Data Structure Distribution & Multi-threading of Linux File System for Multiprocessors

Anish Sheth & K. Gopinath
Computer Science and Automation,
Indian Institute of Science,
Bangalore 560012, India,
Email: gopi@csa.iisc.ernet.in

Abstract

The standard Linux design assumes a uniprocessor architecture. Allowing several processors to execute simultaneously in the kernel mode on behalf of different processes can cause consistency problems unless appropriate exclusion mechanisms are used. In addition, if the file system data structures are not distributed, performance can be affected. In this paper, we discuss a multiprocessor file system design for Linux ext2fs with various data structures, such as super block, inodes, buffer cache, directory cache (name cache), distributed wrt different processors with appropriate exclusion mechanisms.

1 Introduction

The classic design of the UNIX system assumes a uniprocessor architecture. Support for shared-memory multiprocessors in UNIX has evolved over a number of years. Early implementations of UNIX allowed concurrent execution of single-threaded processes but many of these implementations serialized execution within the system kernel. In this paper, we present a redesign of the ext2fs of single processor Linux for a multiprocessor.

Allowing several processors to execute simultaneously in the kernel mode on behalf of different processes can cause consistency problems unless appropriate exclusion mechanisms are used. On a multiprocessor, if two or more processes execute simultaneously in the kernel on separate processors, the kernel data could become inconsistent if exclusion mechanisms sufficient for uniprocessor systems are used. There are three methods for preventing such inconsistency:
1. Execute all critical activity on one processor, relying on standard uniprocessor methods for preventing corruption.
2. Serialize access to critical regions of code with locking primitives.
3. Redesign algorithms to avoid contention for data structures.

The first method is a solution with master and slave processors, with master executing in kernel mode while slaves execute only in user mode. The master processor is responsible for handling all system calls. Here inconsistency in kernel can occur only in the scheduler algorithm. Because of the single ready queue, all the processors look for ready processes in this queue and two processors can schedule the same process simultaneously. In one solution, master specifies the slave processor on which the process should execute. In the second solution, kernel allows only one processor to execute the scheduler at a time, using mechanisms such as *semaphores*. More recently, Linux has been extended in Linux 2.x so that only one processor can execute code at a time, without the master/slave dichotomy.

Another method for supporting UNIX systems on multiprocessor configurations is to partition the kernel into critical regions such that at most one processor can execute code in a critical region at a time (Solaris 2.x, Unixware 2.x, etc.). As the sleep-locks used by the uniprocessors cannot be used by multiprocessor systems, semaphores are implemented, which allow atomic operations on them. One example where these semaphores can be used is the buffer allocation algorithm(*getblk()*). For the free buffer list, each hash queue and each buffer one separate semaphore can be provided. A down operation is done on the hash queue semaphore before searching the queue for the buffer. A buffer semaphore is used to get exclusive access of the buffer so that it can be updated. If a new buffer is to be obtained from free list, a down operation is done on free list semaphore before accessing it. While using semaphores, we need to take care of possible deadlocks.

In this paper, we study the later approach. In addition, to enhance locality, we distribute various data structures, such as super block, inodes, buffer cache, directory cache (name cache), wrt different processors with appropriate exclusion mechanisms.

2 Linux File System

Every Linux file system implements a basic set of common concepts derived from the Unix operating system: files are represented by inodes, directories are simply files contain-

ing a list of entries and devices can be accessed by requesting I/O on special files.

2.1 The Second Extended File System

The Second Extended File System (ext2fs) [Car95] is the most popular file system on Linux, with support for big partitions (up to 4 TB) and big file sizes (up to 2 GB), long file names (up to 255 characters), reserved blocks for root (default to 5%), symbolic links and directories with variable length entries. Each directory entry contains the inode number, the entry length, the file name length and the file name. (Figure 1).

