UNIX INTERNALS

Dr. Radha Senthilkumar, AP(Sr. Gr.)
Department of IT
MIT, Chromepet
Anna University, Chennai.

General Overview of the system

Topics

- History
- System Structure
- User Perspective
- Operating System Services
- Assumption about Hardware

Reference:

The Design of the UNIX Operating System by Maurice J. Bach

A Little History First: UNIX

- Initial design by Ken Thompson, Dennis Ritchie and others at AT&T's Bell Telephone Laboratories (BTL) in 1969: 32 years ago!
- AT&T made the source available to Universities for research and educational use.
- 1973 UNIX was rewritten in C resulting in Version 4.
  - The C language was also originally designed and developed for use on the UNIX system by Dennis Ritchie
  - C was evolved from 'B', developed by Thompson.

UNIX History

- AT&T sold UNIX to Novel; Novel passed the UNIX trademark to X/OPEN and sold source code to Santa Cruz Operation (SCO).
- Plan 9 is AT&T's successor to UNIX
- AT&T was unable to market UNIX as a product so they made the source code available to Universities for use in Research and Education.
- Influential variant: Berkeley Software Distributions (**BSD**) distributed by the Computer Systems Research Group, University of California at Berkeley
  - Berkeley obtained UNIX from AT&T in December of 1974

UNIX History

- UNIX ported to many different architectures
- Microsoft and SCO collaborated to port UNIX to the Intel 8086 architecture: **XENIX**
- AT&T purchased 20% of Sun: result is joint effort to develop SVR4
- In 1982 AT&T was broken up and was now able to market UNIX. They released System III in 1982 and System V the following year.
- System V UNIX introduced virtual memory (different from BSD, called *regions*), IPC (shared memory, semaphores, message queues), remote file sharing, shared libraries and ***STREAMS.***

BSD UNIX

- 2 BSD: text editor **vi**
- 3 BSD: demand-paged virtual memory
- 4.0BSD: performance improvements
- 4.1BSD: job control, autoconfiguration
- 4.2/4.3BSD: reliable signals, fast filesystem, improved networking (**TCP/IP reference implementation**), sophisticated IPC primitives
- 4.4 BSD: stackable and extensible vnode interface, network file system, log-structured filesystem, other filesystems, POSIX support, and other enhancements.

Why UNIX

- Historical significance
- Advanced features developed for or ported to UNIX
- Availability of source code and research papers
- Highlights key design and architectural issues

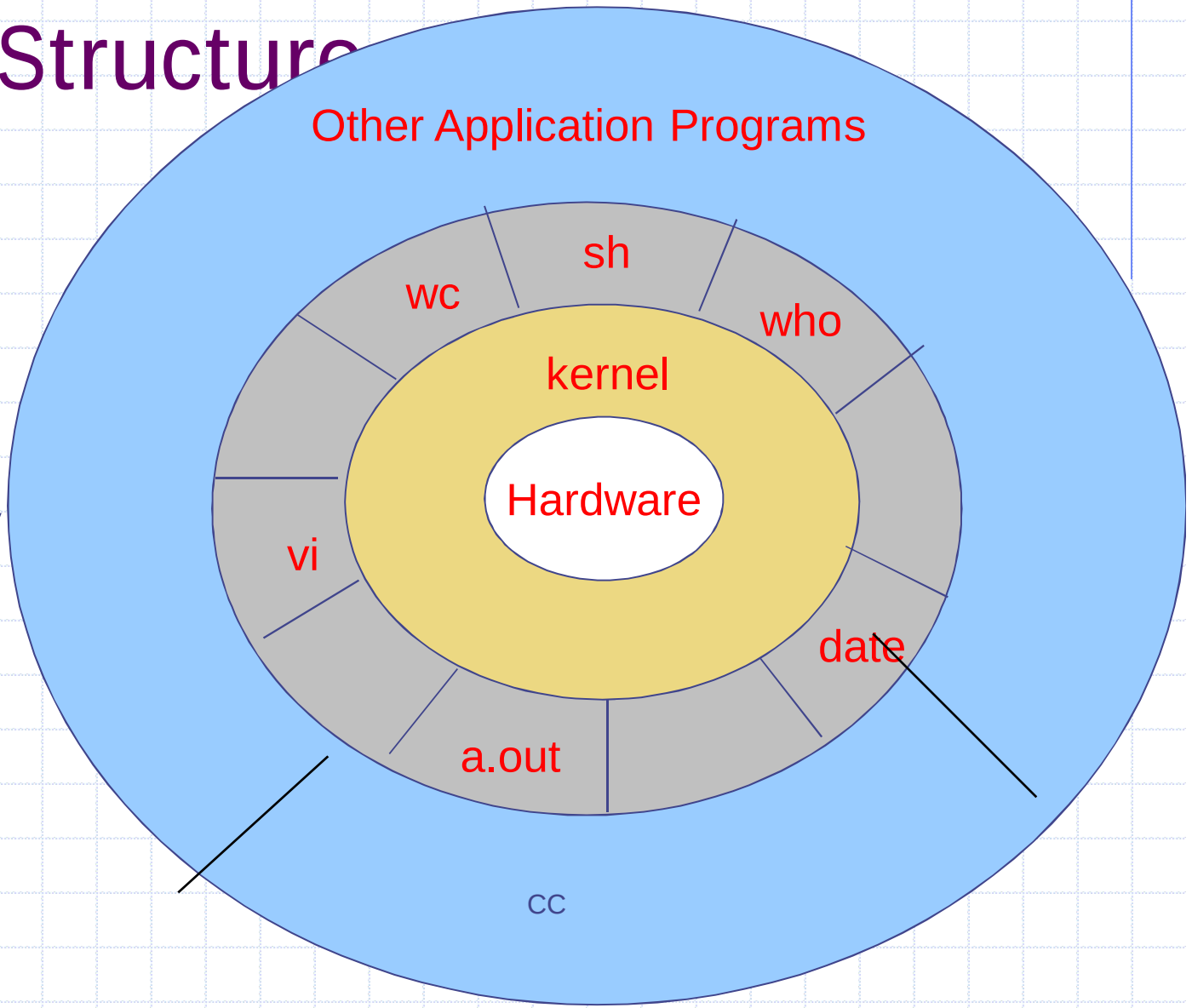
System Structure

The UNIX system is functionally organized at three levels:

- The kernel, which schedules tasks and manages Storage;
- The shell, which connects and interprets users Commands, calls programs from memory, and executes them; and
- The tools and applications that offer additional functionality to the operating system

System Structure

- There are 64 system calls in System V.
- 32 are used more frequently



System Structure: The Kernel

The kernel

The heart of the operating system, the kernel controls the hardware and turns part of the system on and off at the programmer's command.

originally found in /usr/sys, and composed of several sub-components:

- *conf* — originally found in /usr/sys/conf, and composed of configuration and machine-dependent parts, often including boot code
- *dev* — Device drivers (originally /usr/sys/dev) for control of hardware (and sometimes pseudo-hardware)
- *sys* — The "kernel" of the operating system, handling memory management, system calls, etc.
- *h* (or *include*) — Header files, generally defining key interfaces within the system, and important system-specific invariables

System Structure

Commands — Most Unix implementations make little distinction between commands (user-level programs) for system operation and maintenance (e.g. *cron*)

some major categories are:

- *sh* — The Shell, the primary user-interface on Unix and the center of the command environment.
- *Utilities* — the core of the Unix command set, including *ls*, *grep*, *find* and many others. This category could be subcategorized:
  - *System utilities* — such as *mkfs*, *fsck*, and many others; and
  - *User utilities* — *passwd*, *kill*, and others.
- *Document*
- *Communications*

Development Environment

Most implementations of Unix contained a development environment sufficient to recreate the system from source code.

The development environment included:

- ***cc*** **— The C language compiler (first appearing in V3 Unix)**
- ***as*** **— The machine-language assembler for the machine**
- ***ld*** **— The linker, for combining object files**
- ***lib*** **— Libraries. Originally *libc*, the system library**
- **make - The build manager (designed to effectively automate the build process**

User Perspective : File System

The UNIX file system is characterized by

- A Hierarchical structure
- Consistent treatment of file data
- The ability to create and delete files
- Dynamic growth of files
- The protection of file data
- The treatment of peripheral devices as files

  - Is organized as tree with single root node called root, every non-leaf node of the file system structure is a directory of files and files at the leaf nodes of the tree are directories, regular files or special device files

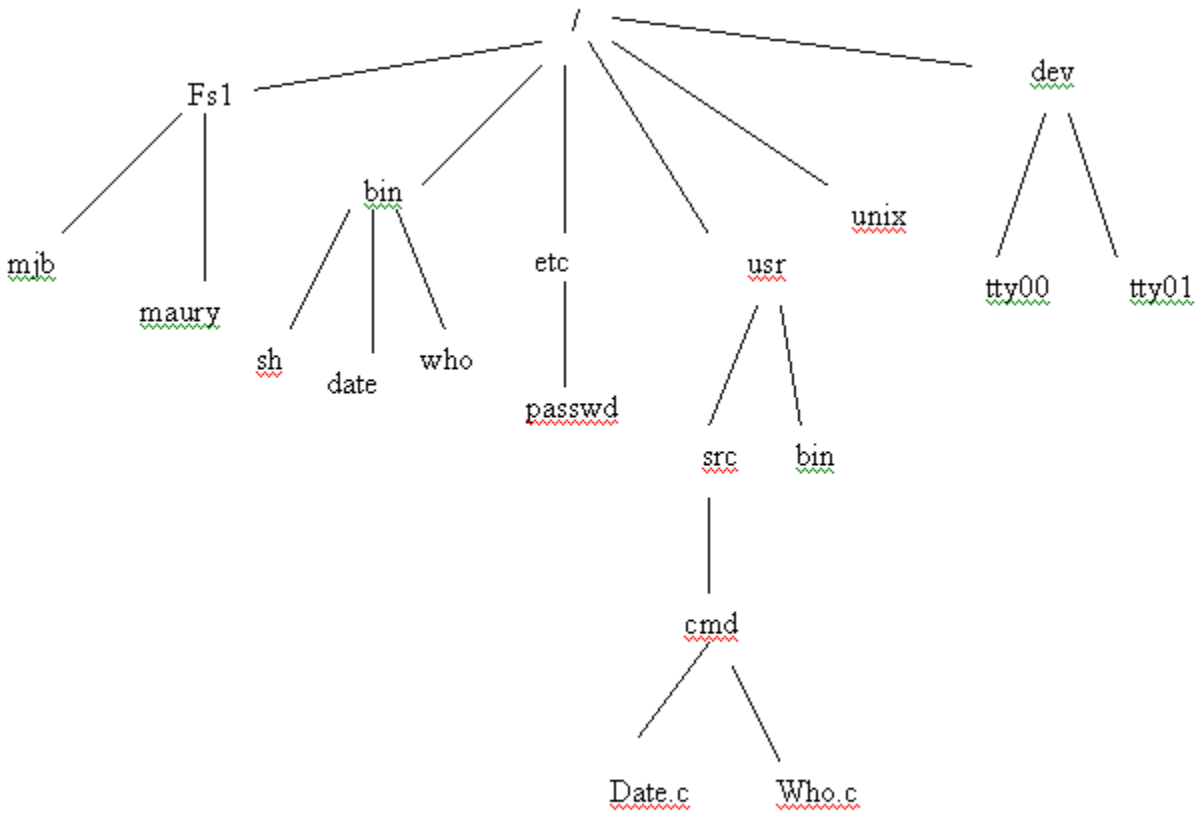
  - The name of the file is given by a path name that describes how to locate the file in the file system hierarchy.

