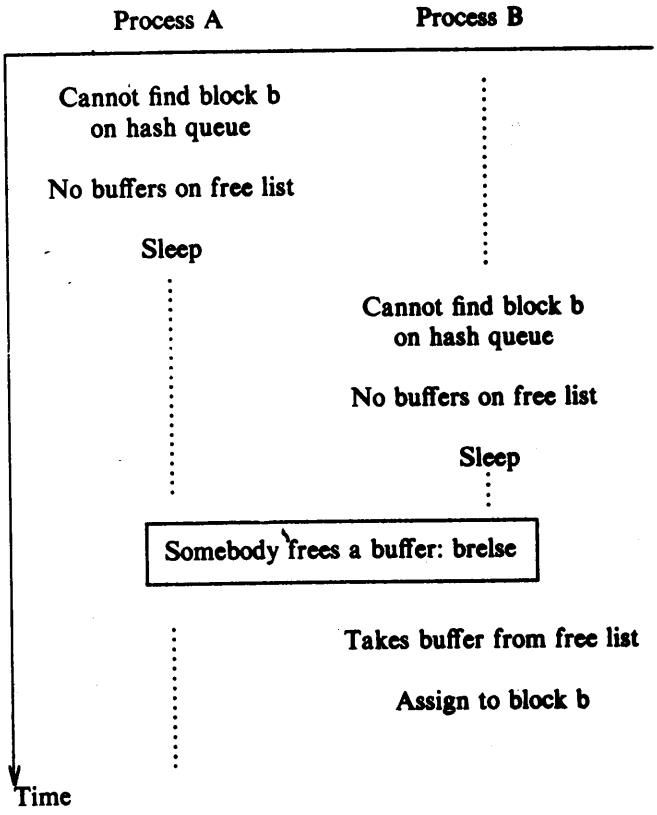
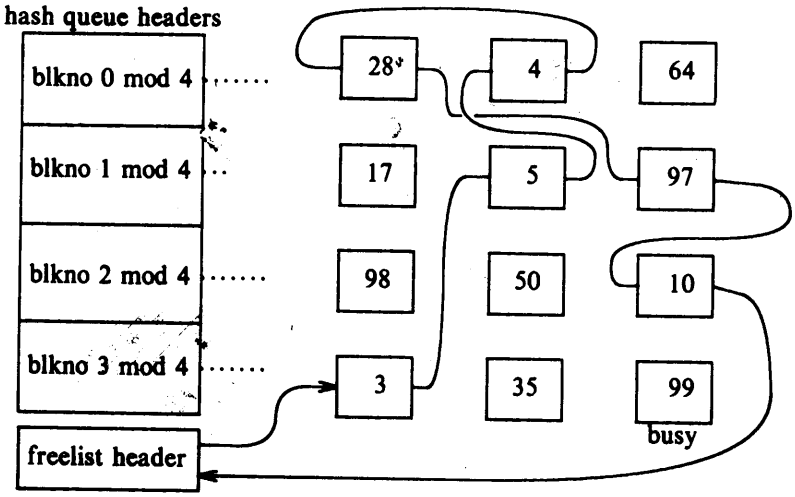


Figure 3.10. Race for Free Buffer

In the end, process B will find its block, possibly allocating a new buffer from the free list as in the second scenario. In Figure 3.11, for example, a process searching for block 99 finds it on its hash queue, but the block is marked busy. The process sleeps until the block becomes free and then restarts the algorithm from the beginning. Figure 3.12 depicts the contention for a locked buffer.

The algorithm for buffer allocation must be safe; processes must not sleep forever, and they must eventually get a buffer. The kernel guarantees that all processes waiting for buffers will wake up, because it allocates buffers during the execution of system calls and frees them before returning.[3] Processes in user mode



Search for Block 99, Block Busy

Figure 3.11. Fifth Scenario for Buffer Allocation

do not control the allocation of kernel buffers directly, so they cannot purposely "hog" buffers. The kernel loses control over a buffer only when it waits for the completion of I/O between the buffer and the disk. It is conceivable that a disk drive is corrupt so that it cannot interrupt the CPU, preventing the kernel from ever releasing the buffer. The disk driver must monitor the hardware for such cases and return an error to the kernel for a bad disk job. In short, the kernel can guarantee that processes sleeping for a buffer will wake up eventually.

It is also possible to imagine cases where a process is starved out of accessing a buffer. In the fourth scenario, for example, if several processes sleep while waiting for a buffer to become free, the kernel does not guarantee that they get a buffer in the order that they requested one. A process could sleep and wake up when a buffer becomes free, only to go to sleep again because another process got control of the buffer first. Theoretically, this could go on forever, but practically, it is not a problem because of the many buffers that are typically configured in the system.

---

3. The *mount* system call is an exception, because it allocates a buffer until a later *umount* call. This exception is not critical, because the total number of buffers far exceeds the number of active mounted file systems.

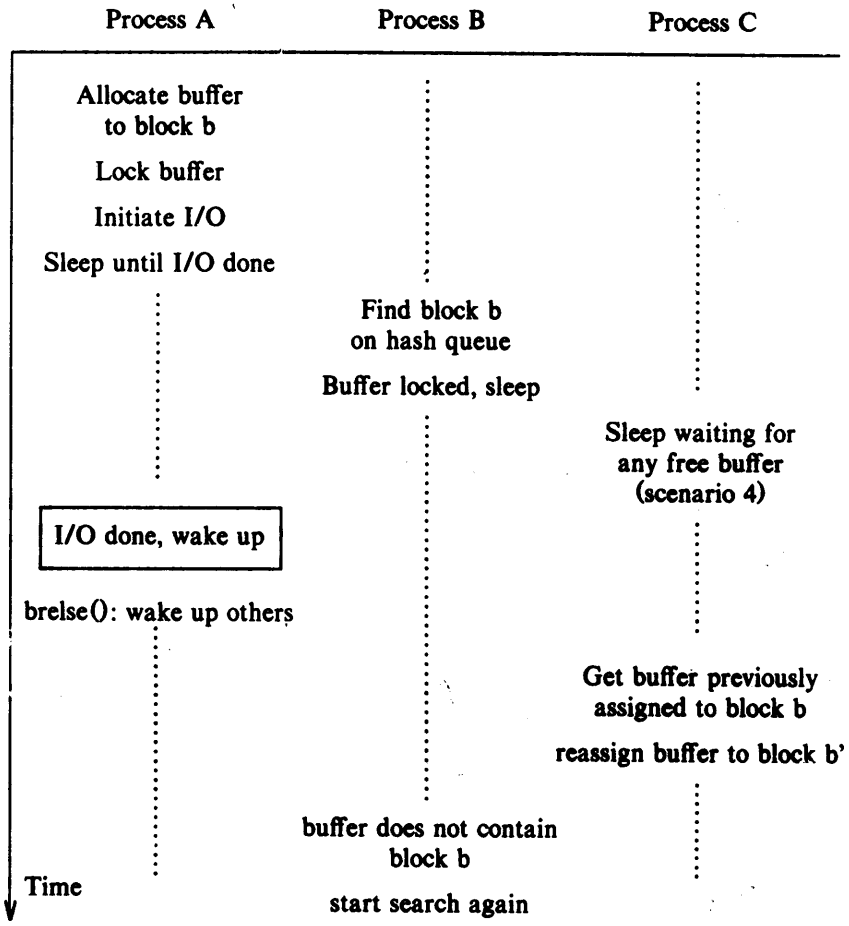


Figure 3.12. Race for a Locked Buffer

3.4 READING AND WRITING DISK BLOCKS

Now that the buffer allocation algorithm has been covered, the procedures for reading and writing disk blocks should be easy to understand. To read a disk block (Figure 3.13), a process uses algorithm *getblk* to search for it in the buffer cache. If it is in the cache, the kernel can return it immediately without physically reading the block from the disk. If it is not in the cache, the kernel calls the disk driver to "schedule" a read request and goes to sleep awaiting the event that the I/O completes. The disk driver notifies the disk controller hardware that it wants to read data, and the disk controller later transmits the data to the buffer.

```
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);
}
```

Figure 3.13. Algorithm for Reading a Disk Block

the disk controller interrupts the processor when the I/O is complete, and the disk interrupt handler awakens the sleeping process; the contents of the disk block are now in the buffer. The modules that requested the particular block now have the data; when they no longer need the buffer they release it so that other processes can access it.

Chapter 5 shows how higher-level kernel modules (such as the file subsystem) may anticipate the need for a second disk block when a process reads a file sequentially. The modules request the second I/O asynchronously in the hope that the data will be in memory when needed, improving performance. To do this, the kernel executes the block read-ahead algorithm *breada* (Figure 3.14): The kernel checks if the first block is in the cache and, if it is not there, invokes the disk driver to read that block. If the second block is not in the buffer cache, the kernel instructs the disk driver to read it asynchronously. Then the process goes to sleep awaiting the event that the I/O is complete on the first block. When it awakens, it returns the buffer for the first block, and does not care when the I/O for the second block completes. When the I/O for the second block does complete, the disk controller interrupts the system; the interrupt handler recognizes that the I/O was asynchronous and releases the buffer (algorithm *brelse*). If it would not release the buffer, the buffer would remain locked and, therefore, inaccessible to all processes. It is impossible to unlock the buffer beforehand, because I/O to the buffer was active, and hence the buffer contents were not valid. Later, if the process wants to read the second block, it should find it in the buffer cache, the I/O having completed in the meantime. If, at the beginning of *breada*, the first block was in the buffer cache, the kernel immediately checks if the second block is in the cache and proceeds as just described.

The algorithm for writing the contents of a buffer to a disk block is similar (Figure 3.15). The kernel informs the disk driver that it has a buffer whose contents should be output, and the disk driver schedules the block for I/O. If the write is synchronous, the calling process goes to sleep awaiting I/O completion and

```
algorithm breada        /* block read and read ahead */
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;
      }
      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;
}
```

Figure 3.14. Algorithm for Block Read Ahead

releases the buffer when it awakens. If the write is asynchronous, the kernel starts the disk write but does not wait for the write to complete. The kernel will release the buffer when the I/O completes.

There are occasions, described in the next two chapters, when the kernel does not write data immediately to disk. If it does a "delayed write," it marks the buffer accordingly, releases the buffer using algorithm *brelse*, and continues without scheduling I/O. The kernel writes the block to disk before another process can reallocate the buffer to another block, as described in scenario 3 of *getblk*. In the meantime, the kernel hopes that a process accesses the block before the buffer must be written to disk; if that process subsequently changes the contents of the buffer, the kernel saves an extra disk operation.

A delayed write is different from an *asynchronous write*. When doing an asynchronous write, the kernel starts the disk operation immediately but does not wait for its completion. For a "delayed write," the kernel puts off the physical write to disk as long as possible; then, recalling the third scenario in algorithm

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

Figure 3.15. Algorithm for Writing a Disk Block

getblk, it marks the buffer "old" and writes the block to disk asynchronously. The disk controller later interrupts the system and releases the buffer, using algorithm *brelse*; the buffer ends up on the head of the free list, because it was "old." Because of the two asynchronous I/O operations — block read ahead and delayed write — the kernel can invoke *brelse* from an interrupt handler. Hence, it must prevent interrupts in any procedure that manipulates the buffer free list, because *brelse* places buffers on the free list.

3.5 ADVANTAGES AND DISADVANTAGES OF THE BUFFER CACHE

Use of the buffer cache has several advantages and, unfortunately, some disadvantages.

- The use of buffers allows uniform disk access, because the kernel does not need to know the reason for the I/O. Instead, it copies data to and from buffers, regardless of whether the data is part of a file, an inode, or a super block. The buffering of disk I/O makes the code more modular, since the parts of the kernel that do the I/O with the disk have one interface for all purposes. In short, system design is simpler.
- The system places no data alignment restrictions on user processes doing I/O, because the kernel aligns data internally. Hardware implementations frequently require a particular alignment of data for disk I/O, such as aligning the data on a two-byte boundary or on a four-byte boundary in memory. Without a buffer mechanism, programmers would have to make sure that their data buffers were correctly aligned. Many programmer errors would result, and programs would not be portable to UNIX systems running on machines with stricter address alignment properties. By copying data from user buffers to system buffers (and vice versa), the kernel eliminates the need for special alignment of user buffers,

making user programs simpler and more portable.

