

used buffer not currently in use and claims it, sending a message to the disk device driver to rewrite it if it is dirty. Then the disk driver is asked to fetch the block to be read.

Once the block is in the cache, the file system sends a message to the system task asking it to copy the data to the appropriate place in the user's buffer (i.e., bytes 600 to 1023 to the start of the buffer, and bytes 1024 to 1623 to offset 424 within the buffer). After the copy has been done, the file system sends a reply message to the user specifying how many bytes have been copied.

When the reply comes back to the user, the library function *read* extracts the reply code and returns it as the function value to the caller.

One extra step is not really part of the *read* call itself. After the file system completes a read and sends a reply, it initiates reading additional blocks, provided that the read is from a block device and certain other conditions are met. Since sequential file reads are common, it is reasonable to expect that the next blocks in a file will be requested in the next read request, and this makes it likely that the desired block will already be in the cache when it is needed. The number of blocks requested depends upon the size of the block cache; as many as 32 additional blocks may be requested. The device driver does not necessarily return this many blocks, and if at least one block is returned a request is considered successful.

## 5.7 IMPLEMENTATION OF THE MINIX 3 FILE SYSTEM

The MINIX 3 file system is relatively large (more than 100 pages of C) but quite straightforward. Requests to carry out system calls come in, are carried out, and replies are sent. In the following sections we will go through it a file at a time, pointing out the highlights. The code itself contains many comments to aid the reader.

In looking at the code for other parts of MINIX 3 we have generally looked at the main loop of a process first and then looked at the routines that handle the different message types. We will organize our approach to the file system differently. First we will go through the major subsystems (cache management, i-node management, etc.). Then we will look at the main loop and the system calls that operate upon files. Next we will look at systems call that operate upon directories, and then, we will discuss the remaining system calls that fall into neither category. Finally we will see how device special files are handled.

### 5.7.1 Header Files and Global Data Structures

Like the kernel and process manager, various data structures and tables used in the file system are defined in header files. Some of these data structures are placed in system-wide header files in *include/* and its subdirectories. For instance,

*include/sys/stat.h* defines the format by which system calls can provide i-node information to other programs and the structure of a directory entry is defined in *include/sys/dir.h*. Both of these files are required by POSIX. The file system is affected by a number of definitions contained in the global configuration file *include/minix/config.h*, such as *NR\_BUFS* and *NR\_BUF\_HASH*, which control the size of the block cache.

## File System Headers

The file system's own header files are in the file system source directory *src/fs/*. Many file names will be familiar from studying other parts of the MINIX 3 system. The FS master header file, *fs.h* (line 20900), is quite analogous to *src/kernel/kernel.h* and *src/pm/pm.h*. It includes other header files needed by all the C source files in the file system. As in the other parts of MINIX 3, the file system master header includes the file system's own *const.h*, *type.h*, *proto.h*, and *glo.h*. We will look at these next.

*Const.h* (line 21000) defines some constants, such as table sizes and flags, that are used throughout the file system. MINIX 3 already has a history. Earlier versions of MINIX had different file systems. Although MINIX 3 does not support the old V1 and V2 file systems, some definitions have been retained, both for reference and in expectation that someone will add support for these later. Support for older versions is useful not only for accessing files on older MINIX file systems, it may also be useful for exchanging files.

Other operating systems may use older MINIX file systems—for instance, Linux originally used and still supports MINIX file systems. (It is perhaps somewhat ironic that Linux still supports the original MINIX file system but MINIX 3 does not.) Some utilities are available for MS-DOS and Windows to access older MINIX directories and files. The superblock of a file system contains a **magic number** to allow the operating system to identify the file system's type; the constants *SUPER\_MAGIC*, *SUPER\_V2*, and *SUPER\_V3* define these numbers for the three versions of the MINIX file system. There are also *\_REV*-suffixed versions of these for V1 and V2, in which the bytes of the magic number are reversed. These were used with ports of older MINIX versions to systems with a different byte order (little-endian rather than big-endian) so a removable disk written on a machine with a different byte order could be identified as such. As of the release of MINIX 3.1.0 defining a *SUPER\_V3\_REV* magic number has not been necessary, but it is likely this definition will be added in the future.

*Type.h* (line 21100) defines both the old V1 and new V2 i-node structures as they are laid out on the disk. The i-node is one structure that did not change in MINIX 3, so the V2 i-node is used with the V-3 file system. The V2 i-node is twice as big as the old one, which was designed for compactness on systems with no hard drive and 360-KB diskettes. The new version provides space for the three time fields which UNIX systems provide. In the V1 i-node there was only one

time field, but a `stat` or `fstat` would “fake it” and return a `stat` structure containing all three fields. There is a minor difficulty in providing support for the two file system versions. This is flagged by the comment on line 21116. Older MINIX 3 software expected the `gid_t` type to be an 8-bit quantity, so `d2_gid` must be declared as type `u16_t`.

*Proto.h* (line 21200) provides function prototypes in forms acceptable to either old K&R or newer ANSI Standard C compilers. It is a long file, but not of great interest. However, there is one point to note: because there are so many different system calls handled by the file system, and because of the way the file system is organized, the various `do_XXX` functions are scattered through a number of files. *Proto.h* is organized by file and is a handy way to find the file to consult when you want to see the code that handles a particular system call.

Finally, *glo.h* (line 21400) defines global variables. The message buffers for the incoming and reply messages are also here. The now-familiar trick with the *EXTERN* macro is used, so these variables can be accessed by all parts of the file system. As in the other parts of MINIX 3, the storage space will be reserved when *table.c* is compiled.

The file system’s part of the process table is contained in *fproc.h* (line 21500). The *fproc* array is declared with the *EXTERN* macro. It holds the mode mask, pointers to the i-nodes for the current root directory and working directory, the file descriptor array, uid, gid, and terminal number for each process. The process id and the process group id are also found here. The process id is duplicated in the part of the process table located in the process manager.

Several fields are used to store the parameters of those system calls that may be suspended part way through, such as reads from an empty pipe. The fields *fp\_suspended* and *fp\_revived* actually require only single bits, but nearly all compilers generate better code for characters than bit fields. There is also a field for the *FD\_CLOEXEC* bits called for by the POSIX standard. These are used to indicate that a file should be closed when an `exec` call is made.

Now we come to files that define other tables maintained by the file system. The first, *buf.h* (line 21600), defines the block cache. The structures here are all declared with *EXTERN*. The array *buf* holds all the buffers, each of which contains a data part, *b*, and a header full of pointers, flags, and counters. The data part is declared as a union of five types (lines 21618 to 21632) because sometimes it is convenient to refer to the block as a character array, sometimes as a directory, etc.

The truly proper way to refer to the data part of buffer 3 as a character array is *buf[3].b.b\_ \_data* because *buf[3].b* refers to the union as a whole, from which the *b\_ \_data* field is selected. Although this syntax is correct, it is cumbersome, so on line 21649 we define a macro *b\_ \_data*, which allows us to write *buf[3].b\_ \_data* instead. Note that *b\_ \_data* (the field of the union) contains two underscores, whereas *b\_ \_data* (the macro) contains just one, to distinguish them. Macros for other ways of accessing the block are defined on lines 21650 to 21655.

The buffer hash table, *buf\_hash*, is defined on line 21657. Each entry points to a list of buffers. Originally all the lists are empty. Macros at the end of *buf.h* define different block types. The *WRITE\_IMMED* bit signals that a block must be rewritten to the disk immediately if it is changed, and the *ONE\_SHOT* bit is used to indicate a block is unlikely to be needed soon. Neither of these is used currently but they remain available for anyone who has a bright idea about improving performance or reliability by modifying the way blocks in the cache are queued.

Finally, in the last line *HASH\_MASK* is defined, based upon the value of *NR\_BUF\_HASH* configured in *include/minix/config.h*. *HASH\_MASK* is ANDed with a block number to determine which entry in *buf\_hash* to use as the starting point in a search for a block buffer.

*File.h* (line 21700) contains the intermediate table *filp* (declared as *EXTERN*), used to hold the current file position and i-node pointer (see Fig. 5-39). It also tells whether the file was opened for reading, writing, or both, and how many file descriptors are currently pointing to the entry.

The file locking table, *file\_lock* (declared as *EXTERN*), is in *lock.h* (line 21800). The size of the array is determined by *NR\_LOCKS*, which is defined as 8 in *const.h*. This number should be increased if it is desired to implement a multi-user data base on a MINIX 3 system.

In *inode.h* (line 21900) the i-node table *inode* is declared (using *EXTERN*). It holds i-nodes that are currently in use. As we said earlier, when a file is opened its i-node is read into memory and kept there until the file is closed. The *inode* structure definition provides for information that is kept in memory, but is not written to the disk i-node. Notice that there is only one version, and nothing is version-specific here. When the i-node is read in from the disk, differences between V1 and V2/V3 file systems are handled. The rest of the file system does not need to know about the file system format on the disk, at least until the time comes to write back modified information.

Most of the fields should be self-explanatory at this point. However, *i\_seek* deserves some comment. It was mentioned earlier that, as an optimization, when the file system notices that a file is being read sequentially, it tries to read blocks into the cache even before they are asked for. For randomly accessed files there is no read ahead. When an *lseek* call is made, the field *i\_seek* is set to inhibit read ahead.

The file *param.h* (line 22000) is analogous to the file of the same name in the process manager. It defines names for message fields containing parameters, so the code can refer to, for example, *m\_in.buffer*, instead of *m\_in.m1\_p1*, which selects one of the fields of the message buffer *m\_in*.

In *super.h* (line 22100), we have the declaration of the superblock table. When the system is booted, the superblock for the root device is loaded here. As file systems are mounted, their superblocks go here as well. As with other tables, *super\_block* is declared as *EXTERN*.

## File System Storage Allocation

The last file we will discuss in this section is not a header. However, just as we did when discussing the process manager, it seems appropriate to discuss *table.c* immediately after reviewing the header files, since they are all included when *table.c* (line 22200) is compiled. Most of the data structures we have mentioned—the block cache, the *filp* table, and so on—are defined with the *EXTERN* macro, as are also the file system's global variables and the file system's part of the process table. In the same way we have seen in other parts of the MINIX 3 system, the storage is actually reserved when *table.c* is compiled. This file also contains one major initialized array. *Call\_vector* contains the pointer array used in the main loop for determining which procedure handles which system call number. We saw a similar table inside the process manager.

### 5.7.2 Table Management

Associated with each of the main tables—blocks, i-nodes, superblocks, and so forth—is a file that contains procedures that manage the table. These procedures are heavily used by the rest of the file system and form the principal interface between tables and the file system. For this reason, it is appropriate to begin our study of the file system code with them.

## Block Management

The block cache is managed by the procedures in the file *cache.c*. This file contains the nine procedures listed in Fig. 5-40. The first one, *get\_block* (line 22426), is the standard way the file system gets data blocks. When a file system procedure needs to read a user data block, a directory block, a superblock, or any other kind of block, it calls *get\_block*, specifying the device and block number.

When *get\_block* is called, it first examines the block cache to see if the requested block is there. If so, it returns a pointer to it. Otherwise, it has to read the block in. The blocks in the cache are linked together on *NR\_BUF\_HASH* linked lists. *NR\_BUF\_HASH* is a tunable parameter, along with *NR\_BUFS*, the size of the block cache. Both of these are set in *include/minix/config.h*. At the end of this section we will say a few words about optimizing the size of the block cache and the hash table. The *HASH\_MASK* is *NR\_BUF\_HASH* - 1. With 256 hash lists, the mask is 255, so all the blocks on each list have block numbers that end with the same string of 8 bits, that is 00000000, 00000001, ..., or 11111111.

