Verification of buffer cache properties in xv6 operating system

Sumith Kulal[1]

IIT Bombay
sumith@cse.iitb.ac.in

Abstract. I present verification of the buffer cache module implemented in the xv6[1] operating system. In particular, we consider the doubly-linked list setup of the buffer cache, appropriate read and write properties along with the flags set and most importantly, the LRU eviction strategy. We use predicate analysis and depend on the CPAchecker[2] tool.

Keywords: abstract interpretation, verification, operating systems

1 Introduction

Xv6 is a teaching operating system developed in the summer of 2006 for MIT's operating systems course. It is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix Version (v6). It loosely follws the structure and style of v6, but is implemented in ANSI C for an x86 based multiprocessor.

The xv6 file system is organised in 6 layers and has been illustrated in the final figure of appendix. The top layers consisting file descriptors and system calls, intermediate layers consisting inode structure and the lower layers consisting logging and buffer cache. Buffer cache layer was chosen for the project as it has very interesing properties (MRU maintenance), good data-structures (doubly-linked list) and less dependencies on the layers below it.

1.1 Buffer Cache Layer

The buffer cache is a doubly-linked list of buffers. The function `binit`, called by `main`, initializes the list with the `NBUF` buffers in the static array `buf`. All other access to the buffer cache refer to the linked list via `bcache.head`, not the `buf` array.

A buffer has three state bits associated with it. `B_VALID` indicates that the buffer contains a copy of the block. `B_DIRTY` indicates that the buffer content has been modified and needs to be written to the disk. `B_BUSY` indicates that some kernel thread has a reference to this buffer and has not yet released it.

1.2 Functions

The `binit` function does the initial setup of the buffer cache array. After it returns, `bcache.head.next` points to the last location of the array `buf[NBUF-1]` and the first element of the array `buf[0]` points to `bcache.head`. For any intermediate element `buf[i]`, `buf[i].next` points to `buf[i-1]` and `buf[i].prev` points to `buf[i+1]`. The above has been illustrated in the figure in appendix.

The `bget` function returns a pointer to the buffer with the specified `dev` and `blockno`. If such a block does not already exists in the buffer, it evicts the LRU and makes it available. If no such block can be evicted, then the kernel panics and exits. Also note that the buffer returned is also locked for future use.

The `bread` function reads from the disk. If the `B_VALID` is not set, a call to `iderw` is made. bread always returns with a valid buffer.

The `bwrite` function writes on to the disk. It calls the `iderw` function after setting the dirty flag. Once `iderw` returns, it returns a valid buffer.

The `brelse` function releases the buffer and updates the LRU list. If the buffer released has no process waiting for it, it is brought to the top of the MRU list. This has been illustrated in the figure in appendix.

2 Verification

The following section highlights the properties that were verified, specifications written for the same and the changes to the source code that had to be made.

2.1 Properties

The `binit` function does the correct pointer assignments and the doubly-linked list is correctly setup. This is a strong property to be verified.

The `bget` function does proper locking and unlocking of `bcache.lock` and also, on successfully returning buffer `b`, `b->dev` == `dev`, `b->blockno` == `blockno` and `b->refcnt` ≥ 1.

The `bread` function always sets the `B_VALID` flag on return.

The `bwrite` function sets the `B_DIRTY` flag before the call to `iderw`. After `bwrite` returns, the buffer has `B_VALID` flag set as it is now in sync with the disc.

The `brelse` function does proper locking and unlocking of locks. Also that the argument buffer `b` is now the MRU which implies `head.next` points to it. There is an additional property that the previous MRU is now pointed to by `b.next` that has also been verified i.e. previously the most recently used is now the second most recently used.

2.2 Tool chosen

Predicate analysis had to be used for verifying the properties. Keeping this in mind, I started off using BLAST[3] but was unable to proceed due to resolution of OCaml dependencies. I moved to CPAchecker[2] later and had successfully verified all properties are stated above. Some of the properties that I had in mind could not be proved and is explained in detail below.

3 Specifications

Two non-trivial specifications from the verification process has been highlighted below. The rest of the specifications can be found in source file supplied.

