Release Consistency

1. Introduction
Chapter 17 of CDK (2nd. edition of Distributed Systems – Concepts and Design by Coulouris, Dollimore and Kindberg) deliberately tries to avoid the subtleties of release consistency, because of lack of space. The paper ‘Implementation and Performance of Munin’ [Carter et al 1991] gives an obscure definition (without discussing the background). These notes are intended to clarify matters.

2. Classifying memory accesses
Chapter 17 of CDK only considers DSM systems implemented on networks of workstations, where we implement synchronisation through message passing for reasons of efficiency. However, synchronisation can also be achieved through shared variables – whether using spinlocks set using the atomic \texttt{testAndSet} instruction, or using an algorithm that requires no special atomic instructions (see any OS book for such algorithms). Programmers normally implement synchronisation using shared variables on a dedicated shared-memory multiprocessor.

Pseudo-code procedures for accessing locks follow. Note that the function \texttt{testAndSet()} sets the lock to 1 and returns 0 if it finds it zero; otherwise it returns 1. It does this atomically.

\begin{verbatim}
acquireLock(var int lock):
    while testAndSet(lock) = 1
        skip;

releaseLock(var int lock):
    lock := 0;
\end{verbatim}

Most papers in the literature on DSM assume that programs use shared memory for synchronisation. We gave the above algorithm in high-level pseudo-code, but the underlying accesses are either LOADs or STOREs or both (\texttt{testAndSet}). We shall sometimes refer to these as ‘reads’ and ‘writes’, instead.

In order to develop a memory model that takes synchronisation into account we begin by classifying memory accesses. The Munin paper distinguishes between ordinary and synchronisation accesses. But this is not the whole story. In fact, the main distinction is between competing accesses and non-competing (ordinary) accesses. Two accesses are competing if:

- they may occur concurrently (there is no definite ordering between them)
- at least one is a STORE.

So two LOADs can never be competing; a LOAD and a STORE to the same location made by two processes that synchronise between the operations (and so order them) are non-competing.

We further divide competing accesses into synchronisation and non-synchronisation accesses:

- \textit{synchronisation} accesses – LOADs/STOREs that contribute to synchronisation

\footnote{All references are from CDK.}
• non-synchronisation accesses – LOADs/STOREs that are concurrent but which do not contribute to synchronisation.

For example, the STORE operation implied by ‘lock := 0’ in `releaseLock()` above is a synchronisation access.

We would expect synchronisation accesses to be competing, because potentially synchronising processes must be able to access synchronisation variables concurrently and they must update them: LOADs alone could not achieve synchronisation. But not all competing accesses are synchronisation accesses – there are classes of parallel algorithms in which processes make competing accesses to shared variables just to update and read one another’s results, and not to synchronise.

Synchronisation accesses are further divided into acquire accesses and release accesses, corresponding to their role in potentially blocking the process making the access, or in unblocking some other process.

3. Performing operations

We distinguish between the point at which a LOAD or STORE instruction is issued – when the processor first commences execution of the instruction – and the point when the instruction is performed or completed. Both LOADs and STOREs can be time-consuming operations. To execute a STORE, the memory system must engage in a protocol before other processors see the written value. Similarly, performing a LOAD may require invalidations of copies of the data in remote memory.

Several forms of asynchronous operation are available to increase the rate at which processors execute programs:

First, STORE instructions are often implemented asynchronously. The value is placed in a buffer before being written to memory, and the effects of the STORE are observed later by other processors. In principle, a second STORE by the same processor to a different memory location could be observed before the first.

Second, processors do not always wait for a previous instruction to complete before issuing the next instruction – this is known as dynamic instruction scheduling. Consider the following sequence of instructions:

1. LOAD R1, A1  // Load word at A1 into R1
2. STORE R2, A2  // Store word in R2 into memory at A2
3. MOVE R1, R3  // Move R1’s data to R3

If this program executes in isolation (it does not share memory with another program) then the processor can issue (2) before a value has been fetched and put into R1 (as a result of issuing (1)). However, the processor must stall at instruction (3) until instruction (1) has completed. And it is not clear that the program would behave correctly in combination with another program accessing the same memory locations.

A third source of asynchronicity is that some processors can pre-fetch values in anticipation of LOADing them.

© Tim Kindberg
In summary, to gain performance there is in general a degree of asynchronicity between the processor and the memory. None of the mechanisms we have mentioned would cause a single program to execute incorrectly. But when two or more programs access shared variables in DSM, we must take care that the memory system produces results that the programmer expects.

### 3.1 Instruction issue and performance in a DSM system

We shall assume that our DSM is at least coherent: this means that every processor agrees on the order of STOREs to the same location (but they do not necessarily agree on the ordering of STOREs to different locations). Given this assumption, we may speak unambiguously of the ordering of STOREs to a given location.

