Buffer Cache

Tanaji Patil

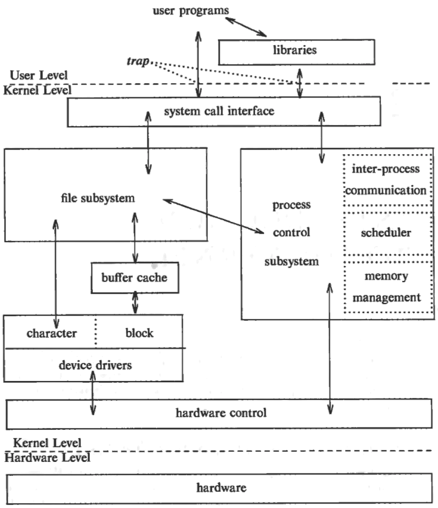
Department of Computer Science and Engineering

Contents

Buffer Cache

- kernel could read/write directly to/from the disk.
  - slow disk transfer rate
  - system response time is poor
- minimize the frequency of disk access by keeping a pool of data buffers
- data buffers are called as buffer cache
- it contains data in recently used blocks

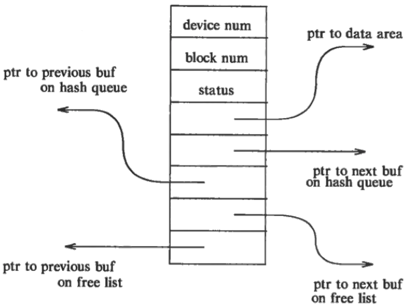
Position of Buffer Cache



Buffer Headers I

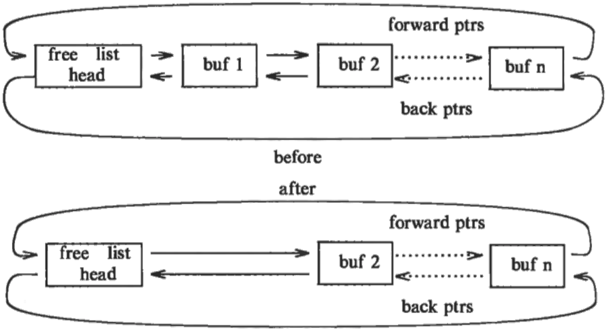
- During system initialization, the kernel allocates space for a number of buffers, configurable according to memory size and performance constraints.
- Two parts of the buffer:
  1. a memory array that contains data from the disk.
  2. **“buffer header”** that identifies the buffer.
- Data in a buffer corresponds to data in a logical disk block on a file system.
- A disk block can **never** map into more than one buffer at a time.

Buffer Headers II



- **device number:** specifies logical file system
- **block number** of the data on disk
- **ptr to data area:** size must be at least as big as the size of a disk block.
- The status of a buffer is a combination of –
  - Buffer is locked / busy
  - Buffer contains valid data
  - Kernel must write the buffer contents to disk before reassigning the buffer; called as **delayed-write**
  - Kernel is currently reading or writing the contexts of the buffer to disk
  - A process is waiting for buffer to become free.
- The two set of pointers are used for traversal of the buffer queues (doubly circular linked lists).

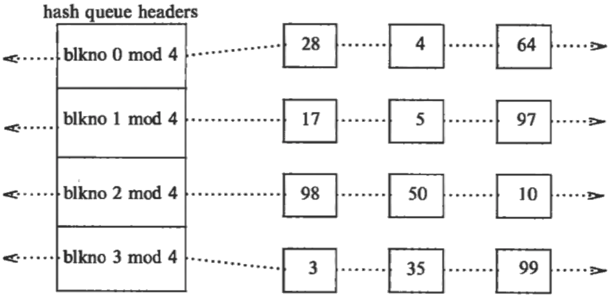
Structure of Buffer Pool



Free list of buffers

- Algorithm: least recently used (LRU)
- System Boot: all buffers are on the free list
- Free buffer is taken from the head of the free list
- After use, free buffers are attached to the end of the list