- Use of the buffer cache can reduce the amount of disk traffic, thereby increasing overall system throughput and decreasing response time. Processes reading from the file system may find data blocks in the cache and avoid the need for disk I/O. The kernel frequently uses "delayed write" to avoid unnecessary disk writes, leaving the block in the buffer cache and hoping for a cache hit on the block. Obviously, the chances of a cache hit are greater for systems with many buffers. However, the number of buffers a system can profitably configure is constrained by the amount of memory that should be kept available for executing processes: if too much memory is used for buffers, the system may slow down because of excessive process swapping or paging.
- The buffer algorithms help insure file system integrity, because they maintain a common, single image of disk blocks contained in the cache. If two processes simultaneously attempt to manipulate one disk block, the buffer algorithms (*getblk* for example) serialize their access, preventing data corruption.
- Reduction of disk traffic is important for good throughput and response time, but the cache strategy also introduces several disadvantages. Since the kernel does not immediately write data to the disk for a delayed write, the system is vulnerable to crashes that leave disk data in an incorrect state. Although recent system implementations have reduced the damage caused by catastrophic events, the basic problem remains: A user issuing a write system call is never sure when the data finally makes its way to disk.[4]
- Use of the buffer cache requires an extra data copy when reading and writing to and from user processes. A process writing data copies the data into the kernel, and the kernel copies the data to disk; a process reading data has the data read from disk into the kernel, and from the kernel to the user process. When transmitting large amounts of data, the extra copy slows down performance, but when transmitting small amounts of data, it improves performance because the kernel buffers the data (using algorithms *getblk* and delayed write) until it is economical to transmit to or from the disk.

3.6 SUMMARY

This chapter has presented the structure of the buffer cache and the various methods by which the kernel locates blocks in the cache. The buffer algorithms combine several simple ideas to provide a sophisticated caching mechanism. The kernel uses the least-recently-used replacement algorithm to keep blocks in the

4. The standard I/O package available to C language programs includes an *fflush* call. This function call flushes data from buffers in the user address space (part of the package) into the kernel. However, the user still does not know when the kernel writes the data to the disk.

buffer cache, assuming that blocks that were recently accessed are likely to be accessed again soon. The order that the buffers appear on the free list specifies the order in which they were last used. Other buffer replacement algorithms, such as first-in-first-out or least-frequently-used, are either more complicated to implement or result in lower cache hit ratios. The hash function and hash queues enable the kernel to find particular blocks quickly, and use of doubly linked lists makes it easy to remove buffers from the lists.

The kernel identifies the block it needs by supplying a logical device number and block number. The algorithm *getblk* searches the buffer cache for a block and, if the buffer is present and free, locks the buffer and returns it. If the buffer is locked, the requesting process sleeps until it becomes free. The locking mechanism ensures that only one process at a time manipulates a buffer. If the block is not in the cache, the kernel reassigns a free buffer to the block, locks it and returns it. The algorithm *bread* allocates a buffer for a block and reads the data into the buffer, if necessary. The algorithm *bwrite* copies data into a previously allocated buffer. If, in execution of certain higher-level algorithms, the kernel determines that it is not necessary to copy the data immediately to disk, it marks the buffer "delayed write" to avoid unnecessary I/O. Unfortunately, the "delayed write" scheme means that a process is never sure when the data is physically on disk. If the kernel writes data synchronously to disk, it invokes the disk driver to write the block to the file system and waits for an I/O completion interrupt.

The kernel uses the buffer cache in many ways. It transmits data between application programs and the file system via the buffer cache, and it transmits auxiliary system data such as inodes between higher-level kernel algorithms and the file system. It also uses the buffer cache when reading programs into memory for execution. The following chapters will describe many algorithms that use the procedures described in this chapter. Other algorithms that cache inodes and pages of memory also use techniques similar to those described for the buffer cache.

3.7 EXERCISES

1. Consider the hash function in Figure 3.3. The best hash function is one that distributes the blocks uniformly over the set of hash queues. What would be an optimal hashing function? Should a hash function use the logical device number in its calculations?
2. In the algorithm *getblk*, if the kernel removes a buffer from the free list, it must raise the processor priority level to block out interrupts before checking the free list. Why?
* 3. In algorithm *getblk*, the kernel must raise the processor priority level to block out interrupts before checking if a block is busy. (This is not shown in the text.) Why?
4. In algorithm *brelse*, the kernel enqueues the buffer at the head of the free list if the buffer contents are invalid. If the contents are invalid, should the buffer appear on a hash queue?
5. Suppose the kernel does a delayed write of a block. What happens when another process takes that block from its hash queue? From the free list?

* 6. If several processes contend for a buffer, the kernel guarantees that none of them sleep forever, but it does not guarantee that a process will not be starved out from use of a buffer. Redesign *getblk* so that a process is guaranteed eventual use of a buffer.
7. Redesign the algorithms for *getblk* and *brelse* such that the kernel does not follow a least-recently-used scheme but a first-in-first-out scheme. Repeat this problem using a least-frequently-used scheme.
8. Describe a scenario where the buffer data is already valid in algorithm *bread*.
* 9. Describe the various scenarios that can happen in algorithm *breada*. What happens on the next invocation of *bread* or *breada* when the current read-ahead block will be read? In algorithm *breada*, if the first or second block are not in the cache, the later test to see if the buffer data is valid implies that the block could be in the buffer pool. How is this possible?
10. Describe an algorithm that asks for and receives any free buffer from the buffer pool. Compare this algorithm to *getblk*.
11. Various system calls such as *umount* and *sync* (Chapter 5) require the kernel to flush to disk all buffers that are "delayed write" for a particular file system. Describe an algorithm that implements a buffer flush. What happens to the order of buffers on the free list as a result of the flush operation? How can the kernel be sure that no other process sneaks in and writes a buffer with delayed write to the file system while the flushing process sleeps waiting for an I/O completion?
12. Define system response time as the average time it takes to complete a system call. Define system throughput as the number of processes the system can execute in a given time period. Describe how the buffer cache can help response time. Does it necessarily help system throughput?

4

INTERNAL REPRESENTATION OF FILES

As observed in Chapter 2, every file on a UNIX system has a unique inode. The inode contains the information necessary for a process to access a file, such as file ownership, access rights, file size, and location of the file's data in the file system. Processes access files by a well defined set of system calls and specify a file by a character string that is the path name. Each path name uniquely specifies a file, and the kernel converts the path name to the file's inode.

This chapter describes the internal structure of files in the UNIX system, and the next chapter describes the system call interface to files. Section 4.1 examines the inode and how the kernel manipulates it, and Section 4.2 examines the internal structure of regular files and how the kernel reads and writes their data. Section 4.3 investigates the structure of directories, the files that allow the kernel to organize the file system as a hierarchy of files, and Section 4.4 presents the algorithm for converting user file names to inodes. Section 4.5 gives the structure of the super block, and Sections 4.6 and 4.7 present the algorithms for assignment of disk inodes and disk blocks to files. Finally, Section 4.8 talks about other file types in the system, namely, pipes and device files.

The algorithms described in this chapter occupy the layer above the buffer cache algorithms explained in the last chapter (Figure 4.1). The algorithm *iget* returns a previously identified inode, possibly reading it from disk via the buffer cache, and the algorithm *iput* releases the inode. The algorithm *bmap* sets kernel parameters for accessing a file. The algorithm *namei* converts a user-level path

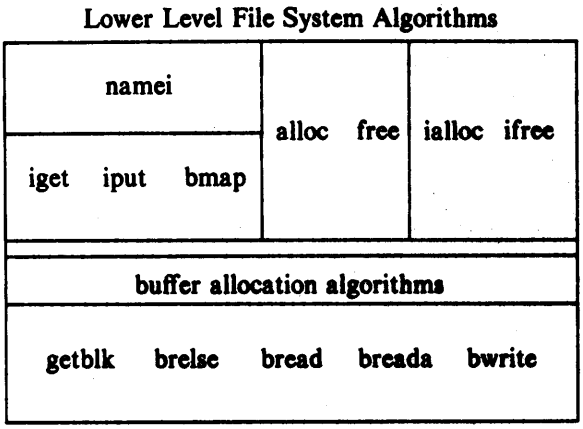


Figure 4.1. File System Algorithms

name to an inode, using the algorithms *iget*, *iput*, and *bmap*. Algorithms *alloc* and *free* allocate and free disk blocks for files, and algorithms *ialloc* and *ifree* assign and free inodes for files.

4.1 INODES

4.1.1 Definition

Inodes exist in a static form on disk, and the kernel reads them into an in-core inode to manipulate them. Disk inodes consist of the following fields:

- File owner identifier. Ownership is divided between an individual owner and a "group" owner and defines the set of users who have access rights to a file. The superuser has access rights to all files in the system.
- File type. Files may be of type regular, directory, character or block special, or FIFO (pipes).
- File access permissions. The system protects files according to three classes: the owner and the group owner of the file, and other users; each class has access rights to read, write and execute the file, which can be set individually. Because directories cannot be executed, execution permission for a directory gives the right to search the directory for a file name.
- File access times, giving the time the file was last modified, when it was last accessed, and when the inode was last modified.

- Number of links to the file, representing the number of names the file has in the directory hierarchy. Chapter 5 explains file links in detail.
- Table of contents for the disk addresses of data in a file. Although users treat the data in a file as a logical stream of bytes, the kernel saves the data in discontiguous disk blocks. The inode identifies the disk blocks that contain the file's data.
- File size. Data in a file is addressable by the number of bytes from the beginning of the file, starting from byte offset 0, and the file size is 1 greater than the highest byte offset of data in the file. For example, if a user creates a file and writes only 1 byte of data at byte offset 1000 in the file, the size of the file is 1001 bytes.

The inode does not specify the path name(s) that access the file.

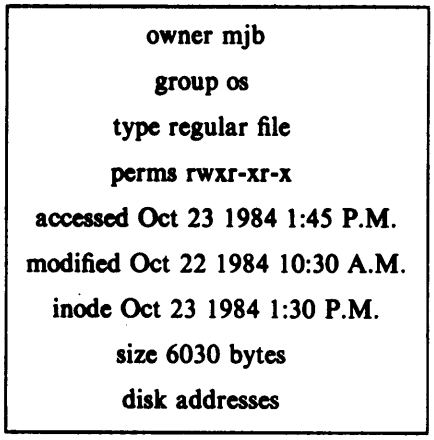


Figure 4.2. Sample Disk Inode

Figure 4.2 shows the disk inode of a sample file. This inode is that of a regular file owned by "mjb," which contains 6030 bytes. The system permits "mjb" to read, write, or execute the file; members of the group "os" and all other users can only read or execute the file, not write it. The last time anyone read the file was on October 23, 1984, at 1:45 in the afternoon, and the last time anyone wrote the file was on October 22, 1984, at 10:30 in the morning. The inode was last changed on October 23, 1984, at 1:30 in the afternoon, although the data in the file was not written at that time. The kernel encodes the above information in the inode. Note the distinction between writing the contents of an inode to disk and writing the contents of a file to disk. The contents of a file change only when *writ*ing it. The contents of an inode change when changing the contents of a file or when changing its owner, permission, or link settings. Changing the contents of a

file automatically implies a change to the inode, but changing the inode does not imply that the contents of the file change.

The in-core copy of the inode contains the following fields in addition to the fields of the disk inode:

- The status of the in-core inode, indicating whether
  - the inode is locked,
  - a process is waiting for the inode to become unlocked,
  - the in-core representation of the inode differs from the disk copy as a result of a change to the data in the inode,
  - the in-core representation of the file differs from the disk copy as a result of a change to the file data,
  - the file is a mount point (Section 5.15).
- The logical device number of the file system that contains the file.
- The inode number. Since inodes are stored in a linear array on disk (recall Section 2.2.1), the kernel identifies the number of a disk inode by its position in the array. The disk inode does not need this field.
- Pointers to other in-core inodes. The kernel links inodes on hash queues and on a free list in the same way that it links buffers on buffer hash queues and on the buffer free list. A hash queue is identified according to the inode's logical device number and inode number. The kernel can contain at most one in-core copy of a disk inode, but inodes can be simultaneously on a hash queue and on the free list.
- A reference count, indicating the number of instances of the file that are active (such as when *open*ed).

Many fields in the in-core inode are analogous to fields in the buffer header, and the management of inodes is similar to the management of buffers. The inode lock, when set, prevents other processes from accessing the inode; other processes set a flag in the inode when attempting to access it to indicate that they should be awakened when the lock is released. The kernel sets other flags to indicate discrepancies between the disk inode and the in-core copy. When the kernel needs to record changes to the file or to the inode, it writes the in-core copy of the inode to disk after examining these flags.

