12 Block Device I/O and Buffer Management

Abstract

This chapter covers block device I/O and buffer management. It explains the principles of block device I/O and advantages of I/O buffering. It discusses the buffer management algorithm of Unix and points out its shortcomings. Then it uses semaphores to design new buffer management algorithms to improve the efficiency and performance of the I/O buffer cache. It is shown that the simple PV algorithm is easy to implement, has good cache effect and is free of deadlock and starvation. It presents a programming project to compare the performances of the Unix buffer management algorithm and the PV algorithm. The programming project should also help the reader to better understand I/O operations in file systems.

12.1 Block Device I/O Buffers

In Chap. 11, we showed the algorithms of read/write regular files. The algorithms rely on two key operations, get_block and put_block, which read/write a disk block to/from a buffer in memory. Since disk I/O are slow in comparison with memory access, it is undesirable to do disk I/O on every read/write file operation. For this reason, most file systems use I/O buffering to reduce the number of physical I/O to/from storage devices. A well designed I/O buffering scheme can significantly improve file I/O efficiency and increase system throughput.

The basic principle of I/O buffering is very simple. The file system uses a set of I/O buffers as a cache memory for block devices. When a process tries to read a disk block identified by (dev, blk), it first searches the buffer cache for a buffer already assigned to the disk block. If such a buffer exists and contains valid data, it simply reads from the buffer without reading the block from disk again. If such a buffer does not exist, it allocates a buffer for the disk block, reads data from disk into the buffer, then reads data from the buffer. Once a block is read in, the buffer will be kept in the buffer cache for the next read/write requests of the same block by any process. Similarly, when a process writes to a disk block, it first gets a buffer assigned to the block. Then it writes data to the buffer, marks the buffer as dirty for delay write and releases it to the buffer cache. Since a dirty buffer contains valid data, it can be used to satisfy subsequent read/write requests for the same block without incurring real disk I/O. Dirty buffers will be written to disk only when they are to be reassigned to different blocks.

Before discussing buffer management algorithms, we first introduce the following terms. In read_file/write_file, we have assumed that they read/write from/to a dedicated buffer in memory. With I/O buffering, the buffer will be allocated dynamically from a buffer cache. Assume that BUFFER is the structure type of buffers (defined below) and getblk(dev, blk) allocates a buffer assigned to (dev, blk) from the buffer cache. Define a bread(dev, blk) function, which returns a buffer (pointer) containing valid data.

```
BUFFER *bread(dev,blk) // return a buffer containing valid data
{
   BUFFER *bp = getblk(dev,blk); // get a buffer for (dev,blk)
   if (bp data valid)
      return bp;
   bp->opcode = READ;            // issue READ operation
   start_io(bp);                 // start I/O on device
   wait for I/O completion;
   return bp;
}
```

After reading data from a buffer, the process releases the buffer back to the buffer cache by brelse(bp). Similarly, define a write_block(dev, blk, data) function as

```
write_block(dev, blk, data)      // write data from U space
{
   BUFFER *bp = bread(dev,blk);  // read in the disk block first
   write data to bp;
   (synchronous write)? bwrite(bp) : dwrite(bp);
}
```

where bwrite(bp) is for synchronous write and dwrite(bp) is for delay-write, as shown below.

```
------------------------------------------------------------------
bwrite(BUFFER *bp){              | dwrite(BUFFER *bp){
   bp->opcode = WRITE;           |   mark bp dirty for delay_write;
   start_io(bp);                 |   brelse(bp); // release bp
   wait for I/O completion;      | }
   brelse(bp); // release bp     |
}                                |
------------------------------------------------------------------
```

Synchronous write waits for the write operation to complete. It is used for sequential or removable block devices, e.g. USB drives. For random access devices, e.g. hard disks, all writes can be delay writes. In delay write, dwrite(bp) marks the buffer as dirty and releases it to the buffer cache. Since dirty buffers contain valid data, they can be used to satisfy subsequent read/write requests of the same block. This not only reduces the number of physical disk I/O but also improves the buffer cache effect. A dirty buffer will be written to disk only when it is to be reassigned to a different disk block, at which time the buffer is written out by

```
awrite(BUFFER *bp)
{
    bp->opcode = ASYNC;     // for ASYNC write;
    start_io(bp);
 }
```

awrite() calls start_io() to start I/O operation on the buffer but does not wait for the operation to complete. When an ASYNC write operation completes, the disk interrupt handler will release the buffer.

Physical block device I/O: Each device has an I/O queue which contains buffers of pending I/O. The start_io() operation on a buffer is

```
start_io(BUFFER *bp)
{
    enter bp into device I/O queue;
    if (bp is first buffer in I/O queue)
       issue I/O command for bp to device;
 }
```

When an I/O operation completes, the device interrupt handler finishes the I/O operation on the current buffer and starts I/O for the next buffer in the I/O queue if it is non-empty. The algorithm of the device interrupt handler is

```
InterruptHandler()
{
   bp = dequeue(device I/O queue); // bp = remove head of I/O queue
  (bp->opcode == ASYNC)? brelse(bp) : unblock process on bp;
   if (!empty(device I/O queue))
      issue I/O command for first bp in I/O queue;
}
```

12.2 Unix I/O Buffer Management Algorithm

Unix I/O buffer management algorithm first appeared in V6 Unix (Ritchie and Thompson 1978; Lion 1996). It is discussed in detail in Chapter 3 of Bach (Bach 1990). The Unix buffer management subsystem consists of the following components.