STRUCTURE OF BUFFER POOL



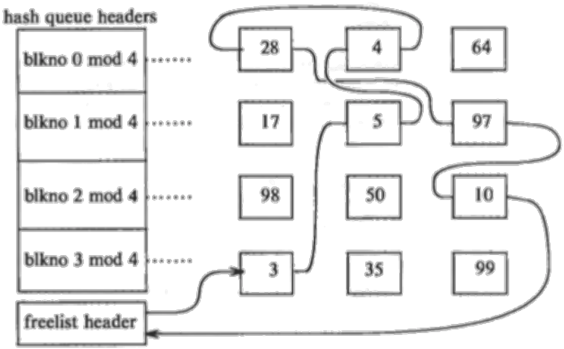
Buffers on the Hash Queues

- Hash queues are doubly linked circular lists.
- Separate queues, **hashed** as a function of the device and block number
- Every disk block mapped to only one hash queue and only once
- A buffer is always on a hash queue, but it may or may not be on the free list

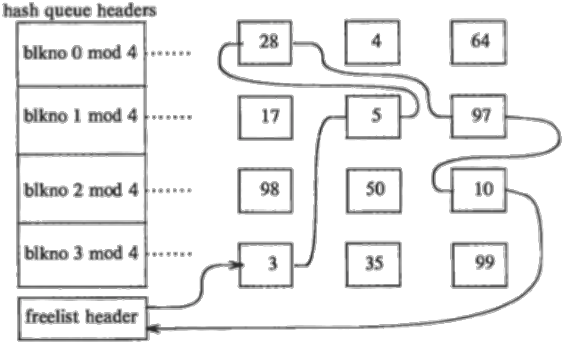
Scenarios for Retrieval of a Buffer

1. Block is found on its hash queue and its buffer is free.
2. Block could not be found on the hash queue, so a buffer from the free list is allocated.
3. Block could not be found on the hash queue, and when allocating a buffer from free list, a buffer marked **delayed write** is allocated. Then the kernel must write the **delayed write** buffer to disk and allocate another buffer.
4. Block could not be found on the hash queue and the free list of buffers is empty.
5. Block was found on the hash queue, but its buffer is currently busy.