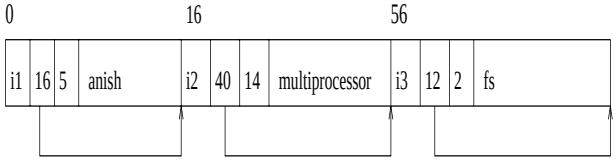

Figure 1: EXT2FS Directory Entry

Each ext2fs file system consists of block groups; the layout is given in Figure 2. Each block group contains a redundant copy of crucial file system control informations (super-block and the file system descriptors) and also contains a part of the file system (a block bitmap, an inode bitmap, a piece of the inode table, and data blocks). The structure of a block group is shown in Figure 3.

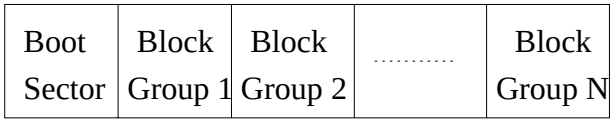

Figure 2: Physical Structure of an EXT2 File System

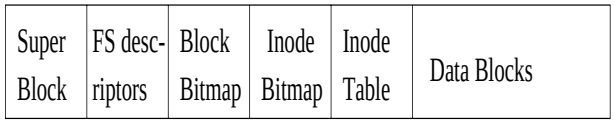

Figure 3: Structure of a Block Group

2.2 The Linux Virtual File System

In order to ease the addition of new file systems into the Linux kernel, a Virtual File System(VFS) layer was developed. Details about the VFS interface in 4.4BSD can be found in [McK95] whereas details for UNIX System V Release 4 are available in [GC94]. When a process issues a file oriented system call, the kernel calls a function contained in the VFS. This function handles the structure independent manipulations and redirects the call to a function contained in the physical file system code, which is responsible for handling the structure dependent operations. File system code uses the buffer cache functions to request I/O on devices.

The VFS layer not only supplies an object oriented interface to the underlying file systems, but also provides a set of management routines that can be used by all the client file systems. The following services are common:

1. In-core Inodes for all the file system types are kept in a global list which is hashed for better search performance. A reference count is used for each in-core inode indicating how many links are currently active to this inode. When the last reference to a file is closed, the reference count becomes zero but the in-core inode is also kept on hash chains so that it can be reactivated quickly if the file is reopened.

The list for in-core inodes is common to all the file systems. When an application opens a file that does not currently have an in-core inode, the VFS routine calls *get_empty_inode()* routine to allocate a new one. This routine removes the first unused inode from the front of the inode list and used by the client file system.

2. Buffer Cache All read/write operations from/to disk are through buffer cache. Each disk block, when in memory, is kept in a buffer. Each buffer has a user count similar to the in-core inode. When a particular block is accessed for read/write, user count for the buffer containing that block is incremented. When operation is over and buffer is released, count is decremented. When buffer is updated, it is marked dirty and it is written[1] back to the disk.

For different types of buffers (e.g. clean, dirty, shared, locked etc.), different LRU queues are maintained. Whenever buffer type changes, buffer is removed from original LRU list and is added to the new LRU list at the time of releasing the buffer (i.e. when *brelse()* is called). Different buffer sizes are supported, each with separate free lists.

3. The Directory Cache This interface provides a facility to add a name and its corresponding inode, lookup a name to get the corresponding inode, and to delete a specific name from the cache. In addition to providing a facility for deleting specific names, the interface also provides an efficient way to invalidate all names that reference a specific inode. Directory inodes can have many names that reference them, notably the .. entries in all their immediate descendents. Each inode is given a *version* - a 32-bit number. An inode's version is incremented each time inode is reassigned to a new file. When an entry is made in the directory table, the current value of the inode's version is copied to the associated name entry. When a name is found during a cached lookup, the version assigned to the name is compared with that of the inode. If they match,

[1] There are three types of UNIX file system writes: *synchronous, asynchronous and delayed.* A write is synchronous if the process issues it (i.e. sends it to the device driver) immediately and waits for it to complete. A write is asynchronous if the process issues it immediately but does not wait for it to complete. A delayed write is not issued immediately; the affected buffer cache blocks are marked dirty and issued later by a background process (unless the cache runs out of a clean block).

the lookup is successful; if they do not match, failure is returned. The directory cache is a 2-level LRU. It also provides for negative caching.