The first step is usually to search a hash chain for a block, although there is a special case, when a hole in a sparse file is being read, where this search is skipped. This is the reason for the test on line 22454. Otherwise, the next two

Procedure	Function
<code>get_block</code>	Fetch a block for reading or writing
<code>put_block</code>	Return a block previously requested with <code>get_block</code>
<code>alloc_zone</code>	Allocate a new zone (to make a file longer)
<code>free_zone</code>	Release a zone (when a file is removed)
<code>rw_block</code>	Transfer a block between disk and cache
<code>invalidate</code>	Purge all the cache blocks for some device
<code>flushall</code>	Flush all dirty blocks for one device
<code>rw_scattered</code>	Read or write scattered data from or to a device
<code>rm_lru</code>	Remove a block from its LRU chain

**Figure 5-40.** Procedures used for block management.

lines set *bp* to point to the start of the list on which the requested block would be, if it were in the cache, applying *HASH\_MASK* to the block number. The loop on the next line searches this list to see if the block can be found. If it is found and is not in use, it is removed from the LRU list. If it is already in use, it is not on the LRU list anyway. The pointer to the found block is returned to the caller on line 22463.

If the block is not on the hash list, it is not in the cache, so the least recently used block from the LRU list is taken. The buffer chosen is removed from its hash chain, since it is about to acquire a new block number and hence belongs on a different hash chain. If it is dirty, it is rewritten to the disk on line 22495. Doing this with a call to *flushall* rewrites any other dirty blocks for the same device. This call is the way most blocks get written. Blocks that are currently in use are never chosen for eviction, since they are not on the LRU chain. Blocks will hardly ever be found to be in use, however; normally a block is released by *put\_block* immediately upon being used.

As soon as the buffer is available, all of the fields, including *b\_dev*, are updated with the new parameters (lines 22499 to 22504), and the block may be read in from the disk. However, there are two occasions when it may not be necessary to read the block from the disk. *Get\_block* is called with a parameter *only\_search*. This may indicate that this is a prefetch. During a prefetch an available buffer is found, writing the old contents to the disk if necessary, and a new block number is assigned to the buffer, but the *b\_dev* field is set to *NO\_DEV* to signal there are as yet no valid data in this block. We will see how this is used when we discuss the *rw\_scattered* function. *Only\_search* can also be used to signal that the file system needs a block just to rewrite all of it. In this case it is wasteful to first read the old version in. In either of these cases the parameters are updated, but the actual disk read is omitted (lines 22507 to 22513). When the new block has been read in, *get\_block* returns to its caller with a pointer to it.

Suppose that the file system needs a directory block temporarily, to look up a file name. It calls *get\_block* to acquire the directory block. When it has looked up its file name, it calls *put\_block* (line 22520) to return the block to the cache, thus making the buffer available in case it is needed later for a different block.

*Put\_block* takes care of putting the newly returned block on the LRU list, and in some cases, rewriting it to the disk. At line 22544 a decision is made to put it on the front or rear of the LRU list. Blocks on a RAM disk are always put on the front of the queue. The block cache does not really do very much for a RAM disk, since its data are already in memory and accessible without actual I/O. The *ONE\_SHOT* flag is tested to see if the block has been marked as one not likely to be needed again soon, and such blocks are put on the front, where they will be reused quickly. However, this is used rarely, if at all. Almost all blocks except those from the RAM disk are put on the rear, in case they are needed again soon.

After the block has been repositioned on the LRU list, another check is made to see if the block should be rewritten to disk immediately. Like the previous test, the test for *WRITE\_IMMED* is a vestige of an abandoned experiment; currently no blocks are marked for immediate writing.

As a file grows, from time to time a new zone must be allocated to hold the new data. The procedure *alloc\_zone* (line 22580) takes care of allocating new zones. It does this by finding a free zone in the zone bitmap. There is no need to search through the bitmap if this is to be the first zone in a file; the *s\_zsearch* field in the superblock, which always points to the first available zone on the device, is consulted. Otherwise an attempt is made to find a zone close to the last existing zone of the current file, in order to keep the zones of a file together. This is done by starting the search of the bitmap at this last zone (line 22603). The mapping between the bit number in the bitmap and the zone number is handled on line 22615, with bit 1 corresponding to the first data zone.

When a file is removed, its zones must be returned to the bitmap. *Free\_zone* (line 22621) is responsible for returning these zones. All it does is call *free\_bit*, passing the zone map and the bit number as parameters. *Free\_bit* is also used to return free i-nodes, but then with the i-node map as the first parameter, of course.

Managing the cache requires reading and writing blocks. To provide a simple disk interface, the procedure *rw\_block* (line 22641) has been provided. It reads or writes one block. Analogously, *rw\_inode* exists to read and write i-nodes.

The next procedure in the file is *invalidate* (line 22680). It is called when a disk is unmounted, for example, to remove from the cache all the blocks belonging to the file system just unmounted. If this were not done, then when the device were reused (with a different floppy disk), the file system might find the old blocks instead of the new ones.

We mentioned earlier that *flushall* (line 22694), called from *get\_block* whenever a dirty block is removed from the LRU list, is the function responsible for writing most data. It is also called by the sync system call to flush to disk all dirty buffers belonging to a specific device. Sync is activated periodically by the

update daemon, and calls *flushall* once for each mounted device. *Flushall* treats the buffer cache as a linear array, so all dirty buffers are found, even ones that are currently in use and are not in the LRU list. All buffers in the cache are scanned, and those that belong to the device to be flushed and that need to be written are added to an array of pointers, *dirty*. This array is declared as *static* to keep it off the stack. It is then passed to *rw\_scattered*.

In MINIX 3 scheduling of disk writing has been removed from the disk device drivers and made the sole responsibility of *rw\_scattered* (line 22711). This function receives a device identifier, a pointer to an array of pointers to buffers, the size of the array, and a flag indicating whether to read or write. The first thing it does is sort the array it receives on the block numbers, so the actual read or write operation will be performed in an efficient order. It then constructs vectors of contiguous blocks to send to the device driver with a call to *dev\_io*. The driver does not have to do any additional scheduling. It is likely with a modern disk that the drive electronics will further optimize the order of requests, but this is not visible to MINIX 3. *Rw\_scattered* is called with the *WRITING* flag only from the *flushall* function described above. In this case the origin of these block numbers is easy to understand. They are buffers which contain data from blocks previously read but now modified. The only call to *rw\_scattered* for a read operation is from *rahead* in *read.c*. At this point, we just need to know that before calling *rw\_scattered*, *get\_block* has been called repeatedly in prefetch mode, thus reserving a group of buffers. These buffers contain block numbers, but no valid device parameter. This is not a problem, since *rw\_scattered* is called with a device parameter as one of its arguments.

There is an important difference in the way a device driver may respond to a read (as opposed to a write) request, from *rw\_scattered*. A request to write a number of blocks *must* be honored completely, but a request to read a number of blocks may be handled differently by different drivers, depending upon what is most efficient for the particular driver. *Rahead* often calls *rw\_scattered* with a request for a list of blocks that may not actually be needed, so the best response is to get as many blocks as can be gotten easily, but not to go wildly seeking all over a device that may have a substantial seek time. For instance, the floppy driver may stop at a track boundary, and many other drivers will read only consecutive blocks. When the read is complete, *rw\_scattered* marks the blocks read by filling in the device number field in their block buffers.

The last function in Fig. 5-40 is *rm\_lru* (line 22809). This function is used to remove a block from the LRU list. It is used only by *get\_block* in this file, so it is declared *PRIVATE* instead of *PUBLIC* to hide it from procedures outside the file.

Before we leave the block cache, let us say a few words about fine-tuning it. *NR\_BUF\_HASH* must be a power of 2. If it is larger than *NR\_BUFS*, the average length of a hash chain will be less than one. If there is enough memory for a large number of buffers, there is space for a large number of hash chains, so the usual choice is to make *NR\_BUF\_HASH* the next power of 2 greater than



*NR\_BUFS*. The listing in the text shows settings of 128 blocks and 128 hash lists. The optimal size depends upon how the system is used, since that determines how much must be buffered. The full source code used to compile the standard MINIX 3 binaries that are installed from the CD-ROM that accompanies this text has settings of 1280 buffers and 2048 hash chains. Empirically it was found that increasing the number of buffers beyond this did not improve performance when recompiling the MINIX 3 system, so apparently this is large enough to hold the binaries for all compiler passes. For some other kind of work a smaller size might be adequate or a larger size might improve performance.

The buffers for the standard MINIX 3 system on the CD-ROM occupy more than 5 MB of RAM. An additional binary, designated *image\_small* is provided that was compiled with just 128 buffers in the block cache, and the buffers for this system need only a little more than 0.5 MB. This one can be installed on a system with only 8 MB of RAM. The standard version requires 16 MB of RAM. With some tweaking, it could no doubt be shoehorned into a memory of 4 MB or smaller.

## I-Node Management

The block cache is not the only file system table that needs support procedures. The i-node table does, too. Many of the procedures are similar in function to the block management procedures. They are listed in Fig. 5-41.

Procedure	Function
<code>get_inode</code>	Fetch an i-node into memory
<code>put_inode</code>	Return an i-node that is no longer needed
<code>alloc_inode</code>	Allocate a new i-node (for a new file)
<code>wipe_inode</code>	Clear some fields in an i-node
<code>free_inode</code>	Release an i-node (when a file is removed)
<code>update_times</code>	Update time fields in an i-node
<code>rw_inode</code>	Transfer an i-node between memory and disk
<code>old_icopy</code>	Convert i-node contents to write to V1 disk i-node
<code>new_icopy</code>	Convert data read from V1 file system disk i-node
<code>dup_inode</code>	Indicate that someone else is using an i-node

**Figure 5-41.** Procedures used for i-node management.

The procedure *get\_inode* (line 22933) is analogous to *get\_block*. When any part of the file system needs an i-node, it calls *get\_inode* to acquire it. *Get\_inode* first searches the *inode* table to see if the i-node is already present. If so, it increments the usage counter and returns a pointer to it. This search is contained on

lines 22945 to 22955. If the i-node is not present in memory, the i-node is loaded by calling *rw\_inode*.

When the procedure that needed the i-node is finished with it, the i-node is returned by calling the procedure *put\_inode* (line 22976), which decrements the usage count *i\_count*. If the count is then zero, the file is no longer in use, and the i-node can be removed from the table. If it is dirty, it is rewritten to disk.

If the *i\_link* field is zero, no directory entry is pointing to the file, so all its zones can be freed. Note that the usage count going to zero and the number of links going to zero are different events, with different causes and different consequences. If the i-node is for a pipe, all the zones must be released, even though the number of links may not be zero. This happens when a process reading from a pipe releases the pipe. There is no sense in having a pipe for one process.

When a new file is created, an i-node must be allocated by *alloc\_inode* (line 23003). MINIX 3 allows mounting of devices in read-only mode, so the superblock is checked to make sure the device is writable. Unlike zones, where an attempt is made to keep the zones of a file close together, any i-node will do. In order to save the time of searching the i-node bitmap, advantage is taken of the field in the superblock where the first unused i-node is recorded.

After the i-node has been acquired, *get\_inode* is called to fetch the i-node into the table in memory. Then its fields are initialized, partly in-line (lines 23038 to 23044) and partly using the procedure *wipe\_inode* (line 23060). This particular division of labor has been chosen because *wipe\_inode* is also needed elsewhere in the file system to clear certain i-node fields (but not all of them).

When a file is removed, its i-node is freed by calling *free\_inode* (line 23079). All that happens here is that the corresponding bit in the i-node bitmap is set to 0 and the superblock's record of the first unused i-node is updated.

The next function, *update\_times* (line 23099), is called to get the time from the system clock and change the time fields that require updating. *Update\_times* is also called by the *stat* and *fstat* system calls, so it is declared *PUBLIC*.

The procedure *rw\_inode* (line 23125) is analogous to *rw\_block*. Its job is to fetch an i-node from the disk. It does its work by carrying out the following steps:

1. Calculate which block contains the required i-node.
2. Read in the block by calling *get\_block*.
3. Extract the i-node and copy it to the *inode* table.
4. Return the block by calling *put\_block*.

*Rw\_inode* is a bit more complex than the basic outline given above, so some additional functions are needed. First, because getting the current time requires a kernel call, any need for a change to the time fields in the i-node is only marked by setting bits in the i-node's *i\_update* field while the i-node is in memory. If this field is nonzero when an i-node must be written, *update\_times* is called.