(1). I/O buffers: A set of NBUF buffers in kernel is used as a buffer cache. Each buffer is represented by a structure.

```
typdef struct buf{
  struct buf *next_free;     // freelist pointer
  struct buf *next_dev;      // dev_list pointer
  int dev,blk;               // assigned disk block;
  int opcode;                 // READ|WRITE
  int dirty;                  // buffer data modified
  int async;                  // ASYNC write flag
  int valid;                  // buffer data valid
  int busy;                   // buffer is in use
  int wanted;                 // some process needs this buffer
  struct semaphore lock=1;    // buffer locking semaphore; value=1
  struct semaphore iodone=0;  // for process to wait for I/O completion;
  char buf[BLKSIZE];          // block data area
} BUFFER;
BUFFER buf[NBUF], *freelist; // NBUF buffers and free buffer list
```

The buffer structure consists of two parts; a header part for buffer management and a data part for a block of data. To conserve kernel memory, the status fields may be defined as a bit vector, in which each bit represents a unique status condition. They are defined as int here for clarity and ease of discussion.

(2). Device Table: Each block device is represented by a device table structure.

```
struct devtab{
  u16  dev;              // major device number
  BUFFER *dev_list;      // device buffer list
  BUFFER *io_queue;      // device I/O queue
} devtab[NDEV];
```

Each devtab has a dev_list, which contains I/O buffers currently assigned to the device, and an io_queue, which contains buffers of pending I/O operations on the device. The I/O queue may be organized for optimal I/O operations. For instance, it may implement the various disk scheduling algorithms, such as the elevator algorithm or the linear-sweep algorithm, etc. For the sake of simplicity, Unix uses FIFO I/O queues.

(3). Buffer Initialization: When the system starts, all I/O buffers are in the freelist and all device lists and I/O queues are empty.

(4). Buffer Lists: When a buffer is assigned to a (dev, blk), it is inserted into the devtab's dev_list. If the buffer is currently in use, it is marked as BUSY and removed from the freelist. A BUSY buffer may also be in the I/O queue of a devtab. Since a buffer cannot be free and busy at the same time, the device I/O queue is maintained by using the same next_free pointer. When a buffer is no longer BUSY, it is released back to the freelist but remains in the dev_list for possible reuse. A buffer may change from one dev_list to another only when it is reassigned. As shown before, read/write disk blocks can be expressed in terms of bread, bwrite and dwrite, all of which depend on getblk and brelse. Therefore, getblk and brelse form the core of the Unix buffer management scheme. The algorithm of getblk and brelse is as follows.

(5). Unix getblk/brelse algorithm: (Lion 1996; Chapter 3 of (Bach 1990)).

```
/* getblk: return a buffer=(dev,blk) for exclusive use */
BUFFER *getblk(dev,blk){
   while(1){
     (1). search dev_list for a bp=(dev, blk);
```

```
        (2). if (bp in dev_lst){
                 if (bp BUSY){
                    set bp WANTED flag;
                    sleep(bp);      // wait for bp to be released
                    continue;       // retry the algorithm
                 }
                 /* bp not BUSY */
                 take bp out of freelist;
                 mark bp BUSY;
                 return bp;
               }
        (3). /* bp not in cache; try to get a free buf from freelist */
             if (freelist empty){
                 set freelist WANTED flag;
                 sleep(freelist); // wait for any free buffer
                 continue;        // retry the algorithm
             }
        (4). /* freelist not empty */
             bp = first bp taken out of freelist;
             mark bp BUSY;
             if (bp DIRTY){       // bp is for delayed write
                 awrite(bp);      // write bp out ASYNC;
                 continue;        // from (1) but not retry
             }
        (5). reassign bp to (dev,blk); // set bp data invalid, etc.
             return bp;
}

/** brelse: releases a buffer as FREE to freelist **/
brelse(BUFFER *bp){
  if (bp WANTED)
     wakeup(bp);       // wakeup ALL proc's sleeping on bp;
  if (freelist WANTED)
     wakeup(freelist); // wakeup ALL proc's sleeping on freelist;
  clear bp and freelist WANTED flags;
  insert bp to (tail of) freelist;
}
```

It is noted that in (Bach 1990), buffers are maintained in hash queues. When the number of buffers is large, hashing may reduce the search time. If the number of buffers is small, hashing may actually increases the execution time due to additional overhead. Furthermore, studies (Wang 2002) have shown that hashing has almost no effect on the buffer cache performance. In fact, we may consider the device lists as hash queues by the simple hashing function hash(dev, blk) = dev. So there is no loss of generality by using the device lists as hash queues. The Unix algorithm is very simple and easy to understand. Perhaps because of its extreme simplicity, most people are not very impressed by it at first sight. Some may even consider it naive because of the repeated retry loops. However, the more you look at it, the more it makes sense. This amazingly simple but effective algorithm attests to the ingenuity of the original Unix designers. Some specific comments about the Unix algorithm follow.