Sample file system tree

The path names “/etc/passwd”, “/bin/who”, and “/usr/src/cmd/who.c” designate files in the tree, but “/bin/passwd” and “/usr/src/date.c” do not.

A path name does not have to start from root

e.g. “/dev/tty01”



User Perspective : File System

```
#include <fcntl.h>
Char buffer[2048];
Int version=1;
main (argc, argv)
 int argc;
Char *argv[];
{
 int fdold, fdnew;
 if (argc!=3)
{
  printf("need! Argument for copy program\n");
  exit(1);
}
fdold=open(argv[1],0_RDONLY);
If (fdold == -1)
{
  printf("cannot open the file %s\n", argv[1]);
  exit(1);
}
  fdnew = creat(argv[2], 0666);
```

```
If (fdnew ==-1 )
 {
  printf("cannot create file %s\n, argv[2]);
  exit();
}
Copy (fdold,fdnew); exit(0); }
Copy (old, new)
 int old, new;
{
  int count;
   while (( count = read(old, buffer,
       sizeof(buffer)))>0)
  write (new, buffer, count); }
```

Program to copy a file

User Perspective : Processing Environment

A program is an executable file, and a process is an instance of the program in execution.

Process control

- fork,
- Exec
- Wait
- Exit

The shell executes the command synchronously and asynchronous

User Perspective : Processing Environment

```
Main(argc, argv)
Int argc;
Char *argv[];
{
  if (fork() == 0)
 execl("copy", "copy", argv[1],
     argv[2], 0);
Wait((int *), 0);
Printf("copy done\n");
}
```

Program that create a new process to copy file

The returned value of **fork()**:

- If **fork()** returns a negative value, the creation of a child process was unsuccessful.
- **fork()** returns a zero to the newly created child process.
- **fork()** returns a positive value, the ***process ID*** of the child process, to the parent.

User Perspective : Processing Environment

- Shell is the command interpreter program that users typically execute after logging into the system.
  - The shell usually execute a command synchronously
    - Eg who
  - Shell also execute asynchronously i.e execute in the background
    - Who &
  - Shell is a user program and not part of the kernel , it is easy to modify it and tailor it to a particular environment.

User Perspective : Building a Block Primitives

- Unix system is to provide operating system primitives that enable users to write small, modular programs that can be used as building blocks to build more complex programs.
  - Redirect I/O
    - ls > output
    - mail aravind<letter
    - nroff –mm <doc1 > doc1.out 2>errors
  - Pipe
    - grep main a.c b.c c.c
    - grep main a.c b.c c.c | wc -l

Operating system services

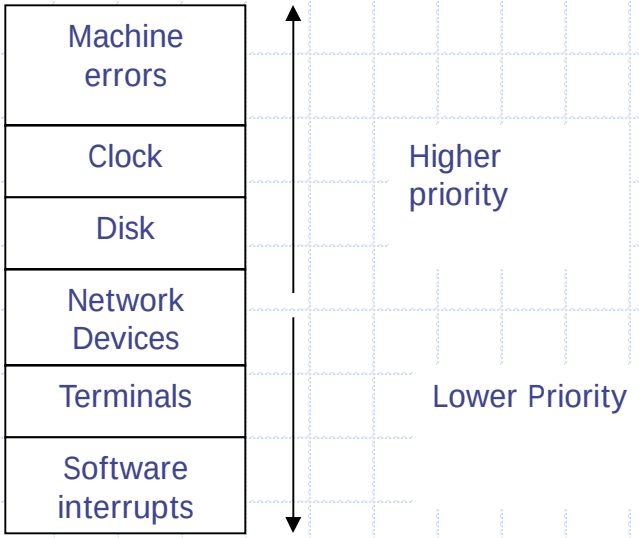
- Controlling and execution of process by allowing process creation, termination or suspension and communication
- Scheduling processes fairly for execution on the CPU.
- Allocating main memory for an executing process.
- Allocating secondary memory for efficient storage and retrieval of user data.
- Allowing processes controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.

Assumptions about hardware

- Two level
  - User
    - Process in this level can access their won instruction and data but not kernel instruction and data.
  - Kernel
    - Can access kernel and user addresses.
- Interrupts and Exception
  - Exception refers to unexpected events caused by a process such as addressing illegal memory, executing privileged instruction, dividing by zero and so on.
  - Interrupts are caused by events that are external to a process.
  - Exceptions happen in the middle of the execution of a an instruction
  - Interrupts happen between the execution of two instructions.

Assumptions about hardware

- Processor Level
  - The kernel must prevent the occurrence of interrupts during critical activity
  - Corrupt the data
- Memory Management
  - The kernel permanently resides in main memory as does the currently executing process.
  - Virtual address
  - physical address



Typical interrupt level

Introduction to Kernel

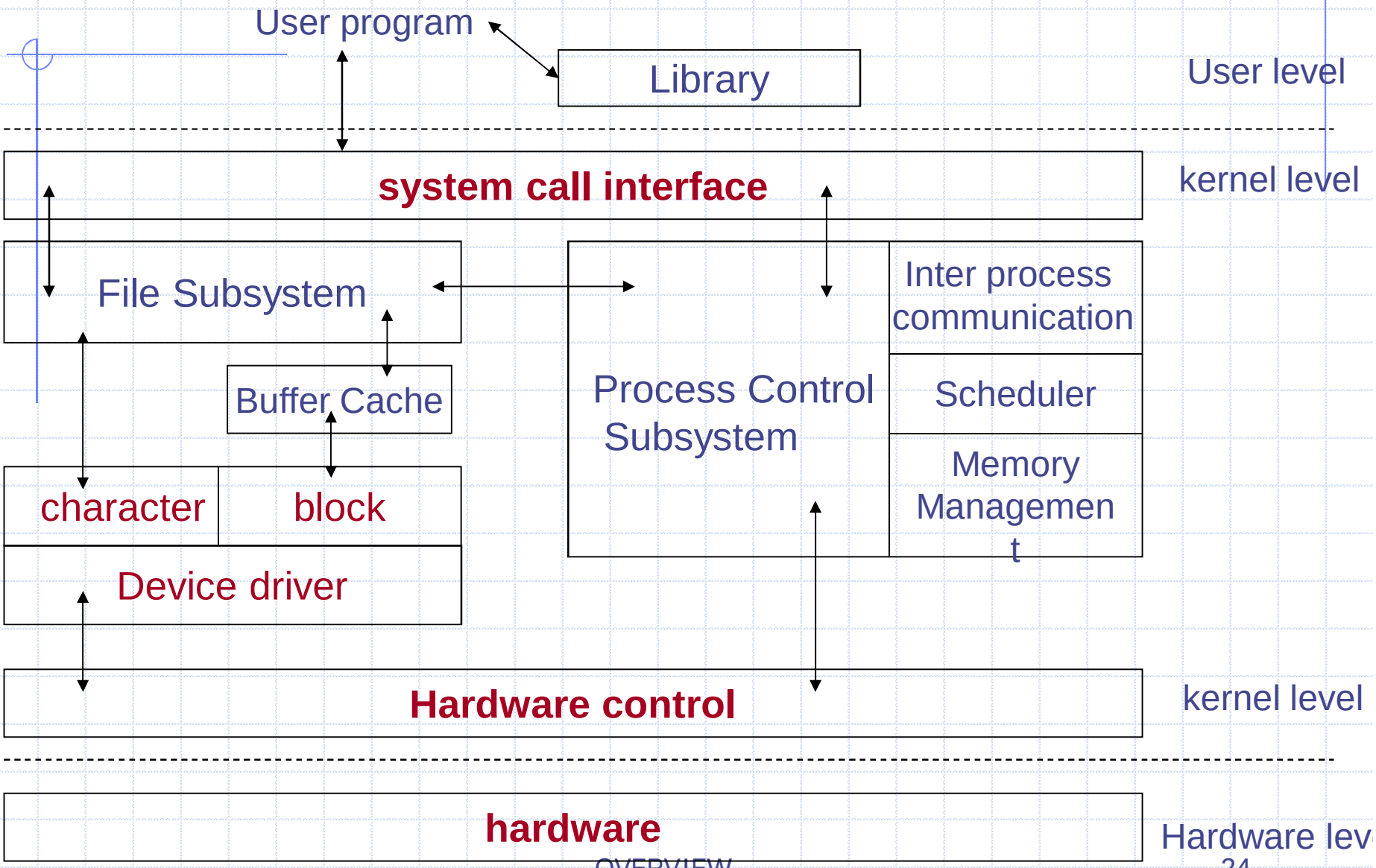
Topics

- Kernel Architecture
- File System
- Process

Reference:

The Design of the UNIX Operating System
by Maurice J. Bach

kernel Architecture (UNIX)



kernel Architecture - Cont

- The libraries map these system calls to the primitive needed to enter the OS.
- Assembly language invokes system call directly without a system call library.
- The libraries are linked with programs at compile time and are thus part of the user program.
- The file subsystem manages files, allocating file space, administering free space, controlling access to files, and retrieving data for users.

kernel Architecture - Cont

- The process interact with the file subsystem via a specific set of system calls, such as open, close , read, write,chown, chmod.
- The file subsystem access file data using buffering mechanism that regulates data flow between kernel and secondary storage devices.
  - Block I/O device drivers
  - Raw data I/O device drivers

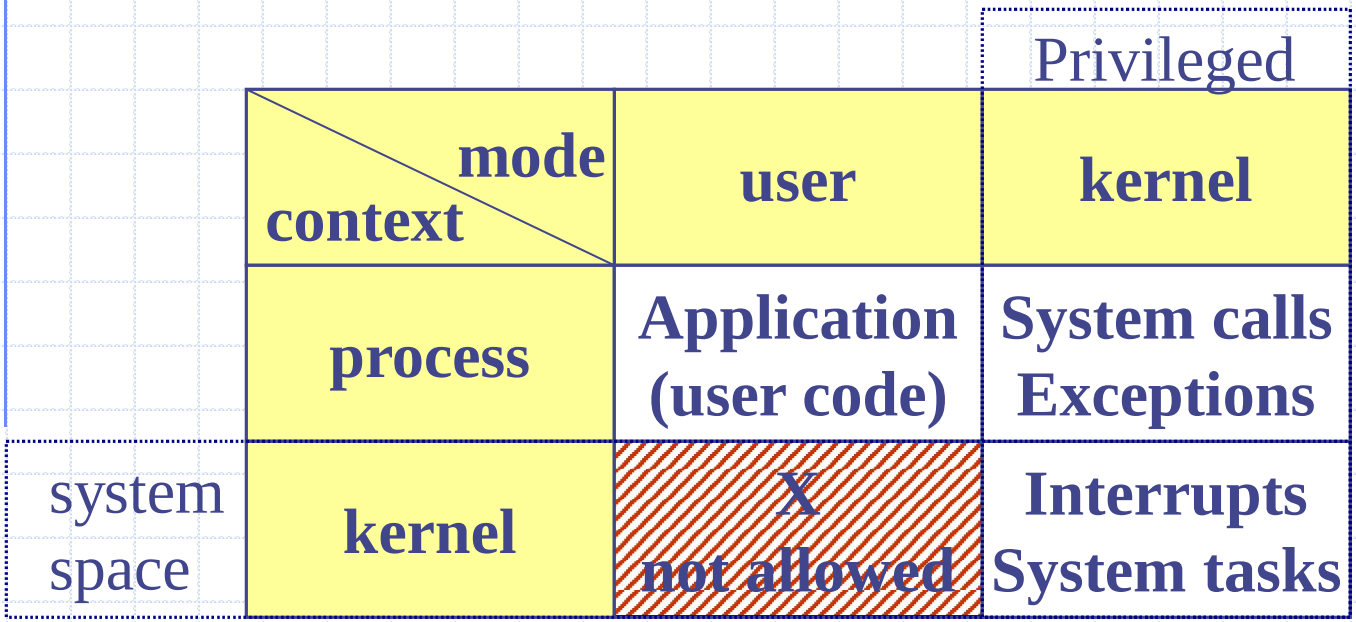
kernel Architecture - Cont

- Process control subsystem is responsible for
  - Process synchronization
  - Inter process communication
  - Memory Management
  - Process scheduling.
- System calls for controlling processes:
  - Fork
  - Exec
  - Exit
  - Wait
  - brk (control the size of memory allocated to a process)
  - signal

kernel Architecture - Cont

- Memory management module control the allocation of memory
  - Swapping
  - Demand paging
- The scheduler module allocate the CPU to processes.
- H/w control is responsible for handling interrupts for communicating with the m/c.
- There are several forms of IPC ranging from asynchronous signaling of events to synchronous transmission of messages between process.

