Homework 12

Group Number 56 | Anmol Singhal 2017332, Daksh Shah 2017336, Tejas Oberoi 2017367

begin_op

```
/* This function checks whether the log is committing
its blocks to the disk or not. It also checks if
there is enough space in the log to store the
writes from the current call and all the
currently executing system calls. In case the
log is committing or enough space does not exist,
begin_op() waits until the commit gets
completed or there is enough space to write to
the log. It is called at the start of each fs system call.
MAXOPBLOCKS = 10
LOGSIZE = MAXOPBLOCKS*3 = 30
*/
 void
 begin_op(void)
 {
      /*acquires a lock on the log */
      acquire(&log.lock);

      while(1){

            /* If the log is committing its blocks to disk,
            sleep until completed */

            if(log.committing){
                  sleep(&log, &log.lock);
            }

            /*
            log.outstanding counts that number of system calls
            currently running, other than the current call.
            Each system call can write up to MAXOPBLOCKS
            distinct blocks.
```

```
            If the number of blocks in the corresponding log (log.lh.n) +
no. of system calls (log.outstanding+1) * size taken by each call
(MAXOPBLOCKS) more than the log size, sleep until the size of the data to
be written can be accommodated by the log */

            else if(log.lh.n + (log.outstanding+1)*MAXOPBLOCKS > LOGSIZE){
                  // this op might exhaust log space; wait for commit.
                  sleep(&log, &log.lock);
            }

            /* Increment log.outstanding, to add the
            current call to the number of currently executing
            fs system calls.
            Release the lock on the log and return */

            else {
                  log.outstanding += 1;
                  release(&log.lock);
                  break;
            }
      }
 }
```

end_op

```
/* It decrements the total number of currently
 executing system calls by 1. It commits if the current call
  was the last outstanding operation.
 It is called at the end of each FS system call. */
```

```
 void
 end_op(void)
 {
      //the do_commit is a flag which indicates whether the log needs to
commit or not. It is by default not set.
      int do_commit = 0;
      //acquires a lock on the log
      acquire(&log.lock);
      //decrementing the count of currently executing system calls
      log.outstanding -= 1;
      //check if the log is not committing currently
      if(log.committing)
            panic("log.committing");
      //commit flag set to one if no system calls are executing currently
      //the commit flag in the log is also set
      if(log.outstanding == 0){
            do_commit = 1;
            log.committing = 1;
      }

      else {
            // wake up the log if currently in sleep state
            //begin_op() may be waiting for log space.
            wakeup(&log);
      }
      //release the lock on the log
      release(&log.lock);

      //if the commit flag is set
      if(do_commit){
            // call commit w/o holding locks, since not allowed
            // to sleep with locks.
            //commit function is called
            commit();
            //acquire the lock on the log
            acquire(&log.lock);
            //commit bit of the log set to 0, as commit() returns
            log.committing = 0;
            //wake up the log if currently in sleep state
            wakeup(&log);
            //release the lock on the log
```

```
            release(&log.lock);
      }
 }
```

log_write-

```
/* takes a buffer as an argument.
records the block's sector number (where the
buffer is stored) in memory,
and reserves it a slot in the log on disk, and marks
the buffer B_DIRTY bit to prevent the buffer cache
from evicting it. This is needed as the cached copy is
the only record of the modification of the buffer.

Notices when a block is written multiple
times when a single transaction is getting processed
and allocates that block the same slot in the log on disk.
This optimization to save log space is called absorption.
*/
void
log_write(struct buf *b)
{
      int i;
      /*log.lh.n = log header’s n variable, this indicates how many blocks
      are stored in the corresponding log. If this exceeds LOGSIZE, we have
      a problem, as the array we use to store the block numbers is of the
      size LOGSIZE. Hence only n < LOGSIZE is valid

      Additionally, there is a problem if the header says that there are
      more blocks than the actual number of blocks in the log
      */
      if (log.lh.n >= LOGSIZE || log.lh.n >= log.size - 1)
      panic("too big a transaction");
      //if no transactions (system calls) currently running
      if (log.outstanding < 1)
            panic("log_write outside of trans");
      //acquires a lock on the log
      acquire(&log.lock);
      /*checking if a block is written multiple times in a
      single transaction*/
      for (i = 0; i < log.lh.n; i++) {
```

```
            if (log.lh.block[i] == b->blockno) // log absorption
                  break;
      }
      //reserve a slot for the block in the log on disk
      log.lh.block[i] = b->blockno;
      //update the log header size
      if (i == log.lh.n)
            log.lh.n++;
      // prevent eviction by setting B_DIRTY flag
      b->flags |= B_DIRTY;
      //release the lock on the log
      release(&log.lock);
}
```

commit-

```
static void
commit()
{ // checks if log header size > 0 i.e. if there are any blocks which need
to be modified in the disk

  if (log.lh.n > 0) {
    write_log();     // Copies each block modified in the transaction from
the buffer cache to its slot in the log on disk
    /* Writes the header block to disk. This is a commit point, and a crash
    after the write will result in recovery redoing the transaction's
writes from the log. */
    write_head();    // Write header to disk -- the real commit

    install_trans(); // Reads each block from the log and writes it to the
proper place in the file system.

    log.lh.n = 0; // Setting log header size to 0 as the blocks from the
log have been written to the proper places mentioned above

    /* Finally end_op writes the log header with a count of zero; this has
to happen before the next transaction starts writing logged blocks, so that
a crash doesn't result in recovery using one transaction's header with the
subsequent transaction's logged blocks*/
```

```
    write_head();    // Erase the transaction from the log
  }
}
```

bread-

```
// Return a B_BUSY buf with the contents of the indicated disk sector.
struct buf*
bread(uint dev, uint sector)
{
  struct buf *b; // defined a buffer b

  b = bget(dev, sector); //bread calls bget to get a buffer for the given
sector;
  if(!(b->flags & B_VALID))// checks if valid flag is not set
                           // B_VALID indicates that the buffer contains a
copy of the block
    iderw(b); // if valid flag is not set, buffer needs to be read from
disk
  return b; // return the buffer with contents of indicated disk sector
}
```

brelse-

```
//When the caller is done with reading from/writing to the sector using the
associated allocated buffer, it must call brelse to release the buffer.
// Release a B_BUSY buffer.
// Move the buffer to the head of the MRU(Most recently used) list.
void
brelse(struct buf *b)
{
  if((b->flags & B_BUSY) == 0)
    panic("brelse");//Before brelse, B_BUSY flag must be set. Something is
wrong, if it isn't set.
```

```
  acquire(&bcache.lock);// Acquire lock on buffercache.

/* Moving buffer to the front of the buffer cache(doubly linked list of
buffers) */
/*
This makes the list ordered by how recently the buffers were used(meaning
released). The first buffer in the list is the most recently used and the
last is the least recently used.
*/
  b->next->prev = b->prev;
  b->prev->next = b->next;
  b->next = bcache.head.next;
  b->prev = &bcache.head;
  bcache.head.next->prev = b;
  bcache.head.next = b;

  b->flags &= ~B_BUSY;// Clears the B_BUSY bit
  wakeup(b);// Wakes any processes sleeping on the buffer

  release(&bcache.lock);// Releases the lock on buffercache.
}
```