Scenario-1

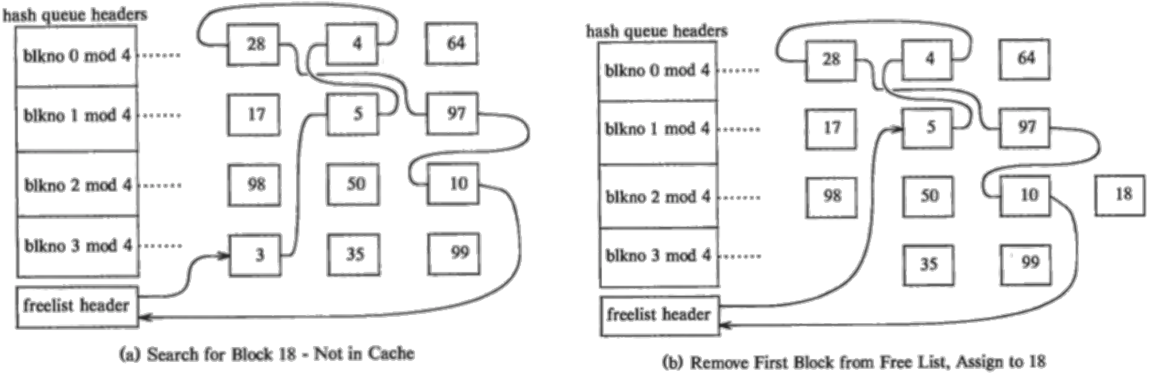


(a) Search for Block 4 on First Hash Queue

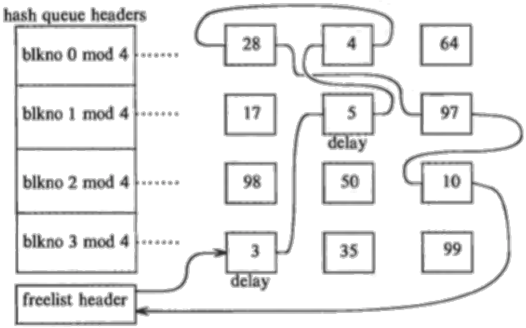


(b) Remove Block 4 from Free List

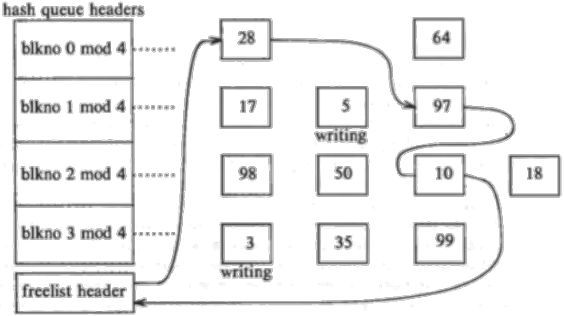
SCENARIO-2


hash queue headers
blkno 0 mod 4
blkno 1 mod 4
blkno 2 mod 4
blkno 3 mod 4
freelist header
28
4
64
17
5
97
98
50
10
3
35
99
(a) Search for Block 18 - Not in Cache
hash queue headers
blkno 0 mod 4
blkno 1 mod 4
blkno 2 mod 4
blkno 3 mod 4
freelist header
28
4
64
17
5
97
98
50
10
18
35
99
(b) Remove First Block from Free List, Assign to 18

Scenario-3

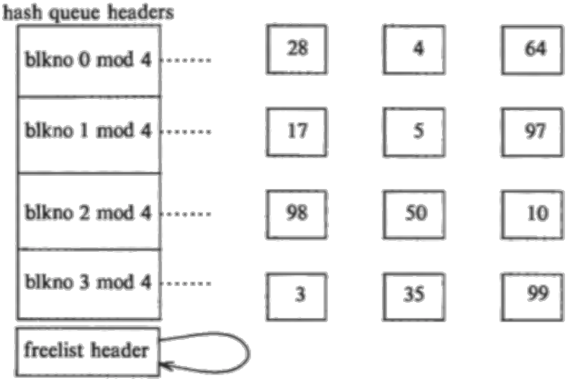


(a) Search for Block 18, Delayed Write Blocks on Free List



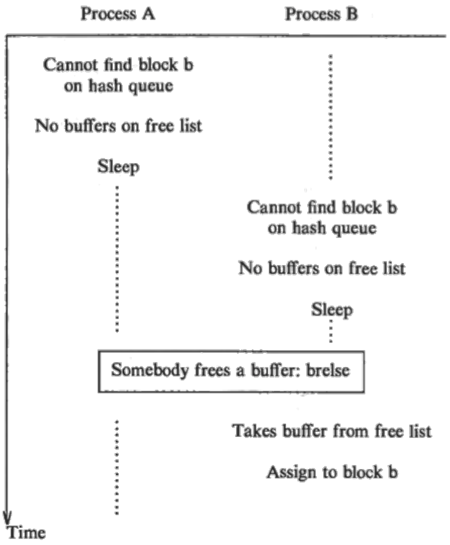
(b) Writing Blocks 3, 5, Reassign 4 to 18

