15. Concurrent Disks and Directories

In this handout we give examples of more elaborate concurrent programs:

An implementation of Disk.read using the same kind of caching used in BufferedDisk from handout 7 on file systems, but now with concurrent clients.

An implementation of a directory tree or graph, as discussed in handout 12 on naming, but again with concurrent clients.

**Concurrent buffered disk**

The ConcurrentDisk module below is similar to BufferedDisk in handout 7 on file systems; both implement the Disk spec. For simplicity, we omit the complications of crashes. As in handout 7, the buffered disk is based on an underlying implementation of Disk called UDisk, and calls on UDisk routines are in bold so you can pick them out easily.

We add a level of indirection so that we can have names (called b’s) for the buffers; a B is just an integer, and we keep the buffers in a sequence called bv. B has methods that let us write bv.db for bv(b).db and similarly for other fields.

The cache is protected by a mutex mc. Each cache buffer is protected by a mutex b.m; when this is held, we say that the buffer is locked. Each buffer also has a count users of the number of b’s to the buffer that are outstanding. This count is also protected by mc. It plays roughly the role of a readers lock on the cache reference to the buffer during a disk transfer; if it’s non-zero, it is not OK to reassign the buffer to a different disk page. GetBufs increments users, and InstallData decrements it. No one waits explicitly for this lock. Instead, read just waits on the condition moreSpace for more space to become available.

Thus there are three levels of locking, allowing successively more concurrency and held for longer times:

- mc is global, but is held only during pointer manipulations;
- b.m is per buffer, but exclusive, and is held during data transfers;
- b.users is per buffer and shared; it keeps the assignment of a buffer to a DA from changing.

There are three design criteria for the implementation:

1. Don’t hold mc during an expensive operation (a disk access or a block copy).
2. Don’t deadlock.
3. Handle additional threads that want to read a block being read from the disk.

You can check by inspection that the first is satisfied. As you know, the simple way to ensure the second is to define a partial order on the locks, and check that you only acquire a lock when it is greater than one you already have. In this case the order is mc < every b.m. The users count takes care of the third.

The loop in read calls GetBufs to get space for blocks that have to be read from the disk (this work was done by MakeCacheSpace in handout 7). GetBufs may not find enough free buffers, in which case it returns an empty set to read, which waits on moreSpace. This condition is signaled by the demon thread FlushBuf. A real system would have signaling in the other direction too, from GetBufs to FlushBuf, to trigger flushing when the number of clean buffers drops below some threshold.

The boxes in ConcurrentDisk highlight places where it differs from BufferedDisk. These are only highlights, however, since the code differs in many details.

**CLASS ConcurrentDisk**

| EXPORT read, write, size, check, sync = |

| TYPE | % Data, DA, DB, Blocks, Dsk, E as in Disk |
| I = | Int |
| J = | Int |
| Buf = | [db, B, users: I, clean: Bool] % m protects db, mc the rest |
| M = | Mutex |
| B = | Int WITH {m := (\b | bv(b).m), db := (\b | bv(b).db), users:= (\b | bv(b).users), clean:= (\b | bv(b).clean)} |
| BS = | SET B |

| CONST | DBSize := Disk.DBSize |
| DBsize := | 100 |
| moreSpace := | Condition.C |

| VAR | % uses UDisk’s disk, so there’s no state for that |
| udisk := | Disk |
| cache := | (DA -> B) {} |
| mc := | M |
| mc := | M |
| moreSpace := | Condition.C |
| bv := | (B -> Buf) |
| flushing := | (DA + Null) := nil |

| % ABSTRACTION FUNCTION Disk.disk(0) = (\da | ( cache(da) \[ (cache(da).m not held \& da = flushing) \| cache(da).db = U Disk.disk(0)(da) ] ) |

The following invariants capture the reasons why this code works. They are not strong enough for a formal proof of correctness.