The most striking difference between an in-core inode and a buffer header is the in-core reference count, which counts the number of active instances of the file. An inode is active when a process allocates it, such as when *open*ing a file. An inode is on the free list only if its reference count is 0, meaning that the kernel can reallocate the in-core inode to another disk inode. The free list of inodes thus serves as a cache of inactive inodes: If a process attempts to access a file whose inode is not currently in the in-core inode pool, the kernel reallocates an in-core inode from the free list for its use. On the other hand, a buffer has no reference count; it is on the free list if and only if it is unlocked.

```
algorithm iget
input:  file system inode number
output: locked inode
{
        while (not done)
        {
                if (inode in inode cache)
                {
                        if (inode locked)
                        {
                                sleep (event inode becomes unlocked);
                                continue;       /* loop back to while */
                        }
                        /* special processing for mount points (Chapter 5) */
                        if (inode on inode free list)
                                remove from free list;
                        increment inode reference count;
                        return (inode);
                }

                /* inode not in inode cache */
                if (no inodes on free list)
                        return(error);
                remove new inode from free list;
                reset inode number and file system;
                remove inode from old hash queue, place on new one;
                read inode from disk (algorithm bread);
                initialize inode (e.g. reference count to 1);
                return(inode);
        }
}
```

Figure 4.3. Algorithm for Allocation of In-Core Inodes

4.1.2 Accessing Inodes

The kernel identifies particular inodes by their file system and inode number and allocates in-core inodes at the request of higher-level algorithms. The algorithm *iget* allocates an in-core copy of an inode (Figure 4.3); it is almost identical to the algorithm *getblk* for finding a disk block in the buffer cache. The kernel maps the device number and inode number into a hash queue and searches the queue for the inode. If it cannot find the inode, it allocates one from the free list and locks it. The kernel then prepares to read the disk copy of the newly accessed inode into the in-core copy. It already knows the inode number and logical device and computes the logical disk block that contains the inode according to how many disk inodes fit into a disk block. The computation follows the formula

block num = ((inode number − 1) / number of inodes per block) +
start block of inode list

where the division operation returns the integer part of the quotient. For example, assuming that block 2 is the beginning of the inode list and that there are 8 inodes per block, then inode number 8 is in disk block 2, and inode number 9 is in disk block 3. If there are 16 inodes in a disk block, then inode numbers 8 and 9 are in disk block 2, and inode number 17 is the first inode in disk block 3.

When the kernel knows the device and disk block number, it reads the block using the algorithm *bread* (Chapter 2), then uses the following formula to compute the byte offset of the inode in the block:

((inode number − 1) modulo (number of inodes per block)) * size of disk inode

For example, if each disk inode occupies 64 bytes and there are 8 inodes per disk block, then inode number 8 starts at byte offset 448 in the disk block. The kernel removes the in-core inode from the free list, places it on the correct hash queue, and sets its in-core reference count to 1. It copies the file type, owner fields, permission settings, link count, file size, and the table of contents from the disk inode to the in-core inode, and returns a locked inode.

The kernel manipulates the inode lock and reference count independently. The lock is set during execution of a system call to prevent other processes from accessing the inode while it is in use (and possibly inconsistent). The kernel releases the lock at the conclusion of the system call: an inode is never locked across system calls. The kernel increments the reference count for every active reference to a file. For example, Section 5.1 will show that it increments the inode reference count when a process *opens* a file. It decrements the reference count only when the reference becomes inactive, for example, when a process *closes* a file. The reference count thus remains set across multiple system calls. The lock is free between system calls to allow processes to share simultaneous access to a file; the reference count remains set between system calls to prevent the kernel from reallocating an active in-core inode. Thus, the kernel can lock and unlock an allocated inode independent of the value of the reference count. System calls other than *open* allocate and release inodes, as will be seen in Chapter 5.

Returning to algorithm *iget*, if the kernel attempts to take an inode from the free list but finds the free list empty, it reports an error. This is different from the philosophy the kernel follows for disk buffers, where a process sleeps until a buffer becomes free: Processes have control over the allocation of inodes at user level via execution of *open* and *close* system calls, and consequently the kernel cannot guarantee when an inode will become available. Therefore, a process that goes to sleep waiting for a free inode to become available may never wake up. Rather than leave such a process "hanging," the kernel fails the system call. However, processes do not have such control over buffers: Because a process cannot keep a buffer locked across system calls, the kernel can guarantee that a buffer will become free soon, and a process therefore sleeps until one is available.

The preceding paragraphs cover the case where the kernel allocated an inode that was not in the inode cache. If the inode is in the cache, the process (A) would find it on its hash queue and check if the inode was currently locked by another process (B). If the inode is locked, process A sleeps, setting a flag in the in-core inode to indicate that it is waiting for the inode to become free. When process B later unlocks the inode, it awakens all processes (including process A) waiting for the inode to become free. When process A is finally able to use the inode, it locks the inode so that other processes cannot allocate it. If the reference count was previously 0, the inode also appears on the free list, so the kernel removes it from there: the inode is no longer free. The kernel increments the inode reference count and returns a locked inode.

To summarize, the *iget* algorithm is used toward the beginning of system calls when a process first accesses a file. The algorithm returns a locked inode structure with reference count 1 greater than it had previously been. The in-core inode contains up-to-date information on the state of the file. The kernel unlocks the inode before returning from the system call so that other system calls can access the inode if they wish. Chapter 5 treats these cases in greater detail.

```
algorithm iput              /* release (put) access to in-core inode */
input:  pointer to in-core inode
output: none
{
        lock inode if not already locked;
        decrement inode reference count;
        if (reference count == 0)
        {
                if (inode link count == 0)
                {
                        free disk blocks for file (algorithm free, section 4.7);
                        set file type to 0;
                        free inode (algorithm ifree, section 4.6);
                }
                if (file accessed or inode changed or file changed)
                        update disk inode;
                put inode on free list;
        }
        release inode lock;
}
```

Figure 4.4. Releasing an Inode

4.1.3 Releasing Inodes

When the kernel releases an inode (algorithm *iput*, Figure 4.4), it decrements its in-core reference count. If the count drops to 0, the kernel writes the inode to disk if the in-core copy differs from the disk copy. They differ if the file data has changed, if the file access time has changed, or if the file owner or access permissions have changed. The kernel places the inode on the free list of inodes, effectively caching the inode in case it is needed again soon. The kernel may also release all data blocks associated with the file and free the inode if the number of links to the file is 0.

4.2 STRUCTURE OF A REGULAR FILE

As mentioned above, the inode contains the table of contents to locate a file's data on disk. Since each block on a disk is addressable by number, the table of contents consists of a set of disk block numbers. If the data in a file were stored in a contiguous section of the disk (that is, the file occupied a linear sequence of disk blocks), then storing the start block address and the file size in the inode would suffice to access all the data in the file. However, such an allocation strategy would not allow for simple expansion and contraction of files in the file system without running the risk of fragmenting free storage area on the disk. Furthermore, the kernel would have to allocate and reserve contiguous space in the file system before allowing operations that would increase the file size.

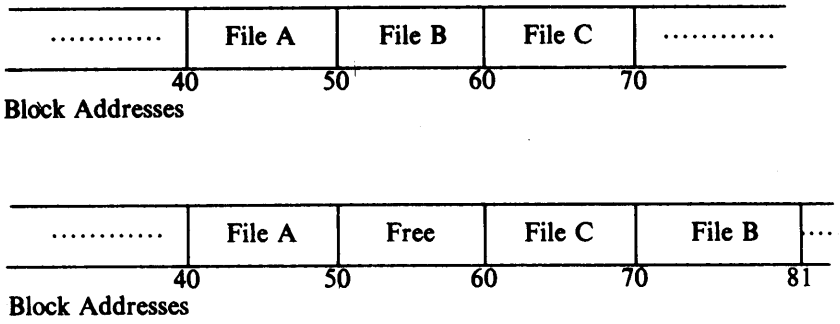


Figure 4.5. Allocation of Contiguous Files and Fragmentation of Free Space

For example, suppose a user creates three files, A, B and C, each consisting of 10 disk blocks of storage, and suppose the system allocated storage for the three files contiguously. If the user then wishes to add 5 blocks of data to the middle file, B, the kernel would have to copy file B to a place in the file system that had room for 15 blocks of storage. Aside from the expense of such an operation, the disk

blocks previously occupied by file B's data would be unusable except for files smaller than 10 blocks (Figure 4.5). The kernel could minimize fragmentation of storage space by periodically running garbage collection procedures to compact available storage, but that would place an added drain on processing power.

For greater flexibility, the kernel allocates file space one block at a time and allows the data in a file to be spread throughout the file system. But this allocation scheme complicates the task of locating the data. The table of contents could consist of a list of block numbers such that the blocks contain the data belonging to the file, but simple calculations show that a linear list of file blocks in the inode is difficult to manage. If a logical block contains 1K bytes, then a file consisting of 10K bytes would require an index of 10 block numbers, but a file containing 100K bytes would require an index of 100 block numbers. Either the size of the inode would vary according to the size of the file, or a relatively low limit would have to be placed on the size of a file.

To keep the inode structure small yet still allow large files, the table of contents of disk blocks conforms to that shown in Figure 4.6. The System V UNIX system runs with 13 entries in the inode table of contents, but the principles are independent of the number of entries. The blocks marked "direct" in the figure contain the numbers of disk blocks that contain real data. The block marked "single indirect" refers to a block that contains a list of direct block numbers. To access the data via the indirect block, the kernel must read the indirect block, find the appropriate direct block entry, and then read the direct block to find the data. The block marked "double indirect" contains a list of indirect block numbers, and the block marked "triple indirect" contains a list of double indirect block numbers.

In principle, the method could be extended to support "quadruple indirect blocks," "quintuple indirect blocks," and so on, but the current structure has sufficed in practice. Assume that a logical block on the file system holds 1K bytes and that a block number is addressable by a 32 bit (4 byte) integer. Then a block can hold up to 256 block numbers. The maximum number of bytes that could be held in a file is calculated (Figure 4.7) at well over 16 gigabytes, using 10 direct blocks and 1 indirect, 1 double indirect, and 1 triple indirect block in the inode. Given that the file size field in the inode is 32 bits, the size of a file is effectively limited to 4 gigabytes (2^{32}).

Processes access data in a file by byte offset. They work in terms of byte counts and view a file as a stream of bytes starting at byte address 0 and going up to the size of the file. The kernel converts the user view of bytes into a view of blocks: The file starts at logical block 0 and continues to a logical block number corresponding to the file size. The kernel accesses the inode and converts the logical file block into the appropriate disk block. Figure 4.8 gives the algorithm *bmap* for converting a file byte offset into a physical disk block.

Consider the block layout for the file in Figure 4.9 and assume that a disk block contains 1024 bytes. If a process wants to access byte offset 9000, the kernel calculates that the byte is in direct block 8 in the file (counting from 0). It then accesses block number 367; the 808th byte in that block (starting from 0) is byte

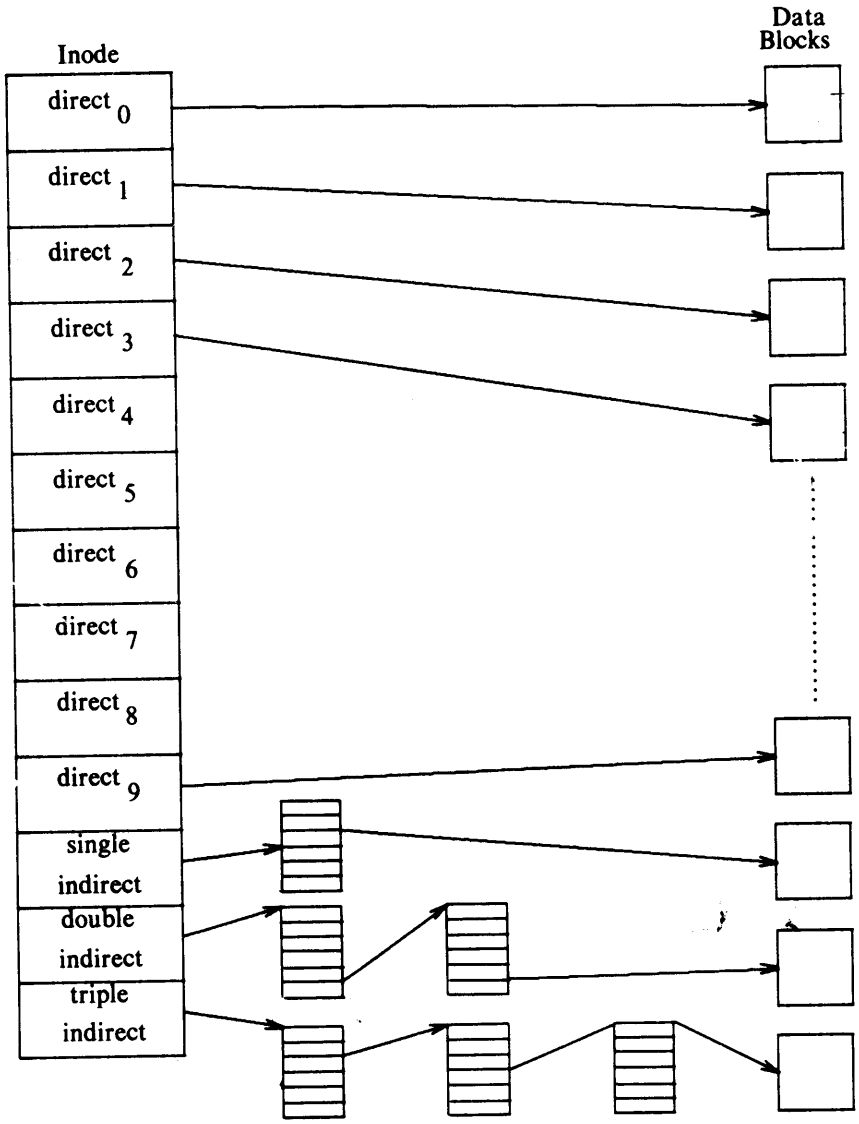


Figure 4.6. Direct and Indirect Blocks in Inode

10 direct blocks with 1K bytes each =	10K bytes
1 indirect block with 256 direct blocks =	256K bytes
1 double indirect block with 256 indirect blocks =	64M bytes
1 triple indirect block with 256 double indirect blocks =	16G bytes

