLockPs` be those with a read-lock. A new `Inv7a` can replace `Inv7`:

```
% INVARIANT Inv7a: (ALL p | dirty_p ==> p IN wLockPs)  % dirty data is write-locked
```

We also have the following new invariants:

```
% INVARIANT Inv8:                                     % lock exclusion
     (wLockPs # {} ==> rLockPs = {}) /\ wLockPs.size <= 1

% INVARIANT Inv9:                                     % data is current
    (ALL p | dirty_p \/ (p IN rLockPs /\ Live_p) ==> Current_p())
```

With these invariants, the identity abstraction to `GlobalImpl` works:

```
% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.
```

We note some differences between `ExternalLocks` and `ExclusiveLocks`, which also uses exclusive locks for writing:

- In `ExclusiveLocks`, `Read` can always proceed if there is a cache copy. In `ExternalLocks`, `Read` has a stronger guard in `ReadAcquire` (requiring a read lock).
- In `ExclusiveLocks`, `MtoC` checks that no other processor has a lock on the item. In `ExternalLocks`, an `MtoC` can occur as long as it doesn't overwrite dirty writes.
- In `ExternalLocks`, the guard for `Acquire` only involves lock conflicts, and does not check `Only`. (In fact, `ExternalLocks` doesn't use `Only` at all.)
- Additional `Barrier` actions are required in `ExternalLocks`.
- In `ExclusiveLocks`, the data in the cache is always current. In `ExternalLocks`, it is only guaranteed to be current for read-lock holders, and for write-lock holders who have already written.

Remarks

`IncoherentMemory` allows a multiprocessor shared memory to respond to `Read` and `Write` actions without any interprocessor communication. Furthermore, these actions only require communication between a processor and the global memory when a processor reads from an address that isn't in its cache. The expensive operation in this spec is `Barrier`, since the sequence $\texttt{Write}_p$; $\texttt{Barrier}_p$; $\texttt{Barrier}_q$; $\texttt{Read}_q$ requires the value written by `p` to be communicated to `q`. In most implementations `Barrier` is even more expensive because it acts on all addresses at once. This means that roughly speaking there must be at least enough communication to record globally every address that `p` wrote before the $\texttt{Barrier}_p$, and to drop from `p`'s cache every address that is globally recorded as dirty.

Although this isn't strictly necessary, all current implementations have additional external actions that make it easier to program mutual exclusion. These usually take the form of some kind of atomic read-modify-write operation, for example an atomic swap or compare-and-swap of a register value and a memory value. A currently popular scheme is two actions: `ReadLinked(a)` and `WriteConditional(a)`, with the property that if any other processor writes to `a` between a $\texttt{ReadLinked}_p\texttt{(a)}$ and the next $\texttt{WriteConditional}_p\texttt{(a)}$, the `WriteConditional` leaves the memory unchanged and returns an indication of failure. The effect is that if the `WriteConditional` succeeds, the entire sequence is an atomic read-modify-write from the viewpoint of another processor, and if it fails the sequence is a `SKIP`. Of course these operations also incur communication costs, at least if the address `a` is shared.

We have shown that a program that touches shared memory only inside a critical section cannot distinguish memory that satisfies `IncoherentMemory` from memory that satisfies the serial specification `CoherentMemory`. This is not the only way to use `IncoherentMemory`, however. It is possible to program other standard idioms, such as producer-consumer buffers, without relying on mutual exclusion. We leave these programs as an exercise for the reader.

We developed the coherent caching code by evolving from the obviously correct `GlobalImpl` to code that has no global operations except to acquire locks. Another way to look at it is that coherent caching is just a variation on easy concurrency. Each `Read` or `Write` touches a shared variable and therefore must be done with a lock held, but there are no bigger atomic operations. The read lock is `Live` and the write lock is `lock`. In order to avoid the overhead of acquiring and releasing a lock on every memory operation, we use the optimization of holding onto a lock until some other cache needs it.

Hardware caches, especially the 'level 1' caches closest to the processor, usually come in two parts, called the cache and the write buffer. The latter holds dirty data temporarily before it's written back to the memory (or the level 2 cache in most modern systems). It is small and optimized for high write bandwidth, and for combining writes to the same cache block that happen close together in time into a single write of the entire line.