Second, the history of MINIX adds a complication: in the old *V1* file system the i-nodes on the disk have a different structure from *V2*. Two functions, *old\_icopy* (line 23168) and *new\_icopy* (line 23214) are provided to take care of the conversions. The first converts between i-node information in memory and the format used by the *V1* filesystem. The second does the same conversion for *V2* and *V3* filesystem disks. Both of these functions are called only from within this file, so they are declared *PRIVATE*. Each function handles conversions in both directions (disk to memory or memory to disk).

Older versions of MINIX were ported to systems which used a different byte order from Intel processors and MINIX 3 is also likely to be ported to such architectures in the future. Every implementation uses the native byte order on its disk; the *sp->native* field in the superblock identifies which order is used. Both *old\_icopy* and *new\_icopy* call functions *conv2* and *conv4* to swap byte orders, if necessary. Of course, much of what we have just described is not used by MINIX 3, since it does not support the *V1* filesystem to the extent that *V1* disks can be used. And as of this writing nobody has ported MINIX 3 to a platform that uses a different byte order. But these bits and pieces remain in place for the day when someone decides to make MINIX 3 more versatile.

The procedure *dup\_inode* (line 23257) just increments the usage count of the i-node. It is called when an open file is opened again. On the second open, the i-node need not be fetched from disk again.

## Superblock Management

The file *super.c* contains procedures that manage the superblock and the bitmaps. Six procedures are defined in this file, listed in Fig. 5-42.

Procedure	Function
<i>alloc_bit</i>	Allocate a bit from the zone or i-node map
<i>free_bit</i>	Free a bit in the zone or i-node map
<i>get_super</i>	Search the superblock table for a device
<i>get_block_size</i>	Find block size to use
<i>mounted</i>	Report whether given i-node is on a mounted (or root) file system
<i>read_super</i>	Read a superblock

**Figure 5-42.** Procedures used to manage the superblock and bitmaps.

When an i-node or zone is needed, *alloc\_inode* or *alloc\_zone* is called, as we have seen above. Both of these call *alloc\_bit* (line 23324) to actually search the relevant bitmap. The search involves three nested loops, as follows:

1. The outer one loops on all the blocks of a bitmap.
2. The middle one loops on all the words of a block.
3. The inner one loops on all the bits of a word.

The middle loop works by seeing if the current word is equal to the one's complement of zero, that is, a complete word full of 1s. If so, it has no free i-nodes or zones, so the next word is tried. When a word with a different value is found, it must have at least one 0 bit in it, so the inner loop is entered to find the free (i.e., 0) bit. If all the blocks have been tried without success, there are no free i-nodes or zones, so the code *NO\_BIT* (0) is returned. Searches like this can consume a lot of processor time, but the use of the superblock fields that point to the first unused i-node and zone, passed to *alloc\_bit* in *origin*, helps to keep these searches short.

Freeing a bit is simpler than allocating one, because no search is required. *Free\_bit* (line 23400) calculates which bitmap block contains the bit to free and sets the proper bit to 0 by calling *get\_block*, zeroing the bit in memory and then calling *put\_block*.

The next procedure, *get\_super* (line 23445), is used to search the superblock table for a specific device. For example, when a file system is to be mounted, it is necessary to check that it is not already mounted. This check can be performed by asking *get\_super* to find the file system's device. If it does not find the device, then the file system is not mounted.

In MINIX 3 the file system server is capable of handling file systems with different block sizes, although within a given disk partition only a single block size can be used. The *get\_block\_size* function (line 23467) is meant to determine the block size of a file system. It searches the superblock table for the given device and returns the block size of the device if it is mounted. Otherwise the minimum block size, *MIN\_BLOCK\_SIZE* is returned.

The next function, *mounted* (line 23489), is called only when a block device is closed. Normally, all cached data for a device are discarded when it is closed. But, if the device happens to be mounted, this is not desirable. *Mounted* is called with a pointer to the i-node for a device. It just returns *TRUE* if the device is the root device, or if it is a mounted device.

Finally, we have *read\_super* (line 23509). This is partially analogous to *rw\_block* and *rw\_inode*, but it is called only to read. The superblock is not read into the block cache at all, a request is made directly to the device for 1024 bytes starting at an offset of the same amount from the beginning of the device. Writing a superblock is not necessary in the normal operation of the system. *Read\_super* checks the version of the file system from which it has just read and performs conversions, if necessary, so the copy of the superblock in memory will have the standard structure even when read from a disk with a different superblock structure or byte order.

Even though it is not currently used in MINIX 3, the method of determining whether a disk was written on a system with a different byte order is clever and worth noting. The magic number of a superblock is written with the native byte order of the system upon which the file system was created, and when a superblock is read a test for reversed-byte-order superblocks is made.

## File Descriptor Management

MINIX 3 contains special procedures to manage file descriptors and the *filp* table (see Fig. 5-39). They are contained in the file *filedes.c*. When a file is created or opened, a free file descriptor and a free *filp* slot are needed. The procedure *get\_fd* (line 23716) is used to find them. They are not marked as in use, however, because many checks must first be made before it is known for sure that the *creat* or *open* will succeed.

*Get\_filp* (line 23761) is used to see if a file descriptor is in range, and if so, returns its *filp* pointer.

The last procedure in this file is *find\_filp* (line 23774). It is needed to find out when a process is writing on a broken pipe (i.e., a pipe not open for reading by any other process). It locates potential readers by a brute force search of the *filp* table. If it cannot find one, the pipe is broken and the write fails.

## File Locking

The POSIX record locking functions are shown in Fig. 5-43. A part of a file can be locked for reading and writing, or for writing only, by an *fcntl* call specifying a *F\_SETLK* or *F\_SETLKW* request. Whether a lock exists over a part of a file can be determined using the *F\_GETLK* request.

Operation	Meaning
F_SETLK	Lock region for both reading and writing
F_SETLKW	Lock region for writing
F_GETLK	Report if region is locked

**Figure 5-43.** The POSIX advisory record locking operations. These operations are requested by using an *fcntl* system call.

The file *lock.c* contains only two functions. *Lock\_op* (line 23820) is called by the *fcntl* system call with a code for one of the operations shown in Fig. 5-43. It does some error checking to be sure the region specified is valid. When a lock is being set, it must not conflict with an existing lock, and when a lock is being cleared, an existing lock must not be split in two. When any lock is cleared, the other function in this file, *lock\_revive* (line 23964), is called. It wakes up all the processes that are blocked waiting for locks.

This strategy is a compromise; it would take extra code to figure out exactly which processes were waiting for a particular lock to be released. Those processes that are still waiting for a locked file will block again when they start. This strategy is based on an assumption that locking will be used infrequently. If a major multiuser data base were to be built upon a MINIX 3 system, it might be desirable to reimplement this.

*Lock\_revive* is also called when a locked file is closed, as might happen, for instance, if a process is killed before it finishes using a locked file.

### 5.7.3 The Main Program

The main loop of the file system is contained in file *main.c*, (line 24040). After a call to *fs\_init* for initialization, the main loop is entered. Structurally, this is very similar to the main loop of the process manager and the I/O device drivers. The call to *get\_work* waits for the next request message to arrive (unless a process previously suspended on a pipe or terminal can now be handled). It also sets a global variable, *who*, to the caller's process table slot number and another global variable, *call\_nr*, to the number of the system call to be carried out.

Once back in the main loop the variable *fp* is pointed to the caller's process table slot, and the *super\_user* flag tells whether the caller is the superuser or not. Notification messages are high priority, and a *SYS\_SIG* message is checked for first, to see if the system is shutting down. The second highest priority is a *SYN\_ALARM*, which means that a timer set by the file system has expired. A *NOTIFY\_MESSAGE* means a device driver is ready for attention, and is dispatched to *dev\_status*. Then comes the main attraction—the call to the procedure that carries out the system call. The procedure to call is selected by using *call\_nr* as an index into the array of procedure pointers, *call\_vecs*.

When control comes back to the main loop, if *dont\_reply* has been set, the reply is inhibited (e.g., a process has blocked trying to read from an empty pipe). Otherwise a reply is sent by calling *reply* (line 24087). The final statement in the main loop has been designed to detect that a file is being read sequentially and to load the next block into the cache before it is actually requested, to improve performance.

Two other functions in this file are intimately involved with the file system's main loop. *Get\_work* (line 24099) checks to see if any previously blocked procedures have now been revived. If so, these have priority over new messages. When there is no internal work to do the file system calls the kernel to get a message, on line 24124. Skipping ahead a few lines, we find *reply* (line 24159) which is called after a system call has been completed, successfully or otherwise. It sends a reply back to the caller. The process may have been killed by a signal, so the status code returned by the kernel is ignored. In this case there is nothing to be done anyway.

## Initialization of the File System

The functions that remain to be discussed in *main.c* are used at system startup. The major player is *fs\_init*, which is called by the file system before it enters its main loop during startup of the entire system. In the context of discussing process scheduling in Chapter 2 we showed in Fig. 2-43 the initial queueing of processes as the MINIX 3 system starts up. The file system is scheduled on a queue with lower priority than the process manager, so we can be sure that at startup time the process manager will get a chance to run before the file system. In Chapter 4 we examined the initialization of the process manager. As the PM builds its part of the process table, adding entries for itself and all other processes in the boot image, it sends a message to the file system for each one so the FS can initialize the corresponding entry in the FS part of the file system. Now we can see the other half of this interaction.

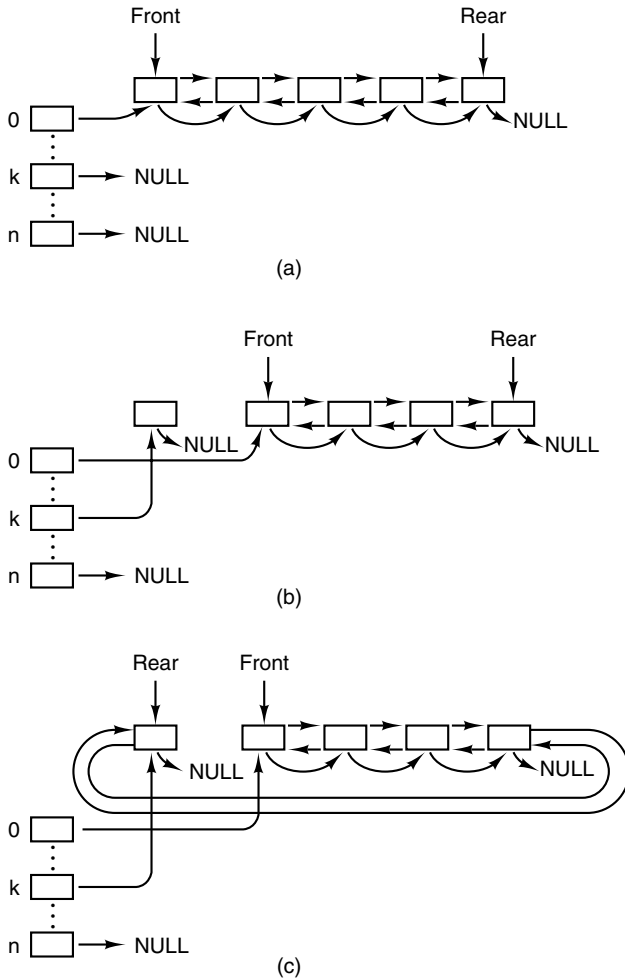
When the file system starts it immediately enters a loop of its own in *fs\_init*, on lines 24189 to 24202. The first statement in the loop is a call to *receive*, to get a message sent at line 18235 in the PM's *pm\_init* initialization function. Each message contains a process number and a PID. The first is used as an index into the file system's process table and the second is saved in the *fp\_pid* field of each selected slot. Following this the real and effective uid and gid for the superuser and a ~0 (all bits set) umask is set up for each selected slot. When a message with the symbolic value *NONE* in the process number field is received the loop terminates and a message is sent back to the process manager to tell it all is OK.