(1). Data Consistency: In order to ensure data consistency, getblk must never assign more than one buffer to the same (dev, blk). This is achieved by having the process re-execute the "retry loops" after waking up from sleep. The reader may verify that every assigned buffer is unique. Second, dirty buffers are written out before they are reassigned, which guarantees data consistency.
(2). Cache effect: Cache effect is achieved by the following means. A released buffer remains in the device list for possible reuse. Buffers marked for delay-write do not incur immediate I/O and are available for reuse. Buffers are released to the tail of freelist but allocated from the front of freelist. This is based on the LRU ((Least-Recent-Used) principle, which helps prolong the lifetime of assigned buffers, thereby increasing their cache effect.
(3). Critical Regions: Device interrupt handlers may manipulate the buffer lists, e.g. remove a bp from a devtab's I/O queue, change its status and call brelse(bp). So in getblk and brelse, device interrupts are masked out in these critical regions. These are implied but not shown in the algorithm.

12.2.1 Shortcomings of Unix Algorithm

The Unix algorithm is very simple and elegant, but it also has the following shortcomings.

(1). Inefficiency: the algorithm relies on retry loops. For example, releasing a buffer may wake up two sets of processes; those who want the released buffer, as well as those who just need a free buffer. Since only one process can get the released buffer, all other awakened processes must go back to sleep again. After waking up from sleep, every awakened process must re-execute the algorithm again from beginning because the needed buffer may already exist. This would cause excessive process switches.
(2). Unpredictable cache effect: In the Unix algorithm, every released buffer is up for grabs. If the buffer is obtained by a process which needs a free buffer, the buffer would be reassigned, even though there may be processes which still need the buffer.
(3). Possible starvation: The Unix algorithm is based on the principle of "free economy", in which every process is given chances to try but with no guarantee of success. Therefore, process starvation may occur.
(4). The algorithm uses sleep/wakeup, which is only suitable for uniprocessor systems.

12.3 New I/O Buffer Management Algorithm

In this section, we shall show a new algorithm for I/O buffer management. Instead of using sleep/ wakeup, we shall use P/V on semaphores for process synchronization. The main advantages of semaphores over sleep/wakeup are

(1). Counting semaphores can be used to represent the number of available resources, e.g. the number of free buffers.
(2). When many processes wait for a resource, the V operation on a semaphore unblocks only one waiting process, which does not have to retry since it is guaranteed to have the resource.

These semaphore properties can be used to design more efficient algorithms for buffer management. Formally, we specify the problem as follows.

12.3.1 Buffer Management Algorithm using Semaphores

Assume a uniprocessor kernel (one process runs at a time). Use P/V on counting semaphores to design new buffer management algorithms which meet the following requirements:

(1). Guarantee data consistency.
(2). Good cache effect.
(3). High efficiency: No retry loops and no unnecessary process "wakeups".
(4). Free of deadlocks and starvation.

It is noted that merely replacing sleep/wakeup in the Unix algorithm by P/V on semaphores is not an acceptable solution because doing so would retain all the retry loops. We must redesign the algorithm to meet all the above requirements and justify that the new algorithm is indeed better than the Unix algorithm. First, we define the following semaphores.

```
BUFFER buf[NBUF];       // NBUF I/O buffers
SEMAPHORE free = NBUF;  // counting semaphore for FREE buffers
SEMAPHORE buf[i].sem = 1; // each buffer has a lock sem=1;
```

To simplify the notations, we shall refer to the semaphore of each buffer by the buffer itself. As in the Unix algorithm, initially all buffers are in the freelist and all device lists and I/O queues are empty. The following shows a simple buffer management algorithm using semaphores.

12.4 PV Algorithm

```
BUFFER *getblk(dev, blk)
{
     while(1){
 (1).   P(free);           // get a free buffer first
 (2).   if (bp in dev_list){
 (3).       if (bp not BUSY){
                remove bp from freelist;
                P(bp);     // lock bp but does not wait
                return bp;
            }
            // bp in cache but BUSY
            V(free);       // give up the free buffer
 (4).       P(bp);         // wait in bp queue
            return bp;
        }
        // bp not in cache, try to create a bp=(dev, blk)
```

```
   (5).   bp = frist buffer taken out of freelist;
          P(bp);            // lock bp, no wait
   (6).   if (bp dirty){
              awrite(bp);   // write bp out ASYNC, no wait
              continue;     // continue from (1)
          }
   (7).   reassign bp to (dev,blk); // mark bp data invalid, not dirty
          return bp;
      }                     // end of while(1)
  }

  brelse(BUFFER *bp)
  {
   (8). if (bp queue has waiter){ V(bp); return; }
   (9). if (bp dirty && free queue has waiter){ awrite(bp); return; }
  (10). enter bp into (tail of) freelist; V(bp); V(free);
  }
```

Next, we show that the PV algorithm is correct and meets the requirements.