3 A Design for a Multiprocessor Linux File System

The design outlined below uses critical regions and redesigns algorithms to avoid contention. We also describe the data structures used in the file system, the distribution of data structures, their use in the new system and locks needed in the design. Note that both the file system independent portion as well as ext2fs itself need changes. In the following, we assume a standard single-threaded kernel and use the word "processor" and its single kernel thread interchangeably.

First, we discuss the implementation of a semaphore for the multiprocessor case whose structure is as follows:

```
struct semaphore {
   int share;
   int excl;
   struct wait_queue * wait;
   struct wait_queue * excl_wait;
}
struct wait_queue {
   struct task_struct * w_task;
   struct wait_queue  * w_next;
   struct wait_queue  * w_prev;
}
```

share - Number of threads in shared mode
excl - inode locked in exclusive mode
wait - Wait queue of threads
excl_wait - Wait queue of threads
- that require lock in exclusive mode

Some system calls use inodes only for reading and some use it for update. System calls like *write()*, *link()*, *unlink()* update the file/directory entries and hence should have exclusive access to the inode. We define the following down and up operations for exclusive access:

```
void down_excl(struct semaphore *sem){
   struct wait_queue *waiter;
   struct wait_queue *excl_waiter;
   init waiter & excl_waiter (curr thread)
   lock(sem);
   add_wait_queue(&sem->wait, waiter);
   add_wait_queue(&sem->excl_wait, excl_waiter);
  repeat:
   if(sem->share!=0 || sem->excl==1  ||
      sem->excl_wait!=excl_waiter) {
      unlock(sem);
      schedule();
      lock(sem);
      goto repeat;
   }
   sem->excl++;
   remove_wait_queue(&sem->wait, waiter);
   remove_wait_queue(&sem->excl_wait,excl_waiter)
   unlock(sem);
}
void up_excl(struct semaphore *sem){
   lock(sem);
   sem->excl--;
   wakeup(sem, ALL_WAIT);
   unlock(sem);
}
```

Functions like *lookup()* used in *namei()* use directories only for searching. Hence we can provide shared access to more than one such operation. Similar is the case with system calls like *read()*. Semaphore operations that provide shared access are defined as follows:

```
void down_shared(struct semaphore * sem){
  struct wait_queue *waiter;
  initialize waiter
  lock(sem);
  add_wait_queue(&sem->wait, waiter);
 repeat:
  if(sem->excl!=0 ||
     (sem->excl_wait && sem->wait->w_task==
        sem->excl_wait->w_task)) {
    unlock(sem);
    schedule();
    lock(sem);
    goto repeat;
  }
  sem->share++;
  remove_wait_queue(&sem->wait, waiter);
  unlock(sem);
}
void up_shared(struct semaphore *sem){
   lock(sem);
   sem->share--;
   if(sem->share==0)
      wakeup(sem, FIRST_WAIT);
   unlock(sem);
}
```

wakeup() puts waiting processes in the ready queue; its second parameter has the following meaning: FIRST_WAIT (wake up only first process from the wait queue), ALL_WAIT (wake up all the processes before the first *excl_wait* process; if the first process in wait queue is same as that in exclusive wait queue then wake up only first process).

Functions *down_excl()* and *down_shared()*, for a given semaphore, are called only once during the system call execution. The above implementation does not support multiple down operations for a given semaphore in any system call. A kernel thread will deadlock if a down operation is done a second time on the same semaphore.

3.1 Super Block Management

The fields of the super block that are of interest are:
1. group descriptors In allocating or deallocating a disk block or an inode, group descriptors and inode bitmaps or block bitmaps for that group are used. To allow only one processor to update these structures, each processor can haves some group descriptors assigned to it. Whenever it needs to allocate new blocks or inodes, it allocates them from those groups. However, if the processor has used all its blocks/inodes, it cannot allocate a new block/inode even if disk groups on other processors have free blocks/inodes. Also, deallocating a block or an inode from a group of another processor is not possible. An alternative is to provide a lock per group descriptor. Each processor before accessing these structures, gets the lock, does the necessary operations and releases the lock. By providing a lock per group we can allow more than one processors to do inode and/or block operations if they are doing so in different groups. If multiple processors intend to update the same group data, only one can continue and others have to wait till the first one completes.
2. free blocks count Each group descriptor maintains counts of free blocks in the group. A global counter (*s_free_blocks_count*) is also provided for redundancy. During the mount operation or *fsck*, both are used for file system check. In the existing system, each time a disk block is allocated/deallocated, global counter is updated along with the free disk block count in the group descriptor. In a multiprocessor system, this will serialize all the disk block allocation/deallocation operations.