% INVARIANT 1: ( ALL da :IN cache.dom, b | b = cache(da) \& b.m not held \& b.clean \implies b.db = UDisk.disk(0)(da) ) A buffer in the cache, not locked, and clean agrees with the disk (if it’s locked, the code in FlushBuf and the caller of GetBufs is responsible for keeping track of whether it agrees with the disk).
% INVARIANT 2: (ALL b | (da | cache!da \ cache(da) = b).size <= 1)
A buffer is in the cache at most once.
% INVARIANT 3: mc not held => (ALL b :IN bv.dom | b.clean \ b.users = 0
==> b.m not held)
If mc is not held, a clean buffer with users = 0 isn't locked.

PROC new(size: Int) -> Disk =
self := StdNew(); udisk :=
udisk(new(size);
mc.acq; DO VAR b | ~ bv!b => VAR m := m.new() |
vb(b) := Buf{m := m, db := {}, users := 0, clean := true}
OD; mc.rel
RET self

PROC read(e) -> Data RAISES {notThere} =
udisk.check(e);
VAR data := Data{}, da := e.da, upto := da + e.size, i |
mc.acq;
% Note that we release mc before a slow operation (bold below)
% and reacquire it afterward.
DO da < upto => VAR b, bs |
% read all the blocks
IF cache!da =>
b := cache(da);
% yes, in buffer b; copy it
b.m.acq; bv(b).users + := 1; mc.rel;
% Now acquire m before copying the data.
% May have to wait for m if the block is being read.
b.m.acq; data + := b.db; b.m.rel;
mc.acq; bv(b).users - := 1; mc.rel;
da := da + 1
I[i] := RunNotInCache(da, upTo);
% i not in the cache
bs := GetBufs(da, i); i := bs.size; % GetBufs is fast
IF i > 0 =>
mc.rel; data + := InstallData(da, i); mc.acq
da := da + i
[*] moreSpace.wait(mc)
FI
FI
OD; RET self

PROC GetBufs(da, i) -> BS =
% GetBufs tries to return i buffers, but it returns at least minDiskRead buffers (unless i is less than this)
% so that read won't do lots of tiny disk transfers. It's tempting to make GetBufs always succeed, but this
% means that it must do a Wait if there's not enough space. While mc is released in the Wait, the state of the
% cache can change so that we no longer want to read i pages. So the choice of i must be made again after the
% Wait, and it's most natural to do the Wait in read.
% must lock. Return some buffers assigned to da, da+1, .... locked, and
% with users = 1 or () if there's not enough space. No slow operations.
VAR bs := (b | b.users = 0 \ b.clean) |
% the usable buffers
IF bs.size >= (i, minDiskRead).min =>
i := (i, bs.size).min;
DO VAR b | b IN bs \ b.users = 0 =>
% Remove the buffer from the cache if it's there.
IF VAR da' | cache(da') = b => cache := cache{da' -> } [*] SKIP FI;
mc.acq; bv(b).users - := 1; cache(da) := b; da := da + 1
OD; RET {b :IN bs | b.users > 0}
[*] RET () FI

In handout 7, InstallData is done inline in read.

PROC InstallData(da, i) = VAR data, j := 0 |
% Pre: cache(da) .. cache(da+i-1) locked by SELF with users > 0.
data := udisk.read(E{da, i});
DO j < i => VAR b := cache(da + j) |
% acquire m before checking the other fields.
IF VAR da' | cache(da') = b => cache := cache{da' -> } [*] SKIP FI;
mc.acq; bv(b).clean := true; mc.rel;
mc.acq; bv(b).users := 1; mc.rel;
bv(b).db := udisk.DToB(data).sub(j); b.m.acq; b.m.rel;
bv(b).users - := 1; mc.rel;
mc.rel; moreSpace.signal
FI
OD; RET data

PROC write is omitted. It sets clean to false for each block it writes. The background thread
FlushBuf does the writes to disk. Here is a simplified version that does not preserve write order.
Note that, like read, it releases mc during a slow operation.

