2.6 IMPLEMENTATION OF PROCESSES IN MINIX 3

We are now moving closer to looking at the actual code, so a few words about the notation we will use are perhaps in order. The terms "procedure," "function," and "routine" will be used interchangeably. Names of variables, procedures, and files will be written in italics, as in *rw_flag*. When a variable, procedure, or file name starts a sentence, it will be capitalized, but the actual names begin with lower case letters. There are a few exceptions, the tasks which are compiled into the kernel are identified by upper case names, such as *CLOCK*, *SYSTEM*, and *IDLE*. System calls will be in lower case Helvetica, for example, read.

The book and the software, both of which are continuously evolving, did not "go to press" on the same day, so there may be minor discrepancies between the references to the code, the printed listing, and the CD-ROM version. Such differences generally only affect a line or two, however. The source code printed in the book has been simplified by omitting code used to compile options that are not discussed in the book. The complete version is on the CD-ROM. The MINIX 3 Web site (*www.minix3.org*) has the current version, which has new features and additional software and documentation.

2.6.1 Organization of the MINIX 3 Source Code

The implementation of MINIX 3 as described in this book is for an IBM PC-type machine with an advanced processor chip (e.g., 80386, 80486, Pentium, Pentium Pro, II, III, 4, M, or D) that uses 32-bit words. We will refer to all of these as Intel 32-bit processors. The full path to the C language source code on a standard Intel-based platform is */usr/src/* (a trailing "/" in a path name indicates that it refers to a directory). The source directory tree for other platforms may be in a different location. Throughout the book, MINIX 3 source code files will be referred to using a path starting with the top *src/* directory. An important subdirectory of the source tree is *src/include/*, where the master copy of the C header files are located. We will refer to this directory as *include/*.

Each directory in the source tree contains a file named **Makefile** which directs the operation of the UNIX-standard *make* utility. The *Makefile* controls compilation of files in its directory and may also direct compilation of files in one or more subdirectories. The operation of *make* is complex and a full description is beyond the scope of this section, but it can be summarized by saying that *make* manages efficient compilation of programs involving multiple source files. *Make* assures that all necessary files are compiled. It tests previously compiled modules to see if they are up to date and recompiles any whose source files have been modified since the previous compilation. This saves time by avoiding recompilation of files that do not need to be recompiled. Finally, *make* directs the combination of separately compiled modules into an executable program and may also manage installation of the completed program.

All or part of the *src/* tree can be relocated, since the *Makefile* in each source directory uses a relative path to C source directories. For instance, you may want to make a source directory on the root filesystem, */src/*, for speedy compilation if the root device is a RAM disk. If you are developing a special version you can make a copy of *src/* under another name.

The path to the C header files is a special case. During compilation every *Makefile* expects to find header files in */usr/include/* (or the equivalent path on a non-Intel platform). However, *src/tools/Makefile*, used to recompile the system, expects to find a master copy of the headers in */usr/src/include* (on an Intel system). Before recompiling the system, however, the entire */usr/include/* directory tree is deleted and */usr/src/include/* is copied to */usr/include/*. This was done to make it possible to keep all files needed in the development of MINIX 3 in one place. This also makes it easy to maintain multiple copies of the entire source and headers tree for experimenting with different configurations of the MINIX 3 system. However, if you want to edit a header file as part of such an experiment, you must be sure to edit the copy in the *src/include* directory and not the one in */usr/include/*.

This is a good place to point out for newcomers to the C language how file names are quoted in a #include statement. Every C compiler has a default header directory where it looks for include files. Frequently, this is */usr/include/*. When the name of a file to include is quoted between less-than and greater-than symbols ("< ... >") the compiler searches for the file in the default header directory or a specified subdirectory, for example,

```
#include <filename>
```

includes a file from */usr/include/*.