The solution is to provide a count per processor. Each processor, while allocating or deallocating a block, updates this local count. Free block count per processor can be a positive or negative number. These local counts are stored in the super block structure. Current structure uses only 84 bytes whereas disk space allocated to super block is 1024 bytes. A new field (unsigned long s_nr_procs) has been added to the disk super block structure (struct ext2_super_block) to indicate number of processors in the system. Also, an array of free block counts for processors that can be accessed as an offset from the new field.

While mounting or unmounting the file system, all local counts are added to the global count *s_free_blocks_count* and all local counts are reset to 0. In between, whenever a buffer containing super block is written to disk (because it is dirty), these local counters can have nonzero value.

In ext2fs, if total number of free blocks is less than or equal to reserve block count, a new block is not allocated. With the above modification, we do not have the free blocks count in the file system as it is distributed between many counters. Hence, when allocating a disk block, instead of comparing reserve block count with the *s_free_blocks_count* for the file system, a block is allocated from the disk group that has more free disk blocks than the reserve block count per group (⌈reserve blocks count/group count⌉).

3. free inodes count *s_free_inodes_count* indicates the number of free inodes in the file system. It is used in a similar manner as *s_free_blocks_count*. For this counter, a similar solution can be used. Free inode counts per processor are maintained, which are stored after the free blocks count per processor in the disk super block structure.

3.2 Inode Management

All the inodes are kept in the single global structure. Each time *iget()* or *iput()* is called, we need to provide some locking mechanism to keep the inode structure consistent. In addition to a semaphore for each in-core inode, the following locks are provided:
1. Lock for finding free inode The link list of inodes is traversed when unused inode is needed. The first unlocked inode with *i_count* zero is returned as the free inode. The returned inode is put at the end of the link list. To allow only one thread to traverse and update the link list, a lock is provided.
2. Lock per hash bucket To search an inode, *iget()* hashes using the inode number and the device number. That particular hash bucket is locked before searching for the inode. This lock is necessary because some other thread might be removing a free inode from one hash bucket and later adding it to some other hash bucket list depending on the new inode number.
3. Lock for i_count of in-core inode structure Whether an inode is unused or not is decided by the *i_count*. If *i_count* is zero, then the inode is unused. Each time *iget()* is done for an inode, *i_count* is incremented and for each *iput()* it is decremented. To make sure that only one thread is updating/accessing this value, a lock is provided for the *i_count* field of each in-core inode.

Ext2fs inode allocation algorithm tries to keep inodes evenly allocated in each group. A new inode is allocated for a directory in the group which has more number of free inodes than average free inodes per group, which it calculates using following formula: average free inode = free inodes count / number of groups. In our system, *s_free_inodes_count* is not uptodated all the time. So it is not possible to calculate average free inodes per group and use it for inode allocation. We keep one pointer which tells the process from which group descriptor to start the search for new inode. We define an array *start_search* with one element per processor in the array and initialized as below: *start_search[processor number] = processor number % s_group_count*. Whenever a process needs a new inode, depending on which processor it is executing, it starts searching for new inode in the group number given in *start_search[processor number]*. This value is updated to the group number from which last inode was allocated by a process executing on the that processor. Assumption for above procedure is that inode allocation and deallocation are evenly distributed on all processors.

3.3 Buffer Cache Management

Buffer headers are kept globally as well as per processor with the global one being the union of all the local ones. However, the buffer data is never copied. Buffer headers containing meta-data (i.e. super block, group descriptors, bitmaps, inodes) are kept globally only (for reasons, see Section 3.3.3).