Next, the file system's own initialization is completed. First important constants are tested for valid values. Then several other functions are invoked to initialize the block cache and the device table, to load the RAM disk if necessary, and to load the root device superblock. At this point the root device can be accessed, and another loop is made through the FS part of the process table, so each process loaded from the boot image will recognize the root directory and use the root directory as its working directory (lines 24228 to 24235).

The first function called by *fs\_init* after it finishes its interaction with the process manager is *buf\_pool*, which begins on line 24132. It builds the linked lists used by the block cache. Figure 5-37 shows the normal state of the block cache, in which all blocks are linked on both the LRU chain and a hash chain. It may be helpful to see how the situation of Fig. 5-37 comes about. Immediately after the cache is initialized by *buf\_pool*, all the buffers will be on the LRU chain, and all will be linked into the 0th hash chain, as in Fig. 5-44(a). When a buffer is requested, and while it is in use, we have the situation of Fig. 5-44(b), in which we see that a block has been removed from the LRU chain and is now on a different hash chain.

Normally, blocks are released and returned to the LRU chain immediately. Figure 5-44(c) shows the situation after the block has been returned to the LRU chain. Although it is no longer in use, it can be accessed again to provide the

same data, if need be, and so it is retained on the hash chain. After the system has been in operation for awhile, almost all of the blocks can be expected to have been used and to be distributed among the different hash chains at random. Then the LRU chain will look like Fig. 5-37.



**Figure 5-44.** Block cache initialization. (a) Before any buffers have been used. (b) After one block has been requested. (c) After the block has been released.

The next thing called after *buf\_pool* is *build\_dmap*, which we will describe later, along with other functions dealing with device files. After that, *load\_ram* is called, which uses the next function we will examine, *igetenv* (line 2641). This



function retrieves a numeric device identifier from the kernel, using the name of a boot parameter as a key. If you have used the *sysenv* command to look at the boot parameters on a working MINIX 3 system, you have seen that *sysenv* reports devices numerically, displaying strings like

```
rootdev=912
```

The file system uses numbers like this to identify devices. The number is simply  $256 \times \text{major} + \text{minor}$ , where *major* and *minor* are the major and minor device numbers. In this example, the major, minor pair is 3, 144, which corresponds to */dev/c0d1p0s0*, a typical place to install MINIX 3 on a system with two disk drives.

*Load\_ram* (line 24260) allocates space for a RAM disk, and loads the root file system on it, if required by the boot parameters. It uses *igetenv* to get the *rootdev*, *ramimagedev*, and *ramsize* parameters set in the boot environment (lines 24278 to 24280). If the boot parameters specify

```
rootdev = ram
```

the root file system is copied from the device named by *ramimagedev* to the RAM disk block by block, starting with the boot block, with no interpretation of the various file system data structures. If the *ramsize* boot parameter is smaller than the size of *ramimagedev*, the RAM disk is made large enough to hold it. If *ramsize* specifies a size larger than the boot device file system the requested size is allocated and the RAM disk file system is adjusted to use the full size specified (lines 24404 to 24420). This is the only time that the file system ever writes a superblock, but, just as with reading a superblock, the block cache is not used and the data is written directly to the device using *dev\_io*.

Two items merit note at this point. The first is the code on lines 24291 to 24307 which deals with the case of booting from a CD-ROM. The *cdprobe* function, not discussed in this text, is used. Interested readers are referred to the code in *fs/cdprobe.c*, which can be found on the CD-ROM or the Web site. Second, regardless of the disk block size used by MINIX 3 for ordinary disk access, the boot block is always a 1 KB block and the superblock is loaded from the second 1 KB of the disk device. Anything else would be complicated, since the block size cannot be known until the superblock has been loaded.

*Load\_ram* allocates space for an empty RAM disk if a nonzero *ramsize* is specified without a request to use the RAM disk as the root file system. In this case, since no file system structures are copied, the RAM device cannot be used as a file system until it has been initialized by the *mkfs* command. Alternatively, such a RAM disk can be used for a secondary cache if support for this is compiled into the file system.

The last function in *main.c* is *load\_super* (line 24426). It initializes the superblock table and reads in the superblock of the root device.

### 5.7.4 Operations on Individual Files

In this section we will look at the system calls that operate on individual files one at a time (as opposed to, say, operations on directories). We will start with how files are created, opened, and closed. After that we will examine in some detail the mechanism by which files are read and written. Then that we will look at pipes and how operations on them differ from those on files.

#### Creating, Opening, and Closing Files

The file *open.c* contains the code for six system calls: *creat*, *open*, *mknod*, *mknod*, *close*, and *lseek*. We will examine *creat* and *open* together, and then look at each of the others.

In older versions of UNIX, the *creat* and *open* calls had distinct purposes. Trying to open a file that did not exist was an error, and a new file had to be created with *creat*, which could also be used to truncate an existing file to zero length. The need for two distinct calls is no longer present in a POSIX system, however. Under POSIX, the *open* call now allows creating a new file or truncating an old file, so the *creat* call now represents a subset of the possible uses of the *open* call and is really only necessary for compatibility with older programs. The procedures that handle *creat* and *open* are *do\_creat* (line 24537) and *do\_open* (line 24550). (As in the process manager, the convention is used in the file system that system call XXX is performed by procedure *do\_XXX*.) Opening or creating a file involves three steps:

1. Finding the i-node (allocating and initializing if the file is new).
2. Finding or creating the directory entry.
3. Setting up and returning a file descriptor for the file.

Both the *creat* and the *open* calls do two things: they fetch the name of a file and then they call *common\_open* which takes care of tasks common to both calls.

*Common\_open* (line 24573) starts by making sure that free file descriptor and *filp* table slots are available. If the calling function specified creation of a new file (by calling with the *O\_CREAT* bit set), *new\_node* is called on line 24594. *New\_node* returns a pointer to an existing i-node if the directory entry already exists; otherwise it will create both a new directory entry and i-node. If the i-node cannot be created, *new\_node* sets the global variable *err\_code*. An error code does not always mean an error. If *new\_node* finds an existing file, the error code returned will indicate that the file exists, but in this case that error is acceptable (line 24597). If the *O\_CREAT* bit is not set, a search is made for the i-node using an alternative method, the *eat\_path* function in *path.c*, which we will discuss further on. At this point, the important thing to understand is that if an i-node is

not found or successfully created, *common\_open* will terminate with an error before line 24606 is reached. Otherwise, execution continues here with assignment of a file descriptor and claiming of a slot in the *filp* table. Following this, if a new file has just been created, lines 24612 to 24680 are skipped.

If the file is not new, then the file system must test to see what kind of a file it is, what its mode is, and so on, to determine whether it can be opened. The call to *forbidden* on line 24614 first makes a general check of the *rw**x* bits. If the file is a regular file and *common\_open* was called with the *O\_TRUNC* bit set, it is truncated to length zero and *forbidden* is called again (line 24620), this time to be sure the file may be written. If the permissions allow, *wipe\_inode* and *rw\_inode* are called to re-initialize the i-node and write it to the disk. Other file types (directories, special files, and named pipes) are subjected to appropriate tests. In the case of a device, a call is made on line 24640 (using the *dmap* structure) to the appropriate routine to open the device. In the case of a named pipe, a call is made to *pipe\_open* (line 24646), and various tests relevant to pipes are made.

The code of *common\_open*, as well as many other file system procedures, contains a large amount of code that checks for various errors and illegal combinations. While not glamorous, this code is essential to having an error-free, robust file system. If something is wrong, the file descriptor and *filp* slot previously allocated are deallocated and the i-node is released (lines 24683 to 24689). In this case the value returned by *common\_open* will be a negative number, indicating an error. If there are no problems the file descriptor, a positive value, is returned.

This is a good place to discuss in more detail the operation of *new\_node* (line 24697), which does the allocation of the i-node and the entering of the path name into the file system for *creat* and *open* calls. It is also used for the *mknod* and *mkdir* calls, yet to be discussed. The statement on line 24711 parses the path name (i.e., looks it up component by component) as far as the final directory; the call to *advance* three lines later tries to see if the final component can be opened.

For example, on the call

```
fd = creat("/usr/ast/foobar", 0755);
```

*last\_dir* tries to load the i-node for */usr/ast/* into the tables and return a pointer to it. If the file does not exist, we will need this i-node shortly in order to add *foobar* to the directory. All the other system calls that add or delete files also use *last\_dir* to first open the final directory in the path.

If *new\_node* discovers that the file does not exist, it calls *alloc\_inode* on line 24717 to allocate and load a new i-node, returning a pointer to it. If no free i-nodes are left, *new\_node* fails and returns *NIL\_INODE*.

If an i-node can be allocated, the operation continues at line 24727, filling in some of the fields, writing it back to the disk, and entering the file name in the final directory (on line 24732). Again we see that the file system must constantly check for errors, and upon encountering one, carefully release all the resources, such as i-nodes and blocks that it is holding. If we were prepared to just let

MINIX 3 panic when we ran out of, say, i-nodes, rather than undoing all the effects of the current call and returning an error code to the caller, the file system would be appreciably simpler.

As mentioned above, pipes require special treatment. If there is not at least one reader/writer pair for a pipe, *pipe\_open* (line 24758) suspends the caller. Otherwise, it calls *release*, which looks through the process table for processes that are blocked on the pipe. If it is successful, the processes are revived.

The *mknod* call is handled by *do\_mknod* (line 24785). This procedure is similar to *do\_creat*, except that it just creates the i-node and makes a directory entry for it. In fact, most of the work is done by the call to *new\_node* on line 24797. If the i-node already exists, an error code will be returned. This is the same error code that was an acceptable result from *new\_node* when it was called by *common\_open*; in this case, however, the error code is passed back to the caller, which presumably will act accordingly. The case-by-case analysis we saw in *common\_open* is not needed here.

The *mkdir* call is handled by the function *do\_mkdir* (line 24805). As with the other system calls we have discussed here, *new\_node* plays an important part. Directories, unlike files, always have links and are never completely empty because every directory must contain two entries from the time of its creation: the “.” and “..” entries that refer to the directory itself and to its parent directory. The number of links a file may have is limited, it is *LINK\_MAX* (defined in *include/limits.h* as *SHRT\_MAX*, 32767 for MINIX 3 on a standard 32-bit Intel system). Since the reference to a parent directory in a child is a link to the parent, the first thing *do\_mkdir* does is to see if it is possible to make another link in the parent directory (lines 24819 and 24820). Once this test has been passed, *new\_node* is called. If *new\_node* succeeds, then the directory entries for “.” and “..” are made (lines 24841 and 24842). All of this is straightforward, but there could be failures (for instance, if the disk is full), so to avoid making a mess of things provision is made for undoing the initial stages of the process if it can not be completed.

Closing a file is easier than opening one. The work is done by *do\_close* (line 24865). Pipes and special files need some attention, but for regular files, almost all that needs to be done is to decrement the *filp* counter and check to see if it is zero, in which case the i-node is returned with *put\_inode*. The final step is to remove any locks and to revive any process that may have been suspended waiting for a lock on the file to be released.

Note that returning an i-node means that its counter in the *inode* table is decremented, so it can be removed from the table eventually. This operation has nothing to do with freeing the i-node (i.e., setting a bit in the bitmap saying that it is available). The i-node is only freed when the file has been removed from all directories.

The final procedure in *open.c* is *do\_lseek* (line 24939). When a seek is done, this procedure is called to set the file position to a new value. On line 24968

reading ahead is inhibited; an explicit attempt to seek to a position in a file is incompatible with sequential access.

## Reading a File

Once a file has been opened, it can be read or written. Many functions are used during both reading and writing. These are found in the file *read.c*. We will discuss these first and then proceed to the following file, *write.c*, to look at code specifically used for writing. Reading and writing differ in a number of ways, but they have enough similarities that all that is required of *do\_read* (line 25030) is to call the common procedure *read\_write* with a flag set to *READING*. We will see in the next section that *do\_write* is equally simple.