Many programs also require definitions in local header files that are not meant to be shared system-wide. Such a header may have the same name as and be meant to replace or supplement a standard header. When the name is quoted between ordinary quote characters (" ″ ... ″") the file is searched for first in the same directory as the source file (or a specified subdirectory) and then, if not found there, in the default directory. Thus

```
#include "filename"
```

reads a local file.

The *include/* directory contains a number of POSIX standard header files. In addition, it has three subdirectories:

sys/	– additional POSIX headers.
minix/	– header files used by the MINIX 3 operating system.
ibm/	– header files with IBM PC-specific definitions.

To support extensions to MINIX 3 and programs that run in the MINIX 3 environment, other files and subdirectories are also present in *include/* as provided on the

CD-ROM and also on the MINIX 3 Web site. For instance, *include/arpa/* and the *include/net/* directory and its subdirectory *include/net/gen/* support network extensions. These are not necessary for compiling the basic MINIX 3 system, and files in these directories are not listed in Appendix B.

In addition to *src/include/*, the *src/* directory contains three other important subdirectories with operating system source code:

kernel/	– layer 1 (scheduling, messages, clock and system tasks).
drivers/	– layer 2 (device drivers for disk, console, printer, etc.).
servers/	–layer 3 (process manager, file system, other servers).

Three other source code directories are not printed or discussed in the text, but are essential to producing a working system:

src/lib/	– source code for library procedures (e.g., open, read).
src/tools/	– Makefile and scripts for building the MINIX 3 system.
src/boot/	– the code for booting and installing MINIX 3.

The standard distribution of MINIX 3 includes many additional source files not discussed in this text. In addition to the process manager and file system source code, the system source directory *src/servers/* contains source code for the *init* program and the reincarnation server, *rs*, both of which are essential parts of a running MINIX 3 system. The network server source code is in *src/servers/inet/*. *Src/drivers/* has source code for device drivers not discussed in this text, including alternative disk drivers, sound cards, and network adapters. Since MINIX 3 is an experimental operating system, meant to be modified, there is a *src/test/* directory with programs designed to test thoroughly a newly compiled MINIX 3 system. An operating system exists, of course, to support commands (programs) that will run on it, so there is a large *src/commands/* directory with source code for the utility programs (e.g., *cat*, *cp*, *date*, *ls*, *pwd* and more than 200 others). Source code for some major open source applications originally developed by the GNU and BSD projects is here, too.

The "book" version of MINIX 3 is configured with many of the optional parts omitted (trust us: we cannot fit everything into one book or into your head in a semester-long course). The "book" version is compiled using modified *Makefile*s that do not refer to unnecessary files. (A standard *Makefile* requires that files for optional components be present, even if not to be compiled.) Omitting these files and the conditional statements that select them makes reading the code easier.

For convenience we will usually refer to simple file names when it it is clear from the context what the complete path is. However, be aware that some file names appear in more than one directory. For instance, there are several files named *const.h*. *Src/kernel/const.h* defines constants used in the kernel, while *src/servers/pm/const.h* defines constants used by the process manager, etc.

The files in a particular directory will be discussed together, so there should not be any confusion. The files are listed in Appendix B in the order they are discussed in the text, to make it easier to follow along. Acquisition of a couple of bookmarks might be of use at this point, so you can go back and forth between the text and the listing. To keep the size of the listing reasonable, code for every file is not printed. In general, those functions that are described in detail in the text are listed in Appendix B; those that are just mentioned in passing are not listed, but the complete source is on the CD-ROM and Web site, both of which also provide an index to functions, definitions, and global variables in the source code.

Appendix C contains an alphabetical list of all files described in Appendix B, divided into sections for headers, drivers, kernel, file system, and process manager. This appendix and the Web site and CD-ROM indices reference the listed objects by line number in the source code.