1. **Buffer uniqueness:** In getblk(), if there are free buffers, the process does not wait at (1). Then it searches the dev_list. If the needed buffer already exits, the process does not create the same buffer again. If the needed buffer does not exist, the process creates the needed buffer by using a free buffer, which is guaranteed to have. If there are no free buffers, it is possible that several processes, all of which need the same buffer, are blocked at (1). When a free buffer is released at (10), it unblocks only one process to create the needed buffer. Once a buffer is created, it will be in the dev_list, which prevents other processes from creating the same buffer again. Therefore, every assigned buffer is unique.
2. **No retry loops:** The only place a process re-executes the while(1) loop is at (6), but that is not a retry because the process is continually executing.
3. **No unnecessary wakeups:** In getblk(), a process may wait for a free buffer at (1) or a needed buffer at (4). In either case, the process is not woken up to run again until it has a buffer. Furthermore, at (9), when a dirty buffer is to be released as free and there are waiters for free buffers at (1), the buffer is not released but written out directly. This avoids an unnecessary process wakeup.
4. **Cache effect:** In the Unix algorithm, every released buffer is up for grabs. In the new algorithm, a buffer with waiters is always kept for reuse. A buffer is released as free only if it has no waiters. This should improve the buffer's cache effect.
5. **No deadlocks and starvation:** In getblk(), the semaphore locking order is always unidirectional, i.e. P(free), then P(bp), but never the other way around, so deadlock cannot occur. If there are no free buffers, all requesting processes will be blocked at (1). This implies that while there are processes waiting for free buffer, all buffers in use cannot admit any new users. This guarantees that a BUSY buffer will eventually be released as free. Therefore, starvation for free buffers cannot occur.

Although we have shown that the new algorithm is correct, whether it can perform better than the Unix algorithm remains an open question, which can only be answered by real quantitative data. For this purpose, we have designed a programming project to compare the performances of the buffer management algorithms. The programming project should also help the reader to better understand I/O operations in file systems.

12.5 Programming Project: Comparison of I/O Buffer Management Algorithms

The programming project is to implement a simulation system to compare the performances of the Unix I/O buffer management algorithms and the PV algorithm using semaphores. Although the immediate goal is I/O buffer performance, the real goal is for the reader to understand I/O operations of file systems. The following describes the project.

12.5.1 System Organization

Figure 12.1 shows the organization of the simulation system.

(1). Box#1: User Interface: This is the user interface part of the simulation system. It prompts for input commands, show command executions, displays system status and execution results, etc. During development, the reader may input commands manually for tasks to execute. During final testing, tasks should have their own sequence of input commands. For example, each task may read an input file containing commands. After a task switch, task inputs will be read from a different file.

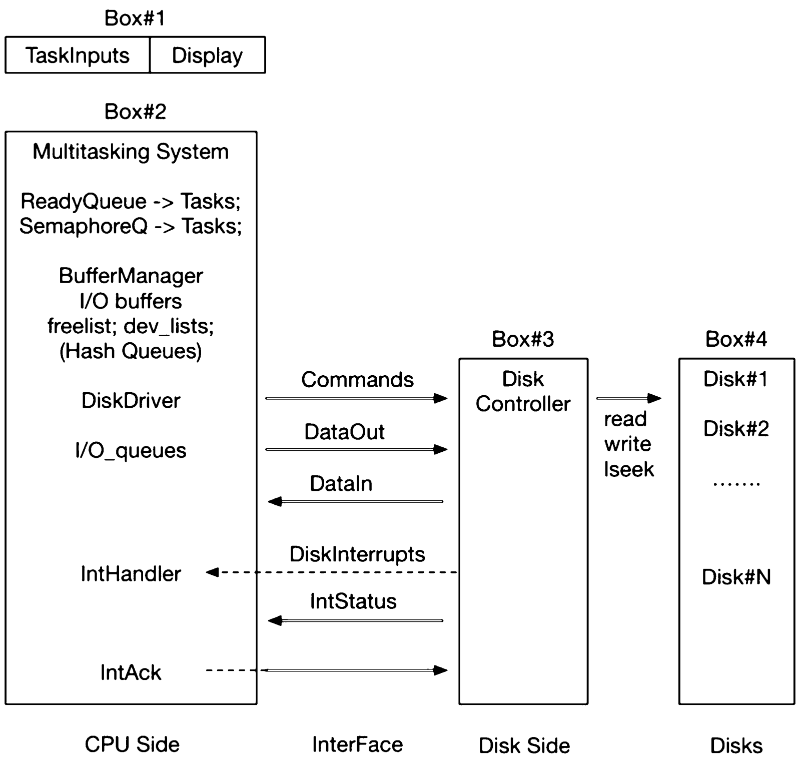


Fig. 12.1 System Organization Diagram

12.5.2 Multitasking System

(2). Box#2: This is the CPU side of a multitasking system, which simulates the kernel mode of a uniprocessor (single CPU) file system. It is essentially the same multitasking system described in Chap. 4 for user-level threads, except for the following modifications. When the system starts, it creates and runs a main task with the lowest priority, but it creates ntask working tasks, all with priority 1, and enters them into the readyQueue. Then the main task executes the following code, which switches task to run working tasks from the readyQueue.

```
/******* Main Task Code *******/
// after system initialization
while(1){
  while(task && readyQ == 0);  // loop if no task runnable
  if (readyQ)                  // if readyQueue nonempty
     kswitch();                // switch to run a working task
  else
     end_task();               // all working tasks have ended
}
```

Since the main task has the lowest priority, it will run again if there are no runnable tasks or if all tasks have ended. In the latter case, the main task execute end_task(), in which it collects and displays the simulation results and terminates, thus ending the simulation run.