*Read\_write* begins on line 25038. Some special code on lines 25063 to 25066 is used by the process manager to have the file system load entire segments in user space for it. Normal calls are processed starting on line 25068. Some validity checks follow (e.g., reading from a file opened only for writing) and some variables are initialized. Reads from character special files do not go through the block cache, so they are filtered out on line 25122.

The tests on lines 25132 to 25145 apply only to writes and have to do with files that may get bigger than the device can hold, or writes that will create a hole in the file by writing *beyond* the end-of-file. As we discussed in the MINIX 3 overview, the presence of multiple blocks per zone causes problems that must be dealt with explicitly. Pipes are also special and are checked for.

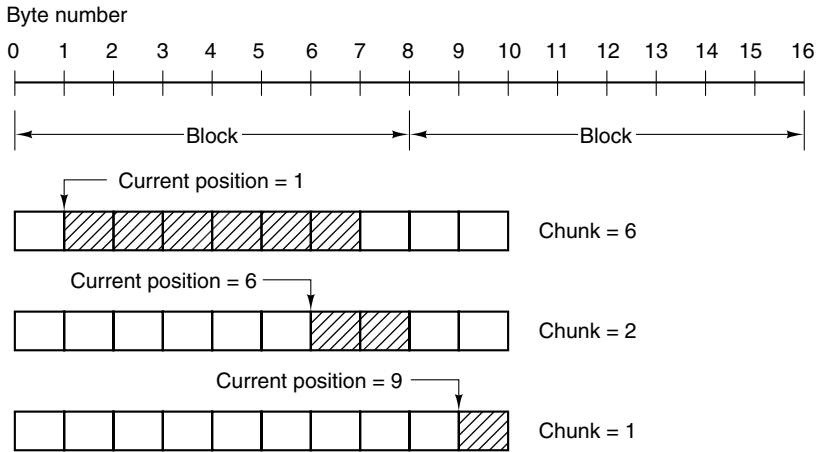
The heart of the read mechanism, at least for ordinary files, is the loop starting on line 25157. This loop breaks the request up into chunks, each of which fits in a single disk block. A chunk begins at the current position and extends until one of the following conditions is met:

1. All the bytes have been read.
2. A block boundary is encountered.
3. The end-of-file is hit.

These rules mean that a chunk never requires two disk blocks to satisfy it. Figure 5-45 shows three examples of how the chunk size is determined, for chunk sizes of 6, 2, and 1 bytes, respectively. The actual calculation is done on lines 25159 to 25169.

The actual reading of the chunk is done by *rw\_chunk*. When control returns, various counters and pointers are incremented, and the next iteration begins. When the loop terminates, the file position and other variables may be updated (e.g., pipe pointers).

Finally, if read ahead is called for, the i-node to read from and the position to read from are stored in global variables, so that after the reply message is sent to the user, the file system can start getting the next block. In many cases the file



**Figure 5-45.** Three examples of how the first chunk size is determined for a 10-byte file. The block size is 8 bytes, and the number of bytes requested is 6. The chunk is shown shaded.

system will block, waiting for the next disk block, during which time the user process will be able to work on the data it just received. This arrangement overlaps processing and I/O and can improve performance substantially.

The procedure *rw\_chunk* (line 25251) is concerned with taking an i-node and a file position, converting them into a physical disk block number, and requesting the transfer of that block (or a portion of it) to the user space. The mapping of the relative file position to the physical disk address is done by *read\_map*, which understands about i-nodes and indirect blocks. For an ordinary file, the variables *b* and *dev* on line 25280 and line 25281 contain the physical block number and device number, respectively. The call to *get\_block* on line 25303 is where the cache handler is asked to find the block, reading it in if need be. Calling *rahead* on line 25295 then ensures that the block is read into the cache.

Once we have a pointer to the block, the *sys\_vircopy* kernel call on line 25317 takes care of transferring the required portion of it to the user space. The block is then released by *put\_block*, so that it can be evicted from the cache later. (After being acquired by *get\_block*, it will not be in the LRU queue and it will not be returned there while the counter in the block's header shows that it is in use, so it will be exempt from eviction; *put\_block* decrements the counter and returns the block to the LRU queue when the counter reaches zero.) The code on line 25327 indicates whether a write operation filled the block. However, the value passed to *put\_block* in *n* does not affect how the block is placed on the queue; all blocks are now placed on the rear of the LRU chain.

*Read\_map* (line 25337) converts a logical file position to the physical block number by inspecting the i-node. For blocks close enough to the beginning of the

file that they fall within one of the first seven zones (the ones right in the i-node), a simple calculation is sufficient to determine which zone is needed, and then which block. For blocks further into the file, one or more indirect blocks may have to be read.

*Rd\_indir* (line 25400) is called to read an indirect block. The comments for this function are a bit out of date; code to support the 68000 processor has been removed and the support for the MINIX V1 file system is not used and could also be dropped. However, it is worth noting that if someone wanted to add support for other file system versions or other platforms where data might have a different format on the disk, problems of different data types and byte orders could be relegated to this file. If messy conversions were necessary, doing them here would let the rest of the file system see data in only one form.

*Read\_ahead* (line 25432) converts the logical position to a physical block number, calls *get\_block* to make sure the block is in the cache (or bring it in), and then returns the block immediately. It cannot do anything with the block, after all. It just wants to improve the chance that the block is around if it is needed soon,

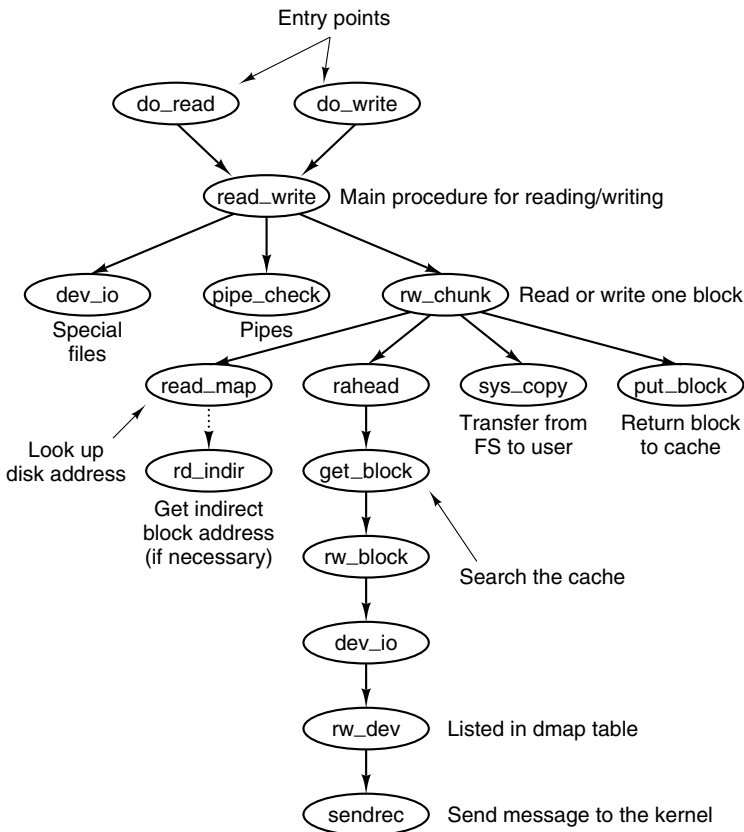
Note that *read\_ahead* is called only from the main loop in *main*. It is not called as part of the processing of the read system call. It is important to realize that the call to *read\_ahead* is performed *after* the reply is sent, so that the user will be able to continue running even if the file system has to wait for a disk block while reading ahead.

*Read\_ahead* by itself is designed to ask for just one more block. It calls the last function in *read.c*, *rahead*, to actually get the job done. *Rahead* (line 25451) works according to the theory that if a little more is good, a lot more is better. Since disks and other storage devices often take a relatively long time to locate the first block requested but then can relatively quickly read in a number of adjacent blocks, it may be possible to get many more blocks read with little additional effort. A prefetch request is made to *get\_block*, which prepares the block cache to receive a number of blocks at once. Then *rw\_scattered* is called with a list of blocks. We have previously discussed this; recall that when the device drivers are actually called by *rw\_scattered*, each one is free to answer only as much of the request as it can efficiently handle. This all sounds fairly complicated, but the complications make possible a significant speedup of applications which read large amounts of data from the disk.

Figure 5-46 shows the relations between some of the major procedures involved in reading a file—in particular, who calls whom.

## Writing a File

The code for writing to files is in *write.c*. Writing a file is similar to reading one, and *do\_write* (line 25625) just calls *read\_write* with the *WRITING* flag. A major difference between reading and writing is that writing requires allocating new disk blocks. *Write\_map* (line 25635) is analogous to *read\_map*, only instead



**Figure 5-46.** Some of the procedures involved in reading a file.

of looking up physical block numbers in the i-node and its indirect blocks, it enters new ones there (to be precise, it enters zone numbers, not block numbers).

The code of *write\_map* is long and detailed because it must deal with several cases. If the zone to be inserted is close to the beginning of the file, it is just inserted into the i-node on (line 25658).

The worst case is when a file exceeds the size that can be handled by a single-indirect block, so a double-indirect block is now required. Next, a single-indirect block must be allocated and its address put into the double-indirect block. As with reading, a separate procedure, *wr\_indir*, is called. If the double-indirect block is acquired correctly, but the disk is full so the single-indirect block cannot be allocated, then the double one must be returned to avoid corrupting the bitmap.

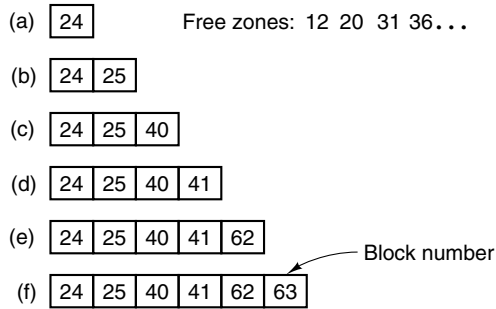
Again, if we could just toss in the sponge and panic at this point, the code would be much simpler. However, from the user's point of view it is much nicer that running out of disk space just returns an error from *write*, rather than crashing the computer with a corrupted file system.



*Wr\_indir* (line 25726) calls the conversion routines, *conv4* to do any necessary data conversion and puts a new zone number into an indirect block. (Again, there is leftover code here to handle the old V1 filesystem, but only the V2 code is currently used.) Keep in mind that the name of this function, like the names of many other functions that involve reading and writing, is not literally true. The actual writing to the disk is handled by the functions that maintain the block cache.

The next procedure in *write.c* is *clear\_zone* (line 25747), which takes care of the problem of erasing blocks that are suddenly in the middle of a file. This happens when a seek is done beyond the end of a file, followed by a write of some data. Fortunately, this situation does not occur very often.

*New\_block* (line 25787) is called by *rw\_chunk* whenever a new block is needed. Figure 5-47 shows six successive stages of the growth of a sequential file. The block size is 1-KB and the zone size is 2-KB in this example.



**Figure 5-47.** (a) – (f) The successive allocation of 1-KB blocks with a 2-KB zone.

The first time *new\_block* is called, it allocates zone 12 (blocks 24 and 25). The next time it uses block 25, which has already been allocated but is not yet in use. On the third call, zone 20 (blocks 40 and 41) is allocated, and so on. *Zero\_block* (line 25839) clears a block, erasing its previous contents. This description is considerably longer than the actual code.

## Pipes

Pipes are similar to ordinary files in many respects. In this section we will focus on the differences. The code we will discuss is all in *pipe.c*.

First of all, pipes are created differently, by the *pipe* call, rather than the *creat* call. The *pipe* call is handled by *do\_pipe* (line 25933). All *do\_pipe* really does is allocate an i-node for the pipe and return two file descriptors for it. Pipes are owned by the system, not by the user, and are located on the designated pipe de-

vice (configured in *include/minix/config.h*), which could very well be a RAM disk, since pipe data do not have to be preserved permanently.

Reading and writing a pipe is slightly different from reading and writing a file, because a pipe has a finite capacity. An attempt to write to a pipe that is already full will cause the writer to be suspended. Similarly, reading from an empty pipe will suspend the reader. In effect, a pipe has two pointers, the current position (used by readers) and the size (used by writers), to determine where data come from or go to.