In a distributed shared memory system we can draw a time line for any memory operation \( o \) that processor \( P \) executes, as follows:

\[
\begin{array}{c}
\text{Real time} \\
\hline
P \text{ issues } o & o \text{ performed} \\
\hline
\text{w.r.t. } P' \text{ at time } t
\end{array}
\]

We say that a STORE has performed with respect to a processor \( P' \) at real time \( t \) if from that point either \( P' \) reads the value \( v \) as written by \( P \); or it reads a value written then. In other words, from time \( t \) we rule out \( P' \) reading first a different value and then \( v \) (as written by \( o \) – another operation may STORE the same value!).

Similarly, we say that a LOAD has performed with respect to processor \( P' \) by time \( t \) if thereafter no STORE issued by \( P' \) to the same location could possibly supply the value that \( P \) reads. For example, the processor may have pre-fetched the value it needs to LOAD.

Finally, the operation \( o \) has performed if it has performed with respect to all processors.

### 3. Release consistency

The requirements we wish to meet are:

- to preserve the synchronisation semantics of locks and barriers
- to gain performance, we allow a degree of asynchronicity between processor and memory operations
- but we need to constrain the overlap between competing memory accesses – synchronisation accesses in particular – in order to ensure correct execution.

Release consistent memory is designed to satisfy these requirements. The normal definition of release consistency\(^2\) [Gharachorloo et al 1990] stipulates:

\begin{itemize}
  \item (RC1) before an ordinary LOAD or STORE access is allowed to perform with respect to any other processor, all previous acquire accesses must be performed
  \item (RC2) before a release access is allowed to perform with respect to any other processor, all previous ordinary LOAD and STORE accesses must be performed
  \item (RC3) competing (‘special’) accesses are processor consistent with respect to one another.
\end{itemize}

Under processor consistency as defined in [Gharachorloo et al 1990], the memory is coherent and in addition:

\(^2\)RC1 and RC2 agree with the Munin definition [Carter et al 1991], but RC3 appears to differ. The definition of processor consistency in [Gharachorloo et al 1990] also differs from other definitions!

© Tim Kindberg
(PC1) before a LOAD is allowed to perform with respect to any other processor, all previous LOAD accesses must be performed.

(PC2) before a STORE is allowed to perform with respect to any other processor, all previous accesses (LOADs and STOREs) must be performed.

The simplest way to think of processor consistency is that all processors agree on the ordering of any two (competing) STORE accesses made by the same processor – that is, they agree with its program order.

Let us denote an ordinary memory access by \( o \); an acquire access by \( a \) and a release access by \( r \).

Consider the following program fragment representing operations before, within and after a critical section:

\[
 o_{\text{beforeAcq}} \ldots; \quad a \; \ldots; \quad o_{\text{between}} \ldots; \quad r \; \ldots; \quad o_{\text{afterRel}}.
\]

Applying the release consistency conditions in this example:

(1) \( o_{\text{between}} \) may not perform at any other processor until \( a \) has performed.

(2) \( r \) may not perform at any other processor until \( o_{\text{between}} \) has performed.

Note that this allows \( a \) to be issued before \( o_{\text{beforeAcq}} \) is performed; similarly, \( o_{\text{afterRel}} \) may be issued before \( r \) is performed.

If (1) did not apply then processor \( P \) may enter a critical section and either:

- STORE data which another processor may LOAD before \( P \) has acquired mutual exclusion; or,
- render myself unable to LOAD a final value written by the processor that is leaving the critical section.

If (2) did not apply then, when processor \( P \) exits the critical section:

- another processor may enter it but LOAD stale data; or,
- \( P \) may LOAD a value after another processor entered the critical section, which \( P \) should have read while still in it.

The programmer (or a compiler) is responsible for labelling LOAD and STORE instructions as release, acquire or non-synchronisation – other instructions are assumed to be ordinary. To label the program is to direct the DSM system to enforce the release consistency conditions.

Gharachorloo et al [1990] describe the concept of a properly-labelled program. They prove that such a program cannot distinguish between a release-consistent (RC) DSM and a (less efficient) processor-consistent (PC) DSM. Moreover, if condition (RC3) is replaced to make competing accesses sequentially consistent (SC) instead of PC, then a properly labelled program cannot distinguish between RC DSM and SC DSM. Many programs execute equivalently under PC and SC DSM, and the weaker form of RC can be used with them.

4. An example

The following code implements a critical section using the logic of the acquireLock() and releaseLock() procedures that we gave above.

```c
while testAndSet(lock) = 1       # LOAD: acquire; STORE: non-sync
    skip;
... Critical Section ...
lock := 0;                      # STORE: release
```

In the example, the testAndSet instruction is treated as a LOAD followed by a STORE. The acquire and release labels ensure consistent memory accesses with respect to the critical section, as we
described above. The *non-sync* label specifies that the STORE is a non-synchronisation access. The label is necessary to order the *testAndSet* with respect to other synchronisation operations. For more information about release consistency, Gharachorloo et al [1990] give a rigorous account of their design and implementation for the DASH multiprocessor.

*Tim Kindberg, QMW*