6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 11: October 21, 1992

Lecturer: Bill Weihl *Scribe: Henry Minsky, Ken Mackenzie*[1]

1 Handouts

This lecture is the last of three on concurrency. The example concurrent system is described in the accompanying Handout 28: Cached Concurrent Disk.

2 Placement of Locks

This lecture covers some moderately complex examples of concurrency, and examines some of the issues raised. Some examples are given of the programming techniques used to manage these issues.

The implementation of a cached concurrent disk described in Handout 28 demonstrates a basic technique, which consists of breaking down the problem as follows:

- Identify separable shared blocks of data.
- Create one lock per data block — Protect each block of data with its own lock, implemented with Mutex. A process must acquire the lock before accessing any of the data.
- Maintain an invariant outside of the lock — Protected data must be left in a consistent state when a process leaves a critical section.

Figure 1 shows two threads, **T1** and **T2**, both executing code which wants to access a shared data block, protected by lock l. Each thread must first acquire the lock, using *l.acq* before altering the data. The code between the acquisition of the lock and its release is called a *critical section*. The mutual exclusion property of the mutex ensures that only one thread at a time has access to its critical section.

It is important to realize that in general, data which has been cached inside a critical section (such as a copy of a piece of shared data, assigned to a local variable) will not be valid outside of the critical section, unless some invariant is maintained on the data.

2.1 Invariants For Shared Data

It is possible to show by induction that an invariant $I(d)$ on a piece of shared data is preserved, if you maintain a couple of conditions:

[1] Prepared by Henry Minsky in Fall 1991; refashioned by Ken Mackenzie.

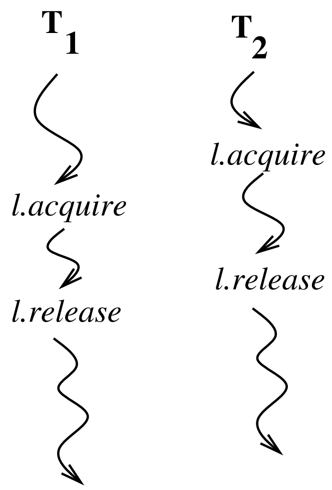

Figure 1: Two threads attempting to access the same data

- The invariant $I(d)$ on the protected data is true at initialization.
- The invariant is maintained whenever its lock is released.

As long as you leave the data obeying the invariant when you exit the critical section, and don't touch it outside of the critical section, it will still obey the invariant the next time a process acquires the lock.

2.2 Fine-Grain Concurrency

Having all the data in a program protected by one lock can create a bottleneck in the processing. In many cases there are logically independent regions of data, in the sense that operations can be performed concurrently on the regions by different processes, without interference between them.

Thus it is often desirable to break the data up into several independent shared blocks each protected by its own lock. This allows multiple threads to proceed in parallel as long as they require data in different blocks.

Another technique to increase concurrency is to use read-write locks instead of mutual exclusion locks. Read-write locks permit arbitrary numbers of readers (since readers don't interfere with one another) but only a single writer for a block.

Muliple locks increase the opportunities for concurrent processing but also introduce more possibilities of deadlock.

3 Deadlock

Figure 2 shows a potential deadlock situation with two threads, each trying to hold two locks. A situation has arisen where neither process can advance, because they are both blocking on a resource

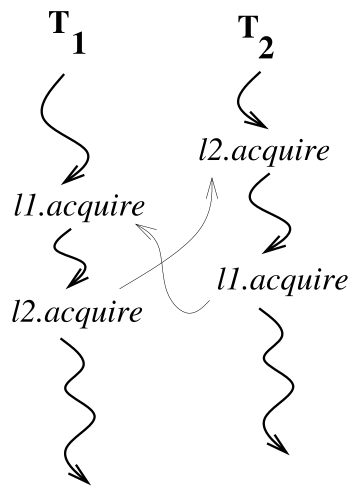

Figure 2: Two threads, each attempting to acquire locks *l1* and *l2*

which the other owns.