Figure 4.7. Byte Capacity of a File — 1K Bytes Per Block

```
algorithm bmap    /* block map of logical file byte offset to file system block */
input:  (1) inode
        (2) byte offset
output: (1) block number in file system
        (2) byte offset into block
        (3) bytes of I/O in block
        (4) read ahead block number
{
      calculate logical block number in file from byte offset;
      calculate start byte in block for I/O;              /* output 2 */
      calculate number of bytes to copy to user;          /* output 3 */
      check if read-ahead applicable, mark inode;         /* output 4 */
      determine level of indirection;
      while (not at necessary level of indirection)
      {
            calculate index into inode or indirect block from
                        logical block number in file;
            get disk block number from inode or indirect block;
            release buffer from previous disk read, if any (algorithm brelse);
            if (no more levels of indirection)
                  return (block number);
            read indirect disk block (algorithm bread);
            adjust logical block number in file according to level of indirection;
      }
}
```

Figure 4.8. Conversion of Byte Offset to Block Number in File System

9000 in the file. If a process wants to access byte offset 350,000 in the file, it must access a double indirect block, number 9156 in the figure. Since an indirect block has room for 256 block numbers, the first byte accessed via the double indirect block is byte number 272,384 (256K + 10K); byte number 350,000 in a file is therefore byte number 77,616 of the double indirect block. Since each single indirect block accesses 256K bytes, byte number 350,000 must be in the 0th single indirect block of the double indirect block — block number 331. Since each direct block in a single indirect block contains 1K bytes, byte number 77,616 of a single

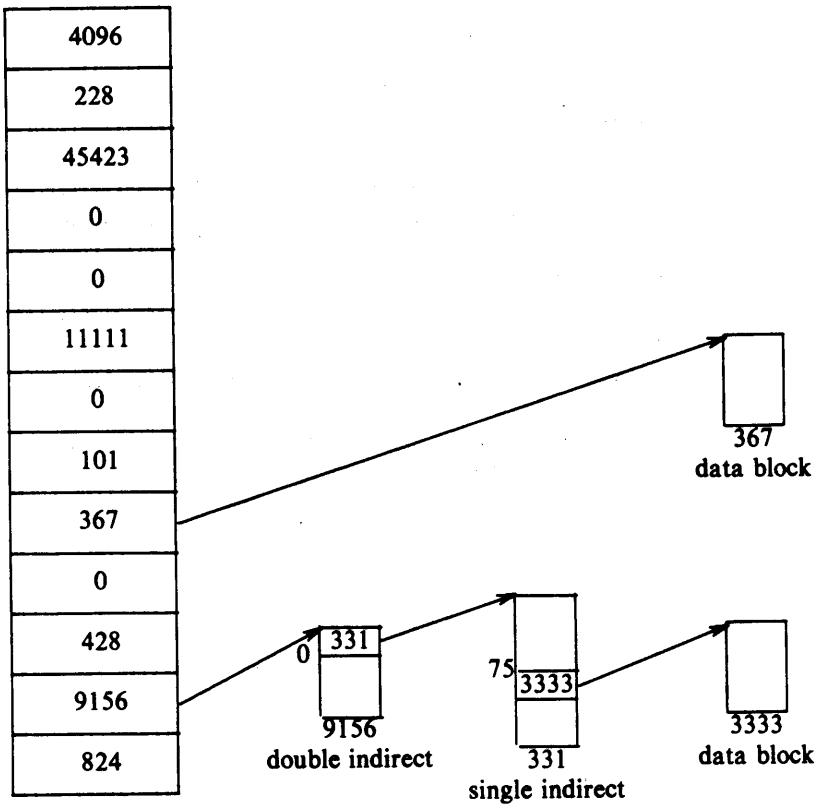


Figure 4.9. Block Layout of a Sample File and its Inode

indirect block is in the 75th direct block in the single indirect block — block number 3333. Finally, byte number 350,000 in the file is at byte number 816 in block 3333.

Examining Figure 4.9 more closely, several block entries in the inode are 0, meaning that the logical block entries contain no data. This happens if no process ever wrote data into the file at any byte offsets corresponding to those blocks and hence the block numbers remain at their initial value, 0. No disk space is wasted for such blocks. Processes can cause such a block layout in a file by using the *lseek* and *write* system calls, as described in the next chapter. The next chapter also describes how the kernel takes care of *read* system calls that access such blocks.

The conversion of a large byte offset, particularly one that is referenced via the triple indirect block, is an arduous procedure that could require the kernel to access three disk blocks in addition to the inode and data block. Even if the kernel finds

the blocks in the buffer cache, the operation is still expensive, because the kernel must make multiple requests of the buffer cache and may have to sleep awaiting locked buffers. How effective is the algorithm in practice? That depends on how the system is used and whether the user community and job mix are such that the kernel accesses large files or small files more frequently. It has been observed [Mullender 84], however, that most files on UNIX systems contain less than 10K bytes, and many contain less than 1K bytes![1] Since 10K bytes of a file are stored in direct blocks, most file data can be accessed with one disk access. So in spite of the fact that accessing large files is an expensive operation, accessing common-sized files is fast.

Two extensions to the inode structure just described attempt to take advantage of file size characteristics. A major principle in the 4.2 BSD file system implementation [McKusick 84] is that the more data the kernel can access on the disk in a single operation, the faster file access becomes. That argues for having larger logical disk blocks, and the Berkeley implementation allows logical disk blocks of 4K or 8K bytes. But having larger block sizes on disk increases block fragmentation, leaving large portions of disk space unused. For instance, if the logical block size is 8K bytes, then a file of size 12K bytes uses 1 complete block and half of a second block. The other half of the second block (4K bytes) is wasted; no other file can use the space for data storage. If the sizes of files are such that the number of bytes in the last block of a file is uniformly distributed, then the average wasted space is half a block per file; the amount of wasted disk space can be as high as 45% for a file system with logical blocks of size 4K bytes [McKusick 84]. The Berkeley implementation remedies the situation by allocating a block *fragment* to contain the last data in a file. One disk block can contain fragments belonging to several files. An exercise in Chapter 5 explores some details of the implementation.

The second extension to the classic inode structure described here is to store file data in the inode (see [Mullender 84]). By expanding the inode to occupy an entire disk block, a small portion of the block can be used for the inode structures and the remainder of the block can store the entire file, in many cases, or the end of a file otherwise. The main advantage is that only one disk access is necessary to get the inode and its data if the file fits in the inode block.

---

1. For a sample of 19,978 files, Mullender and Tannenbaum say that approximately 85% of the files were smaller than 8K bytes and that 48% were smaller than 1K bytes. Although these percentages will vary from one installation to the next, they are representative of many UNIX systems.

4.3 DIRECTORIES

Recall from Chapter 1 that directories are the files that give the file system its hierarchical structure; they play an important role in conversion of a file name to an inode number. A directory is a file whose data is a sequence of entries, each consisting of an inode number and the name of a file contained in the directory. A path name is a null terminated character string divided into separate components by the slash ("/") character. Each component except the last must be the name of a directory, but the last component may be a non-directory file. UNIX System V restricts component names to a maximum of 14 characters; with a 2 byte entry for the inode number, the size of a directory entry is 16 bytes.

Byte Offset in Directory	Inode Number (2 bytes)	File Names
0	83	
16	2	..
32	1798	init
48	1276	fsck
64	85	clri
80	1268	motd
96	1799	mount
112	88	mknod
128	2114	passwd
144	1717	umount
160	1851	checklist
176	92	fsdb1b
192	84	config
208	1432	getty
224	0	crash
240	95	mkfs
256	188	inittab

Figure 4.10. Directory Layout for /etc

Figure 4.10 depicts the layout of the directory "etc". Every directory contains the file names dot and dot-dot ("." and "..") whose inode numbers are those of the directory and its parent directory, respectively. The inode number of "." in "/etc" is located at offset 0 in the file, and its value is 83. The inode number of ".." is located at offset 16, and its value is 2. Directory entries may be empty, indicated by an inode number of 0. For instance, the entry at address 224 in "/etc" is empty, although it once contained an entry for a file named "crash". The program *mkfs* initializes a file system so that "." and ".." of the root directory have the root inode number of the file system.

The kernel stores data for a directory just as it stores data for an ordinary file, using the inode structure and levels of direct and indirect blocks. Processes may read directories in the same way they read regular files, but the kernel reserves exclusive right to write a directory, thus insuring its correct structure. The access permissions of a directory have the following meaning: read permission on a directory allows a process to read a directory; write permission allows a process to create new directory entries or remove old ones (via the *creat*, *mknod*, *link*, and *unlink* system calls), thereby altering the contents of the directory; execute permission allows a process to search the directory for a file name (it is meaningless execute a directory). Exercise 4.6 explores the difference between reading and searching a directory.

4.4 CONVERSION OF A PATH NAME TO AN INODE

The initial access to a file is by its path name, as in the *open*, *chdir* (change directory), or *link* system calls. Because the kernel works internally with inodes rather than with path names, it converts the path names to inodes to access files. The algorithm *namei* parses the path name one component at a time, converting each component into an inode based on its name and the directory being searched, and eventually returns the inode of the input path name (Figure 4.11).

Recall from Chapter 2 that every process is associated with (resides in) a current directory; the *u area* contains a pointer to the current directory inode. The current directory of the first process in the system, process 0, is the root directory. The current directory of every other process starts out as the current directory of its parent process at the time it was created (see Section 5.10). Processes change their current directory by executing the *chdir* (change directory) system call. All path name searches start from the current directory of the process unless the path name starts with the slash character, signifying that the search should start from the root directory. In either case, the kernel can easily find the inode where the path name search starts: The current directory is stored in the process *u area*, and the system root inode is stored in a global variable.[2]

Namei uses intermediate inodes as it parses a path name; call them working inodes. The inode where the search starts is the first working inode. During each iteration of the *namei* loop, the kernel makes sure that the working inode is indeed that of a directory. Otherwise, the system would violate the assertion that non-directory files can only be leaf nodes of the file system tree. The process must also have permission to search the directory (read permission is insufficient). The user ID of the process must match the owner or group ID of the file, and execute

2. A process can execute the *chroot* system call to change its notion of the file system root. The changed root is stored in the *u area*.

```
algorithm namei       /* convert path name to inode */
input:  path name
output: locked inode
{
      if (path name starts from root)
            working inode = root inode (algorithm iget);
      else
            working inode = current directory inode (algorithm iget);

      while (there is more path name)
      {
            read next path name component from input;
            verify that working inode is of directory, access permissions OK;
            if (working inode is of root and component is "..")
                  continue;       /* loop back to while */
            read directory (working inode) by repeated use of algorithms
                  bmap, bread and brelse;
            if (component matches an entry in directory (working inode))
            {
                  get inode number for matched component;
                  release working inode (algorithm iput);
                  working inode = inode of matched component (algorithm iget);
            }
            else      /* component not in directory */
                  return (no inode);
      }
      return (working inode);
}
```

Figure 4.11. Algorithm for Conversion of a Path Name to an Inode

permission must be granted, or the file must allow search to all users. Otherwise the search fails.

The kernel does a linear search of the directory file associated with the working inode, trying to match the path name component to a directory entry name. Starting at byte offset 0, it converts the byte offset in the directory to the appropriate disk block according to algorithm *bmap* and reads the block using algorithm *bread*. It searches the block for the path name component, treating the contents of the block as a sequence of directory entries. If it finds a match, it records the inode number of the matched directory entry, releases the block (algorithm *brelse*) and the old working inode (algorithm *iput*), and allocates the inode of the matched component (algorithm *iget*). The new inode becomes the working inode. If the kernel does not match the path name with any names in the block, it releases the block, adjusts the byte offset by the number of bytes in a block, converts the new offset to a disk block number (algorithm *bmap*), and reads

the next block. The kernel repeats the procedure until it matches the path name component with a directory entry name, or until it reaches the end of the directory.

For example, suppose a process wants to open the file "/etc/passwd". When the kernel starts parsing the file name, it encounters "/" and gets the system root inode. Making root its current working inode, the kernel gathers in the string "etc". After checking that the current inode is that of a directory ("/") and that the process has the necessary permissions to search it, the kernel searches root for a file whose name is "etc": It accesses the data in the root directory block by block and searches each block one entry at a time until it locates an entry for "etc". On finding the entry, the kernel releases the inode for root (algorithm *iput*) and allocates the inode for "etc" (algorithm *iget*) according to the inode number of the entry just found. After ascertaining that "etc" is a directory and that it has the requisite search permissions, the kernel searches "etc" block by block for a directory structure entry for the file "passwd". Referring to Figure 4.10, it would find the entry for "passwd" as the ninth entry of the directory. On finding it, the kernel releases the inode for "etc", allocates the inode for "passwd", and — since the path name is exhausted — returns that inode.