All working tasks execute the same body() function, in which each task reads commands from an input file to execute read or write disk block operations until end of the command file.

```
#define CMDLEN 10
int cmdfile[NTASK]; // opened command file descriptors
int task = ntask;   // number of active tasks
int body()
{
   int dev, blk;
   char opcode, cmd[CMDLEN];
   while(1){
      if (read(cmdfile[running->pid], cmd, CMDLEN)==0){
          running->status = DEAD; // task ends
          task--;                 // dec task count by 1
          tswtich();
      }
      sscanf(cmd, "%c%4d%5d", &opcode, &dev, &blk);
      if (opcode=='r')         // read (dev, blk)
          readBlk(dev, blk);
      if (opcode=='w')
          writeBlk(dev, blk);  // write (dev, blk)
   }
}
```

The commands of each task are generated randomly by the rand() function modulo a limit value. Each command is a triple of the form

```
    [r xxx yyyy] or [w xxx yyyy], where xxx=dev, yyyy=blkno;
```

For a [r dev blk] command, the task calls

```
BUFFER *bread(dev, blk)
```

to get a buffer (pointer) containing valid data. After reading data from the buffer, it releases the buffer back to the buffer cache for reuse. In bread(), the task may wait for a buffer in getblk() or until the buffer data become valid. If so, it sleeps (in Unix algorithm) or becomes blocked (in PV algorithm), which switches task to run the next task from the readyQueue.

For a [w dev blk] command, the task tries to get a buffer for (dev, blk). If it has to wait for a buffer in getblk(), it will sleep or become blocked and switch task to run the next task. After writing data to a buffer, it marks the buffer as DIRTY for delayed write and releases the buffer to the buffer cache. Then it tries to execute the next command, etc.

12.5.3 Buffer Manager

The Buffer Manager implements buffer management functions, which consist of

```
BUFFER *bread(dev, blk)
        dwrite(BUFFER *bp)
        awrite(BUFFER *bp)
BUFFER *getblk(dev, blk)
        brelse(BUFFER *bp)
```

These functions may invoke the Disk Driver to issue physical disk I/O.

12.5.4 Disk Driver

The Disk Driver consists of two parts:

(1). **start_io():** maintain device I/O queues and issue I/O operation for buffers in I/O queue.
(2). **Interrupt Handler:** At the end of each I/O operation the Disk Controller interrupts the CPU. Upon receiving an interrupt, the Interrupt Handler first reads the Interrupt Status from IntStatus, which contains

```
[dev  | R|W |  StatusCode]
|<----- e.g.10 chars---->|
```

where dev identifies the device. The Interrupt Handler removes the buffer from the head of the device I/O queue and processes the interrupt for the buffer. For a disk READ interrupt, the Interrupt Handler first moves the block of data from DataIn into the buffer's data area. Then it marks the buffer data valid and wakes up or unblocks the task that's waiting on the buffer for valid data. The awakened process will read data from the buffer and release the buffer back to the buffer cache. For a disk WRITE interrupt, there is no process waiting on the buffer for I/O completion. Such an ASYNC write buffer is handled by the Interrupt handler. It turns off the buffer's ASYNC write flag and releases the buffer to the buffer cache. Then, it examines the device I/O queue. If the I/O queue is nonempty, it issues I/O commands for the first buffer in the I/O queue. Before issuing a disk WRITE operation, it copies the buffer's data to DataOut for the Disk Controller to get. Finally, it writes an ACK to IntAck to acknowledge the current interrupt, allowing the Disk Controller to continue When the Interrupt Handler finishes processing the current interrupt, normal task execution resumes from the last interrupted point.

12.5.5 Disk Controller

(3). Box#3: This is the Disk Controller, which is a **child process** of the **main process**. As such, it operates independently with the CPU side except for the communication channels between them, which are shown as the **Interface** between CPU and Disk Controller. The communication channels are implemented by pipes between the main process and the child process.

Commands:	I/O commands from CPU to Disk Controller
DataOut :	data out from CPU to Disk Controller in write operations
DataIn :	data in from Disk Controller to CPU in read operations
IntStatus :	interrupt status from Disk Controller to CPU
IntAck :	interrupt ACK from CPU to Disk Controller

12.5.6 Disk Interrupts

Interrupts from Disk Controller to CPU are implemented by the SIGUSR1 (#10) signal. At the end of each I/O operation, the Disk Controller issues a kill(ppid, SIGUSR1) system call to send a SIGUSR1 signal to the parent process, which acts as an interrupt to the virtual CPU. As usual, the virtual CPU may mask out/in disk interrupts (signals) in Critical Regions. To prevent race condition, the Disk Controller must receive an interrupt ACK from the CPU before it can interrupt again.

12.5.7 Virtual Disks

(4). Box#4: These are virtual disks, which are simulated by Linux files. Using the Linux system calls lseek(), read() and write(), we may support any block I/O operations on the virtual disks. For simplicity, the disk block size is set to 16 bytes. Since the data contents do not matter, they can be set to a fixed sequence of 16 chars.

12.5.8 Project Requirements

Implement two versions of buffer management algorithms:

```
Unix algorithm using sleep/wakeup
New algorithm using semaphores.
```

Compare their performances in terms of buffer cache hit-ratio, numbers of actual I/O operations, task switches, retries and total run time, etc. under different task/buffer ratios.