There are several ways to deal with deadlock in a system. A general solution to this problem is to put a partial order on all locks in the system, and require that for each process, "to acquire lock *l*, it must be the case that all locks already held are less-than *l*". This solution is general but may not be feasible in cases where there is no natural order for the resources.

A second solution, suggested in class, is to dynamically detect deadlock situations and then correct them. Detection is possible but potentially complex as it involves reasoning about all the resources in the system. Correcting the deadlock generally requires that operations be capable of aborting in mid-operation. This is an extra requirement that complicates the programming problem.

The simplest form of deadlock detection is a timeout. A timeout-and-retry mechanism can be an effective way to recover from deadlock. A scheme similar to the exponential backoff schemes used in Ethernet CSMA with random timers will allow deadlocks (actually, livelocks) to be broken with high probablility.

4 Concurrent Disk Example

The `ConcurrentDisk` module of Handout 28 performs several types of operations, ordered here by expense in terms of latency and possibly CPU usage:

- *cache lookup* = cheap
- *block-copy* within main memory = medium
- *disk access* = expensive

What benefit would be gained here by using concurrency? Consider the cases of cache-lookup and block-copy, and a single CPU system. If we only care about CPU utlization, then there is no overall benefit to be gained in making these operations concurrent. If a certain number of CPU operations must be done for each task, then time-slicing the tasks does not get the whole job done any faster.

If we have two or more CPUs, or a CPU and disk drives, then it is possible to make real gains in performance, if several tasks can actually be executed simultaneously. In general, if we have several engines of computation which are capable of making progress simultaneously, then concurrency will be useful.

Actually, even in the "single CPU with no disk" case, if what we care about is the real-time response of the system, then it is desirable to run operations concurrently, using a scheduler. This means that a short operation does not have to wait because another very long operation is running. The responsiveness of a system is an important factor for human users[2].

A task can be made more concurrent by separating out subtasks, each of which can have its own lock. In both the single and multiple CPU examples, we find that when there is a need to distribute load more fairly, either in time or space, then it is time to look at finer grain locking strategies. This can result in

1. better response time, and
2. keeping mutliple resources busy (if you have them)

4.1 Cached Concurrent Disk

We want the implementation of the `ConcurrentDisk` module to support multiple threads reading and writing simultaneously. This could be accomplished by serializing all of the operations, which would basically be the `BufferedDisk` implementation, by making all operations atomic. But this makes threads wishing to access independent cached disk blocks wait needlessly for other's slow disk operations or block-copies to complete.

The `ConcurrentDisk` implementation uses a combination of multiple locks for data blocks in the cache, and a single monolithic lock for the relatively brief bookeeping operations on the cache data-structure itself.

Figure 3 shows the relation between locks and cache blocks in the implementation. The `usrs` field counts outstanding references to the block, and is protected by `mc`, the master mutex for the cache. The `m` field is a mutex which protects the the data.

The invariant on the cache blocks is that when the mutex `m` is *not* held by anyone, then the data in `db`, the cached disk block, represents the true data for that block. As long as the invariant is maintained when a thread releases the lock (`b.m.rel`), then it is maintained when the lock is next acquired.

The `ReadBlocks` procedure is constructed to maintain these invariants, and to manage the allocation of storage in the cache. The purpose of the `i` field in the cache block is to keep track of how many threads are accessing the data in that block. A cache block `b` can only be flushed if that block has no readers (i.e., `usrs(i)` $= 0$).

[2] Birrell (*An Introduction to Programming with Threads*) points out that this is particularly true in interactive applications, such as window systems.