It is natural to question the efficiency of a linear search of a directory for a path name component. Ritchie points out (see page 1968 of [Ritchie 78b]) that a linear search is efficient because it is bounded by the size of the directory. Furthermore, early UNIX system implementations did not run on machines with large memory space, so there was heavy emphasis on simple algorithms such as linear search schemes. More complicated search schemes could require a different, more complex, directory structure, and would probably run more slowly on small directories than the linear search scheme.

4.5 SUPER BLOCK

So far, this chapter has described the structure of a file, assuming that the inode was previously bound to a file and that the disk blocks containing the data were already assigned. The next sections cover how the kernel assigns inodes and disk blocks. To understand those algorithms, let us examine the structure of the super block.

The super block consists of the following fields:

- the size of the file system,
- the number of free blocks in the file system,
- a list of free blocks available on the file system,
- the index of the next free block in the free block list,
- the size of the inode list,
- the number of free inodes in the file system,
- a list of free inodes in the file system,
- the index of the next free inode in the free inode list,

- lock fields for the free block and free inode lists,
- a flag indicating that the super block has been modified.

The remainder of this chapter will explain the use of the arrays, indices and locks. The kernel periodically writes the super block to disk if it had been modified so that it is consistent with the data in the file system.

4.6 INODE ASSIGNMENT TO A NEW FILE

The kernel uses algorithm *iget* to allocate a known inode, one whose (file system and) inode number was previously determined. In algorithm *namei* for instance, the kernel determines the inode number by matching a path name component to a name in a directory. Another algorithm, *ialloc*, assigns a disk inode to a newly created file.

The file system contains a linear list of inodes, as mentioned in Chapter 2. An inode is free if its type field is zero. When a process needs a new inode, the kernel could theoretically search the inode list for a free inode. However, such a search would be expensive, requiring at least one read operation (possibly from disk) for every inode. To improve performance, the file system super block contains an array to cache the numbers of free inodes in the file system.

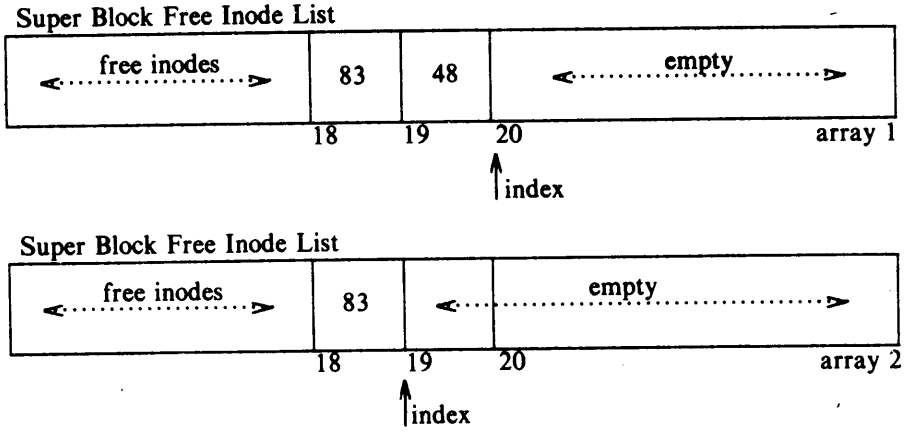
Figure 4.12 shows the algorithm *ialloc* for assigning new inodes. For reasons cited later, the kernel first verifies that no other processes have locked access to the super block free inode list. If the list of inode numbers in the super block is not empty, the kernel assigns the next inode number, allocates a free in-core inode for the newly assigned disk inode using algorithm *iget* (reading the inode from disk if necessary), copies the disk inode to the in-core copy, initializes the fields in the inode, and returns the locked inode. It updates the disk inode to indicate that the inode is now in use: A non-zero file type field indicates that the disk inode is assigned. In the simplest case, the kernel has a good inode, but race conditions exist that necessitate more checking, as will be explained shortly. Loosely defined, a race condition arises when several processes alter common data structures such that the resulting computations depend on the order in which the processes executed, even though all processes obeyed the locking protocol. For example, it is implied here that a process could get a used inode. A race condition is related to the mutual exclusion problem defined in Chapter 2, except that locking schemes solve the mutual exclusion problem there but may not, by themselves, solve all race conditions.

If the super block list of free inodes is empty, the kernel searches the disk and places as many free inode numbers as possible into the super block. The kernel reads the inode list on disk, block by block, and fills the super block list of inode numbers to capacity, remembering the highest-numbered inode that it finds. Call that inode the "remembered" inode; it is the last one saved in the super block. The next time the kernel searches the disk for free inodes, it uses the remembered inode as its starting point, thereby assuring that it wastes no time reading disk blocks

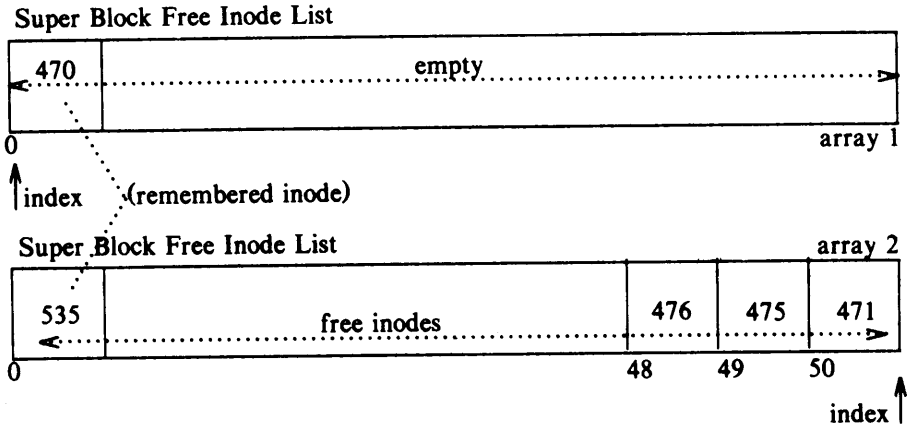
```
algorithm ialloc        /* allocate inode */
input:  file system
output: locked inode
{
        while (not done)
        {
                if (super block locked)
                {
                        sleep (event super block becomes free);
                        continue;       /* while loop */
                }
                if (inode list in super block is empty)
                {
                        lock super block;
                        get remembered inode for free inode search;
                        search disk for free inodes until super block full,
                                or no more free inodes (algorithms bread and brelse);
                        unlock super block;
                        wake up (event super block becomes free);
                        if (no free inodes found on disk)
                                return (no inode);
                        set remembered inode for next free inode search;
                }
                /* there are inodes in super block inode list */
                get inode number from super block inode list;
                get inode (algorithm iget);
                if (inode not free after all)       /* !!! */
                {
                        write inode to disk;
                        release inode (algorithm iput);
                        continue;       /* while loop */
                }
                /* inode is free */
                initialize inode;
                write inode to disk;
                decrement file system free inode count;
                return (inode);
        }
}
```

Figure 4.12. Algorithm for Assigning New Inodes

where no free inodes should exist. After gathering a fresh set of free inode numbers, it starts the inode assignment algorithm from the beginning. Whenever the kernel assigns a disk inode, it decrements the free inode count recorded in the super block.



(a) Assigning Free Inode from Middle of List



(b) Assigning Free Inode - Super Block List Empty

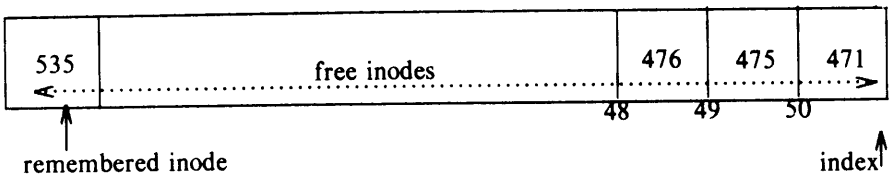
Figure 4.13. Two Arrays of Free Inode Numbers

Consider the two pairs of arrays of free inode numbers in Figure 4.13. If the list of free inodes in the super block looks like the first array in Figure 4.13(a) when the kernel assigns an inode, it decrements the index for the next valid inode number to 18 and takes inode number 48. If the list of free inodes in the super block looks like the first array in Figure 4.13(b), it will notice that the array is empty and search the disk for free inodes, starting from inode number 470, the remembered inode. When the kernel fills the super block free list to capacity, it remembers the last inode as the start point for the next search of the disk. The kernel assigns an inode it just took from the disk (number 471 in the figure) and continues whatever it was doing.

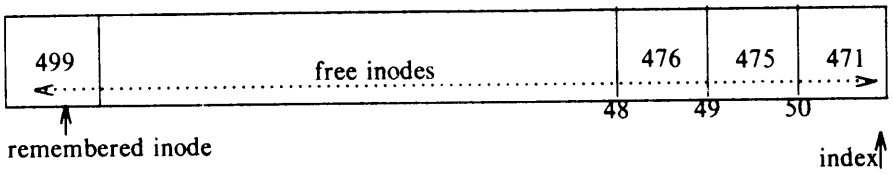
```
algorithm ifree        /* inode free */
input:  file system inode number
output: none
{
        increment file system free inode count;
        if (super block locked)
                return;
        if (inode list full)
        {
                if (inode number less than remembered inode for search)
                        set remembered inode for search = input inode number;
        }
        else
                store inode number in inode list;
        return;
}
```

Figure 4.14. Algorithm for Freeing Inode

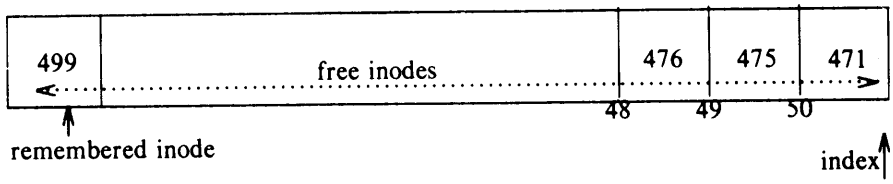
The algorithm for freeing an inode is much simpler. After incrementing the total number of available inodes in the file system, the kernel checks the lock on the super block. If locked, it avoids race conditions by returning immediately: The inode number is not put into the super block, but it can be found on disk and is available for reassignment. If the list is not locked, the kernel checks if it has room for more inode numbers and, if it does, places the inode number in the list and returns. If the list is full, the kernel may not save the newly freed inode there: It compares the number of the freed inode with that of the remembered inode. If the freed inode number is less than the remembered inode number, it "remembers" the newly freed inode number discarding the old remembered inode number from the super block. The inode is not lost, because the kernel can find it by searching the inode list on disk. The kernel maintains the super block list such that the last inode it dispenses from the list is the remembered inode. Ideally, there should never be free inodes whose inode number is less than the remembered inode number, but



(a) Original Super Block List of Free Inodes



(b) Free Inode 499



(c) Free Inode 601

Figure 4.15. Placing Free Inode Numbers into the Super Block

exceptions are possible.

Consider two examples of freeing inodes. If the super block list of free inodes has room for more free inode numbers as in Figure 4.13(a), the kernel places the inode number on the list, increments the index to the next free inode, and proceeds. But if the list of free inodes is full as in Figure 4.15, the kernel compares the inode number it has freed to the remembered inode number that will start the next disk search. Starting with the free inode list in Figure 4.15(a), if the kernel frees inode 499, it makes 499 the remembered inode and evicts number 535 from the free list. If the kernel then frees inode number 601, it does not change the contents of the free list. When it later uses up the inodes in the super block free list, it will search the disk for free inodes starting from inode number 499, and find inodes 535 and 601 again.