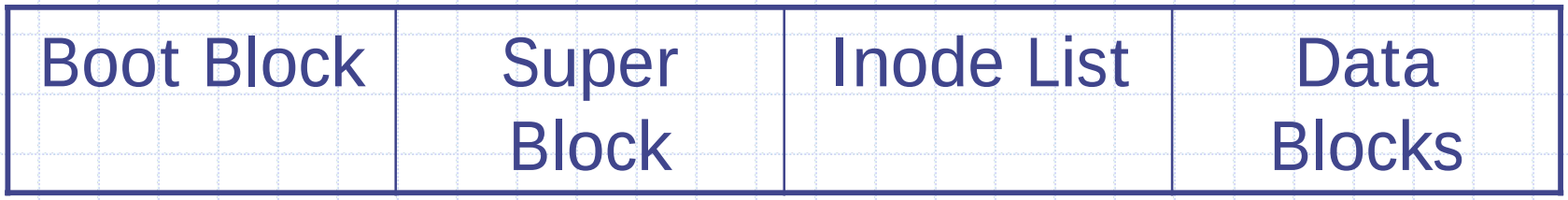
Mode, Space and Context



UNIX uses only two privilege levels

File System

- A file system is consists of a sequence of logical blocks (512/1024 byte etc.)
- A file system has the following structure:



File System: Boot Block

- The beginning of the file system
- Contains bootstrap code to load the operating system
- Initialize the operating system
- Typically occupies the first sector of the disk

File System: Super Block

- Describes the state of a file system
  - How large it is
- Describes the size of the file system
  - How many files it can store
- Where to find free space on the file system
- Other information

File System: Inode List

- Inodes are used to access disk files.
- Inodes maps the disk files
- For each file there is an inode entry in the inode list block
- Inode list also keeps track of directory structure

File System: Data Block

- Starts at the end of the inode list
- Contains disk files
- An allocated data block can belong to one and only one file in the file system

Process

- Process : states + context
- fork & execute
- Executable file
  - Header : describe the attributes of the file
  - text : program text
  - data : data(has initial values) + bbs
  - Symbol table information

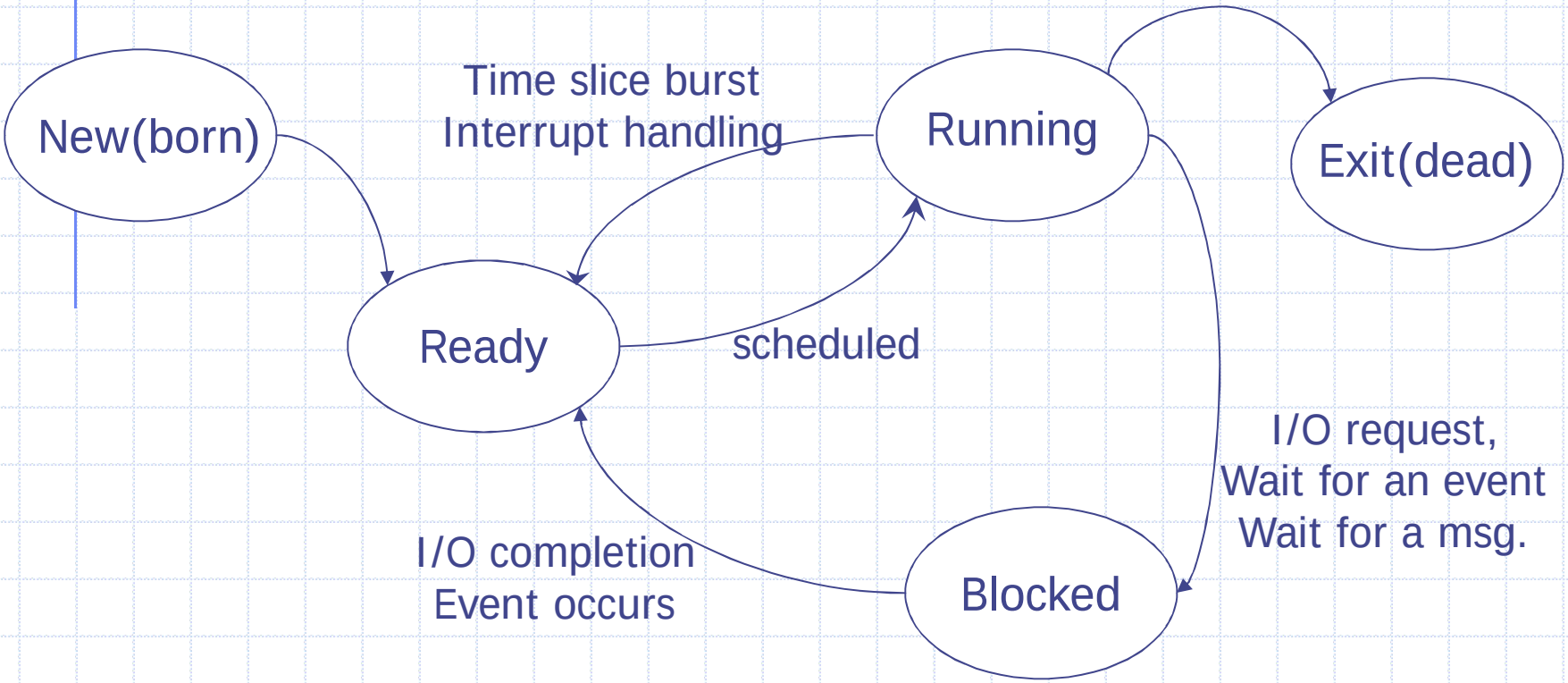
Context switch

```
context_switch (oldPCB, newPCB)
{
   save current register contents into
          oldPCB including PC, SP, ..;
           PC : resume address.
   restore register contents in new PCB
         into registers including PC(jump);
 //   resume here by another instance    of      context
   switch
}
```

Process states(CPU)

- Running :
  - currently has the control of the CPU
  - Executing in user or kernel mode
- Ready :
  - waiting for being scheduled(Queue)
- Blocked :
  - waiting for an event(I/O)
  - cannot be scheduled until

Process States and Transition



The Buffer Cache

TOPICS

- UNIX system Architecture
- Buffer Cache
- Buffer Pool Structure
- Retrieval of Buffer
- Release Buffer
- Reading and Writing Disk Blocks

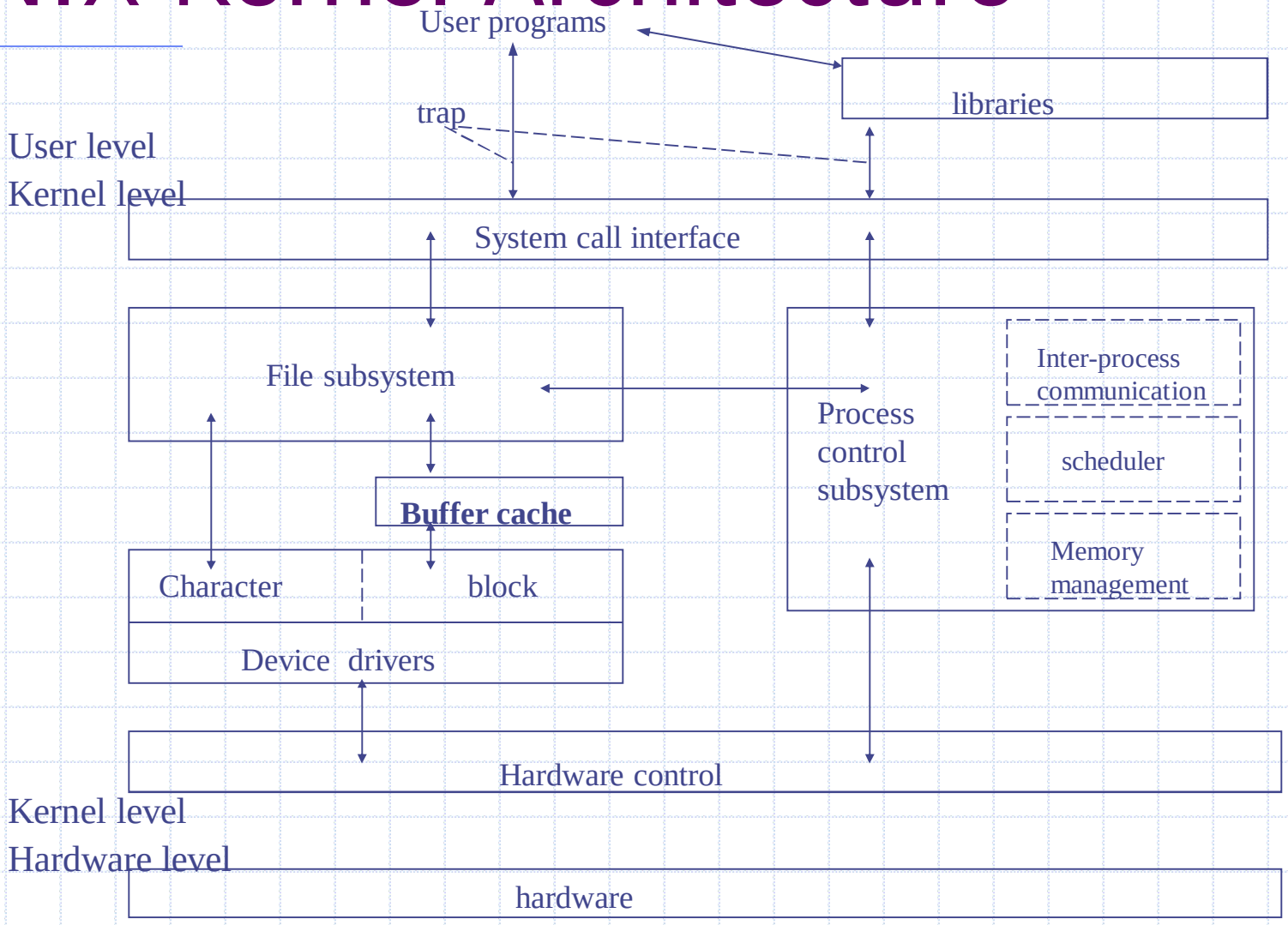
Reference:

The Design of the UNIX Operating System
by Maurice J. Bach

The Buffer Cache

- When a process wants to access data from a file, the kernel brings the data into main memory, alters it and then request to save in the file system
- Example: copy cp one.c two.c
- To increase the response time and throughput, the kernel minimizes the frequency of disk access by keeping a pool of internal data buffer called buffer cache.