The various checks to see if an operation on a pipe is possible are carried out by *pipe\_check* (line 25986). In addition to the above tests, which may lead to the caller being suspended, *pipe\_check* calls *release* to see if a process previously suspended due to no data or too much data can now be revived. These revivals are done on line 26017 and line 26052, for sleeping writers and readers, respectively. Writing on a broken pipe (no readers) is also detected here.

The act of suspending a process is done by *suspend* (line 26073). All it does is save the parameters of the call in the process table and set the flag *dont\_reply* to *TRUE*, to inhibit the file system's reply message.

The procedure *release* (line 26099) is called to check if a process that was suspended on a pipe can now be allowed to continue. If it finds one, it calls *revive* to set a flag so that the main loop will notice it later. This function is not a system call, but is listed in Fig. 5-33(c) because it uses the message-passing mechanism.

The last procedure in *pipe.c* is *do\_unpause* (line 26189). When the process manager is trying to signal a process, it must find out if that process is hanging on a pipe or special file (in which case it must be awakened with an *EINTR* error). Since the process manager knows nothing about pipes or special files, it sends a message to the file system to ask. That message is processed by *do\_unpause*, which revives the process, if it is blocked. Like *revive*, *do\_unpause* has some similarity to a system call, although it is not one.

The last two functions in *pipe.c*, *select\_request\_pipe* (line 26247) and *select\_match\_pipe* (line 26278), support the *select* call, which is not discussed here.

### 5.7.5 Directories and Paths

We have now finished looking at how files are read and written. Our next task is to see how path names and directories are handled.

#### Converting a Path to an I-Node

Many system calls (e.g., *open*, *unlink*, and *mount*) have path names (i.e., file names) as a parameter. Most of these calls must fetch the i-node for the named file before they can start working on the call itself. How a path name is converted

to an i-node is a subject we will now look at in detail. We already saw the general outline in Fig. 5-16.

The parsing of path names is done in the file *path.c*. The first procedure, *eat\_path* (line 26327), accepts a pointer to a path name, parses it, arranges for its i-node to be loaded into memory, and returns a pointer to the i-node. It does its work by calling *last\_dir* to get the i-node to the final directory and then calling *advance* to get the final component of the path. If the search fails, for example, because one of the directories along the path does not exist, or exists but is protected against being searched, *NIL\_INODE* is returned instead of a pointer to the i-node.

Path names may be absolute or relative and may have arbitrarily many components, separated by slashes. These issues are dealt with by *last\_dir*, which begins by examining the first character of the path name to see if it is an absolute path or a relative one (line 26371). For absolute paths, *rip* is set to point to the root i-node; for relative ones, it is set to point to the i-node for the current working directory.

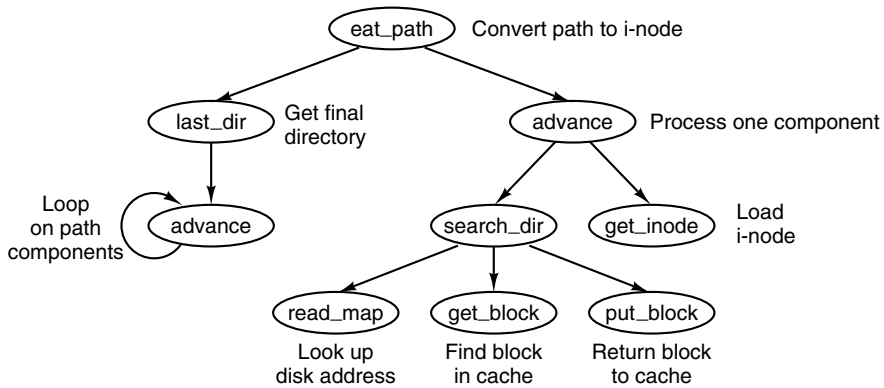
At this point, *last\_dir* has the path name and a pointer to the i-node of the directory to look up the first component in. It enters a loop on line 26382 now, parsing the path name, component by component. When it gets to the end, it returns a pointer to the final directory.

*Get\_name* (line 26413) is a utility procedure that extracts components from strings. More interesting is *advance* (line 26454), which takes as parameters a directory pointer and a string, and looks up the string in the directory. If it finds the string, *advance* returns a pointer to its i-node. The details of transferring across mounted file systems are handled here.

Although *advance* controls the string lookup, the actual comparison of the string against the directory entries is done in *search\_dir* (line 26535), which is the only place in the file system where directory files are actually examined. It contains two nested loops, one to loop over the blocks in a directory, and one to loop over the entries in a block. *Search\_dir* is also used to enter and delete names from directories. Figure 5-48 shows the relationships between some of the major procedures used in looking up path names.

## Mounting File Systems

Two system calls that affect the file system as a whole are *mount* and *umount*. They allow independent file systems on different minor devices to be “glued” together to form a single, seamless naming tree. Mounting, as we saw in Fig. 5-38, is effectively achieved by reading in the root i-node and superblock of the file system to be mounted and setting two pointers in its superblock. One of them points to the i-node mounted on, and the other points to the root i-node of the mounted file system. These pointers hook the file systems together.



**Figure 5-48.** Some of the procedures used in looking up path names.

The setting of these pointers is done in the file *mount.c* by *do\_mount* on lines 26819 and 26820. The two pages of code that precede setting the pointers are almost entirely concerned with checking for all the errors that can occur while mounting a file system, among them:

1. The special file given is not a block device.
2. The special file is a block device but is already mounted.
3. The file system to be mounted has a rotten magic number.
4. The file system to be mounted is invalid (e.g., no i-nodes).
5. The file to be mounted on does not exist or is a special file.
6. There is no room for the mounted file system's bitmaps.
7. There is no room for the mounted file system's superblock.
8. There is no room for the mounted file system's root i-node.

Perhaps it seems inappropriate to keep harping on this point, but the reality of any practical operating system is that a substantial fraction of the code is devoted to doing minor chores that are not intellectually very exciting but are crucial to making a system usable. If a user attempts to mount the wrong floppy disk by accident, say, once a month, and this leads to a crash and a corrupted file system, the user will perceive the system as being unreliable and blame the designer, not himself.

The famous inventor Thomas Edison once made a remark that is relevant here. He said that “genius” is 1 percent inspiration and 99 percent perspiration. The difference between a good system and a mediocre one is not the brilliance of the former's scheduling algorithm, but its attention to getting all the details right.

Unmounting a file system is easier than mounting one—there are fewer things that can go wrong. *Do\_umount* (line 26828) is called to start the job, which is divided into two parts. *Do\_umount* itself checks that the call was made by the superuser, converts the name into a device number, and then calls *unmount* (line 26846), which completes the operation. The only real issue is making sure that no process has any open files or working directories on the file system to be removed. This check is straightforward: just scan the whole i-node table to see if any i-nodes in memory belong to the file system to be removed (other than the root i-node). If so, the *umount* call fails.

The last procedure in *mount.c* is *name\_to\_dev* (line 26893), which takes a special file pathname, gets its i-node, and extracts its major and minor device numbers. These are stored in the i-node itself, in the place where the first zone would normally go. This slot is available because special files do not have zones.

## Linking and Unlinking Files

The next file to consider is *link.c*, which deals with linking and unlinking files. The procedure *do\_link* (line 27034) is very much like *do\_mount* in that nearly all of the code is concerned with error checking. Some of the possible errors that can occur in the call

```
link(file_name, link_name);
```

are listed below:

1. *File\_name* does not exist or cannot be accessed.
2. *File\_name* already has the maximum number of links.
3. *File\_name* is a directory (only superuser can link to it).
4. *Link\_name* already exists.
5. *File\_name* and *link\_name* are on different devices.

If no errors are present, a new directory entry is made with the string *link\_name* and the i-node number of *file\_name*. In the code, *name1* corresponds to *file\_name* and *name2* corresponds to *link\_name*. The actual entry is made by *search\_dir*, called from *do\_link* on line 27086.

Files and directories are removed by unlinking them. The work of both the *unlink* and *rmdir* system calls is done by *do\_unlink* (line 27104). Again, a variety of checks must be made; testing that a file exists and that a directory is not a mount point are done by the common code in *do\_unlink*, and then either *remove\_dir* or *unlink\_file* is called, depending upon the system call being supported. We will discuss these shortly.

The other system call supported in *link.c* is *rename*. UNIX users are familiar with the *mv* shell command which ultimately uses this call; its name reflects

another aspect of the call. Not only can it change the name of a file within a directory, it can also effectively move the file from one directory to another, and it can do this atomically, which prevents certain race conditions. The work is done by *do\_rename* (line 27162). Many conditions must be tested before this command can be completed. Among these are:

1. The original file must exist (line 27177).
2. The old pathname must not be a directory above the new pathname in the directory tree (lines 27195 to 27212).
3. Neither `.` nor `..` is acceptable as an old or new name (lines 27217 and 27218).
4. Both parent directories must be on the same device (line 27221).
5. Both parent directories must be writable, searchable, and on a writable device (lines 27224 and 27225).
6. Neither the old nor the new name may be a directory with a file system mounted upon it.

Some other conditions must be checked if the new name already exists. Most importantly it must be possible to remove an existing file with the new name.

In the code for *do\_rename* there are a few examples of design decisions that were taken to minimize the possibility of certain problems. Renaming a file to a name that already exists could fail on a full disk, even though in the end no additional space is used, if the old file were not removed first, and this is what is done at lines 27260 to 27266. The same logic is used at line 27280, removing the old file name before creating a new name in the same directory, to avoid the possibility that the directory might need to acquire an additional block. However, if the new file and the old file are to be in different directories, that concern is not relevant, and at line 27285 a new file name is created (in a different directory) before the old one is removed, because from a system integrity standpoint a crash that left two filenames pointing to an i-node would be much less serious than a crash that left an i-node not pointed to by any directory entry. The probability of running out of space during a rename operation is low, and that of a system crash even lower, but in these cases it costs nothing more to be prepared for the worst case.

The remaining functions in *link.c* support the ones that we have already discussed. In addition, the first of them, *truncate* (line 27316), is called from several other places in the file system. It steps through an i-node one zone at a time, freeing all the zones it finds, as well as the indirect blocks. *Remove\_dir* (line 27375) carries out a number of additional tests to be sure the directory can be removed, and then it in turn calls *unlink\_file* (line 27415). If no errors are found, the directory entry is cleared and the link count in the i-node is reduced by one.

### 5.7.6 Other System Calls

The last group of system calls is a mixed bag of things involving status, directories, protection, time, and other services.

#### Changing Directories and File Status

The file *stadir.c* contains the code for six system calls: *chdir*, *fchdir*, *chroot*, *stat*, *fstat*, and *fstatfs*. In studying *last\_dir* we saw how path searches start out by looking at the first character of the path, to see if it is a slash or not. Depending on the result, a pointer is then set to the working directory or the root directory.

Changing from one working directory (or root directory) to another is just a matter of changing these two pointers within the caller's process table. These changes are made by *do\_chdir* (line 27542) and *do\_chroot* (line 27580). Both of them do the necessary checking and then call *change* (line 27594), which does some more tests, then calls *change\_into* (line 27611) to open the new directory and replace the old one.

*Do\_fchdir* (line 27529) supports *fchdir*, which is an alternate way of effecting the same operation as *chdir*, with the calling argument a file descriptor rather than a path. It tests for a valid descriptor, and if the descriptor is valid it calls *change\_into* to do the job.

In *do\_chdir* the code on lines 27552 to 27570 is not executed on *chdir* calls made by user processes. It is specifically for calls made by the process manager, to change to a user's directory for the purpose of handling *exec* calls. When a user tries to execute a file, say, *a.out* in his working directory, it is easier for the process manager to change to that directory than to try to figure out where it is.