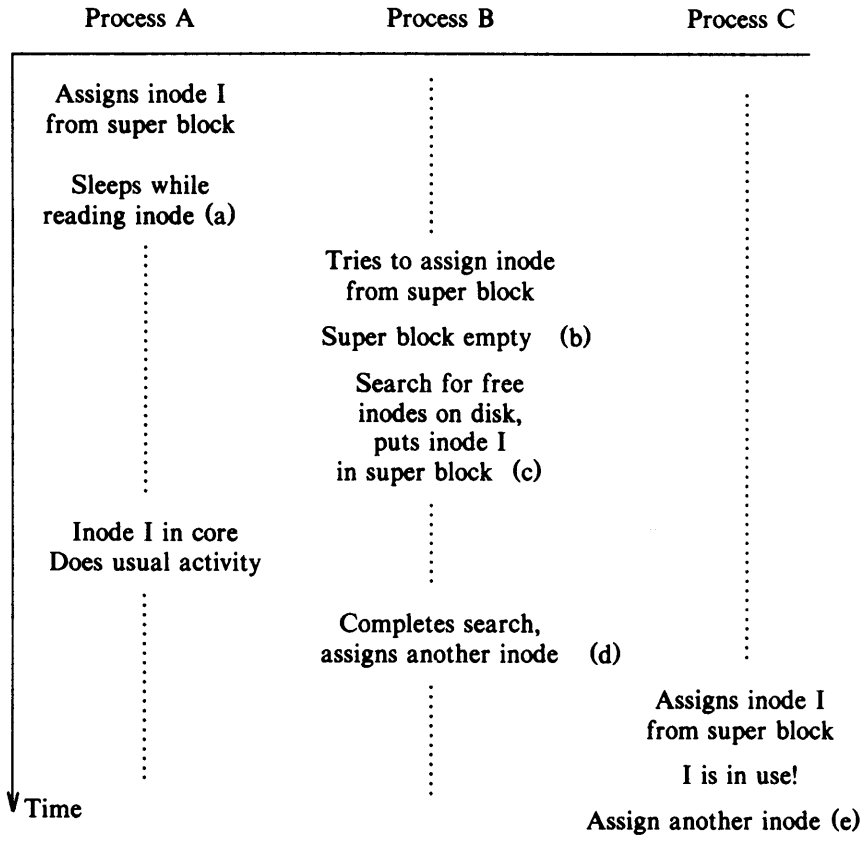


Figure 4.16. Race Condition in Assigning Inodes

The preceding paragraph described the simple cases of the algorithms. Now consider the case where the kernel assigns a new inode and then allocates an in-core copy for the inode. The algorithm implies that the kernel could find that the inode had already been assigned. Although rare, the following scenario shows such a case (refer to Figures 4.16 and 4.17). Consider three processes, A, B, and C, and suppose that the kernel, acting on behalf of process A,[3] assigns inode I but goes to sleep before it copies the disk inode into the in-core copy. Algorithms *iget* (invoked

3. As in the last chapter, the term "process" here will mean "the kernel, acting on behalf of a process."

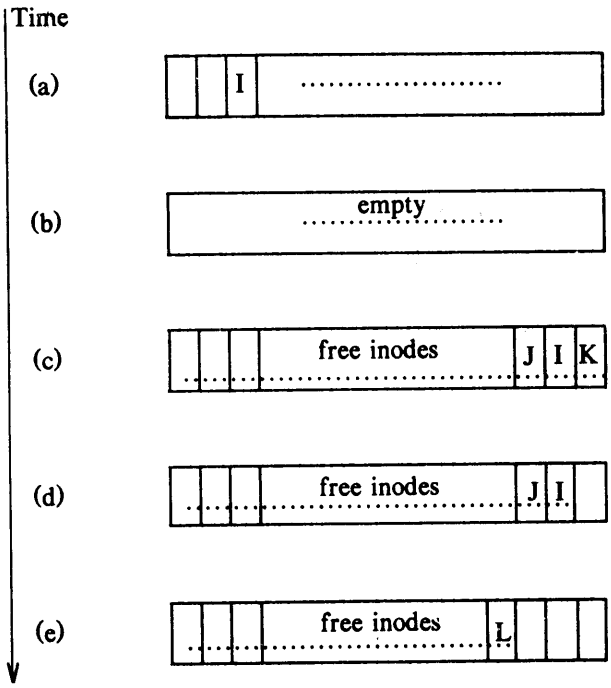


Figure 4.17. Race Condition in Assigning Inodes (continued)

by *ialloc*) and *bread* (invoked by *iget*) give process A ample opportunity to go to sleep. While process A is asleep, suppose process B attempts to assign a new inode but discovers that the super block list of free inodes is empty. Process B searches the disk for free inodes, and suppose it starts its search for free inodes at an inode number lower than that of the inode that A is assigning. It is possible for process B to find inode I free on the disk since process A is still asleep, and the kernel does not know that the inode is about to be assigned. Process B, not realizing the danger, completes its search of the disk, fills up the super block with (supposedly) free inodes, assigns an inode, and departs from the scene. However, inode I is in the super block free list of inode numbers. When process A wakes up, it completes the assignment of inode I. Now suppose process C later requests an inode and happens to pick inode I from the super block free list. When it gets the in-core copy of the inode, it will find its file type set, implying that the inode was already assigned. The kernel checks for this condition and, finding that the inode has been assigned, tries to assign a new one. Writing the updated inode to disk immediately after its assignment in *ialloc* makes the chance of the race smaller, because the file type field will mark the inode in use.

Locking the super block list of inodes while reading in a new set from disk prevents other race conditions. If the super block list were not locked, a process could find it empty and try to populate it from disk, occasionally sleeping while waiting for I/O completion. Suppose a second process also tried to assign a new inode and found the list empty. It, too, would try to populate the list from disk. At best, the two processes are duplicating their efforts and wasting CPU power. At worst, race conditions of the type described in the previous paragraph would be more frequent. Similarly, if a process freeing an inode did not check that the list is locked, it could overwrite inode numbers already in the free list while another process was populating it from disk. Again, the race conditions described above would be more frequent. Although the kernel handles them satisfactorily, system performance would suffer. Use of the lock on the super block free list prevents such race conditions.

4.7 ALLOCATION OF DISK BLOCKS

When a process writes data to a file, the kernel must allocate disk blocks from the file system for direct data blocks and, sometimes, for indirect blocks. The file system super block contains an array that is used to cache the numbers of free disk blocks in the file system. The utility program *mkfs* (make file system) organizes the data blocks of a file system in a linked list, such that each link of the list is a disk block that contains an array of free disk block numbers, and one array entry is the number of the next block of the linked list. Figure 4.18 shows an example of the linked list, where the first block is the super block free list and later blocks on the linked list contain more free block numbers.

When the kernel wants to allocate a block from a file system (algorithm *alloc*, Figure 4.19), it allocates the next available block in the super block list. Once allocated, the block cannot be reallocated until it becomes free. If the allocated block is the last available block in the super block cache, the kernel treats it as a pointer to a block that contains a list of free blocks. It reads the block, populates the super block array with the new list of block numbers, and then proceeds to use the original block number. It allocates a buffer for the block and clears the buffer's data (zeros it). The disk block has now been assigned, and the kernel has a buffer to work with. If the file system contains no free blocks, the calling process receives an error.

If a process *writes* a lot of data to a file, it repeatedly asks the system for blocks to store the data, but the kernel assigns only one block at a time. The program *mkfs* tries to organize the original linked list of free block numbers so that block numbers dispensed to a file are near each other. This helps performance, because it reduces disk seek time and latency when a process reads a file sequentially. Figure 4.18 depicts block numbers in a regular pattern, presumably based on the disk rotation speed. Unfortunately, the order of block numbers on the free block linked lists breaks down with heavy use as processes *write* files and remove them, because block numbers enter and leave the free list at random. The kernel makes no

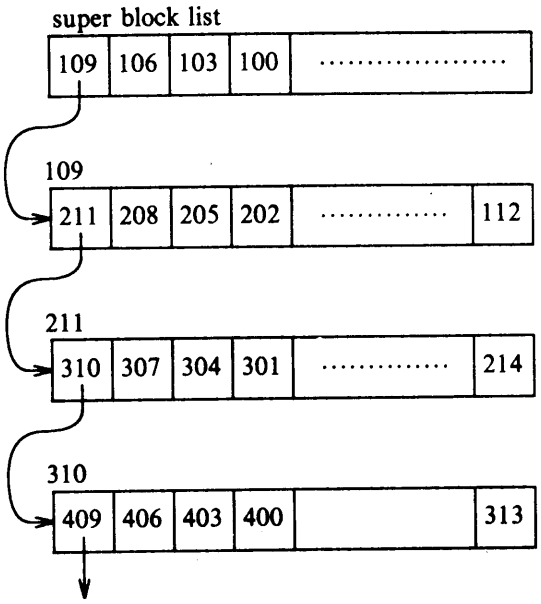


Figure 4.18. Linked List of Free Disk Block Numbers

attempt to sort block numbers on the free list.

The algorithm *free* for freeing a block is the reverse of the one for allocating a block. If the super block list is not full, the block number of the newly freed block is placed on the super block list. If, however, the super block list is full, the newly freed block becomes a link block; the kernel writes the super block list into the block and writes the block to disk. It then places the block number of the newly freed block in the super block list: That block number is the only member of the list.

Figure 4.20 shows a sequence of *alloc* and *free* operations, starting with one entry on the super block free list. The kernel frees block 949 and places the block number on the free list. It then allocates a block and removes block number 949 from the free list. Finally, it allocates a block and removes block number 109 from the free list. Because the super block free list is now empty, the kernel replenishes the list by copying in the contents of block 109, the next link on the linked list. Figure 4.20(d) shows the full super block list and the next link block, block 211.

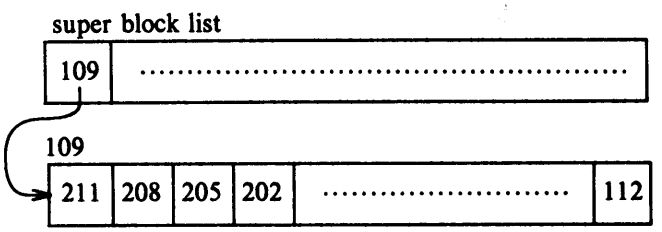
The algorithms for assigning and freeing inodes and disk blocks are similar in that the kernel uses the super block as a cache containing indices of free resources, block numbers, and inode numbers. It maintains a linked list of block numbers such that every free block number in the file system appears in some element of the linked list, but it maintains no such list of free inodes. There are three reasons for

```
algorithm alloc     /* file system block allocation */
input:   file system number
output: buffer for new block
{
        while (super block locked)
                sleep (event super block not locked);
        remove block from super block free list;
        if (removed last block from free list)
        {
                lock super block;
                read block just taken from free list (algorithm bread);
                copy block numbers in block into super block;
                release block buffer (algorithm brelse);
                unlock super block;
                wake up processes (event super block not locked);
        }
        get buffer for block removed from super block list (algorithm getblk);
        zero buffer contents;
        decrement total count of free blocks;
        mark super block modified;
        return buffer;
}
```

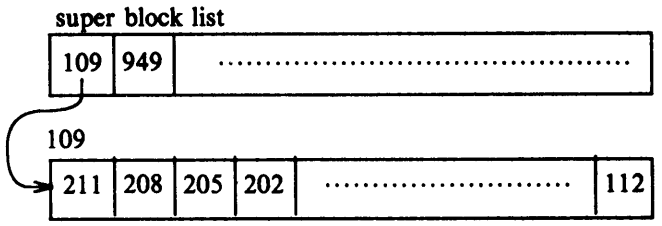
Figure 4.19. Algorithm for Allocating Disk Block

the different treatment.

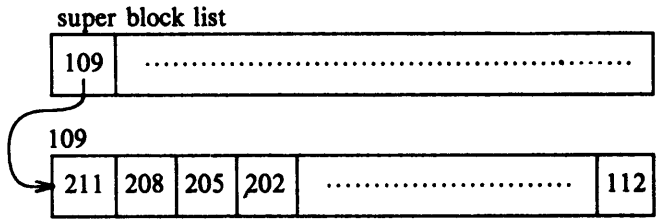
1. The kernel can determine whether an inode is free by inspection: If the file type field is clear, the inode is free. The kernel needs no other mechanism to describe free inodes. However, it cannot determine whether a block is free just by looking at it. It could not distinguish between a bit pattern that indicates the block is free and data that happened to have that bit pattern. Hence, the kernel requires an external method to identify free blocks, and traditional implementations have used a linked list.
2. Disk blocks lend themselves to the use of linked lists: A disk block easily holds large lists of free block numbers. But inodes have no convenient place for bulk storage of large lists of free inode numbers.
3. Users tend to consume disk block resources more quickly than they consume inodes, so the apparent lag in performance when searching the disk for free inodes is not as critical as it would be for searching for free disk blocks.



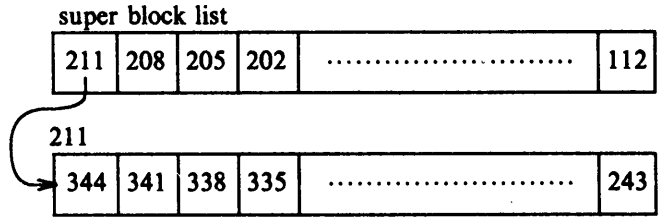
(a) Original configuration



(b) After freeing block number 949



(c) After assigning block number (949)



(d) After assigning block number (109)
replenish super block free list

Figure 4.20. Requesting and Freeing Disk Blocks

4.8 OTHER FILE TYPES

The UNIX system supports two other file types: pipes and special files. A pipe, sometimes called a *fifo* (for "first-in-first-out"), differs from a regular file in that its data is transient: Once data is read from a pipe, it cannot be read again. Also, the data is read in the order that it was written to the pipe, and the system allows no deviation from that order. The kernel stores data in a pipe the same way it stores data in an ordinary file, except that it uses only the direct blocks, not the indirect blocks. The next chapter will examine the implementation of pipes.

The last file types in the UNIX system are special files, including block device special files and character device special files. Both types specify devices, and therefore the file inodes do not reference any data. Instead, the inode contains two numbers known as the major and minor device numbers. The major number indicates a device type such as terminal or disk, and the minor number indicates the unit number of the device. Chapter 10 examines special devices in detail.