UNIX Kernel Architecture



Buffer Cache

- Buffer cache contains the data in recently used disk blocks
- When reading data from disk, the kernel attempts to read from buffer cache.
- If data is already in the buffer cache, the kernel does not need to read from disk
- If data is not in the buffer cache, the kernel reads the data from disk and cache it

Buffer Headers

- A buffer consists of two parts
  - a memory array
  - buffer header
- disk block : buffer = 1 : 1

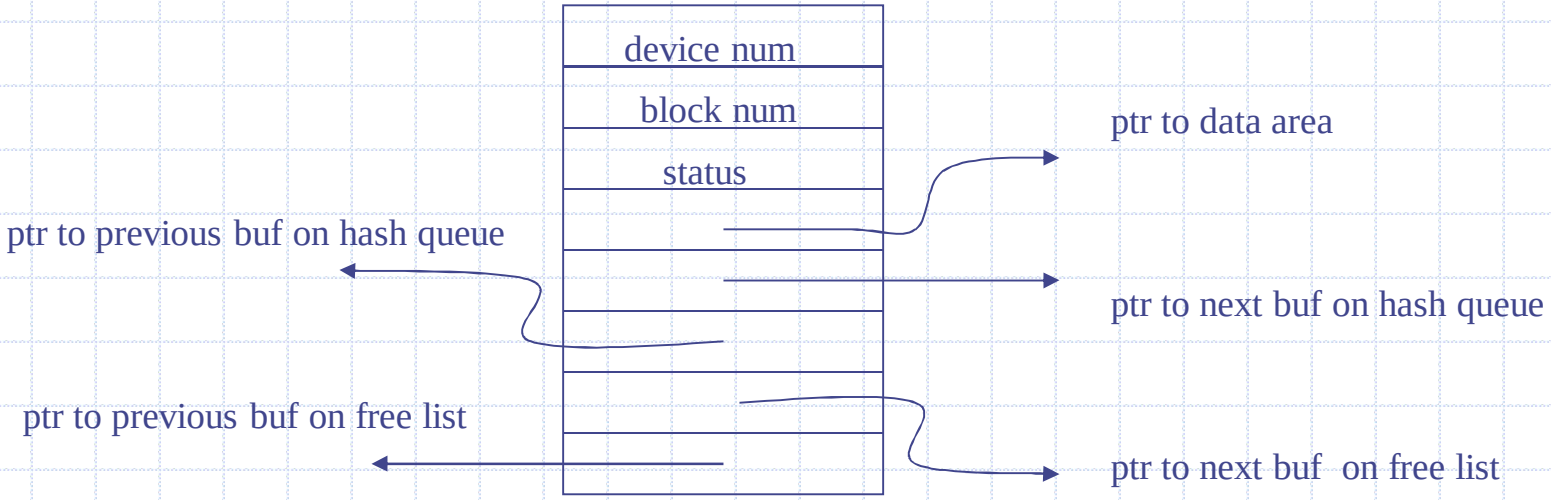


Figure 3.1 Buffer Header

Buffer Headers

- device num
  - logical file system number
- block num
  - block number of the data on disk
- status
  - The buffer is currently locked.
  - The buffer contains valid data.
  - delayed-write
  - The kernel is currently reading or writing the contents of the disk.
  - A process is currently waiting for the buffer to become free.
- kernel identifies the buffer content by examing device num and block num.

Buffer Headers

```
struct buffer_head{
    /*First cache line: */
    struct buffer_head *b_next;    /*Hash queue list*/
    unsigned long b_blocknr;                /*block number*/
    unsigned long b_size;                   /*block size*/
    kdev_t b_dev;                  /*device(B_FREE = free)*/
    kdev_t b_rdev;                 /*Read device*/
    unsigned long b_rsector;                /*Real Buffer location on disk*/
    struct buffer_head *b_this_page;        /*circular list of buffers in one page*/
    unsigned long b_state;                  /*buffer state bitmap(see above)*/
    struct buffer_head *b_next_free;
    unsigned int  b_count;                  /*users using this block*/
    char *b_data;                  /*pointer to data block(1024 bytes)*/
    unsigned int b_list;                    /*List that this buffer appears*/
    unsigned long b_flushtime;     /*Time when this(dirty) buffer should be written*/
    struct wait_queue *b_wait;
    struct buffer_head **b_pprev;           /*doubly linked list of hash-queue*/
    struct buffer_head *b_prev_free;        /* double linked list of buffers*/
    struct buffer_head *b_reqnext;          /*request queue*/

    /*I/O completion*/
    void (*b_end_io)(struct buffer_head *bh,  int uptodate);
    void *b_dev_id;
};
```

Buffer Headers

- **/* buffer head state bits*/**
- #define BH_Uptodate 0 /*1 if the buffer contains valid data*/
- #define BH_Dirty 1 /*1 if the buffer is dirty*/
- #define BH_Lock 2 /*1 if the buffer is locked*/
- #define BH_Req 3 /*0 if the buffer has been invalidated*/
- #define BH_Protected 6 /*1 if the buffer is protected*/

Structures of the buffer pool

- Buffer pool according to LRU
- The kernel maintains a free list of buffer
  - doubly linked list
  - take a buffer from the head of the free list.
  - When returning a buffer, attaches the buffer to the tail.

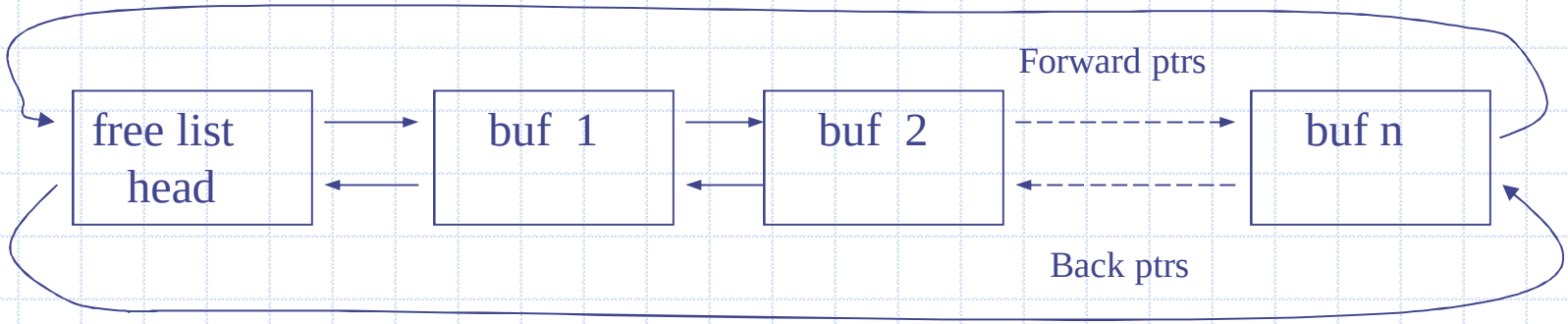


Figure 3.2 Free list of Buffers

Structures of the buffer pool

- When the kernel accesses a disk block
  - separate queue (doubly linked circular list)
  - hashed as a function of the device and block num
  - Every disk block exists on one and only on hash queue and only once on the queue

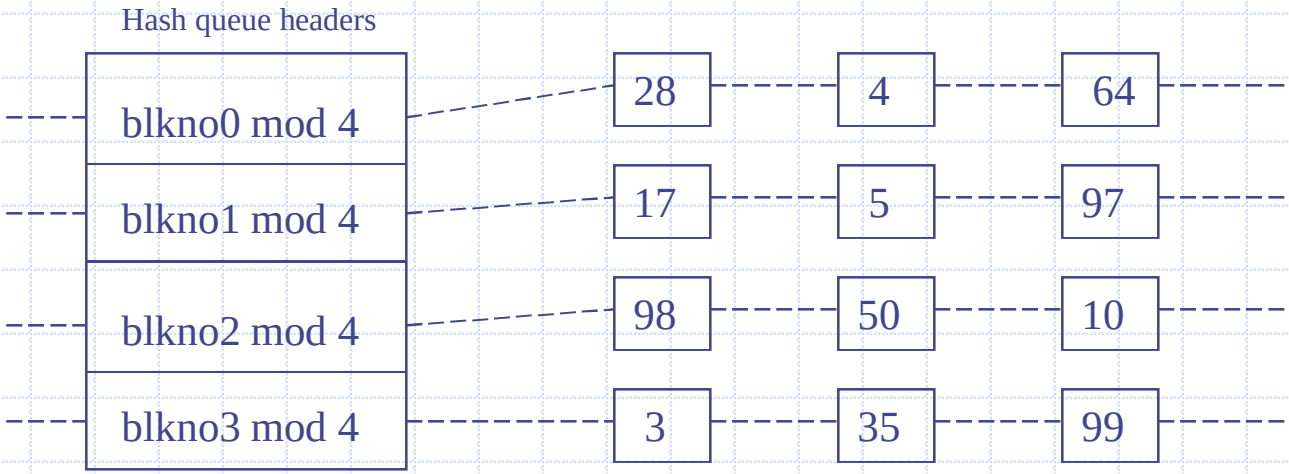


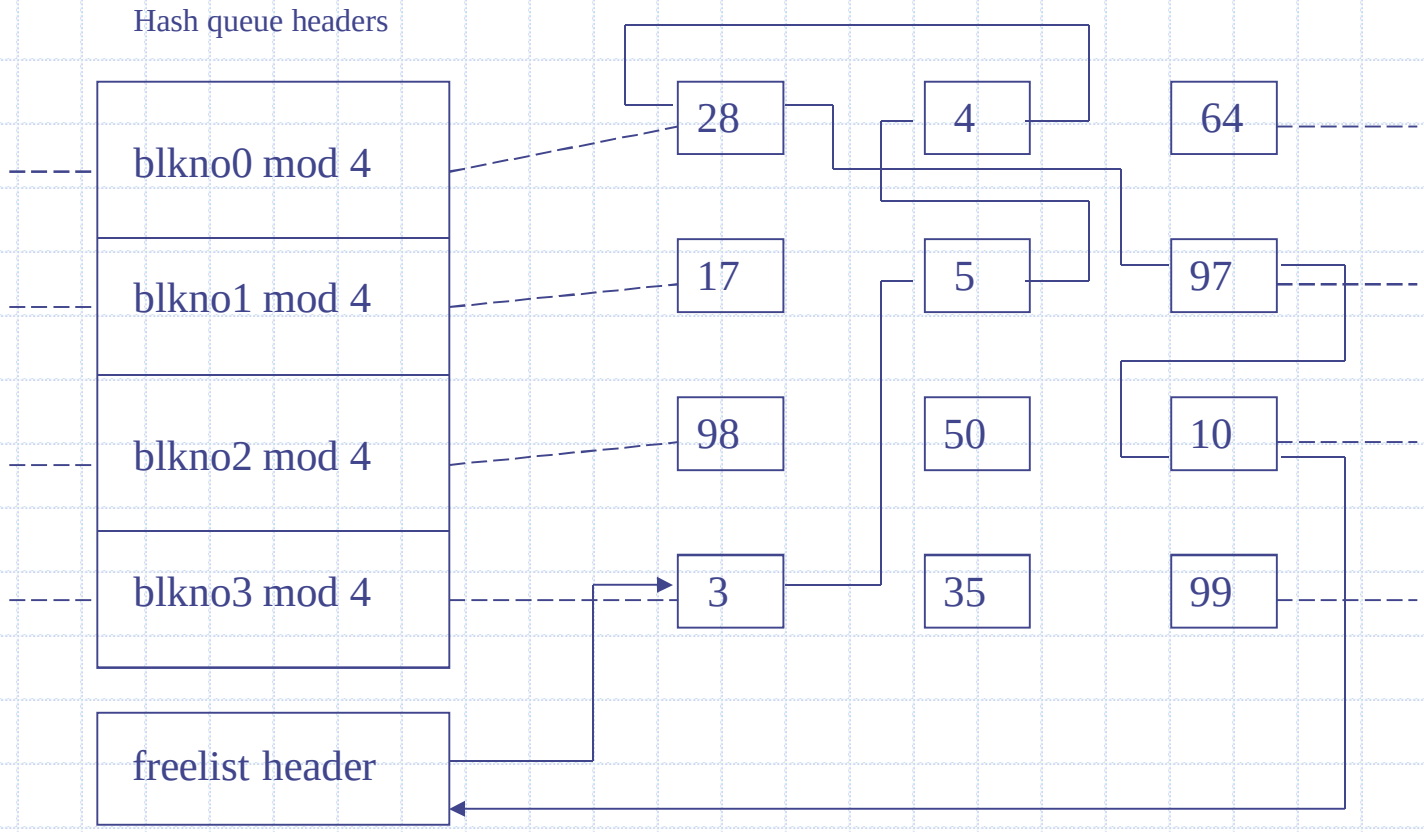
Figure 3.3 Buffers on the Hash Queues

Scenarios for retrieval of a buffer

- Determine the logical device num and block num
- The algorithms for reading and writing disk blocks use the algorithm *getblk*
  - The kernel finds the block on its hash queue
    - The buffer is free.
    - The buffer is currently busy.
  - The kernel cannot find the block on the hash queue
    - The kernel allocates a buffer from the free list.
    - In attempting to allocate a buffer from the free list, finds a buffer on the free list that has been marked “delayed write”.
    - The free list of buffers is empty.