The two system calls *stat* and *fstat* are basically the same, except for how the file is specified. The former gives a path name, whereas the latter provides the file descriptor of an open file, similar to what we saw for *chdir* and *fchdir*. The top-level procedures, *do\_stat* (line 27638) and *do\_fstat* (line 27658), both call *stat\_inode* to do the work. Before calling *stat\_inode*, *do\_stat* opens the file to get its i-node. In this way, both *do\_stat* and *do\_fstat* pass an i-node pointer to *stat\_inode*.

All *stat\_inode* (line 27673) does is to extract information from the i-node and copy it into a buffer. The buffer must be explicitly copied to user space by a *sys\_datacopy* kernel call on lines 27713 and 27714 because it is too large to fit in a message.

Finally, we come to *do\_fstatfs* (line 27721). *Fstatfs* is not a POSIX call, although POSIX defines a similar *fstatvfs* call which returns a much bigger data structure. The MINIX 3 *fstatfs* returns only one piece of information, the block size of a file system. The prototype for the call is

```
_PROTOTYPE( int fstatfs, (int fd, struct statfs *st) );
```

The *statfs* structure it uses is simple, and can be displayed on a single line:

```
struct statfs { off_t f_bsize; /* file system block size */};
```

These definitions are in *include/sys/statfs.h*, which is not listed in Appendix B.

## Protection

The MINIX 3 protection mechanism uses the *rwX* bits. Three sets of bits are present for each file: for the owner, for his group, and for others. The bits are set by the *chmod* system call, which is carried out by *do\_chmod*, in file *protect.c* (line 27824). After making a series of validity checks, the mode is changed on line 27850.

The *chown* system call is similar to *chmod* in that both of them change an internal i-node field in some file. The implementation is also similar although *do\_chown* (line 27862) can be used to change the owner only by the superuser. Ordinary users can use this call to change the group of their own files.

The *umask* system call allows the user to set a mask (stored in the process table), which then masks out bits in subsequent *creat* system calls. The complete implementation would be only one statement, line 27907, except that the call must return the old mask value as its result. This additional burden triples the number of lines of code required (lines 27906 to 27908).

The *access* system call makes it possible for a process to find out if it can access a file in a specified way (e.g., for reading). It is implemented by *do\_access* (line 27914), which fetches the file's i-node and calls the internal procedure, *forbidden* (line 27938), to see if the access is forbidden. *Forbidden* checks the uid and gid, as well as the information in the i-node. Depending on what it finds, it selects one of the three *rwX* groups and checks to see if the access is permitted or forbidden.

*Read\_only* (line 27999) is a little internal procedure that tells whether the file system on which its i-node parameter is located is mounted read only or read-write. It is needed to prevent writes on file systems mounted read only.

## 5.7.7 The I/O Device Interface

As we have mentioned more than once, a design goal was to make MINIX 3 a more robust operating system by having all device drivers run as user-space processes without direct access to kernel data structures or kernel code. The primary advantage of this approach is that a faulty device driver will not cause the entire system to crash, but there are some other implications of this approach. One is that device drivers not needed immediately upon startup can be started at any time after startup is complete. This also implies that a device driver can be stopped, restarted, or replaced by a different driver for the same device at any time while the system is running. This flexibility is subject, of course to some



restrictions—you cannot start multiple drivers for the same device. However, if the hard disk driver crashes, it can be restarted from a copy on the RAM disk.

MINIX 3 device drivers are accessed from the file system. In response to user requests for I/O the file system sends messages to the user-space device drivers. The *dmap* table has an entry for every possible major device type. It provides the mapping between the major device number and the corresponding device driver. The next two files we will consider deal with the *dmap* table. The table itself is declared in *dmap.c*. This file also supports initialization of the table and a new system call, *devctl*, which is intended to support starting, stopping, and restarting of device drivers. After that we will look at *device.c* which supports normal run-time operations on devices, such as *open*, *close*, *read*, *write*, and *ioctl*.

When a device is opened, closed, read, or written, *dmap* provides the name of the procedure to call to handle the operation. All of these procedures are located in the file system's address space. Many of these procedures do nothing, but some call a device driver to request actual I/O. The process number corresponding to each major device is also provided by the table.

Whenever a new major device is added to MINIX 3, a line must be added to this table telling what action, if any, is to be taken when the device is opened, closed, read, or written. As a simple example, if a tape drive is added to MINIX 3, when its special file is opened, the procedure in the table could check to see if the tape drive is already in use.

*Dmap.c* begins with a macro definition, *DT* (lines 28115 to 28117), which is used to initialize the *dmap* table. This macro makes it easier to add a new device driver when reconfiguring MINIX 3. Elements of the *dmap* table are defined in *include/minix/dmap.h*; each element consists of a pointer to a function to be called on an *open* or *close*, another pointer to a function to be called on a *read* or *write*, a process number (index into process table, not a PID), and a set of flags. The actual table is an array of such elements, declared on line 28132. This table is globally available within the file server. The size of the table is *NR\_DEVICES*, which is 32 in the version of MINIX 3 described here, and almost twice as big as needed for the number of devices currently supported. Fortunately, the C language behavior of setting all uninitialized variables to zero will ensure that no spurious information appears in unused slots.

Following the declaration of *dmap* is a *PRIVATE* declaration of *init\_dmap*. It is defined by an array of *DT* macros, one for each possible major device. Each of these macros expands to initialize an entry in the global array at compile time. A look at a few of the macros will help with understanding how they are used. *Init\_dmap[1]* defines the entry for the memory driver, which is major device 1. The macro looks like this:

```
DT(1, gen_opcl, gen_io, MEM_PROC_NR, 0)
```

The memory driver is always present and is loaded with the system boot image. The “1” as first parameter means that this driver must be present. In this case, a

pointer to *gen\_opcl* will be entered as the function to call to open or close, and a pointer to *gen\_io* will be entered to specify the function to call for reading or writing, *MEM\_PROC\_NR* tells which slot in the process table the memory driver uses, and “0” means no flags are set. Now look at the next entry, *init\_dmap[2]*. This is the entry for the floppy disk driver, and it looks like this:

```
DT(0, no_dev, 0, 0, DMAP_MUTABLE)
```

The first “0” indicates this entry is for a driver not required to be in the boot image. The default for the first pointer field specifies a call to *no\_dev* on an attempt to open the device. This function returns an *ENODEV* “no such device” error to the caller. The next two zeros are also defaults: since the device cannot be opened there is no need to specify a function to call to do I/O, and a zero in the process table slot is interpreted as no process specified. The meaning of the flag *DMAP\_MUTABLE* is that changes to this entry are permitted. (Note that the absence of this flag for the memory driver entry means its entry cannot be changed after initialization.) MINIX 3 can be configured with or without a floppy disk driver in the boot image. If the floppy disk driver is in the boot image and it is specified by a *label=FLOPPY* boot parameter to be the default disk device, this entry will be changed when the file system starts. If the floppy driver is not in the boot image, or if it is in the image but is not specified to be the default disk device, this field will not be changed when FS starts. However, it is still possible for the floppy driver to be activated later. Typically this is done by the */etc/rc* script run when *init* is run.

*Do\_devctl* (line 28157) is the first function executed to service a *devctl* call. The current version is very simple, it recognizes two requests, *DEV\_MAP* and *DEV\_UNMAP*, and the latter returns a *ENOSYS* error, which means “function not implemented.” Obviously, this is a stopgap. In the case of *DEV\_MAP* the next function, *map\_driver* is called.

It might be helpful to describe how the *devctl* call is used, and plans for its use in the future. A server process, the **reincarnation server (RS)** is used in MINIX 3 to support starting user-space servers and drivers after the operating system is up and running. The interface to the reincarnation server is the *service* utility, and examples of its use can be seen in */etc/rc*. An example is

```
service up /sbin/floppy -dev /dev/fd0
```

This action results in the reincarnation server making a *devctl* call to start the binary */sbin/floppy* as the device driver for the device special file */dev/fd0*. To do this, RS execs the specified binary, but sets a flag that inhibits it from running until it has been transformed into a system process. Once the process is in memory and its slot number in the process table is known, the major device number for the specified device is determined. This information is then included in a message to the file server that requested the *devctl DEV\_MAP* operation. This is the

most important part of the reincarnation server's job from the point of view of initializing the I/O interface. For the sake of completeness we will also mention that to complete initialization of the device driver, RS also makes a `sys_privctl` call to have the system task initialize the driver process's *priv* table entry and allow it to execute. Recall from Chapter 2 that a dedicated *priv* table slot is what makes an otherwise ordinary user-space process into a system process.

The reincarnation server is new, and in the release of MINIX 3 described here it is still rudimentary. Plans for future releases of MINIX 3 include a more powerful reincarnation server that will be able to stop and restart drivers in addition to starting them. It will also be able to monitor drivers and restart them automatically if problems develop. Check the Web site ([www.minix3.org](http://www.minix3.org)) and the newsgroup (*comp.os.minix*) for the current status.

Continuing with *dmap.c*, the function *map\_driver* begins on line 28178. Its operation is straightforward. If the *DMAP\_MUTABLE* flag is set for the entry in the *dmap* table, appropriate values are written into each entry. Three different variants of the function for handling opening and closing of the device are available; one is selected by a *style* parameter passed in the message from RS to the file system (lines 28204 to 28206). Notice that *dmap\_flags* is not altered. If the entry was marked *DMAP\_MUTABLE* originally it retains this status after the *devctl* call.

The third function in *dmap.c* is *build\_map*. This is called by *fs\_init* when the file system is first started, before it enters its main loop. The first thing done is to loop over all of the entries in the local *init\_dmap* table and copy the expanded macros to the global *dmap* table for each entry that does not have *no\_dev* specified as the *dmap\_opcl* member. This correctly initializes these entries. Otherwise the default values for an uninitialized driver are set in place in *dmap*. The rest of *build\_map* is more interesting. A boot image can be built with multiple disk device drivers. By default *at\_wini*, *bios\_wini*, and *floppy* drivers are added to the boot image by the *Makefile* in the *src/tools/*. A label is added to each of these, and a *label=* item in the boot parameters determines which one will actually be loaded in the image and activated as the default disk driver. The *env\_get\_param* calls on line 28248 and line 28250 use library routines that ultimately use the *sys\_getinfo* kernel call to get the *label* and *controller* boot parameter strings. Finally, *build\_map* is called on line 28267 to modify the entry in *dmap* that corresponds to the boot device. The key thing here is setting the process number to *DRVR\_PROC\_NR*, which happens to be slot 6 in the process table. This slot is magic; the driver in this slot is the default driver.

Now we come to the file *device.c*, which contains the procedures needed for device I/O at run time.

The first one is *dev\_open* (line 28334). It is called by other parts of the file system, most often from *common\_open* in *main.c* when an open operation is determined to be accessing a device special file, but also from *load\_ram* and *do\_mount*. Its operation is typical of several procedures we will see here. It de-

termines the major device number, verifies that it is valid, and then uses it to set a pointer to an entry in the *dmap* table, and then makes a call to the function pointed to in that entry, at line 28349:

```
r = (*dp->dmap_opcl)(DEV_OPEN, dev, proc, flags)
```

In the case of a disk drive, the function called will be *gen\_opcl*, in the case of a terminal device it will be *tty\_opcl*. If a *SUSPEND* return code is received there is a serious problem; an open call should not fail this way.

The next call, *dev\_close* (line 28357) is simpler. It is not expected that a call will be made to an invalid device, and no harm is done if a close operation fails, so the code is shorter than this text describing it, just one line that will end up calling the same *\*\_opcl* procedure as *dev\_open* called when the device was opened.

When the file system receives a notification message from a device driver *dev\_status* (line 28366) is called. A notification means an event has occurred, and this function is responsible for finding out what kind of event and initiating appropriate action. The origin of the notification is specified as a process number, so the first step is to search through the *dmap* table to find an entry that corresponds to the notifying process (lines 18371 to 18373). It is possible the notification could have been bogus, so it is not an error if no corresponding entry is found and *dev\_status* returns without finding a match. If a match is found, the loop on lines 28378 to 28398 is entered. On each iteration a message is sent to the driver process requesting its status. Three possible reply types are expected. A *DEV\_REVIVE* message may be received if the process that originally requested I/O was previously suspended. In this case *revive* (in *pipe.c*, line 26146) is called. A *DEV\_IO\_READY* message may be received if a *select* call has been made on the device. Finally, a *DEV\_NO\_STATUS* message may be received, and is, in fact expected, but possibly not until one or both of the first two message types are received. For this reason, the *get\_more* variable is used to cause the loop to repeat until the *DEV\_NO\_STATUS* message is received.