The code for layer 1 is contained in the directory *src/kernel/*. Files in this directory support process control, the lowest layer of the MINIX 3 structure we saw in Fig. 2-29. This layer includes functions which handle system initialization, interrupts, message passing and process scheduling. Intimately connected with these are two modules compiled into the same binary, but which run as independent processes. These are the system task which provides an interface between kernel services and processes in higher layers, and the clock task which provides timing signals to the kernel. In Chap. 3, we will look at files in several of the subdirectories of *src/drivers*, which support various device drivers, the second layer in Fig. 2-29. Then in Chap. 4, we will look at the process manager files in *src/servers/pm/*. Finally, in Chap. 5, we will study the file system, whose source files are located in *src/servers/fs/*.

2.6.2 Compiling and Running MINIX 3

To compile MINIX 3, run make in *src/tools/*. There are several options, for installing MINIX 3 in different ways. To see the possibilities run make with no argument. The simplest method is make image.

When make image is executed, a fresh copy of the header files in *src/include/* is copied to */usr/include/*. Then source code files in *src/kernel/* and several subdirectories of *src/servers/* and *src/drivers/* are compiled to object files. All the object files in *src/kernel/* are linked to form a single executable program, *kernel*. The object files in *src/servers/pm/* are also linked together to form a single executable program, *pm*, and all the object files in *src/servers/fs/* are linked to form *fs*. The additional programs listed as part of the boot image in Fig. 2-30 are also compiled and linked in their own directories. These include *rs* and *init* in subdirectories of *src/servers/* and *memory/*, *log/*, and *tty/* in subdirectories of *src/drivers/*. The component designated "driver" in Fig. 2-30 can be one of several disk drivers; we discuss here a MINIX 3 system configured to boot from the hard disk using the standard *at_wini* driver, which will be compiled in *src/drivers/at_wini/*. Other drivers can be added, but most drivers need not be compiled into the boot image. The same is true for networking support; compilation of the basic MINIX 3 system is the same whether or not networking will be used.