12.5.9 Sample Base Code

For the sake of brevity, we only show the main.c file of the CPU side and the controller.c file of the Disk Controller. The functions of other included files are only explained briefly. Also, not shown are the code segments used to collect performance statistics, but they can be added easily to the base code.

```
/************ CPU side: main.c file **********/
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <signal.h>
#include <string.h>
#include "main.h"              // constants, PROC struct pipes
struct itimerval itimer;       // for timer interrupt
struct timeval tv0, tv1, tv2;  // count running time
sigset_t sigmask;              // signal mask for critical regions
int stack[NTASK][SSIZE];       // task stacks
int pd[5][2];                  // 5 pipes for communication
int cmdfile[NTASK];            // task command file descriptors
int record[NTASK+2][13];       // record each task use buff status
int task;                      // count alive task
int ntask;                     // number of tasks
int nbuffer;                   // number of buffers
int ndevice;                   // number of devices
int nblock;                    // number of blocks per device
int sample;                    // number of per task inputs
int timer_switch;              // count time switch
char cmd[CMLEN];               // command=[r|w dev blkno]
char buffer[BLOCK];            // save command to commuicate with device
char dev_stat[CMLEN];          // device status
PROC proc[NTASK], MainProc;    // NTASK tasks and main task
PROC *running;                 // pointer to current running task
PROC *readyQ, *freeQ, *waitQ;  // process link lists
FILE *fp;                      // record simulation results file
int body();                    // task body function
void catcher();                // signal catcher (interrupt handler)

#include "proc.c"              // PROC init, queue functions
#include "buf.c"               // buffer struct, link lists
#include "utility.c"           // generate cmds, end_task()
#include "manager.c"           // getblk,brelse, bread,dwrite,awrite

int main(int argc, char *argv[ ])
{
   if(argc != 6){
     printf("usage: a.out ntask nbuffer ndevice nblock nsample\n");
     exit(0);
   }
   task = ntask  = atoi(argv[1]);
   nbuffer= atoi(argv[2]);
   ndevice= atoi(argv[3]);
   nblock = atoi(argv[4]);
   sample = atoi(argv[5]);
   if(ntask>NTASK||nbuffer>NBUFFER||ndevice>NDEVICE||nblock>NBLOCK)
      exit(1) ;
   fp = fopen("record", "a"); // record task activities
   printf("Welcome to UNIX Buffer Management System\n");
   printf("Initialize proc[] & buff[] & signal\n");
   initproc();            // build tasks to compete for buffers
   initbuf();             // build buffer link lists
   initsig();             // set sigmask & init itimer for timing
   printf("Install signal handler\n");
   signal(SIGUSR1, catcher); // install catcher for SIGUSR1 signal
   printf("Open pipes for communication\n");
   open_pipe();           // create communication pipes
   printf("fork(): parent as CPU child as Disk Controller\n");
   switch(fork()){
     case -1: perror("fork call"); exit(2); break;
     case  0: // child process as disk controller
        close_pipe(0);  // configure pipes at child side
        sprintf(buffer,"%d %d %d %d %d",\
                CMDIN, WRITEIN, READOUT, STATOUT, ACKIN);
        execl("controller","controller",buffer,argv[3],argv[4],0);
        break;
     default:           // parent process as virtual CPU
        close_pipe(1);  // configure pipes at parent side
        printf("MAIN: check device status\n");
        check_dev();    // wait for child has created devices
        printf("Generate command file for each task\n");
        gencmd();       // generate command files of tasks
        gettimeofday(&tv0,0);  // start record run time
        while(1){
           INTON               // enable interrupts
           while(tasks && readyQ == 0); // wait for ready tasks
           if(readyQ)
            kswitch();        // switch to run ready task
           else
             end_task();       // end processing
        }
     }
}

/******* task body function **********/
int body(int pid)
{
  char rw;            // opcode in command: read or write
  int dev, blk;       // dev, blk in command
  int n, count=0;     // task commands count
  while(1){
     n = read(cmdfile[running->pid], cmd, CMLEN);
     if (n==0){                    // if end of cmd file
        running->status = DEAD;    // mark task dead
        task--;                    // number of active task -1
        kswitch();                 // switch to next task
     }
     count++;                      // commands count++
     sscanf(cmd, "%c%5d%5d",&rw,&dev,&blk);
     if (rw == 'r')
        readBlk(dev, blk);         // READ (dev, blk)
     else
        writeBlk(dev, blk);        // WRITE (dev, blk)
  }
}
int kswitch()
{
  INTOFF       // disable interrupts
    tswitch(); // switch task in assembly code
  INTON        // enable interrupts

}
```

The main.c file includes the following files, which are only explained briefly.

(1). **main.h:** This file defines system constants, PROC structure type and symbolic constants. It also defines the macros

```
#define INTON  sigprocmask(SIG_UNBLOCK, &sigmask, 0);
#define INTOFF sigprocmask(SIG_BLOCK,   &sigmask, 0);
```

for masking in/out interrupts, where sigmask is a 32-bit vector with SIGUSR1 bit (10)=1.

(2). **proc.c:** This file initializes the multitasking system. It creates ntask tasks to do disk I/O operations through the buffer cache.

(3). **buf.c:** This file defines semaphore, buffer and device structures. It initializes the buffer and device data structures, and it contains code for buffer link lists operations.

(4). **utility.c:** This file contains functions common to both the CPU side and the Disk Controller side. Examples are creation and configuration of pipes, generation of input commands files and data files for simulated disks, etc. It also implements the end_task() function which collects and display simulation statistics when all tasks have ended.

(5). **manager.c:** This file implements the actual buffer management functions, such as readBlk(dev, blk), writeBlk(dev, blk), getblk() and brelse() which are the core functions of buffer management.

In the simulator, all the files are identical except the manager.c file, which has two versions; one version implements the Unix algorithm and the other one implements the PV algorithm using semaphores.