4.9 SUMMARY

The inode is the data structure that describes the attributes of a file, including the layout of its data on disk. There are two versions of the inode: the disk copy that stores the inode information when the file is not in use and the in-core copy that records information about active files. Algorithms *ialloc* and *ifree* control assignment of a disk inode to a file during the *creat*, *mknod*, *pipe*, and *unlink* system calls (next chapter), and the algorithms *iget* and *iput* control the allocation of in-core inodes when a process accesses a file. Algorithm *bmap* locates the disk blocks of a file, according to a previously supplied byte offset in the file. Directories are files that correlate file name components to inode numbers. Algorithm *namei* converts file names manipulated by processes to inodes, used internally by the kernel. Finally, the kernel controls assignment of new disk blocks to a file using algorithms *alloc* and *free*.

The data structures discussed in this chapter consist of linked lists, hash queues, and linear arrays, and the algorithms that manipulate the data structures are therefore simple. Complications arise due to race conditions caused by the interaction of the algorithms, and the text has indicated some of these timing problems. Nevertheless, the algorithms are not elaborate and illustrate the simplicity of the system design.

The structures and algorithms explained here are internal to the kernel and are not visible to the user. Referring to the overall system architecture (Figure 2.1), the algorithms described in this chapter occupy the lower half of the file subsystem. The next chapter examines the system calls that provide the user interface to the file system, and it describes the upper half of the file subsystem that invokes the internal algorithms described here.

4.10 EXERCISES

1. The C language convention counts array indices from 0. Why do inode numbers start from 1 and not 0?
2. If a process sleeps in algorithm *iget* when it finds the inode locked in the cache, why must it start the loop again from the beginning after waking up?
3. Describe an algorithm that takes an in-core inode as input and updates the corresponding disk inode.
4. The algorithms *iget* and *iput* do not require the processor execution level to be raised to block out interrupts. What does this imply?
5. How efficiently can the loop for indirect blocks in *bmap* be encoded?

```
mkdir junk
for i in 1 2 3 4 5
do
echo hello > junk/$i
done
ls -ld junk
ls -l junk
chmod -r junk
ls -ld junk
ls junk
ls -l junk
cd junk
pwd
ls -l
echo *
cd ..
chmod +r junk
chmod -x junk
ls junk
ls -l junk
cd junk
chmod +x junk
```

Figure 4.21. Difference between Read and Search Permission on Directories

6. Execute the shell command script in Figure 4.21. It creates a directory "junk" and creates five files in the directory. After doing some control *ls* commands, the *chmod* command turns off read permission for the directory. What happens when the various *ls* commands are executed now? What happens after changing directory into "junk"? After restoring read permission but removing execute (search) permission from "junk", repeat the experiment. What happens? What is happening in the kernel to cause this behavior?
7. Given the current structure of a directory entry on a System V system, what is the maximum number of files a file system can contain?

8. UNIX System V allows a maximum of 14 characters for a path name component. *Namei* truncates extra characters in a component. How should the file system and respective algorithms be redesigned to allow arbitrary length component names?
9. Suppose a user has a private version of the UNIX system but changes it so that a path name component can consist of 30 characters; the private version of the operating system stores the directory entries the same way that the standard operating system does, except that the directory entries are 32 bytes long instead of 16. If the user mounts the private file system on a standard system, what would happen in algorithm *namei* when a process accesses a file on the private file system?
* 10. Consider the algorithm *namei* for converting a path name into an inode. As the search progresses, the kernel checks that the current working inode is that of a directory. Is it possible for another process to remove (*unlink*) the directory? How can the kernel prevent this? The next chapter will come back to this problem.
* 11. Design a directory structure that improves the efficiency of searching for path names by avoiding the linear search. Consider two techniques: hashing and n-ary trees.
* 12. Design a scheme that reduces the number of directory searches for file names by caching frequently used names.
* 13. Ideally, a file system should never contain a free inode whose inode number is less than the "remembered" inode used by *ialloc*. How is it possible for this assertion to be false?
14. The super block is a disk block and contains other information besides the free block list, as described in this chapter. Therefore, the super block free list cannot contain as many free block numbers as can be potentially stored in a disk block on the linked list of free disk blocks. What is the optimal number of free block numbers that should be stored in a block on the linked list?
* 15. Discuss a system implementation that keeps track of free disk blocks with a bit map instead of a linked list of blocks. What are the advantages and disadvantages of this scheme?

5

SYSTEM CALLS FOR THE FILE SYSTEM

The last chapter described the internal data structures for the file system and the algorithms that manipulate them. This chapter deals with system calls for the file system, using the concepts explored in the previous chapter. It starts with system calls for accessing existing files, such as *open*, *read*, *write*, *lseek*, and *close*, then presents system calls to create new files, namely, *creat* and *mknod*, and then examines the system calls that manipulate the inode or that maneuver through the file system: *chdir*, *chroot*, *chown*, *chmod*, *stat*, and *fstat*. It investigates more advanced system calls: *pipe* and *dup* are important for the implementation of pipes in the shell; *mount* and *umount* extend the file system tree visible to users; *link* and *unlink* change the structure of the file system hierarchy. Then, it presents the notion of file system abstractions, allowing the support of various file systems as long as they conform to standard interfaces. The last section in the chapter covers file system maintenance. The chapter introduces three kernel data structures: the file table, with one entry allocated for every opened file in the system, the user file descriptor table, with one entry allocated for every file descriptor known to a process, and the mount table, containing information for every active file system.

Figure 5.1 shows the relationship between the system calls and the algorithms described previously. It classifies the system calls into several categories, although some system calls appear in more than one category:

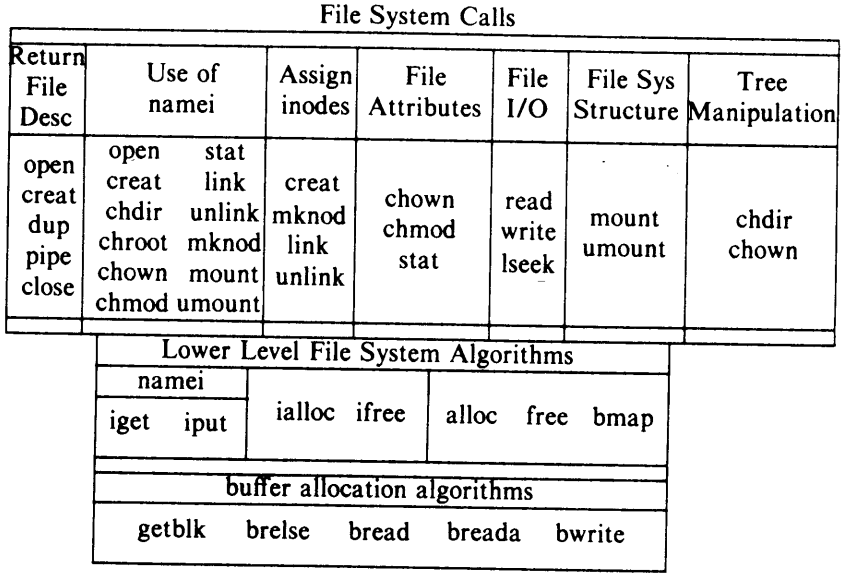


Figure 5.1. File System Calls and Relation to Other Algorithms

- System calls that return file descriptors for use in other system calls;
- System calls that use the *namei* algorithm to parse a path name;
- System calls that assign and free inodes, using algorithms *ialloc* and *ifree*;
- System calls that set or change the attributes of a file;
- System calls that do I/O to and from a process, using algorithms *alloc*, *free*, and the buffer allocation algorithms;
- System calls that change the structure of the file system;
- System calls that allow a process to change its view of the file system tree.

5.1 OPEN

The *open* system call is the first step a process must take to access the data in a file. The syntax for the *open* system call is

```
fd = open(pathname, flags, modes);
```

where *pathname* is a file name, *flags* indicate the type of open (such as for reading or writing), and *modes* give the file permissions if the file is being created. The *open* system call returns an integer[1] called the user *file descriptor*. Other file

operations, such as reading, writing, seeking, duplicating the file descriptor, setting file I/O parameters, determining file status, and closing the file, use the file descriptor that the *open* system call returns.

The kernel searches the file system for the file name parameter using algorithm *namei* (see Figure 5.2). It checks permissions for opening the file after it finds the in-core inode and allocates an entry in the file table for the open file. The file table entry contains a pointer to the inode of the open file and a field that indicates the byte offset in the file where the kernel expects the next *read* or *write* to begin. The kernel initializes the offset to 0 during the *open* call, meaning that the initial *read* or *write* starts at the beginning of a file by default. Alternatively, a process can *open* a file in *write-append* mode, in which case the kernel initializes the offset to the size of the file. The kernel allocates an entry in a private table in the process *u area*, called the user file descriptor table, and notes the index of this entry. The index is the file descriptor that is returned to the user. The entry in the user file table points to the entry in the global file table.

```
algorithm open
inputs: file name
        type of open
        file permissions (for creation type of open)
output: file descriptor
{
      convert file name to inode (algorithm namei);
      if (file does not exist or not permitted access)
            return(error);
      allocate file table entry for inode, initialize count, offset;
      allocate user file descriptor entry, set pointer to file table entry;
      if (type of open specifies truncate file)
            free all file blocks (algorithm free);
      unlock(inode);              /* locked above in namei */
      return(user file descriptor);
}
```

Figure 5.2. Algorithm for Opening a File

Suppose a process executes the following code, opening the file "/etc/passwd" twice, once read-only and once write-only, and the file "local" once, for reading and writing.[2]

1. All system calls return the value −1 if they fail. The return value −1 will not be explicitly mentioned when discussing the syntax of the system calls.

2. The definition of the *open* system call specifies three parameters (the third is used for the *create* mode of open), but programmers usually use only the first two. The C compiler does not check that the number of parameters is correct. System implementations typically pass the first two parameters and a third "garbage" parameter (whatever happens to be on the stack) to the kernel. The kernel