3.3.1 Global Buffer Cache

The global header structure *g_buffer_head* contains all the fields of original *struct buffer_head* along with the new fields. The existing *b_count* has a new meaning indicating the number of processors with active references. To manage the global buffer cache, the following locks are used:
1. Lock for each lru_list: Depending on the buffer type (*Clean, Dirty, Locked etc.*) buffers are put in the respective LRU lists. Locks are provided per LRU list.
2. Lock per hash table bucket: Each buffer is hashed using device number and block number. For updates to the hash queues, locks are provided per hash bucket so that operations on different hash buckets can be carried out simultaneously.
3. Lock for free list: A free list of buffers is maintained for each buffer size separately. To provide exclusive access to the free list, lock is kept for each buffer size.
4. Lock for buffer refill: Once the buffer type (*Clean, Dirty, Locked etc.*) changes, buffer is removed from the original LRU list and put in the new LRU list in the *brelse()* function. More than one *brelse()* can be in progress for the same buffer at the same time. To allow only one *brelse()* to do refilling, a new flag (char b_refill) is provided in the buffer structure. Before refilling, this flag is set for the buffer and is reset after completion of the refill. If the flag is already set, refilling is not done.
5. Lock for unused buffer head list: New buffers are created as they are needed. Buffer heads are created from a free page. From the other pages, space for data area of buffer is created. Depending on the size of the data of the buffer, number of data areas created can be different from the buffer head number. Buffer heads are always more in number than data areas. So unused buffer heads are kept in a separate list pointed to by *unused_list*. These buffer heads are used later when more buffers are needed. To avoid corruption of unused list, it is locked before use.
6. Lock for b_count in each buffer: *b_count* gives number of references to the buffer which are active. It is incremented in the *getblk()* function and decremented in the *brelse()* function. More than one instances of these functions can be in execution for the same buffer at the same time. So a lock is provided to keep the count correct.

Apart from the above buffer management locks, the buffer lock, *b_lock*, has been changed so that buffer can be locked in either shared or exclusive mode. Whenever the buffer is written to/read from disk, it is locked and lock is released when write or read is complete. In the multiprocessor system, while one system call on one processor is updating the buffer, an another processor can write it to disk if dirty. In this case partial modifications from first processor can go to disk. To prevent buffer to be written to the disk while it is being updated, it is locked exclusively while the update is in progress.

A few structures like group descriptors are kept in buffers and accessed using proper type cast. We can lock a group descriptor exclusively so that only one thread can use it. But there are more than one group descriptors in one buffer page. Hence, to allow simultaneous processing of different group descriptors, we need to lock the buffer in shared mode. For this, one more counter is used per buffer which indicates number of references in the shared lock mode (*b_lshare*). Each thread locking a buffer in shared mode also increments *b_lshare* and decrements it while unlocking along with setting/resetting *b_lock*. While thread locking buffer in exclusive mode sets only *b_lock*.

3.3.2 Local Buffer Cache

A local buffer cache has the following structure: hash buckets (buffer hashed using the same algorithm as that of global cache), one LRU list, free list of buffers (one free list for each buffer size) and unused buffer head list. A new field (struct g_buffer_head * b_g_bh) has been added to the *buffer_head* structure that points to the corresponding global buffer. The *b_count* indicates number of references to the given buffer currently active from the processes on that processor.

3.3.3 Maintaining Consistency

A new field (unsigned long b_version) has been added to both *l_buffer_head* and *g_buffer_head* structures for consistency. Whenever a thread updates a buffer, it increments the version number for the global as well as local buffer on that processor. At a time only one thread can update a particular buffer due to the exclusive lock whereas readers use a shared lock. Buffers containing super block, group descriptors, inodes are kept in global cache only and are updated there. They are not moved to the local caches. This is required because of the following reasons:
• **super block** can be updated by more than one thread simultaneously due to the updates to local free block counts and free inodes count on allocation or deallocation of blocks or inodes. However, *bread()* and *brelse()* functions for super block are needed only during mount and unmount system calls.