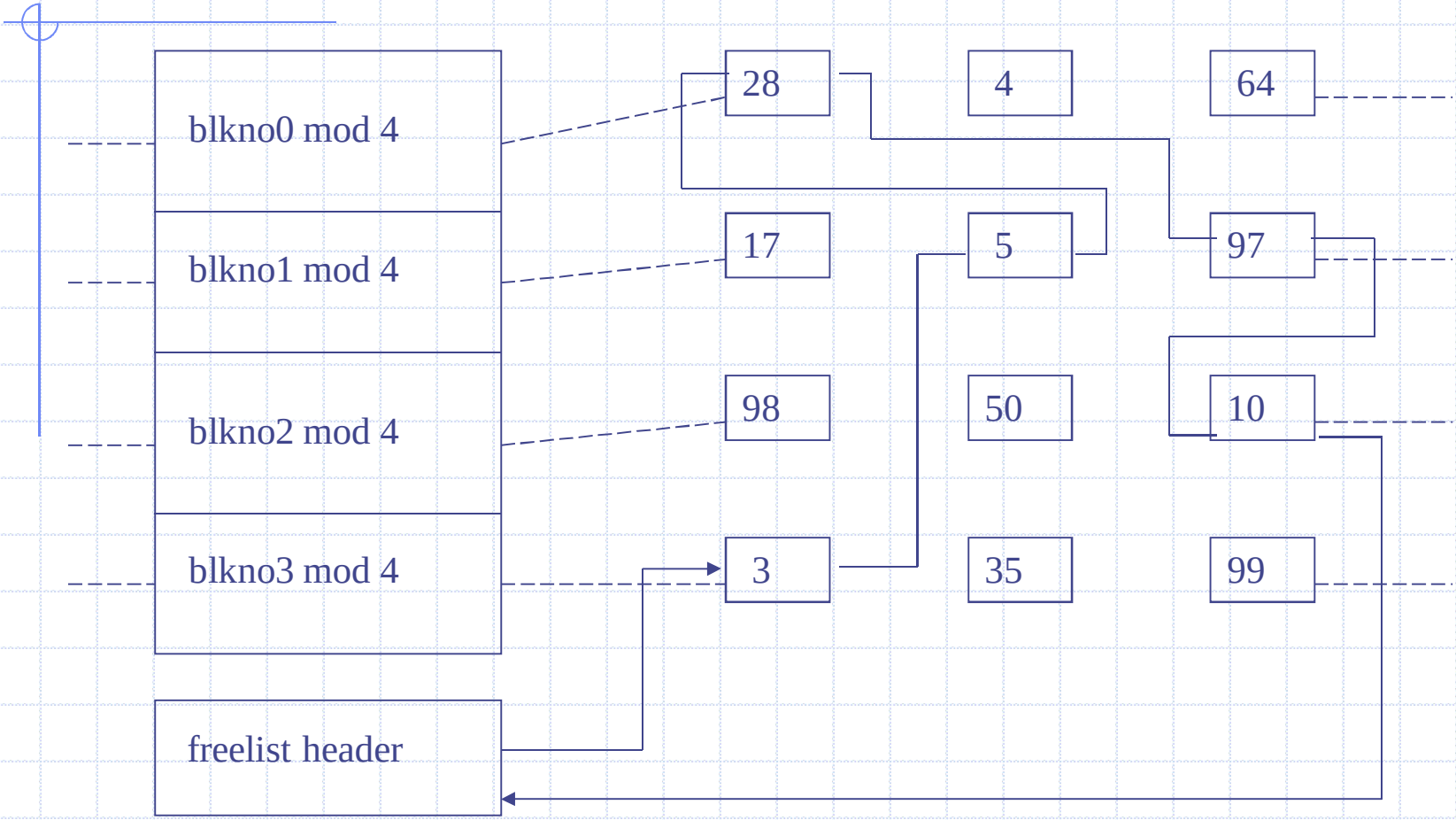
Retrieval of a Buffer:1st Scenario (a)

- The kernel finds the block on the hash queue and its buffer is free



Search for block 4

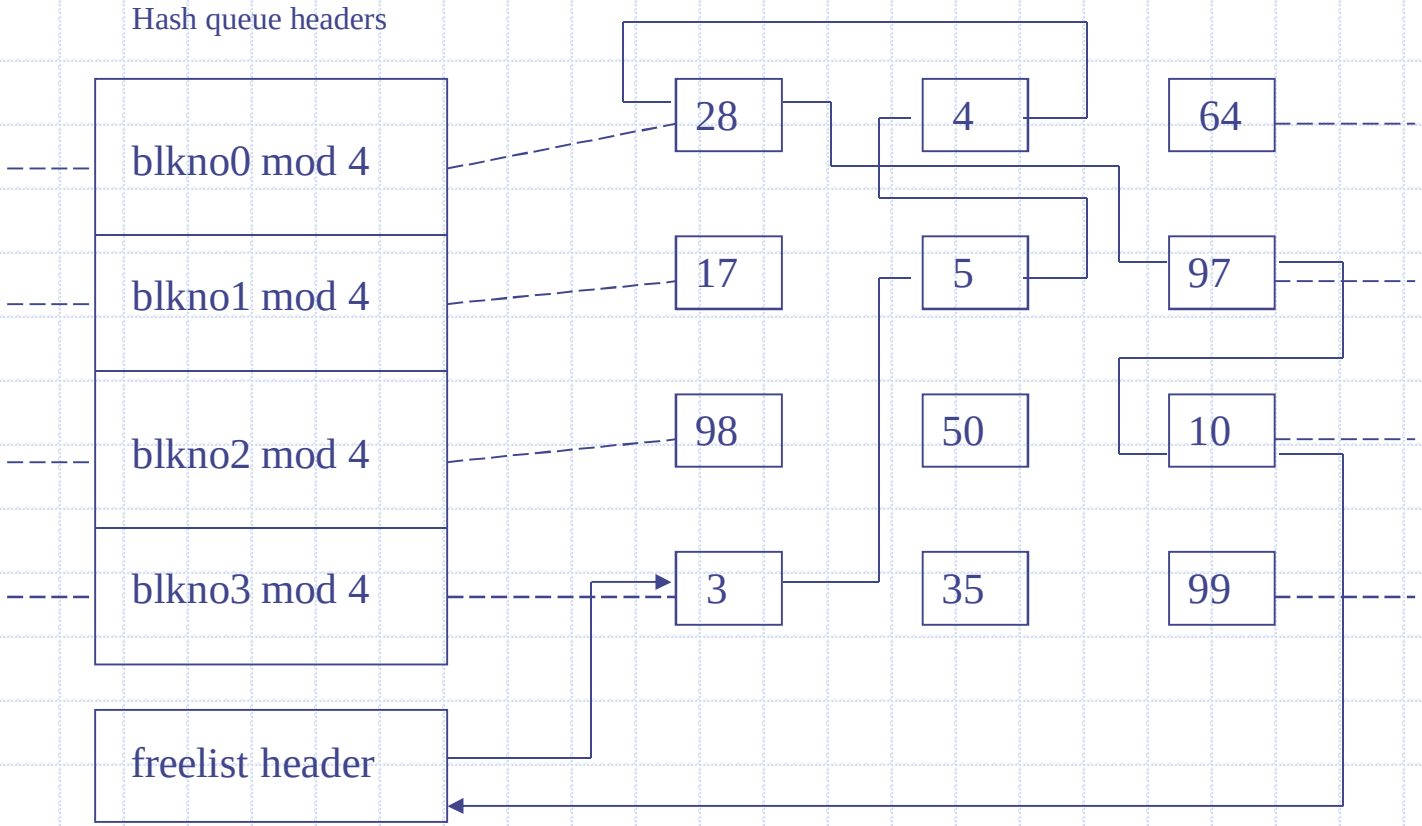
Retrieval of a Buffer:1st Scenario (b)



Remove block 4 from free list

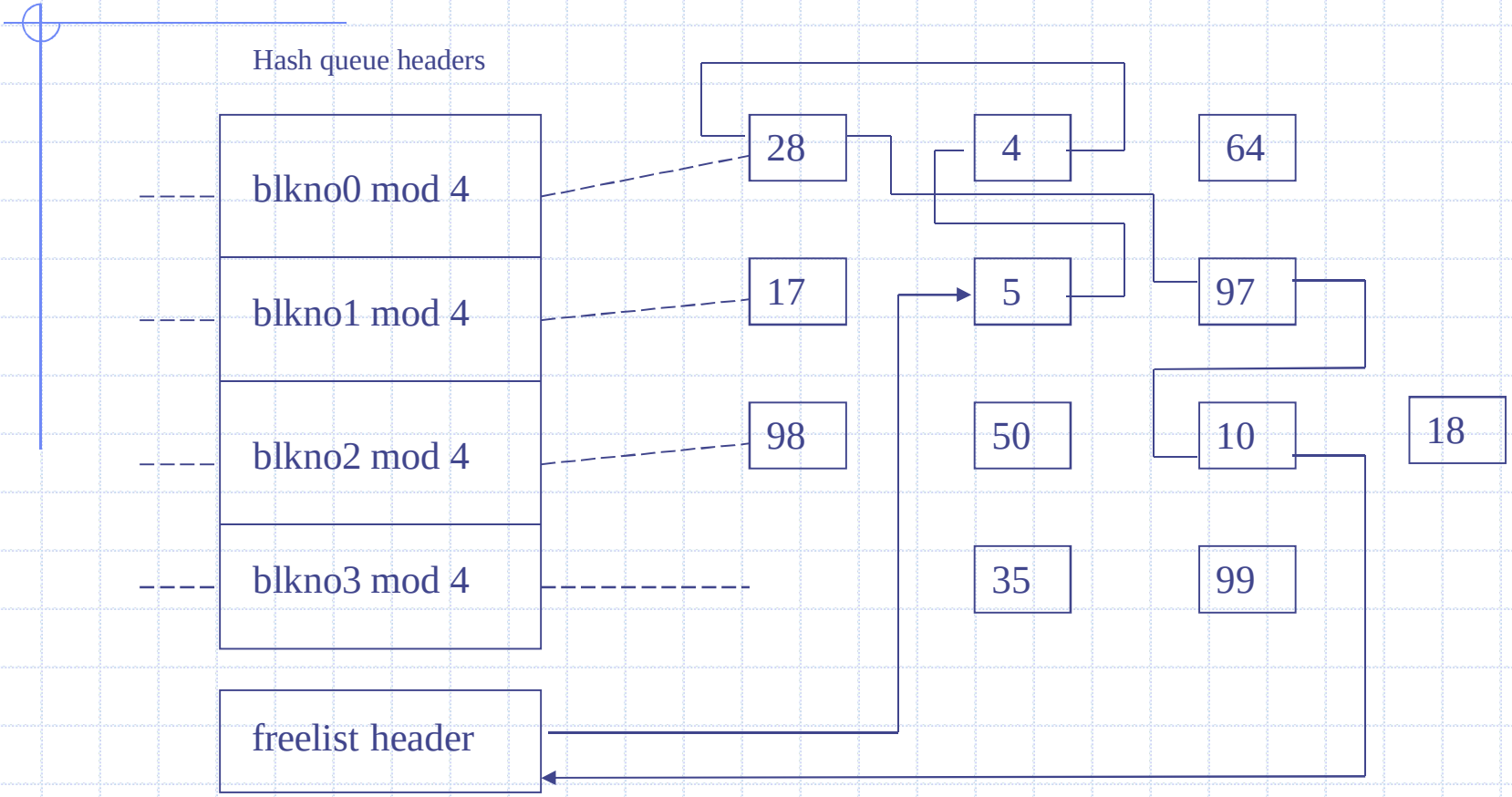
Retrieval of a Buffer: 2nd Scenario (a)