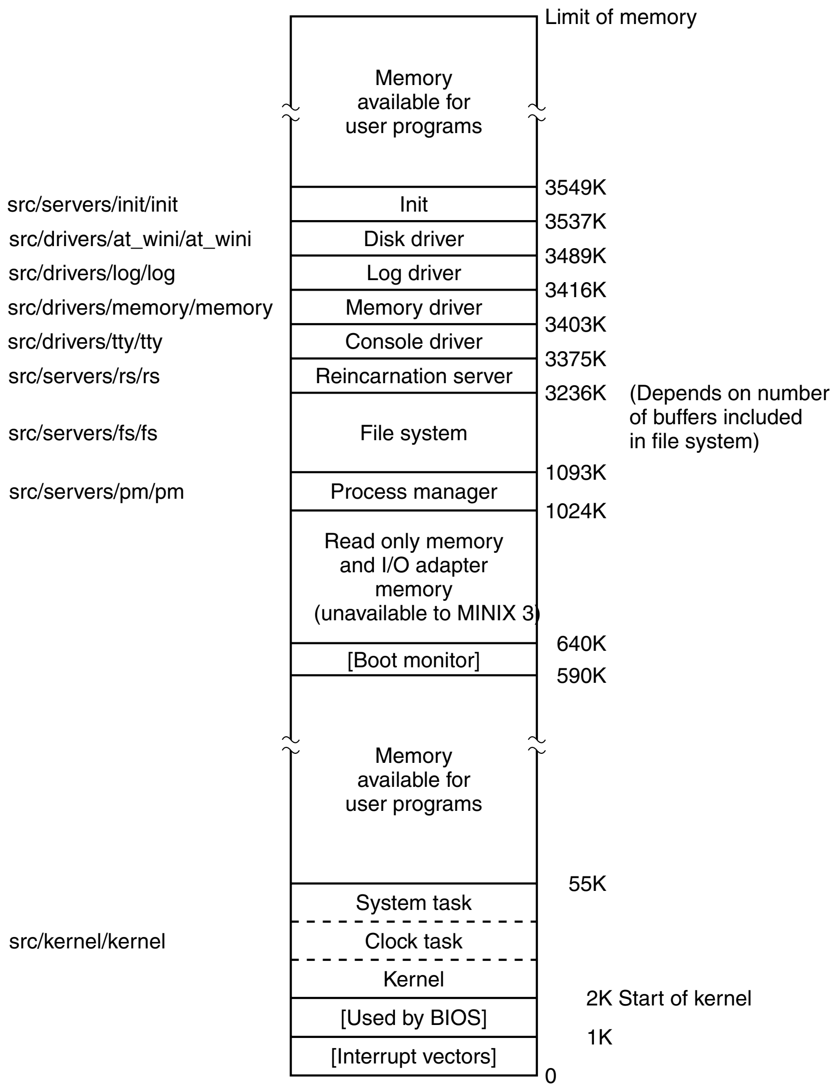

Figure 2-31. Memory layout after MINIX 3 has been loaded from the disk into memory. The kernel, servers, and drivers are independently compiled and linked programs, listed on the left. Sizes are approximate and not to scale.

To install a working MINIX 3 system capable of being booted, a program called *installboot* (whose source is in *src/boot/*) adds names to *kernel*, *pm*, *fs*, *init*, and the other components of the boot image, pads each one out so that its length is a multiple of the disk sector size (to make it easier to load the parts independently), and concatenates them onto a single file. This new file is the boot image and can be copied into the */boot/* directory or the */boot/image/* directory of a floppy disk or a hard disk partition. Later, the boot monitor program can load the boot image and transfer control to the operating system.

Figure 2-31 shows the layout of memory after the concatenated programs are separated and loaded. The kernel is loaded in low memory, all the other parts of the boot image are loaded above 1 MB. When user programs are run, the available memory above the kernel will be used first. When a new program will not fit there, it will be loaded in the high memory range, above *init*. Details, of course, depend upon the system configuration. For instance, the example in the figure is for a MINIX 3 file system configured with a block cache that can hold 512 4-KB disk blocks. This is a modest amount; more is recommended if adequate memory is available. On the other hand, if the size of the block cache were reduced drastically it would be possible to make the entire system fit into less than 640K of memory, with room for a few user processes as well.

It is important to realize that MINIX 3 consists of several totally independent programs that communicate only by passing messages. A procedure called *panic* in the directory *src/servers/fs/* does not conflict with a procedure called *panic* in *src/servers/pm/* because they ultimately are linked into different executable files. The only procedures that the three pieces of the operating system have in common are a few of the library routines in *src/lib/*. This modular structure makes it very easy to modify, say, the file system, without having these changes affect the process manager. It also makes it straightforward to remove the file system altogether and to put it on a different machine as a file server, communicating with user machines by sending messages over a network.

As another example of the modularity of MINIX 3, adding network support makes absolutely no difference to the process manager, the file system, or the kernel. Both an Ethernet driver and the *inet* server can be activated after the boot image is loaded; they would appear in Fig. 2-30 with the processes started by */etc/rc*, and they would be loaded into one of the "Memory available for user programs" regions of Fig. 2-31. A MINIX 3 system with networking enabled can be used as a remote terminal or an ftp and web server. Only if you want to allow incoming logins to the MINIX 3 system over the network would any part of MINIX 3 as described in the text need modification: this is *tty*, the console driver, which would need to be recompiled with pseudo terminals configured to allow remote logins.

2.6.3 The Common Header Files

The *include/* directory and its subdirectories contain a collection of files defining constants, macros, and types. The POSIX standard requires many of these definitions and specifies in which files of the main *include/* directory and its subdirectory *include/sys/* each required definition is to be found. The files in these directories are **header** or **include** files, identified by the suffix *.h*, and used by means of #include statements in C source files. These statements are a built-in feature of the C language. Include files make maintenance of a large system easier.