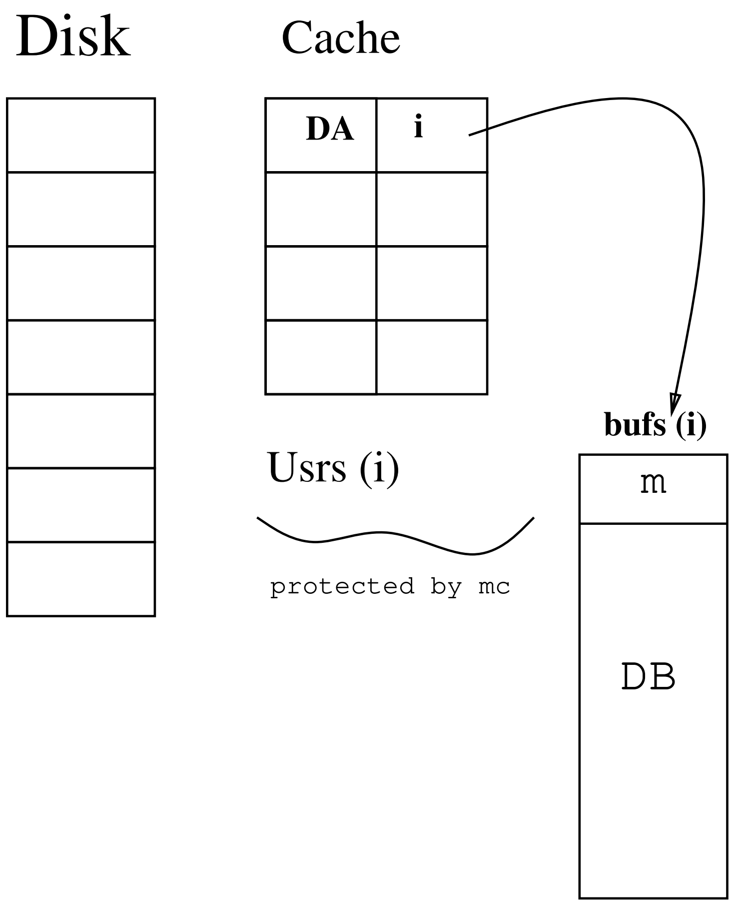

Figure 3: Locking for the ConcurrentDisk module

A call to `ReadBlocks(e)` works by looping to find if the desired data-blocks in the extent are present in the cache. For each block which is present, the number-of-readers counter, `i`, is incremented. The global cache lock must be held during this operation, or else incorrect values of `i` could result. For efficiency, if some of the desired blocks are not present, the algorithm tries to find the longest contiguous string of blocks unavailable in the cache to fill at a time, so as to do as few physical disk-accesses as possible. The global lock can be released now, since the available cache blocks have been effectively "wired down" by making their `usrs` field non-zero. All of the data from cache hits are then copied to the local variable `data`, and the cache replacement algorithm is run. This copy operation requires grabbing the lock for each data block sequentially, but only one of these need be held at a time; a big gain in concurrency over using a single lock for the entire cache.

The cache replacement algorithm now tries to allocate enough free buffers to hold the data it is about to read from disk. When these buffers have been allocated, (i.e., locked), the `Disk.ReadBlocks` operation reads the data from the disk, and copies it into the allocated buffers.

5 Filesystem Example

The filesystem needs to maintain a data structure which maps pathnames to files. In the face of concurrency, it is possible to ensure correct behavior, by serializing all filesystem operations. This is a poor solution for several reasons. If the entire filesystem must be grabbed exclusively by a process wishing to do any filesystem operation, then there will be a lot of processes busy waiting, even if they want to perform totally independent operations. Also, it is hard to make some operations, such as large writes, atomic. This is a case where finer grained locking is a good idea.

Consider the model of a filesystem directory structure as a DAG. Lookup of pathnames is done by traversal of the directory tree from the root. Imagine two processes running concurrently, one performing a `lookup(/b/z)` and another performing `rename(/b)`. With no locking at all, or locking only each entry as it is used, the lookup is liable to return a wrong result. Clearly some sort of locking is needed if the user insists on filesystem operations appearing atomic.