THREAD FlushBuf() = DO
% flush a dirty buffer
mc.acq;
IF VAR da, b | b = cache(da) \ b.users = 0 \ ~ b.clean =
% flushing := true;
flushing := true;
j.m.acq; bv(b).clean := true; mc.rel;
udisk.write(da, b.db);
flushing := false;
mc.acq; bv(b).users := 1; mc.rel;
m.rel; moreSpace.signal
FI
OD

% Other procedures omitted
END ConcurrentDisk
Concurrent directory operations

In handout 12 on naming we gave an ObjNames spec for looking up path names in a tree of graph of directories. Here are the types and state from ObjNames:

```
TYPE D = Int
WITH {get:=GetFromS, set:=SetInS} % Just an internal name

get returns nil if undefined

Link = [d: (D + Null), pn]
% d=nil means the containing D

Z = (V + D + Link + Null)
% nil means undefined

DD = N -> Z

CONST
root : D := 0
s := (D -> DD){}{root -> DD{}} % initially empty root

APROC GetFromS(d, n) -> Z =
% d.get(n)
<< RET s(d)(n) [\*] RET nil >>

APROC SetInS (d, n, z)   =
% d.set(n, z)
% If z = nil, SetInS leaves n undefined in s(d).
<< IF z # nil => s(d)(n) := z [\*] s(d) := s(d){n -> } FI >>
```

We wrote the spec to allow the bindings of names to change during lookups, but it never reuses a D value or an entry in s. If it did, a lookup of /a/b might obtain the D for /a, say dA, and then /a might be deleted and dA reused for some entirely different directory. When the lookup continues it will look for b in that directory. This is definitely not what we have in mind.

An implementation, however, will represent a DD by some data structure on disk (and cached in RAM), and if the directory is deleted it will reuse the space. This code needs to prevent the anomalous behavior we just described. The simplest way to do so is similar to the users device in ConcurrentDisk above: a shared lock that prevents the directory data structure from being deleted or reused.

The situation is trickier here, however. It’s necessary to make sufficiently atomic the steps of first looking up a to obtain dA, and then incrementing s(dA).users. To do this, we make users a true readers lock, which prevents changes to its directory. In particular, it prevents an entry from being deleted or renamed, and thus prevents a subdirectory from being deleted. Then it’s sufficient to hold the lock on dA, look up a to obtain dA, and acquire the lock on dA before releasing the lock on dA. This is called ‘lock coupling’.

As we saw in handout 12, the amount of concurrency allowed there makes it possible for lookups done during renames to produce strange results. For example, Read(/a/x) can return 3 even though there was never any instant at which the path name /a/x had the value 3, or indeed was defined at all. We copy the scenario from handout 12. Suppose:

- initially /a is the directory d1 and /b is undefined;
- initially x is undefined in d1;
- concurrently with Read(/a/x) we do Rename(/a, /b); Write(/b/x, 3).

The following sequence of actions yields Read(/a/x) = 3:

- In the Read, Get(root, a) = d1
- Rename(/a, /b) makes /a undefined and d1 the value of /b
- Write(/b/x, 3) makes 3 the value of x in d1

In the Read, RET d1.get(x) returns 3.

![Diagram]

Obviously, whether this possibility is important or not depends on how clients are using the name space.

To avoid this kind of anomaly, it’s necessary to hold a read lock on every directory on the path. When the directory graph is cyclic, code that acquires each lock in turn can deadlock. To avoid this deadlock, it’s necessary to write more complicated code. Here is the idea.

Define some arbitrary ordering on the directory locks (say based on the numeric value of D). When doing a lookup, if you need to acquire a lock that is less than the biggest one you hold, release the bigger locks, acquire the new one, and then repeat the lookup from the point of the first released lock to reacquire the released locks and check that nothing has changed. This may happen repeatedly as you look up the path name.

This can be made more efficient (and more complicated, alas) with a ‘tentative’ Acquire that returns a failure indication rather than waiting if it can’t acquire the lock. Then it’s only necessary to backtrack when another thread is actually holding a conflicting write lock.