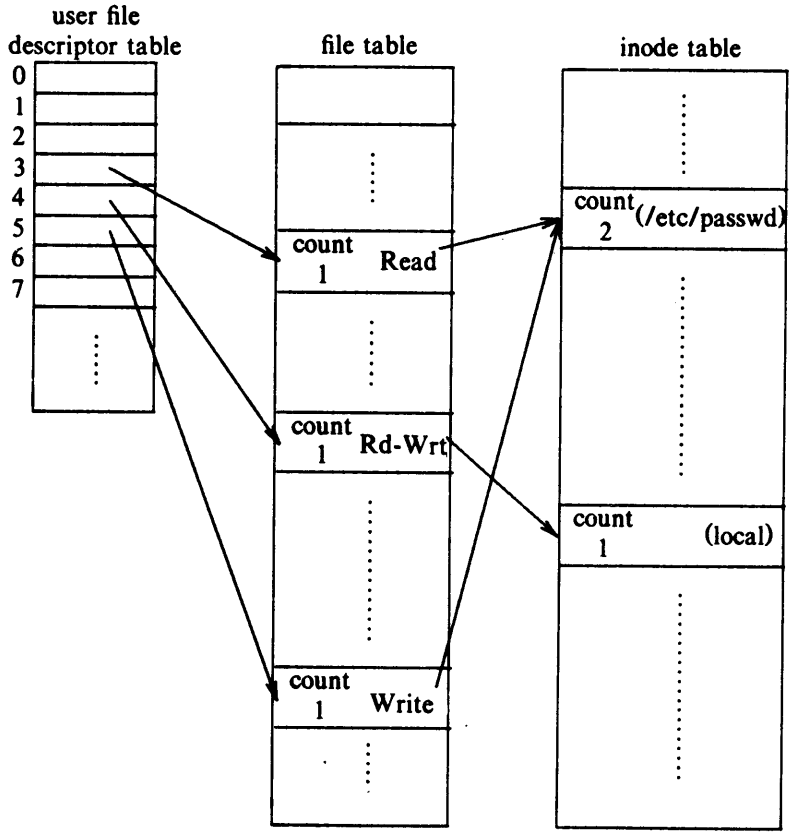


Figure 5.3. Data Structures after Open

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("local", O_RDWR);
fd3 = open("/etc/passwd", O_WRONLY);
```

Figure 5.3 shows the relationship between the inode table, file table, and user file descriptor data structures. Each *open* returns a file descriptor to the process, and the corresponding entry in the user file descriptor table points to a unique entry in

---

does not check the third parameter unless the second parameter indicates that it must, allowing programmers to encode only two parameters.

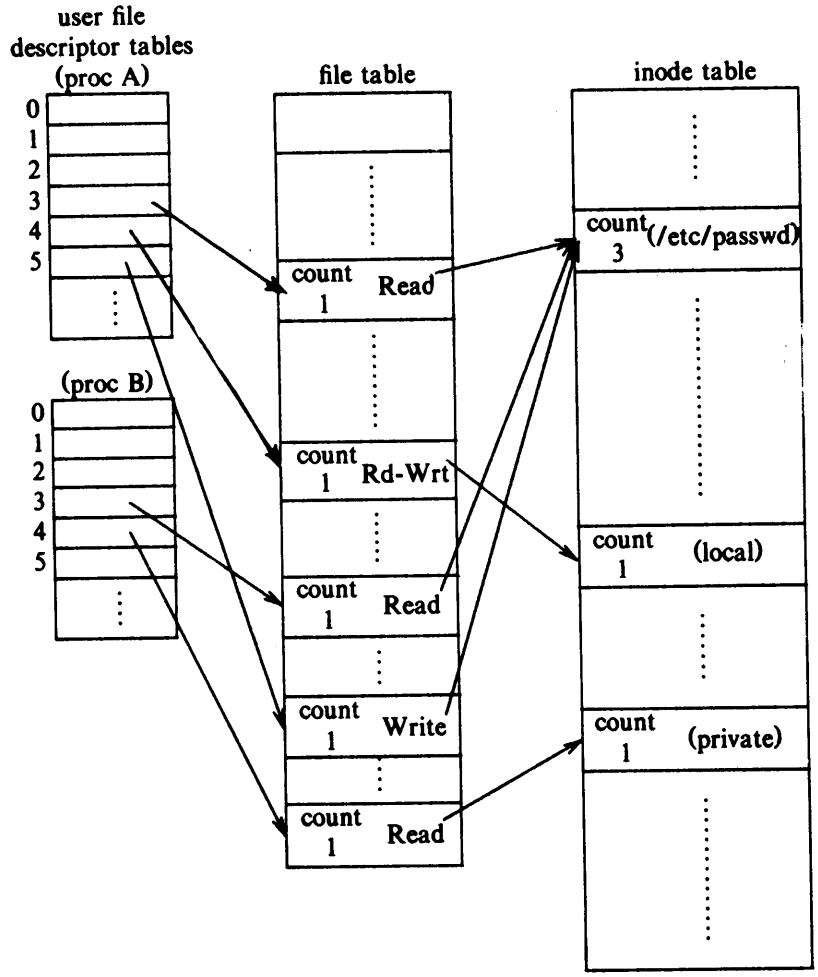


Figure 5.4. Data Structures after Two Processes Open Files

the kernel file table even though one file ("/etc/passwd") is opened twice. The file table entries of all instances of an open file point to one entry in the in-core inode table. The process can *read* or *write* the file "/etc/passwd" but only through file descriptors 3 and 5 in the figure. The kernel notes the capability to read or write the file in the file table entry allocated during the *open* call. Suppose a second process executes the following code.

```
fd1 = open("/etc/passwd", O_RDONLY);
fd2 = open("private", O_RDONLY);
```

Figure 5.4 shows the relationship between the appropriate data structures while both processes (and no others) have the files open. Again, each *open* call results in allocation of a unique entry in the user file descriptor table and in the kernel file table, but the kernel contains at most one entry per file in the in-core inode table.

The user file descriptor table entry could conceivably contain the file offset for the position of the next I/O operation and point directly to the in-core inode entry for the file, eliminating the need for a separate kernel file table. The examples above show a one-to-one relationship between user file descriptor entries and kernel file table entries. Thompson notes, however, that he implemented the file table as a separate structure to allow sharing of the offset pointer between several user file descriptors (see page 1943 of [Thompson 78]). The *dup* and *fork* system calls, explained in Sections 5.13 and 7.1, manipulate the data structures to allow such sharing.

The first three user file descriptors (0, 1, and 2) are called the *standard input*, *standard output*, and *standard error* file descriptors. Processes on UNIX systems conventionally use the standard input descriptor to read input data, the standard output descriptor to write output data, and the standard error descriptor to write error data (messages). Nothing in the operating system assumes that these file descriptors are special. A group of users could adopt the convention that file descriptors 4, 6, and 11 are special file descriptors, but counting from 0 (in C) is much more natural. Adoption of the convention by all user programs makes it easy for them to communicate via *pipes*, as will be seen in Chapter 7. Normally, the control terminal (see Chapter 10) serves as standard input, standard output and standard error.

5.2 READ

The syntax of the *read* system call is

```
number = read(fd, buffer, count)
```

where *fd* is the file descriptor returned by *open*, *buffer* is the address of a data structure in the user process that will contain the read data on successful completion of the call, *count* is the number of bytes the user wants to read, and *number* is the number of bytes actually read. Figure 5.5 depicts the algorithm *read* for reading a regular file. The kernel gets the file table entry that corresponds to

```
algorithm read
input:   user file descriptor
         address of buffer in user process
         number of bytes to read
output: count of bytes copied into user space
{
        get file table entry from user file descriptor;
        check file accessibility;
        set parameters in u area for user address, byte count, I/O to user;
        get inode from file table;
        lock inode;
        set byte offset in u area from file table offset;
        while (count not satisfied)
        {
                convert file offset to disk block (algorithm bmap);
                calculate offset into block, number of bytes to read;
                if (number of bytes to read is 0)
                        /* trying to read end of file */
                        break;                  /* out of loop */
                read block (algorithm breada if with read ahead, algorithm
                                        bread otherwise);
                copy data from system buffer to user address;
                update u area fields for file byte offset, read count,
                                        address to write into user space;
                release buffer;                 /* locked in bread */
        }
        unlock inode;
        update file table offset for next read;
        return(total number of bytes read);
}
```

Figure 5.5. Algorithm for Reading a File

the user file descriptor, following the pointer in Figure 5.3. It now sets several I/O parameters in the *u area* (Figure 5.6), eliminating the need to pass them as function parameters. Specifically, it sets the I/O mode to indicate that a read is being done, a flag to indicate that the I/O will go to user address space, a count field to indicate the number of bytes to read, the target address of the user data buffer, and finally, an offset field (from the file table) to indicate the byte offset into the file where the I/O should begin. After the kernel sets the I/O parameters in the *u area*, it follows the pointer from the file table entry to the inode, locking the inode before it reads the file.

The algorithm now goes into a loop until the *read* is satisfied. The kernel converts the file byte offset into a block number, using algorithm *bmap*, and it notes the byte offset in the block where the I/O should begin and how many bytes

mode	indicates read or write
count	count of bytes to read or write
offset	byte offset in file
address	target address to copy data, in user or kernel memory
flag	indicates if address is in user or kernel memory

Figure 5.6. I/O Parameters Saved in U Area

in the block it should read. After reading the block into a buffer, possibly using block read ahead (algorithms *bread* and *breada*) as will be described, it copies the data from the block to the target address in the user process. It updates the I/O parameters in the *u area* according to the number of bytes it read, incrementing the file byte offset and the address in the user process where the next data should be delivered, and decrementing the count of bytes it needs to read to satisfy the user read request. If the user request is not satisfied, the kernel repeats the entire cycle, converting the file byte offset to a block number, reading the block from disk to a system buffer, copying data from the buffer to the user process, releasing the buffer, and updating I/O parameters in the *u area*. The cycle completes either when the kernel completely satisfies the user request, when the file contains no more data, or if the kernel encounters an error in reading the data from disk or in copying the data to user space. The kernel updates the offset in the file table according to the number of bytes it actually read; consequently, successive *read*s of a file deliver the file data in sequence. The *lseek* system call (Section 5.6) adjusts the value of the file table offset and changes the order in which a process *read*s or *write*s a file.

```
#include <fcntl.h>
main()
{
        int fd;
        char lilbuf[20], bigbuf[1024];

        fd = open("/etc/passwd", O_RDONLY);
        read(fd, lilbuf, 20);
        read(fd, bigbuf, 1024);
        read(fd, lilbuf, 20);
}
```

Figure 5.7. Sample Program for Reading a File

Consider the program in Figure 5.7. The *open* returns a file descriptor that the user assigns to the variable *fd* and uses in the subsequent *read* calls. In the *read* system call, the kernel verifies that the file descriptor parameter is legal, and that

the process had previously *open*ed the file for reading. It stores the values *lilbuf*, 20, and 0 in the *u area*, corresponding to the address of the user buffer, the byte count, and the starting byte offset in the file. It calculates that byte offset 0 is in the 0th block of the file and retrieves the entry for the 0th block in the inode. Assuming such a block exists, the kernel reads the entire block of 1024 bytes into a buffer but copies only 20 bytes to the user address *lilbuf*. It increments the *u area* byte offset to 20 and decrements the count of data to read to 0. Since the *read* has been satisfied, the kernel resets the file table offset to 20, so that subsequent *read*s on the file descriptor will begin at byte 20 in the file, and the system call returns the number of bytes actually read, 20.

For the second *read* call, the kernel again verifies that the descriptor is legal and that the process had *open*ed the file for reading, because it has no way of knowing that the user *read* request is for the same file that was determined to be legal during the last *read*. It stores in the *u area* the user address *bigbuf*, the number of bytes the process wants to read, 1024, and the starting offset in the file, 20, taken from the file table. It converts the file byte offset to the correct disk block, as above, and reads the block. If the time between *read* calls is small, chances are good that the block will be in the buffer cache. But the kernel cannot satisfy the *read* request entirely from the buffer, because only 1004 out of the 1024 bytes for this request are in the buffer. So it copies the last 1004 bytes from the buffer into the user data structure *bigbuf* and updates the parameters in the *u area* to indicate that the next iteration of the read loop starts at byte 1024 in the file, that the data should be copied to byte position 1004 in *bigbuf*, and that the number of bytes to to satisfy the *read* request is 20.

The kernel now cycles to the beginning of the loop in the *read* algorithm. It converts byte offset 1024 to logical block offset 1, looks up the second direct block number in the inode, and finds the correct disk block to read. It reads the block from the buffer cache, reading the block from disk if it is not in the cache. Finally, it copies 20 bytes from the buffer to the correct address in the user process. Before leaving the system call, the kernel sets the offset field in the file table entry to 1044, the byte offset that should be accessed next. For the last *read* call in the example, the kernel proceeds as in the first *read* call, except that it starts reading at byte 1044 in the file, finding that value in the offset field in the file table entry for the descriptor.

The example shows how advantageous it is for I/O requests to start on file system block boundaries and to be multiples of the block size. Doing so allows the kernel to avoid an extra iteration in the *read* algorithm loop, with the consequent expense of accessing the inode to find the correct block number for the data and competing with other processes for access to the buffer pool. The standard I/O library was written to hide knowledge of the kernel buffer size from users; its use avoids the performance penalties inherent in processes that nibble at the file system inefficiently (see exercise 5.4).

As the kernel goes through the *read* loop, it determines whether a file is subject to read-ahead: if a process *read*s two blocks sequentially, the kernel assumes that

all subsequent *read*s will be sequential until proven otherwise. During each iteration through the loop, the kernel saves the next logical block number in the in-core inode and, during the next iteration, compares the current logical block number to the value previously saved. If they are equal, the kernel calculates the physical block number for read-ahead and saves its value in the *u area* for use in the *breada* algorithm. Of course, if a process does not *read* to the end of a block, the kernel does not invoke read-ahead for the next block.

Recall from Figure 4.9 that it is possible for some block numbers in an inode or in indirect blocks to have the value 0, even though later blocks have nonzero value. If a process attempts to *read* data from such a block, the kernel satisfies the request by allocating an arbitrary buffer in the *read* loop, clearing its contents to 0, and copying it to the user address. This case is different from the case where a process encounters the end of a file, meaning that no data was ever written to any location beyond the current point. When encountering end of file, the kernel returns no data to the process (see exercise 5.1).

When a process invokes the *read* system call, the kernel locks the inode for the duration of the call. Afterwards, it could go to sleep reading a buffer associated with data or with indirect blocks of the inode. If another process were allowed to change the file while the first process was sleeping, *read* could return inconsistent data. For example, a process may *read* several blocks of a file; if it slept while reading the first block and a second process were to *write* the other blocks, the returned data would contain a mixture of old and new data. Hence, the inode is left locked for the duration of the *read* call, affording the process a consistent view of the file as it existed at the start of the call.

The kernel can preempt a *read*ing process between system calls in user mode and schedule other processes to run. Since the inode is unlocked at the end of a system call, nothing prevents other processes from accessing the file and changing its contents. It would be unfair for the system to keep an inode locked from the time a process *open*ed the file until it *close*d the file, because one process could keep a file open and thus prevent other processes from ever accessing it. If the file was "/etc/passwd", used by the login process to check a user's password, then one malicious (or, perhaps, just errant) user could prevent all other users from logging in. To avoid such problems, the kernel unlocks the inode at the end of each system call that uses it. If another process changes the file between the two *read* system calls by the first process, the first process may *read* unexpected data, but the kernel data structures are consistent.

For example, suppose the kernel executes the two processes in Figure 5.8 concurrently. Assuming both processes complete their *open* calls before either one starts its *read* or *write* calls, the kernel could execute the *read* and *write* calls in any of six sequences: *read1, read2, write1, write2*, or *read1, write1, read2, write2*, or *read1, write1, write2, read2*, and so on. The data that process A *read*s depends on the order that the system executes the system calls of the two processes; the system does not guarantee that the data in the file remains the same after *open*ing the file. Use of the *file and record locking* feature (Section 5.4) allows a process to