- The kernel cannot find the block on the hash queue, so it allocates a buffer from free list



Search for block 18: Not in cache

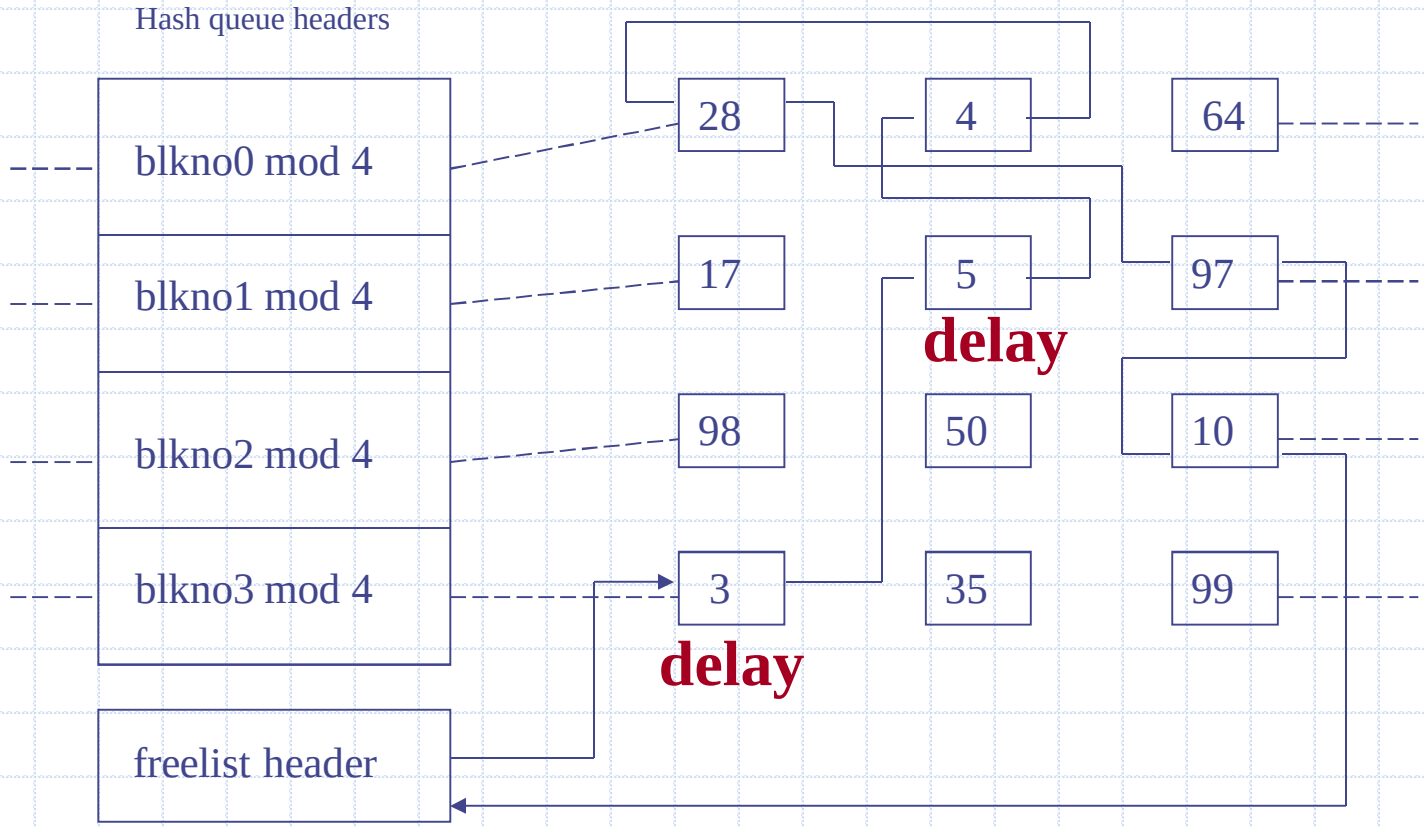
Retrieval of a Buffer: 2nd Scenario (b)



Remove 1st block from free list: Assign to 18

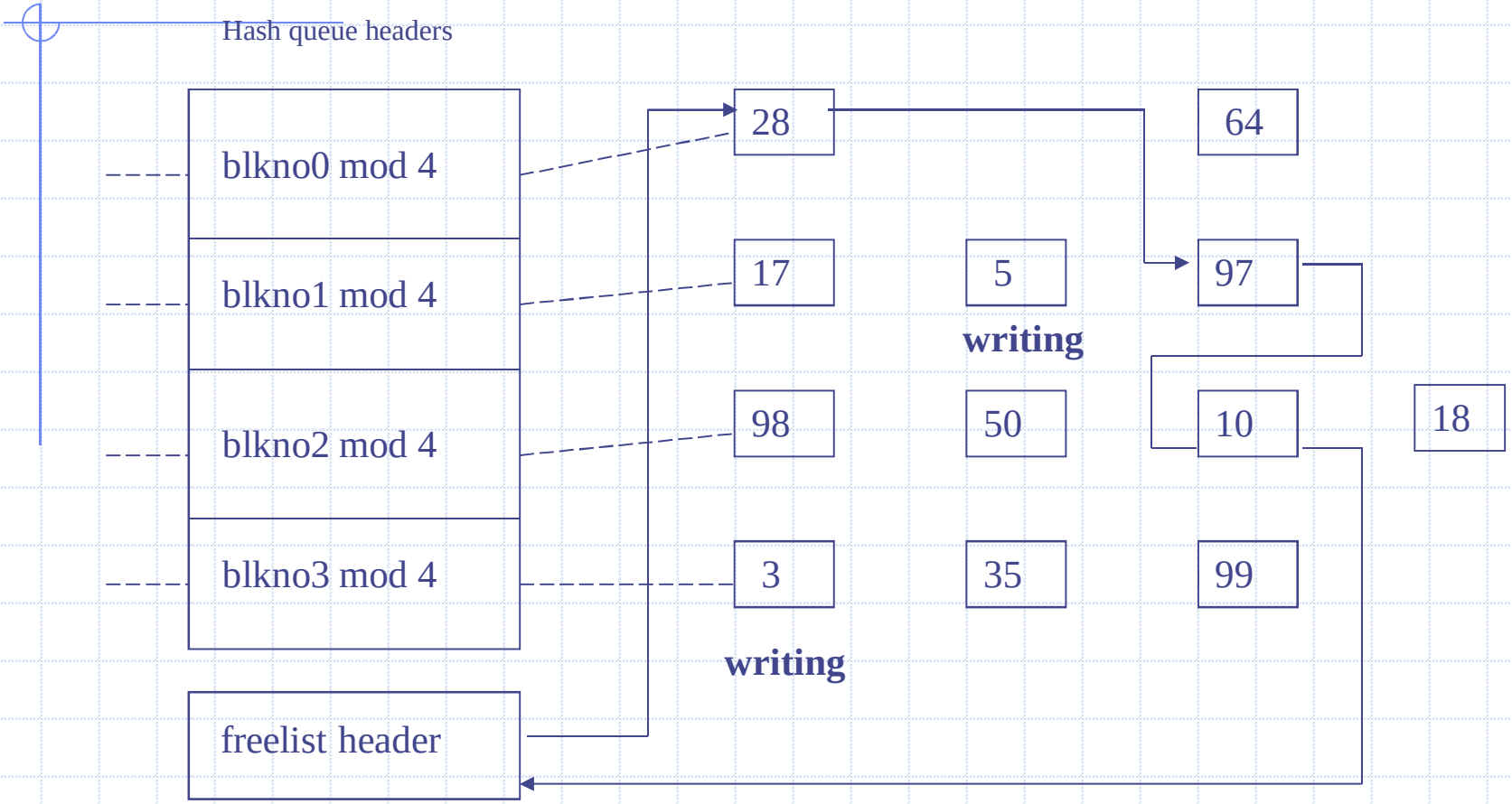
Retrieval of a Buffer: 3rd Scenario (a)

- The kernel cannot find the block on the hash queue, and finds delayed write buffers on hash queue



Search for block 18, Delayed write blocks on free list

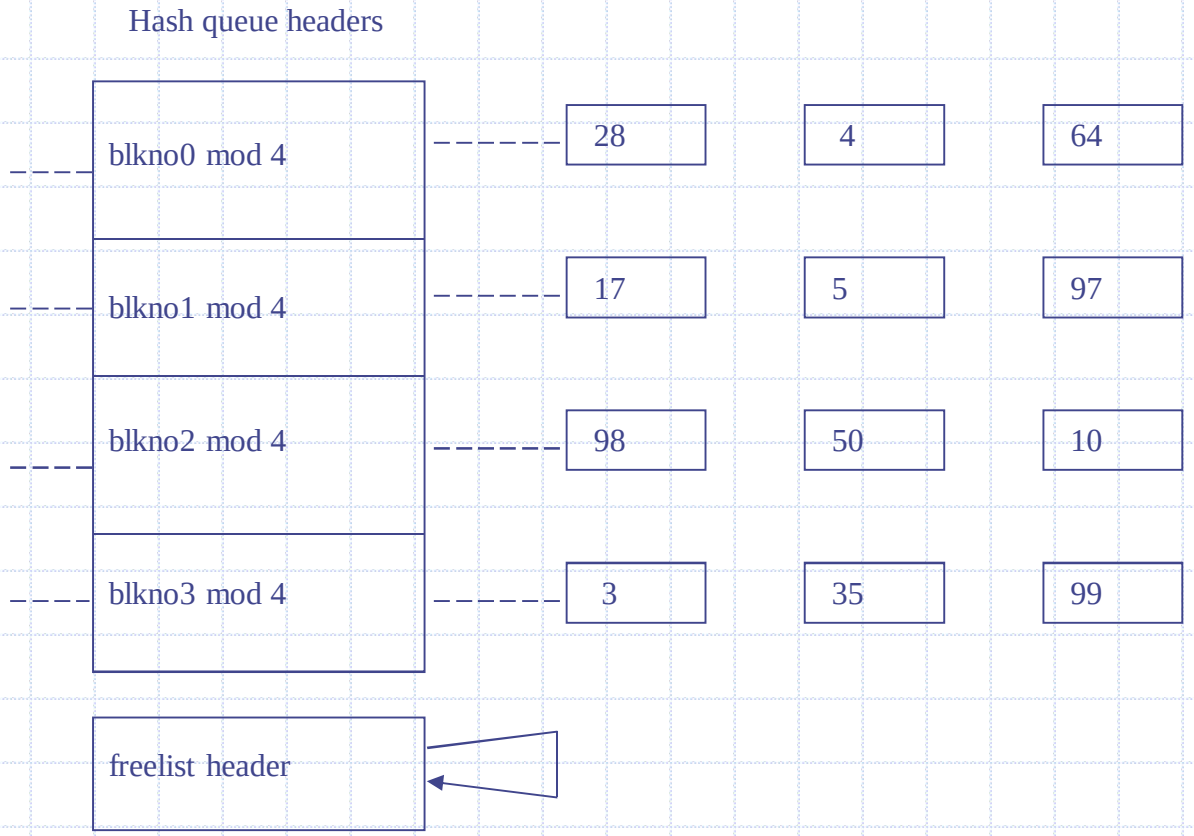
Retrieval of a Buffer: 3rd Scenario (b)