• **group descriptor buffers** We lock one group descriptor at a time. As one buffer page contains more than one group descriptor, more than one thread can be updating the buffer simultaneously. Buffers containing group descriptors and bitmaps are also kept in memory permanently. *bread()* and *brelse()* are also used only at the time of mount and unmount.
• **inode buffers**: While writing inode back to the disk, inode is locked and copied to the buffer. Lock is released when writing is complete. One buffer contains more than

one inode. So a buffer can be updated simultaneously by more than one thread.

To get access to the buffers containing inodes, separate *bread()*, *brelse()* and *mark_buffer_dirty()* functions are used, which work with global buffers instead of local buffers. Due to lack of space, we provide simplified algorithms for buffer cache management for a subset here (with type of gbh struct g_buffer_head* and lbh struct l_buffer_head*):

```
wait_on_buffer_local_read(lbh){
   gbh = lbh->b_g_bh;
   wait_on_buffer_read(gbh);
   // shared lock on data buf
   // excl lock on local/global buf hdrs here
   if(gbh->b_uptodate) {
      copy global buffer header to local
      lbh->b_uptodate = 1
      lbh->b_version=gbh->b_version
   }
   // excl lock on local/global buf hdrs dropped
}
uptodate(lbh) {
  gbh = lbh->b_g_bh;
  lock_buffer_hdr(gbh);
  return (lbh->b_version == gbh->b_version);
}
struct l_buffer_head *bread(){
    lbh = getblk();
    if uptodate(lbh) return lbh;
    unlock_buffer_hdr(gbh);
    wait_on_buffer_local_read(lbh);
    if uptodate(lbh) return lbh;
    brelse(lbh);// releases both hdr locks also
    return NULL;
}
mark_buffer_dirty_local(lbh){
   gbh=lbh->b_g_bh;
   lock_buffer_hdr(gbh)
   gbh->b_version++;
   lbh->b_version=gbh->b_version;
   unlock_buffer_hdr(gbh)
   mark_buffer_dirty(gbh);
}
```

3.4 File Table Management

The file table is maintained as a link list. Each time a new file table entry is required, the link list is searched and the first entry with *f_count* zero is returned. For the multiprocessor system, the file table is kept per processor along with one global free list. Free file pointers are kept in local lists. Following procedures are followed while allocating and deallocating the file pointer:

1. Get new file pointer in open() Whenever a free file pointer is needed it is taken from the local free list. If local free list is empty, it tries to get **NR_GET_FILE** file pointers from the global free list. If global list is empty and total number of file pointers in the system is less than **NR_FILES**, it creates new file pointers and gets required number of free file pointers.

2. Releasing file pointer from close() File pointer with *f_count* zero is returned to the free list. If number of file pointers in the local free list reaches upper threshold (NR_MAX_FILE_PER_PROC), some of the free file pointers are returned to the global free list. The following variables are used: *first_file* (global free list of file pointers), *lock for first_file* (to make global list accesses mutually exclusive), *nr_free_files* (number of free file pointers in global list), *free_files[NR_PROCS]* (free list of file pointers for each processor), and *nr_free_file_per_proc[NR_PROCS]* (number of free file pointers in local lists)

Each time free file pointers from the global list are required, the thread first gets the lock, removes free file pointers from the free list and releases the lock. Additional locks are also provided for the following fields of the file pointer: *f_count* (number of fd's pointing to this entry, 0 indicating free entry), and *f_pos* (offset in the open file).

3.5 Directory Cache Management

Linux maintains a cache of frequently used directory entries. The replacement policy for the cache is a two level LRU policy. Whenever a directory entry is to be searched (in the *namei()* routine) directory cache is first looked up. If the entry is not found, the directory is searched. We can provide a local directory cache per processor along with the global directory cache which contains all the entries kept in local caches. Advantage of keeping local directory caches is that more than one thread can search directory cache simultaneously without locking the cache. Corresponding to each local cache entry there is one entry in the global cache. The size of the global cache is the total size of all the local caches. Each entry keeps track of number of local caches that contain this entry alongwith an array with processor numbers. To prevent other threads from using invalid entries, each time a thread marks the entry invalid, it has to inform all other processor caches about this invalid entry. For this purpose, a stale entry queue is provided per local cache. This is a circular queue which contains the invalid entries.