One solution is to weaken the atomicity requirements for filesystem operations, i.e. alter the filesystem semantics. This is in fact done in Unix where the outcome of concurrent operations are not guaranteed to appear atomic with respect to one another. Even in such systems, some locking is desirable for system reasons. A partial solution is to use *coupled locks* on the directory entries. With this technique a directory node is locked until the lock on the subsequent node is acquired. This at least guarantees that the child node is not deleted during the lookup.

Coupled locks are subject to deadlock if there are cycles in the order that locks are acquired (e.g. Unix's "`..`" reverse links. BSD Unix apparently solves this by abandoning the lock coupling when traversing reverse links.

A solution that preserves atomic semantics is to lock subtrees. But this isn't any better than a global lock since pathnames are always relative to the root. We end up locking the directory tree from the root down, which is amounts to the same thing as a global lock on the filesystem. Read/write locks allow concurrent read operations and ease the bottleneck at the root, but a second problem arises with directories that include reverse- or cross-links. Figure 4 illustrates a deadlock

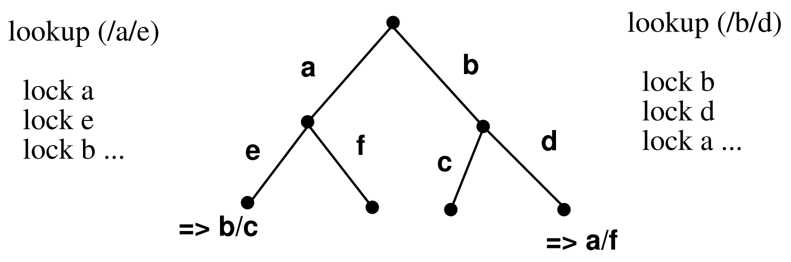

Figure 4: Deadlock in a directory tree with symbolic links

situation caused by the use of symbolic links. The basic problem is that the locks are acquired in the order the directory entries are resolved rather than in a fixed order.

5.1 2-Phase Locking

A technique commonly used in the database world is called *2-Phase Locking* or *2PL*. In this scheme, a thread acquires all the needed locks before releasing any of them. The locks may be aquired in a fixed order to preclude deadlock. When this scheme is adhered to, the operation can be considered to take place at the instant when all of the locks are owned by the thread.

This technique acquires only the needed locks and makes the operation appear atomic but may suffer a performance penalty due to the fact that locks may be held for artificially long periods. 2-phase locking will be discussed more thoroughly in a subsequent lecture.

A major problem in applying 2-phase locking to the directory lookup problem is that it is difficult to apply the simple deadlock-avoidance technique of acquiring the locks in order because the set of locks required is determined dynamically as the lookup proceeds.

5.2 Optimistic Locking

Optimistic locking is a technique that addresses the problem of knowing the set of locks to be acquired. The proceedure is the following:

1. Resolve the complete set of directory nodes that will be accessed (briefly locking each directory to read it). Also, note the version numbers of the nodes.

2. Acquire locks on these nodes according to some predetermined partial order.

3. If any version numbers have changed, release locks and start over.

4. Otherwise perform the operation and release the locks.

This scheme is optimistic in that it 'hopes' that no changes will take place between the first scan and the time it locks the entire set of nodes. This is a fair assumption if the rate of changes to the filesystem structure is sufficiently low. If, however, a node has changed, then the entire cycle is repeated, because the set of nodes to be locked may have changed as a result.

Note that there is no possibility of livelock here (wherein processes keep retrying without any making any progress). A process retries only if a node was found to be modified, and modifications are made only after the second scan is successfully completed. Therefore, at least one process succeeds in its operation.

There are two advantages of optimistic locking:

1. Since the set of locks to be acquired is known after the first scan, they can be locked in some predetermined partial order. This avoids deadlocks.

2. The first scan takes most of the cache misses (for directories not in the cache). The second scan will usually find all directories already in the cache. This avoids having to access the disk while holding locks to a set of directories. Note that a disk access occurring in the first scan is not as bad because it delays the release of only one lock.