(b) Writing Blocks 3, 5, Reassign 4 to 18
Figure 3.8

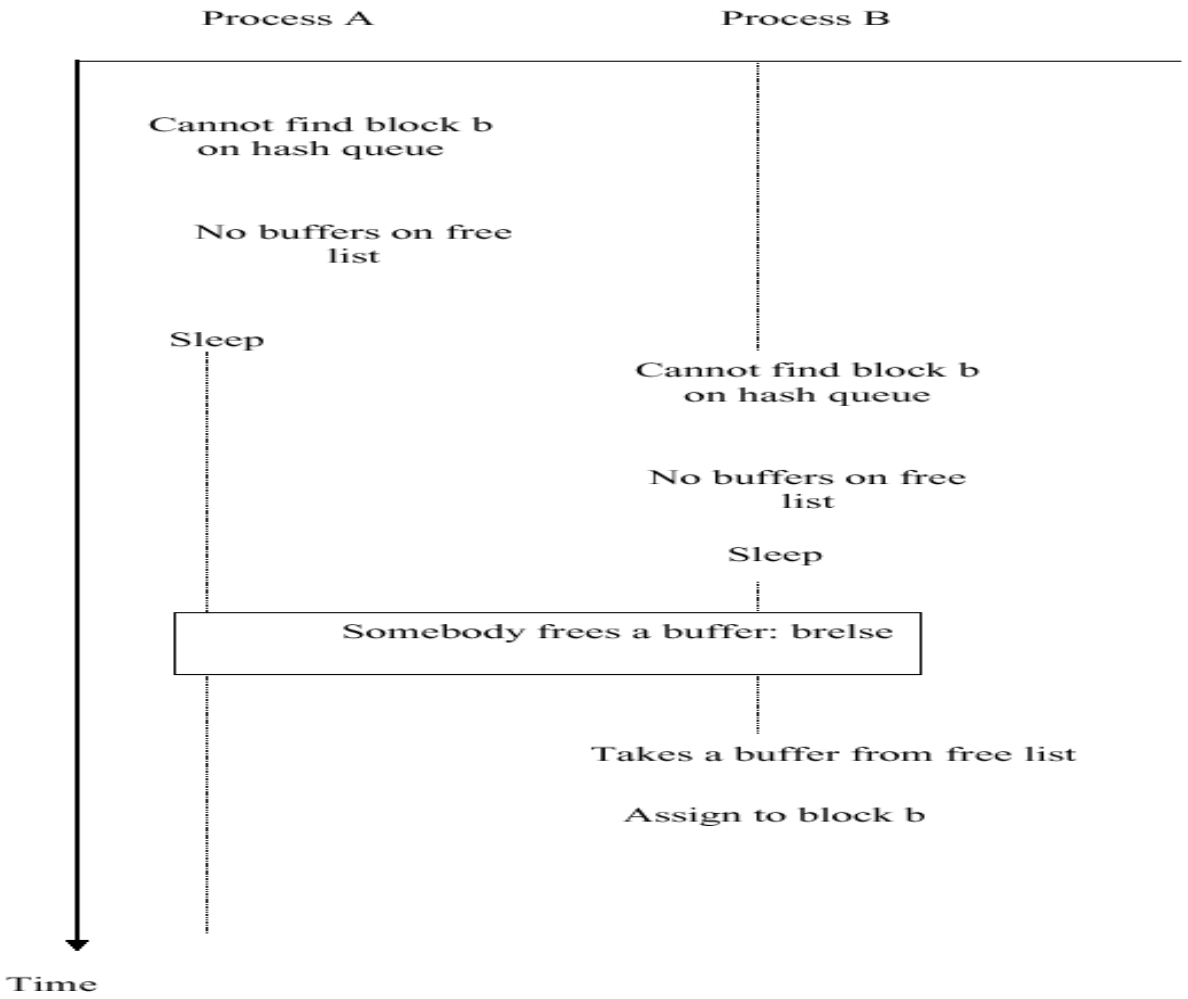
Retrieval of a Buffer: 4th Scenario

- The kernel cannot find the buffer on the hash queue, and the free list is empty