3.1 Doubly-Linked List

```
for(b = bcache.buf; b < &bcache.buf[NBUF]; b++){
  if(b == bcache.buf) { // 1
    if(b->next != &bcache.head) {
      goto ERROR;
    }
    if(b->prev != b+1) {
      goto ERROR;
    }
  } else if(b == &bcache.buf[NBUF-1]) {
    if(b->next != b-1) { // 2
      goto ERROR;
    }
    if(b->prev != &bcache.head) {
      goto ERROR;
    }
  } else {
    if(b->next != b-1) { // 3
      goto ERROR;
    }
    if(b->prev != b+1) {
      goto ERROR;
    }
  }
}
```

The above specification checks that the pointers point to the right locations. For the first location of the array, the `next` pointer points to the `head` and the `prev` pointer points to the next location. For the last location of the array, the `prev` pointer points to the `head` and the `next` pointer points to the previous location. For any location in between, the `prev` pointer points to the next location and the `next` pointer points to the previous location.

3.2 MRU updates

```
if (b->refcnt == 0) {
  if(bcache.head.next != b) {
    goto ERROR;
  }
  if(b->prev != &bcache.head) {
    goto ERROR;
  }
  if(b->next != mru) {
    goto ERROR;
  }
  if(mru->prev != b) {
    goto ERROR;
  }
}
```

The above specification checks that the updates to the MRU list is correct. Let `mru` point to the old MRU. After the updates, `bcache.head.next` which points to the MRU now points to `b` and the `prev` of `b` points to the `head`. We also see that the `next` of `b` (the next most recently used) points to `b` and prev of `mru` points to `b`.

4 Changes to source code

All the headers were preprocessed and the source was reduced to a single `.c` file. The source code had to be modified slightly to model a few elements. Locks were modelled as 0-1 integer variables. initlock was modelled as assigning 0 to the lock variable. Acquiring and releasing locks were modelled as follows.

```
// acquire(&bcache.lock);
if(bcache.lock == 1) {
  goto ERROR;
}
bcache.lock = 1;

// release(&bcache.lock);
if(bcache.lock == 0) {
  goto ERROR;
}
bcache.lock = 0;
```

The for-loop iteration of over the buffer cache array in `binit` previously had `bcache.buf+NBUF;` as the limit but it resulted in a lot of spurious counterexamples. Changing it to `&bcache.buf[NBUF];` got the specifications to go through but I'm unaware as to what resulted in the same.

`bread` and `bwrite` functions have a call to `iderw` which accesses the disk drivers and returns a valid buffer. A dummy `iderw` is written which sets the `B_VALID` flag to `true` and nothing else.

The flags in the buffer was previously an integer and the `B_VALID` and `B_DIRTY` flags were accessed using `flags & 0x2` and `flags & 0x4` respectively. CPAchecker caused errors using the bitwise operators in `if` conditions. Hence this flags variable in the buffer was replaced by individual valid and dirty variables which corresponded to the flags.

The last change made was removing an if condition before the call to `iderw`. This was done because CPAchecker throws spurious counterexamples. Consider the following example

```
// Dummy disk read function for specification
void iderw(struct buf *b) {
  b->valid = 1;
  return;
}

// a snippet in 'bread'
  if(b->valid != 1) {
    iderw(b);
  }
  if(b->valid != 1) {
    goto ERROR;
  }
```

Here `b->valid` is always set to 1 in the check for `ERROR` but CPAchecker pointed counterexamples at this location. Hence, this if condition was eliminated. We note that the correctness of the code here is not affected. Having a call `iderw` even for a valid buffer is an inefficient way of handling things but not an incorrect one. However, not having the original code verified, I feel there is room for improvement.

5 Conclusion and improvements possible

Other than the one stated above, another very interesting property that I could not verify was the bget function returns a locked buffer. CPAchecker stated the verification result as UNKNOWN, it probably had a counterexample that it could not eliminate.

CPAchecker as a tool has been really versatile and easy to use. The report generated automatically has been very useful in debugging the traces. It also has powerful predicate inference and refinement procedures. However it lacks good documentation and better documentation would have helped. At times, it threw traces that seemed impossible. I speculate we have been using the tool incorrectly and haven't gone into great depth in that direction.

Buffer cache and LRU eviction properties are very crucial in any operating system and verification of the same has been an enriching experience.

References

1. xv6, a simple Unix-like teaching operating system. https://github.com/mit-pdos/xv6-public
2. Dirk Beyer and M. Erkan Keremoglu. 2011. CPACHECKER: a tool for configurable software verification. In Proceedings of the 23rd international conference on Computer aided verification (CAV'11), Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer-Verlag, Berlin, Heidelberg, 184-190.
3. Henzinger, TA., Jhala, R., and Majumdar, R., The BLAST Software Verification System, Model Checking Software: 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005.