The following shows the Disk Controller source file, which is executed by a child process of the main process. The information needed by the child process are passed in via command-line parameters in argv[]. After initialization, the Disk Controller process executes an infinite loop until it gets the ‘END’ command from the CPU, indicating all tasks on the CPU side have ended. Then it closes all the virtual disk files and terminates.

```
/******* Pseudo Code of Disk Controller *******/
initialization;
while(1){
   read command from CMDIN
   if  command = “END”, break;
   decode and execute the command
   interrupt CPU;
   read ACK from ACKIN
}
close all virtual disk files and exit
```

```
/******** Disk Controller: controller.c file: *******/
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <string.h>

#define DEVICE  128      // max number of devices
#define BLOCK    16      // disk BLOCK size
#define CMLEN    10      // command record length
int main(int argc, char *argv[ ])
{
   char opcode, cmd[CMLEN], buf[BLOCK], status[CMLEN];
   int CMDIN,READIN,WRITEOUT,STATOUT,ACKIN;  // PIPEs
   int i, ndevice, dev, blk;
   int CPU = getppid();  // ppid for sending signal
   int fd[DEVICE];       // files simulate n devices
   char datafile[8];     // save file name
   // get command line parameters
   sscanf(argv[1], "%d %d %d %d %d",\
        &CMDIN, &READIN, &WRITEOUT, &STATOUT, &ACKIN);
   ndevice = atoi(argv[2]);
   printf("Controller: ndevice=%d\n", ndevice);
   gendata(ndevice);     // generate data files as virtual disks
   for (i=0; i<ndevice i++){  // open files to simulate devices
     sprintf(datafile,"data%d", i%128);
     if(( fd[i] = open(datafile, O_RDWR)) < 0)
       sprintf(status, "%dfail",i) ;
      else
       sprintf(status, " %d ok",i) ;
     write(STATOUT, status, CMLEN);  // send device status to CPU
   }
   // Disk Controller Processing loop
   while(1){
      read(CMDIN, cmd, CMLEN);          // read command from pipe
      if (!strcmp(cmd, "END"))          // end command from CPU, break
         break;
      sscanf(cmd,"%c%4d%5d", &opcode, &dev, &blk);
      if (opcoe == 'r'){
          read(fd[dev], buf, BLOCK);    // read data of (dev, blk)
          write(WRITEOUT, buf, BLOCK); // write data to WRITEOUT pipe
      }
      else if (opcode == 'w'){          // write cmd
          read(READIN, buf, BLOCK);     // read data from READIN pipe
          write(fd[dev], buf, BLOCK);   // write data to (dev, blk)
      }
      else
          strcpy(status, "IOerr");
      write(STATOUT, cmd, CMLEN);       // write INTstatus to STATOUT
      kill(CPU, SIGUSR1);               // interrupt CPU by SIGUSR1
      read(ACKIN, buf, CMLEN);          // read ACK from CPU
   }
   // Disk controller end processing
   for (i=0; i < ndevice;i++)
       close(fd[i]);
}

int gendata(int ndev) // generate data file as virtual disks
{
  char cmd[2048], name[8];
  int fd[128], i, j;
  printf("generate data files\n");
  for(i=0; i<ndev; i++){
     sprintf(name,"data%d", i);
     fd[i]=creat(name, 0644);  // create data files
     for (j=0; j<2048; j++){   // 2 K bytes of '0'-'9' chars
         cmd[j]= i%10 +'0';
     }
     write(fd[i],cmd,2048);
     close(fd[i]);
  }
}
```

The simulator system is compiled and linked into two binary executables, a.out as the CPU side and controller as the Disk Controller side.

```
cc main.c s.s                      # a.out as the CPU side
cc -o controller controller.c      # controller as Disk Controller
```

Then run the simulator system as

```
        a.out  ntask nbuffer ndevice nblock nsample
```

In order to run the simulator with different input parameters, a sh script can be used, as in

```
# run: sh script to run the simulator systems
#! /bin/bash
for TASK in 4 8 16 64; do
  For BUF in 4 16 64 128; do
    a.out $TASK $BUF 16 64 1000   # ndev=16,nblk=64,sample=1000
  done
  echo -------------------------
done
```

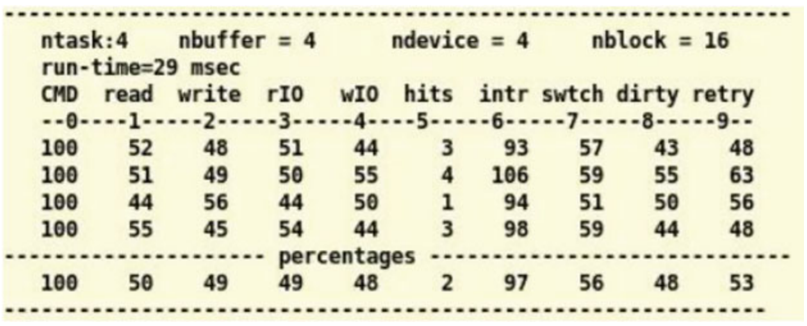
12.5.10 Sample Solutions

Figure 12.2 shows the outputs of the simulator using the Unix algorithm with

4 tasks, 4 buffers, 4 devices, 16 blocks per device and 100 input samples per task.
In the figure, the performance metrics are shown as

```
run-time = total running time
rIO  = number of actual read operations
wIO = number of actual write operations
intr = number of disk interrupts
hits = number of buffer cache hits
swtch = number of task switches
dirty = number of delay write buffers
retry = number of task retries in getblk()
```