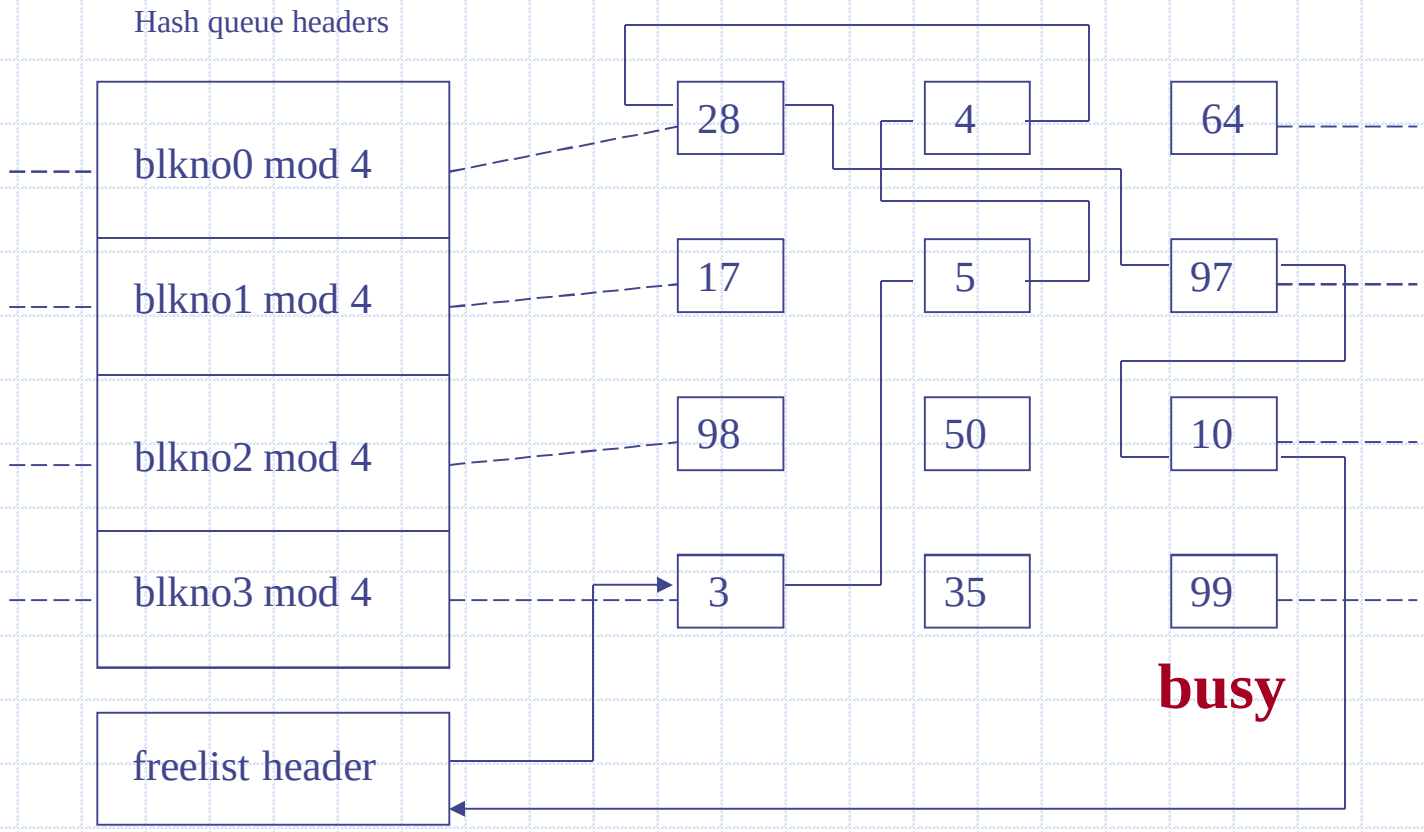
Search for block 18, free list empty

Race for free Buffer



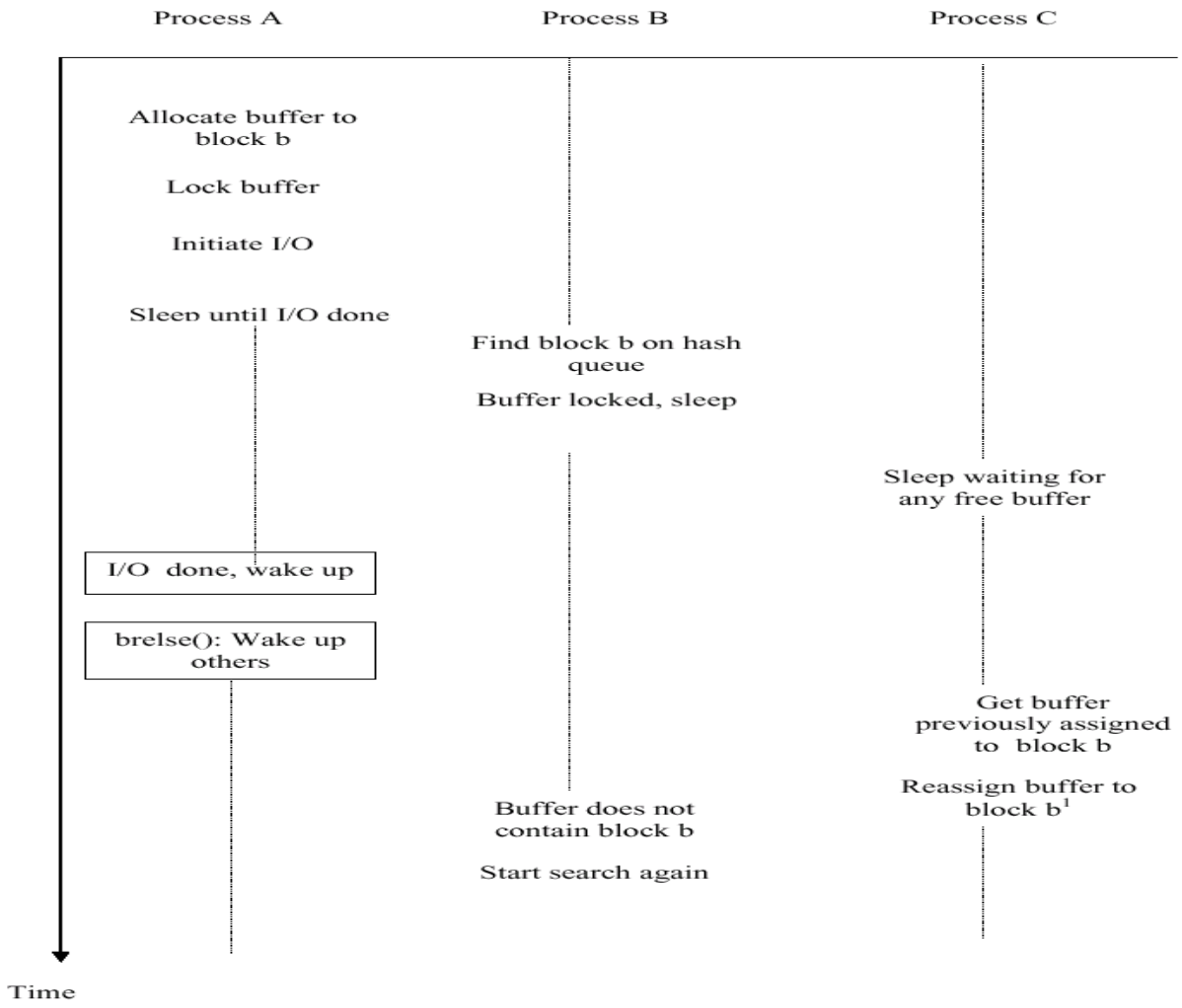
Retrieval of a Buffer: 5th Scenario

- Kernel finds the buffer on hash queue, but it is currently busy



Search for block 99, block busy

Race for a Locked Buffer



Algorithm: GetBlock

- GetBlock (file_system_no,block_no)
  - while (buffer not found)
    - if (buffer in hash queue)
      - if (buffer busy)
        - sleep (event buffer becomes free)
        - continue
      - mark buffer busy
      - remove buffer from free list
      - return buffer
    - else
      - if (there is no buffer on free list)
        - sleep (event any buffer becomes free)
        - continue
      - remove buffer from free list
      - if (buffer marked as delayed write)
        - asyschronous write buffer to disk
        - continue
      - remove buffer from old hash queue
      - put buffer onto new hash queue
      - return buffer

Reading Disk Blocks

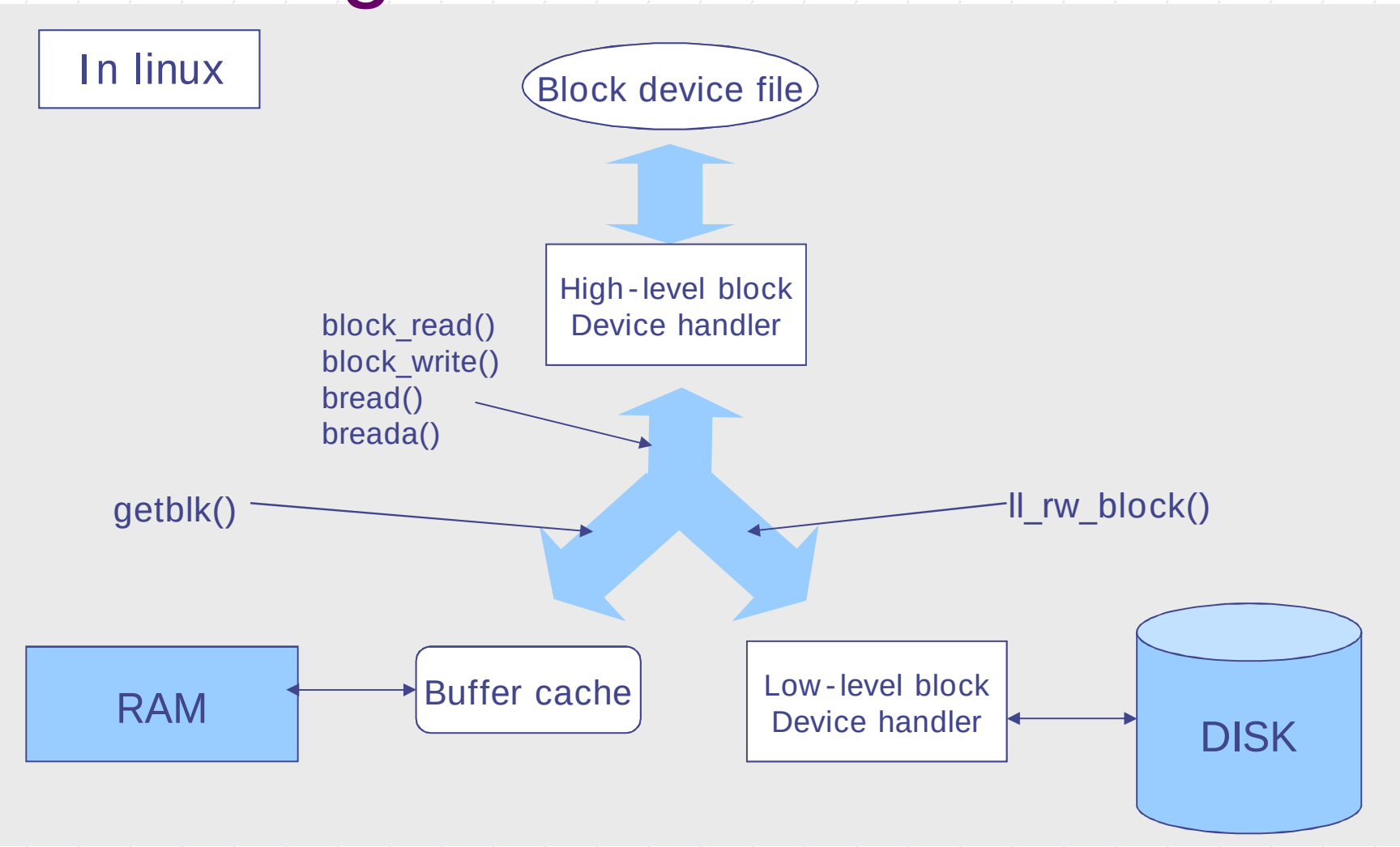


Figure 13-3 block device handler architecture for buffer I/O operation in Understanding the Linux Kernel

Algorithm :Reading a disk block

- Algorithm bread /* block read */
  - Input: file system 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);

- If it is not in cache, the kernel calls the disk driver to schedule a read request.
- The disk driver notifies the disk controller later transmits the data to the buffer.
- Disk interrupt handler awakens the sleeping process.

Reading Disk Blocks

- Read Ahead
  - Improving performance
    - Read additional block before request
  - Use breada()

Algorithm

```
Algorithm breada
Input: (1) file system block number for immediate read
       (2) file system block number for asynchronous read
Output: buffer containing data for immediate read
{
  if (first block not in cache){
    get buffer for first block(algorithm getblk);
    if(buffer data not valid)
        initiate disk read;
  }
```

Reading disk Block

Algorithm-cont

```
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 cache)
      {
          read first block(algorithm bread)
          return buffer;
       }
       sleep(event first buffer contains valid data);
       return buffer;
}
```

Writing disk Block

- Synchronous write
  - the calling process goes the sleep awaiting I/O completion and releases the buffer when awakens.
- Asynchronous write
  - the kernel starts the disk write. The kernel release the buffer when the I/O completes
- Delayed write
  - The kernel put off the physical write to disk until buffer reallocated
  - Look Scenario 3
- Relese
  - Use brelse()

Writing Disk Block

algorithm

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

Release Disk Block

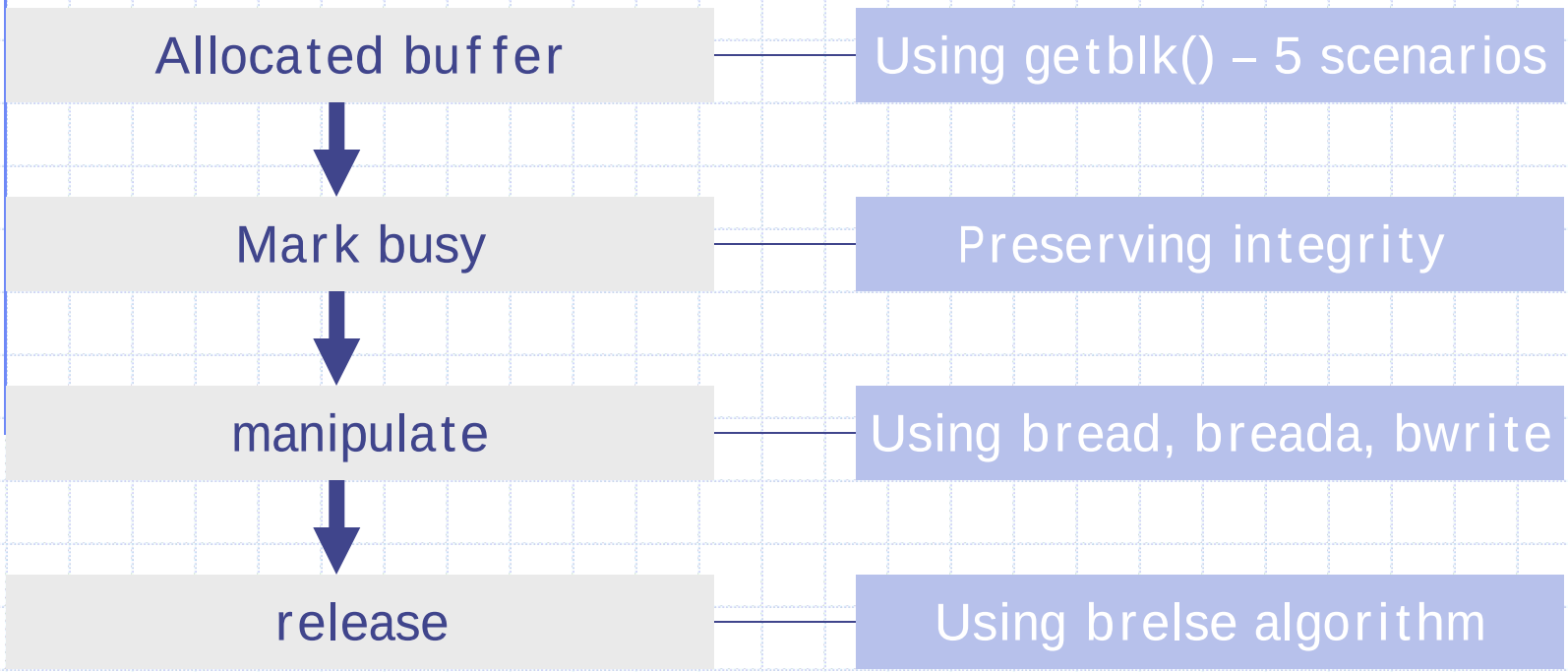
algorithm

```
Algorithm brelse
Input: locked buffer
Output: none
{
   wakeup all process; event,
          waiting for any buffer to become free;
   wakeup all process; 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);
}
```

Advantages and Disadvantages

- Advantages
  - Allows uniform disk access
  - Eliminates the need for special alignment of user buffers
    - by copying data from user buffers to system buffers,
  - Reduce the amount of disk traffic
    - less disk access
  - Insure file system integrity
    - one disk block is in only one buffer
- Disadvantages
  - Can be vulnerable to crashes
    - When delayed write
  - requires an extra data copy
    - When reading and writing to and from user processes

What happen to buffer until now



Reference

- LINUX KERNEL INTERNALS
  - Beck, Bohme, Dziadzka, Kunitz, Magnus, Verworner
- The Design of the Unix operating system
  - Maurice j.bach
- Understanding the LINUX KERNEL
  - Bovet, cesati
- In linux
  - Buffer_head : include/linux/fs.h
  - Bread : fs/buffer.c
  - Brelse : include/linux/fs.h