To maintain the caches uptodate and to prevent processes from using invalid entries, procedures for searching a directory cache (invalidate_stale_entry, dcache_lookup), adding new entry (dcache_add) and marking an entry stale in a directory cache (mark_entry_stale) are provided but omitted here due to lack of space.

4 Implementation

Source code of the Linux 1.x file system has been used for the work. As a multiprocessor system is not available, we decided to do simulation at the implementation level. Actual code has been used as is except for hardware dependent part. The device driver has been replaced

with simplified code, which writes into the simulated disk instead of actual disk. Memory management has been replaced by simple page management routines. We assume that threads do not migrate and that threads know the id of the processor on which it is executing.

4.1 Simulated Disk

The disk has been simulated using multiple UNIX regular files. Layout of the disk is similar to the layout used by the second extended file system (Section 2.1). The disk used for the simulation consists of the following files: The *disk* file stores all the metadata (i.e. super block, group descriptors, inode and block bitmaps and inode table). For regular files, only indirect blocks are stored in a file having block number as the filename (i.e. block number 251 is stored in file 251). For a directory, all the blocks are stored.

4.2 Simulation of Multiple Processors

A separate process is created using *fork()* for each thread. More than one thread is assumed to be running on a single processor. One separate process is used to schedule them. At a time, only one thread (process) is allowed to run on a given processor. There are maximum of N threads (processes) running simultaneously if there are N processors in the system. Once a thread (process) terminates or waits for some resource, the next thread from the ready queue of that processor is activated.

4.3 Sharing Kernel Variables of FS

As threads are simulated using processes, mmap() has been used for sharing of kernel variables of the simulated system. All the variables used by the file system are mapped to a character array (i.e. each variable is '#define' as an offset in the large character array). Portions of the header files used for mapping of variables are shown below

```
char * mp_ptr;
#define SZ_NR_BUFFER_HEADS (sizeof(int))
#define SZ_BUF_HASH_TABLE (NR_HASH*
    sizeof(struct buffer_head *))
#define AD_NR_BUFFER_HEADS (mp_ptr)
#define AD_BUF_HASH_TABLE (AD_NR_BUFFER_HEADS+
    SZ_NR_BUFFER_HEADS)
#define nr_buffer_heads
    (*(int *)AD_NR_BUFFER_HEADS)
#define buf_hash_table
    ((struct buffer_head **)AD_BUF_HASH_TABLE)
```

whereas actual variable declarations are shown below:

```
int nr_buffer_heads;
struct buffer_head * buf_hash_table[NR_HASH];
```

The character array is a UNIX file of required size, which is mapped using *mmap()* system call for each process and is accessed through variable *mp_ptr*.

4.4 Locking

Each thread is simulated using a different UNIX process and variables are mapped to a file. So it is now possible to simulate locking using record locking (i.e. using *fcntl()* call). Offsets of all the variables in the file are available and are used for locking portions of file whenever locking is needed.

5 Validation

To validate the file system, we have to show the following:

No meta-data corruption: Multiple threads must be able to update same meta-data without corruption. This indicates that the locking design is sound.

No deadlocks: Some operations require more than one lock to be obtained before they can continue. This may lead to the deadlock if proper care is not taken in locking and unlocking.

Apart from the above two points, lock frequencies for following structures are measured:

- **The global buffer cache hash table buckets**: Two different designs of buffer cache are considered: single global buffer cache, distributed buffer cache. In the first case buffer cache is kept only in global shared memory. Whenever a buffer is needed, process accesses global cache of buffers. The second option is the buffer cache design given in Section 3.3.
- **The file table**: Number of accesses to the global free list of file table and number of accesses to the local file tables are measured.
- **The directory cache**: Number of accesses to the global directory cache and number of accesses to the local directory caches are measured. Number of global accesses is the total number of times different hash buckets of global cache are locked. Number of local accesses is the sum of two counts for each processor: number of time hash buckets of local cache are accessed and number of times LRU list of local cache is used. Due to the cache management algorithms for local cache, both the counts, hash bucket accesses and LRU accesses, are considered. The locks required will be at least equal to local access count if we use these cache algorithms for a global cache.