When actual device I/O is needed, *dev\_io* (line 28406) is called from *read\_write* (line 25124) to handle character special files, and from *rw\_block* (line 22661) to handle block special files. It builds a standard message (see Fig. 3-17) and sends it to the specified device driver by calling either *gen\_io* or *ctty\_io* as specified in the *dp->dmap\_driver* field of the *dmap* table. While *dev\_io* is waiting for a reply from the driver, the file system waits. It has no internal multiprocessing. Usually, these waits are quite short though (e.g., 50 msec). But it is possible no data will be available—this is especially likely if the data was requested from a terminal device. In that case the reply message may indicate *SUSPEND*, to temporarily suspend the calling application but let the file system continue.

The procedure *gen\_opcl* (line 28455) is called for disk devices, whether floppy disks, hard disks, or memory-based devices. A message is constructed, and, as with reading and writing, the *dmap* table is used to determine whether

*gen\_io* or *ctty\_io* will be used to send the message to the driver process for the device. *Gen\_opcl* is also used to close the same devices.

To open a terminal device *tty\_opcl* (line 28482) is called. It calls *gen\_opcl* after possibly modifying the flags, and if the call made the tty the controlling tty for the active process this is recorded in the process table *fp\_tty* entry for that process.

The device */dev/tty* is a fiction which does not correspond to any particular device. This is a magic designation that an interactive user can use to refer to his own terminal, no matter which physical terminal is actually in use. To open or close */dev/tty*, a call is made to *ctty\_opcl* (line 28518). It determines whether the *fp\_tty* process table entry for the current process has indeed been modified by a previous *ctty\_opcl* call to indicate a controlling tty.

The *setsid* system call requires some work by the file system, and this is performed by *do\_setsid* (line 28534). It modifies the process table entry for the current process to record that the process is a session leader and has no controlling process.

One system call, *ioctl*, is handled primarily in *device.c*. This call has been put here because it is closely tied to the device driver interface. When an *ioctl* is done, *do\_ioctl* (line 28554) is called to build a message and send it to the proper device driver.

To control terminal devices one of the functions declared in *include/termios.h* should be used in programs written to be POSIX compliant. The C library will translate such functions into *ioctl* calls. For devices other than terminals *ioctl* is used for many operations, many of which were described in Chap. 3.

The next function, *gen\_io* (line 28575), is the real workhorse of this file. Whether the operation on a device is an open or a close, a read or a write, or an *ioctl* this function is called to complete the work. Since */dev/tty* is not a physical device, when a message that refers to it must be sent, the next function, *ctty\_io* (line 28652), finds the correct major and minor device and substitutes them into the message before passing the message on. The call is made using the *dmap* entry for the physical device that is actually in use. As MINIX 3 is currently configured a call to *gen\_io* will result.

The function *no\_dev* (line 28677), is called from slots in the table for which a device does not exist, for example when a network device is referenced on a machine with no network support. It returns an *ENODEV* status. It prevents crashes when nonexistent devices are accessed.

The last function in *device.c* is *clone\_opcl* (line 28691). Some devices need special processing upon open. Such a device is “cloned,” that is, on a successful open it is replaced by a new device with a new unique minor device number. In MINIX 3 as described here this capability is not used. However, it is used when networking is enabled. A device that needs this will, of course, have an entry in the *dmap* table that specifies *clone\_opcl* in the *dmap\_opcl* field. This is accomplished by a call from the reincarnation server that specifies *STYLE\_CLONE*.

When *clone\_opcl* opens a device the operation starts in exactly the same way as *gen\_opcl*, but on the return a new minor device number may be returned in the *REP\_STATUS* field of the reply message. If so, a temporary file is created if it is possible to allocate a new i-node. A visible directory entry is not created. That is not necessary, since the file is already open.

## Time

Associated with each file are three 32-bit numbers relating to time. Two of these record the times when the file was last accessed and last modified. The third records when the status of the i-node itself was last changed. This time will change for almost every access to a file except a *read* or *exec*. These times are kept in the i-node. With the *utime* system call, the access and modification times can be set by the owner of the file or the superuser. The procedure *do\_utime* (line 28818) in file *time.c* performs the system call by fetching the i-node and storing the time in it. At line 28848 the flags that indicate a time update is required are reset, so the system will not make an expensive and redundant call to *clock\_time*.

As we saw in the previous chapter, the real time is determined by adding the time since the system was started (maintained by the clock task) to the real time when startup occurred. The *stime* system call returns the real time. Most of its work is done by the process manager, but the file system also maintains a record of the startup time in a global variable, *boottime*. The process manager sends a message to the file system whenever a *stime* call is made. The file system's *do\_stime* (line 28859) updates *boottime* from this message.

## 5.7.8 Additional System Call Support

There are a number of files that are not listed in Appendix B, but which are required to compile a working system. In this section we will review some files that support additional system calls. In the next section we will mention files and functions that provide more general support for the file system.

The file *misc.c* contains procedures for a few system and kernel calls that do not fit in anywhere else.

*Do\_getsysinfo* is an interface to the *sys\_datacopy* kernel call. It is meant to support the information server (IS) for debugging purposes. It allows IS to request a copy of file system data structures so it can display them to the user.

The *dup* system call duplicates a file descriptor. In other words, it creates a new file descriptor that points to the same file as its argument. The call has a variant *dup2*. Both versions of the call are handled by *do\_dup*. This function is included in MINIX 3 to support old binary programs. Both of these calls are obsolete. The current version of the MINIX 3 C library will invoke the *fcntl* system call when either of these are encountered in a C source file.

Operation	Meaning
F_DUPFD	Duplicate a file descriptor
F_GETFD	Get the close-on-exec flag
F_SETFD	Set the close-on-exec flag
F_GETFL	Get file status flags
F_SETFL	Set file status flags
F_GETLK	Get lock status of a file
F_SETLK	Set read/write lock on a file
F_SETLKW	Set write lock on a file

**Figure 5-49.** The POSIX request parameters for the FCNTL system call.

Fcntl, handled by *do\_fcntl* is the preferred way to request operations on an open file. Services are requested using POSIX-defined flags described in Fig. 5-49. The call is invoked with a file descriptor, a request code, and additional arguments as necessary for the particular request. For instance, the equivalent of the old call

```
dup2(fd, fd2);
```

would be

```
fcntl(fd, F_DUPFD, fd2);
```

Several of these requests set or read a flag; the code consists of just a few lines. For instance, the *F\_SETFD* request sets a bit that forces closing of a file when its owner process does an *exec*. The *F\_GETFD* request is used to determine whether a file must be closed when an *exec* call is made. The *F\_SETFL* and *F\_GETFL* requests permit setting flags to indicate a particular file is available in nonblocking mode or for append operations.

*Do\_fcntl* handles file locking, also. A call with the *F\_GETLK*, *F\_SETLK*, or *F\_SETLKW* command specified is translated into a call to *lock\_op*, discussed in an earlier section.

The next system call is *sync*, which copies all blocks and i-nodes that have been modified since being loaded back to the disk. The call is processed by *do\_sync*. It simply searches through all the tables looking for dirty entries. The i-nodes must be processed first, since *rw\_inode* leaves its results in the block cache. After all dirty i-nodes are written to the block cache, then all dirty blocks are written to the disk.

The system calls *fork*, *exec*, *exit*, and *set* are really process manager calls, but the results have to be posted here as well. When a process forks, it is essential that the kernel, process manager, and file system all know about it. These “system calls” do not come from user processes, but from the process manager.

*Do\_fork*, *do\_exit*, and *do\_set* record the relevant information in the file system's part of the process table. *Do\_exec* searches for and closes (using *do\_close*) any files marked to be closed-on-exec.

The last function in *misc.c* is not really a system call but is handled like one. *Do\_revive* is called when a device driver that was previously unable to complete work that the file system had requested, such as providing input data for a user process, has now completed the work. The file system then revives the process and sends it the reply message.

One system call merits a header file as well as a C source file to support it. *Select.h* and *select.c* provide support for the *select* system call. *Select* is used when a single process has to do deal with multiple I/O streams, as, for instance, a communications or network program. Describing it in detail is beyond the scope of this book.

### 5.7.9 File System Utilities

The file system contains a few general purpose utility procedures that are used in various places. They are collected together in the file *utility.c*.

*Clock\_time* sends messages to the system task to find out what the current real time is.

*Fetch\_name* is needed because many system calls have a file name as parameter. If the file name is short, it is included in the message from the user to the file system. If it is long, a pointer to the name in user space is put in the message. *Fetch\_name* checks for both cases, and either way, gets the name.

Two functions here handle general classes of errors. *No\_sys* is the error handler that is called when the file system receives a system call that is not one of its calls. *Panic* prints a message and tells the kernel to throw in the towel when something catastrophic happens. Similar functions can be found in *pm/utility.c* in the process manager's source directory.

The last two functions, *conv2* and *conv4*, exist to help MINIX 3 deal with the problem of differences in byte order between different CPU families. These routines are called when reading from or writing to a disk data structure, such as an i-node or bitmap. The byte order in the system that created the disk is recorded in the superblock. If it is different from the order used by the local processor the order will be swapped. The rest of the file system does not need to know anything about the byte order on the disk.

Finally, there are two other files that provide specialized utility services to the file manager. The file system can ask the system task to set an alarm for it, but if it needs more than one timer it can maintain its own linked list of timers, similar to what we saw for the process manager in the previous chapter. The file *timers.c* provides this support for the file system. Finally, MINIX 3 implements a unique way of using a CD-ROM that hides a simulated MINIX 3 disk with several partitions on a CD-ROM, and allows booting a live MINIX 3 system from the CD-



ROM. The MINIX 3 files are not visible to operating systems that support only standard CD-ROM file formats. The file *cdprobe.c* is used at boot time to locate a CD-ROM device and the files on it needed to start MINIX 3.

### 5.7.10 Other MINIX 3 Components

The process manager discussed in the previous chapter and the file system discussed in this chapter are user-space servers which provide support that would be integrated into a monolithic kernel in an operating system of conventional design. These are not the only server processes in a MINIX 3 system, however. There are other user-space processes that have system privileges and should be considered part of the operating system. We do not have enough space in this book to discuss their internals, but we should at least mention them here.

One we have already mentioned in this chapter. This is the reincarnation server, RS, which can start an ordinary process and turn it into a system process. It is used in the current version of MINIX 3 to launch device drivers that are not part of the system boot image. In future releases it will also be able to stop and restart drivers, and, indeed, to monitor drivers and stop and restart them automatically if they seem to be malfunctioning. The source code for the reincarnation server is in the *src/servers/rs/* directory.

Another server that has been mentioned in passing is the information server, IS. It is used to generate the debugging dumps that can be triggered by pressing the function keys on a PC-style keyboard. The source code for the information server is in the *src/servers/is/* directory.

The information server and the reincarnation servers are relatively small programs. There is a third, optional, server, the network server, or INET. It is quite large. The INET program image on disk is comparable in size to the MINIX 3 boot image. It is started by the reincarnation server in much the same way that device drivers are started. The inet source code is in the *src/servers/inet/* directory.

Finally, we will mention one other system component which is considered a device driver, not a server. This is the log driver. With so many different components of the operating system running as independent processes, it is desirable to provide a standardized way of handling diagnostic, warning, and error messages. The MINIX 3 solution is to have a device driver for a pseudo-device known as */dev/klog* which can receive messages and handle writing them to a file. The source code for the log driver is in the *src/drivers/log/* directory.

## 5.8 SUMMARY

When seen from the outside, a file system is a collection of files and directories, plus operations on them. Files can be read and written, directories can be created and destroyed, and files can be moved from directory to directory. Most