Headers likely to be needed for compiling user programs are mainly found in *include/* whereas *include/sys/* traditionally is used for files that are used primarily for compiling system programs and utilities. The distinction is not terribly important, and a typical compilation, whether of a user program or part of the operating system, will include files from both of these directories. We will discuss here the files that are needed to compile the standard MINIX 3 system, first treating those in *include/* and then those in *include/sys/*. In the next section we will discuss files in the *include/minix/* and *include/ibm/* directories, which, as the directory names indicate, are unique to MINIX 3 and its implementation on IBM-type (really, Intel-type) computers.

The first headers to be considered are truly general purpose ones, so much so that they are not referenced directly by any of the C language source files for the MINIX 3 system. Rather, they are themselves included in other header files. Each major component of MINIX 3 has a master header file, such as *src/kernel/kernel.h*, *src/servers/pm/pm.h*, and *src/servers/fs/fs.h*. These are included in every compilation of these components. Source code for each of the device drivers includes a somewhat similar file, *src/drivers/drivers.h*. Each master header is tailored to the needs of the corresponding part of the MINIX 3 system, but each one starts with a section like the one shown in Fig. 2-32 and includes most of the files shown there. The master headers will be discussed again in other sections of the book. This preview is to emphasize that headers from several directories are used together. In this section and the next one we will mention each of the files referenced in Fig. 2-32.

```
#include <minix/config.h>                /* MUST be first */
#include <ansi.h>                        /* MUST be second */
#include <limits.h>
#include <errno.h>
#include <sys/types.h>
#include <minix/const.h>
#include <minix/type.h>
#include <minix/syslib.h>
#include "const.h"
```

Figure 2-32. Part of a master header which ensures inclusion of header files needed by all C source files. Note that two *const.h* files, one from the *include/* tree and one from the local directory, are referenced.

Let us start with the first header in *include/*, *ansi.h* (line 0000). This is the second header that is processed whenever any part of the MINIX 3 system is compiled; only *include/minix/config.h* is processed earlier. The purpose of *ansi.h* is to test whether the compiler meets the requirements of Standard C, as defined by the International Organization for Standards. Standard C is also often referred to as ANSI C, since the standard was originally developed by the American National Standards Institute before gaining international recognition. A Standard C compiler defines several macros that can then be tested in programs being compiled. *__STDC__* is such a macro, and it is defined by a standard compiler to have a value of 1, just as if the C preprocessor had read a line like

```
#define __STDC__ 1
```

The compiler distributed with current versions of MINIX 3 conforms to Standard C, but older versions of MINIX were developed before the adoption of the standard, and it is still possible to compile MINIX 3 with a classic (Kernighan & Ritchie) C compiler. It is intended that MINIX 3 should be easy to port to new machines, and allowing older compilers is part of this. At lines 0023 to 0025 the statement

```
#define _ANSI
```

is processed if a Standard C compiler is in use. *Ansi.h* defines several macros in different ways, depending upon whether the *_ANSI* macro is defined. This is an example of a **feature test macro**.

Another feature test macro defined here is *_POSIX_SOURCE* (line 0065). This is required by POSIX. Here we ensure it is defined if other macros that imply POSIX conformance are defined.

When compiling a C program the data types of the arguments and the returned values of functions must be known before code that references such data can be generated. In a complex system ordering of function definitions to meet this requirement is difficult, so C allows use of **function prototypes** to **declare** the arguments and return value types of a function before it is **defined**. The most important macro in *ansi.h* is *_PROTOTYPE*. This macro allows us to write function prototypes in the form

```
_PROTOTYPE (return-type function-name, (argument-type argument, ... ) )
```

and have this transformed by the C preprocessor into

```
return-type function-name(argument-type, argument, ...)
```

if the compiler is an ANSI Standard C compiler, or

```
return-type function-name()
```

if the compiler is an old-fashioned (i.e., Kernighan & Ritchie) compiler.