To validate above points, we can use a parallel application which requires meta-data to be updated from multiple threads simultaneously. Correct output of the application is obtained only if data is not corrupted. File system code was tested using following two applications :

- dining philosophers problem (5 philosophers)
- readers and writers problem (8 readers and 1 writer)

None of the above applications is related to the file system. We have modified the locking/unlocking mechanisms used by these applications such that locking/unlocking operations require file system calls. Both applications require binary semaphores. In the *dining philosophers problem*, there is one common semaphore and one semaphore for each philosopher. Semaphore for each philosopher is the

file with philosopher number as the filename. For *readers and writers problem*, there are two semaphores which are used by readers and one of them is also used by writer. So two different files are used, one for each semaphore, which are common for all readers and writers.

A binary semaphore is implemented using *creat()* system call. Using *os_creat()*, *os_close()* and *os_unlink()* system calls of the multiprocessor file system, *down()* and *up()* can be realized as follows:

```
down(char *file){
  int fd;
  while((fd = os_creat(file,0)) < 0) sleep(1);
  os_close(fd);
}
up(char *file){
  if(os_unlink(file) < 0)
    printf("unlink error for %s", file);
}
```

6 Results and Conclusion

Buffer Cache Performance Table 1 and Table 2 show the lock frequencies of hash buckets in both the applications for global buffer cache and distributed buffer cache. The block number 251 is the first block of "/" directory. Files used for semaphores in both the applications are created in this directory. This block maps to bucket number 250 The lock frequency for this block in both applications is reduced for distributed buffer cache as once a processor has a buffer header locally, it doesn't need to lock global buffer hash bucket to access the block buffer.

Hash Bkt #	*Global Buf Cache*		*Distr Buf Cache*	
	Lock Freq	Mean Wait Time (μsec)	Lock Freq	Mean Wait Time (μsec)
0	8	7.125	8	7.125
3	8	11.125	8	10.625
4	236	27.542	236	27.932
5	8	19.250	8	17.875
250	246	30.529	12	22.167

Table 1: Lock Frequency for Buffer Hash Buckets in Dining Philosophers Problem

Other hash buckets have blocks which contain meta-data: super block (block number 1), group descriptor block (block number 2), inode bitmap for group 1 (block number 4) and first inode table block for group 1 (block number 6). The lock frequencies of these blocks are the same as meta-data is kept globally in both the designs.

File Table and Directory Cache Performance The number of accesses to the global free list of file pointers and the global directory cache get reduced in both the applications (from 130 to 5 and 535 to 335 for dining philosophers problem and from 226 to 9 and 850 to 465 for readers and writers problem). Each processor accesses global free file pointer list only once in both the applications. All remaining accesses are satisfied in their local list only.

Hash Bkt #	*Global Buf Cache*		*Distr Buf Cache*	
	Lock Freq	Mean Wait Time (μsec)	Lock Freq	Mean Wait Time (μsec)
0	8	7.375	8	6.875
3	8	11.625	8	10.375
4	255	28.510	250	28.160
5	8	17.750	8	17.125
250	275	30.520	16	25.313

Table 2: Lock Frequency for Buffer Hash Buckets in Readers and Writers Problem

In this paper, a redesign for a multiprocessor Linux file system was presented. While some design decisions were straightforward (like for a file table), others (like for buffer cache and directory cache) were not so. More detailed study can be done by actual implementation in the kernel.

References

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, EngleWood Cliffs, New Jersey, 1986.

[Car95] Remy Card. The Second Extended File System - Current State, Future Developement. In *Second International Linux and Internet Conference*, Berlin, 1995.

[GC94] Berny Goodheart and James Cox. *The Magic Garden Explained : The Internals of UNIX System V Release 4.* Prentice Hall, 1994.

[McK95] Marshall Kirk McKusick. The Virtual Filesystem Interface in 4.4BSD. *The USENIX Association,Computing Systems*, 8(1):3–25, Winter 1995.