The last row of the outputs shows the percentages of the various metrics. The most important performance metrics are the total run time of the simulation and the buffer cache hit ratio. Other metrics may be used as guidance to justify the performance of the algorithms.



```
----------------------------------------------------------
  ntask:4    nbuffer = 4      ndevice = 4     nblock = 16
  run-time=29 msec
  CMD  read  write  rIO   wIO  hits  intr swtch dirty retry
  --0----1-----2-----3-----4----5-----6-----7-----8-----9--
  100    52    48    51    44     3    93    57    43    48
  100    51    49    50    55     4   106    59    55    63
  100    44    56    44    50     1    94    51    50    56
  100    55    45    54    44     3    98    59    44    48
-------------------- percentages --------------------------
  100    50    49    49    48     2    97    56    48    53
--------------------------------------------------------
```

Fig. 12.2 Sample Outputs of Unix Algorithm

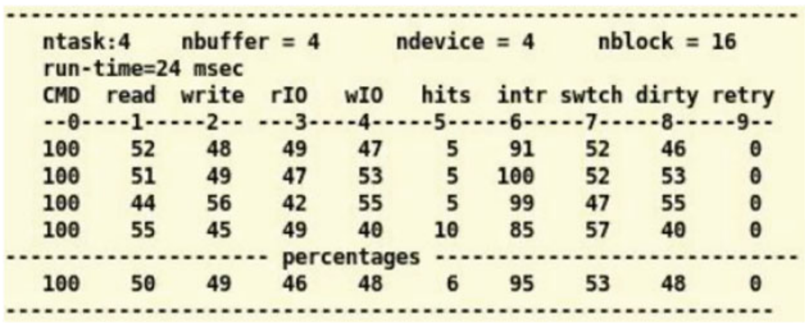


Fig. 12.3 Sample Outputs of PV Algorithm

```
-----------------------------------------------------------------
   ntask:4    nbuffer = 4      ndevice = 4     nblock = 16
   run-time=24 msec
   CMD  read  write  rIO   wIO   hits  intr swtch dirty retry
   --0----1-----2-- ---3----4-----5-----6-----7-----8-----9--
   100    52    48    49    47     5    91    52    46     0
   100    51    49    47    53     5   100    52    53     0
   100    44    56    42    55     5    99    47    55     0
   100    55    45    49    40    10    85    57    40     0
--------------------- percentages --------------------------
   100    50    49    46    48     6    95    53    48     0
--------------------------------------------------------------
```

Figure 12.3 shows the outputs of the simulator using the PV algorithm with the same input parameters as in the Unix algorithm. It shows that the run-time (24 msec) is shorter than the Unix algorithm (29 msec) and the buffer cache hit ratio (6%) is higher than the Unix algorithm (2%). It also shows that, whereas the Unix algorithm has a large number of task retries, there are no task retries in the PV algorithm, which may account for the differences in performance between the two algorithms.

12.6 Refinements of Simulation System

The simulation system can be refined in several ways to make it a better model of real file systems.

(1). Instead of a single disk controller, the simulation system can be extended to support a multiple number of disk controllers, which relives I/O congestions through a single data channel.
(2). The input commands can be generated with non-uniform distributions to better model file operations in a real system. For example, they can be generated with more read commands over write commands, and with heavier I/O demands on some devices, etc.

12.7 Refinements of PV Algorithm

The simple PV algorithm is very simple and easy to implement, but it does has the following two weaknesses. First, its cache effect may not be optimal. This is because as soon as there is no free buffer, all requesting processes will be blocked at (1) in getblk(), even if their needed buffer may already exist in the buffer cache. Second, when a process wakes up from the freelist semaphore queue, it may find the needed buffer already exists but BUSY, in which case it will be blocked again at (4). Strictly speaking, the process has been woken up unnecessarily since it gets blocked twice. The reader may consult (Wang 2015) for an improved algorithm, which is optimal in terms of process switches and its cache performance is also better.

12.8 Summary

This chapter covers block device I/O and buffer management. It explains the principles of block device I/O and the advantages of I/O buffering. It discusses the buffer management algorithm of Unix and points out its shortcomings. Then it uses semaphores to design a new buffer management algorithm to improve the efficiency and performance of the I/O buffer cache. It is shown that the simple PV algorithm is easy to implement, has good cache effect and is free of deadlock and starvation. The programming project is for the reader to implement and compare the performances of the buffer management algorithms in a simulation system. The programming project should help the reader better understand I/O operations and interrupts processing in file systems.

References

Bach, M.J., “The Design of the Unix operating system”, Prentice Hall, 1990

Lion, J., “Commentary on UNIX 6th Edition, with Source Code”, Peer-To-Peer Communications, ISBN 1-57398-013-7, 1996.

Ritchie, D.M., Thompson, K., “The UNIX Time-Sharing System”, Bell System Technical Journal, Vol. 57, No. 6, Part 2, July, 1978,

Wang, X., “Improved I/O Buffer Management Algorithms for Unix Operating System”, M.S. thesis, EECS, WSU, 2002

Wang, K. C., Design and Implementation of the MTX Operating system, Springer A.G, 2015