Scenario-4

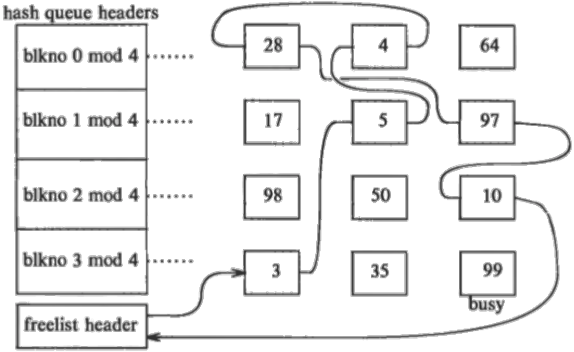


Search for Block 18, Empty Free List

Scenario-4 Race for Free Buffer

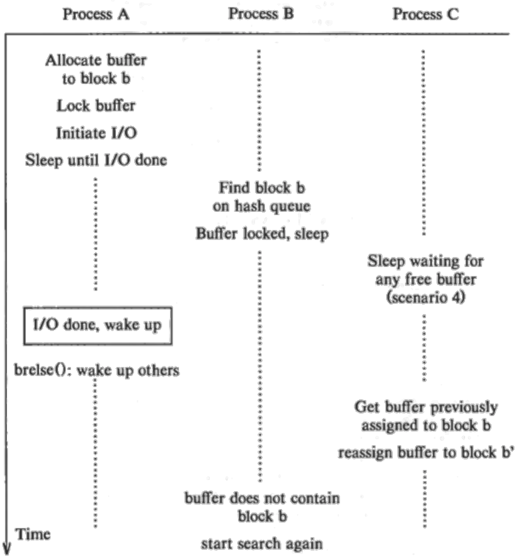


SCENARIO-5



Search for Block 99, Block Busy

Scenario-5 Race for Locked Buffer



Algorithm: getblk

```
while (buffer not found) {
   if (block in hash queue) {
     if (buffer busy) { // scenario 5
       sleep (event: buffer becomes free);
       continue;     // back to while loop
     }
     mark buffer busy; // scenario 1
     remove buffer from free list;
     return buffer;
   } else {
     if (there are no buffers on the free list) {
       sleep (event: any buffer becomes free);
       continue;     // back to while loop
     }
     remove buffer from free list;
     if (buffer marked for delayed write) {
       asynchronous write buffer to disk;
       continue:     // back to while loop;
     }
     remove buffer from old hash queue; // scenario 2
     put buffer onto new hash queue;
     return buffer;
   }
 }
```

Algorithm: brelse

```
/*
* Algorithm: brelse
* Input: locked buffer
* Output: none
*/
{
 wakeup all processes (event: waiting for any buffer to become free);
 wakeup all processes (event: waiting for this buffer to become free);

 raise processor execution level to block interrupts;

 if (buffer contents valid and buffer not old)
  enqueue buffer at end of free list;
 else
  enqueue buffer at beginning of free list;

 lower processor execution level to allow interrupts;

 unlock (buffer);
}
```

Algorithm: bread

```
/*
* Algorithm: bread
* Input: file system number
*      block number
* Output: buffer containing data
*/
{
 get buffer for block (algorithm: getblk);

 if (buffer data valid)
   return buffer;

 initiate disk read;
 sleep (event: disk read complete);

 return buffer;
}
```

Algorithm: breada

```
/* Algorithm: breada
* Input: file system number and block number for immediate read
*        file system number and block number for asynchronous read
* Output: buffer containing data for immediate read
*/
{
 if (first block not in cache) {
  get buffer for first block (algorithm: bread);
  if (buffer data not valid)
   initiate disk read;
 }
 if (second block not in cache) {
  get buffer for second block (algorithm: getblk);
  if (buffer data valid)
   release buffer (algorithm: brelse);
  else
   initiate disk read;
 }
 if (first block was originally in the cache) {
  read first block (algorithm: bread);
  return buffer;
 }
 sleep (event: first buffer contains valid data);
 return buffer;
}
```

Algorithm: bwrite

```
/*
* Algorithm: bwrite
* Input: buffer
* Output: none
*/
{
 initiate disk write;
 if (I/O synchronous)
 {
  sleep (event: I/O complete);
  release buffer (algorithm: brelse);
 }
 else if (buffer marked for delayed write)
  mark buffer to put at head of free list;
}
```

Thank you . . .