Before we leave *ansi.h* let us mention one additional feature. The entire file (except for initial comments) is enclosed between lines that read

```
#ifndef _ANSI_H
```

and

```
#endif /* _ANSI_H */
```

On the line immediately following the #ifndef _ANSI_H itself is defined. A header file should be included only once in a compilation; this construction ensures that the contents of the file will be ignored if it is included multiple times. We will see this technique used in all the header files in the *include/* directory.

Two points about this deserve mention. First, in all of the #ifndef ... #define sequences for files in the master header directories, the filename is preceded by an underscore. Another header with the same name may exist within the C source code directories, and the same mechanism will be used there, but underscores will not be used. Thus inclusion of a file from the master header directory will not prevent processing of another header file with the same name in a local directory. Second, note that the comment /* _ANSI_H */ after the #ifndef is not required. Such comments can be helpful in keeping track of nested #ifndef ... #endif and #ifdef ... #endif sections. However, care is needed in writing such comments: if incorrect they are worse than no comment at all.

The second file in *include/* that is indirectly included in most MINIX 3 source files is the *limits.h* header (line 0100). This file defines many basic sizes, both language types such as the number of bits in an integer, as well as operating system limits such as the length of a file name.

Note that for convenience, the line numbering in Appendix B is ratcheted up to the next multiple of 100 when a new file is listed. Thus do not expect *ansi.h* to contain 100 lines (00000 through 00099). In this way, small changes to one file will (probably) not affect subsequent files in a revised listing. Also note that when a new file is encountered in the listing, a special three-line header consisting of a row of + signs, the file name, and another row of + signs is present (without line numbering). An example of this header is shown between lines 00068 and 00100.

Errno.h (line 0200), is also included by most of the master headers. It contains the error numbers that are returned to user programs in the global variable *errno* when a system call fails. *Errno* is also used to identify some internal errors, such as trying to send a message to a nonexistent task. Internally, it would be inefficient to examine a global variable after a call to a function that might generate an error, but functions must often return other integers, for instance, the number of bytes transferred during an I/O operation. The MINIX 3 solution is to return error numbers as negative values to mark them as error codes within the system, and then to convert them to positive values before being returned to user programs. The trick that is used is that each error code is defined in a line like

```
#define EPERM    (_SIGN 1)
```

(line 0236). The master header file for each part of the operating system defines the *_SYSTEM* macro, but *_SYSTEM* is never defined when a user program is compiled. If *_SYSTEM* is defined, then *_SIGN* is defined as "−"; otherwise it is given a null definition.

The next group of files to be considered are not included in all the master headers, but are nevertheless used in many source files in all parts of the MINIX 3 system. The most important is *unistd.h* (line 0400). This header defines many constants, most of which are required by POSIX. In addition, it includes prototypes for many C functions, including all those used to access MINIX 3 system calls. Another widely used file is *string.h* (line 0600), which provides prototypes for many C functions used for string manipulation. The header *signal.h* (line 0700) defines the standard signal names. Several MINIX 3-specific signals for operating system use are defined, as well. The fact that operating systems functions are handled by independent processes rather than within a monolithic kernel requires some special signal-like communication between the system components. *Signal.h* also contains prototypes for some signal-related functions. As we will see later, signal handling involves all parts of MINIX 3.

Fcntl.h (line 0900) symbolically defines many parameters used in file control operations. For instance, it allows one to use the macro *O_RDONLY* instead of the numeric value 0 as a parameter to a *open* call. Although this file is referenced mostly by the file system, its definitions are also needed in a number of places in the kernel and the process manager.

As we will see when we look at the device driver layer in Chap. 3, the console and terminal interface of an operating system is complex, because many different types of hardware have to interact with the operating system and user programs in a standardized way. *Termios.h* (line 1000) defines constants, macros, and function prototypes used for control of terminal-type I/O devices. The most important structure is the *termios* structure. It contains flags to signal various modes of operation, variables to set input and output transmission speeds, and an array to hold special characters (e.g., the *INTR* and *KILL* characters). This structure is required by POSIX, as are many of the macros and function prototypes defined in this file.

However, as all-encompassing as the POSIX standard is meant to be, it does not provide everything one might want, and the last part of the file, from line 1140 onward, provides extensions to POSIX. Some of these are of obvious value, such as extensions to define standard baud rates of 57,600 baud and higher, and support for terminal display screen windows. The POSIX standard does not forbid extensions, as no reasonable standard can ever be all-inclusive. But when writing a program in the MINIX 3 environment which is intended to be portable to other environments, some caution is required to avoid the use of definitions specific to MINIX 3. This is fairly easy to do. In this file and other files that define MINIX 3-specific extensions the use of the extensions is controlled by the

```
#ifdef _MINIX
```

statement. If the macro *_MINIX* is not defined, the compiler will not even see the MINIX 3 extensions; they will all be completely ignored.

Watchdog timers are supported by *timers.h* (line 1300), which is included in the kernel's master header. It defines a *struct timer*, as well as prototypes of functions used to operate on lists of timers. On line 1321 appears a *typedef* for *tmr_func_t*. This data type is a pointer to a function. At line 1332 its use is seen: within a *timer* structure, used as an element in a list of timers, one element is a *tmr_func_t* to specify a function to be called when the timer expires.

We will mention four more files in the *include/* directory that are not listed in Appendix B. *Stdlib.h* defines types, macros, and function prototypes that are likely to be needed in the compilation of all but the most simple of C programs. It is one of the most frequently used headers in compiling user programs, although within the MINIX 3 system source it is referenced by only a few files in the kernel. *Stdio.h* is familiar to everyone who has started to learn programming in C by writing the famous "Hello World!" program. It is hardly used at all in system files, although, like *stdlib.h*, it is used in almost every user program. *A.out.h* defines the format of the files in which executable programs are stored on disk. An *exec* structure is defined here, and the information in this structure is used by the process manager to load a new program image when an exec call is made. Finally, *stddef.h* defines a few commonly used macros.

Now let us go on to the subdirectory *include/sys/*. As shown in Fig. 2-32, the master headers for the main parts of the MINIX 3 system all cause *sys/types.h* (line 1400) to be read immediately after reading *ansi.h*. *Sys/types.h* defines many data types used by MINIX 3. Errors that could arise from misunderstanding which fundamental data types are used in a particular situation can be avoided by using the definitions provided here. Fig. 2-33 shows the way the sizes, in bits, of a few types defined in this file differ when compiled for 16-bit or 32-bit processors. Note that all type names end with "_t". This is not just a convention; it is a requirement of the POSIX standard. This is an example of a **reserved suffix**, and "_t" should not be used as a suffix of any name which is *not* a type name.

Type	16-Bit MINIX	32-Bit MINIX
gid_t	8	8
dev_t	16	16
pid_t	16	32
ino_t	16	32

Figure 2-33. The size, in bits, of some types on 16-bit and 32-bit systems.

MINIX 3 currently runs natively on 32-bit microprocessors, but 64-bit processors will be increasingly important in the future. A type that is not provided by the hardware can be synthesized if necessary. On line 1471 the *u64_t* type is defined as struct {u32_t[2]}. This type is not needed very often in the current implementation, but it can be useful—for instance, all disk and partition data (offsets and sizes) is stored as 64 bit numbers, allowing for very large disks.

MINIX 3 uses many type definitions that ultimately are interpreted by the compiler as a relatively small number of common types. This is intended to help make the code more readable; for instance, a variable declared as the type *dev_t* is recognizable as a variable meant to hold the major and minor device numbers that identify an I/O device. For the compiler, declaring such a variable as a *short* would work equally well. Another thing to note is that many of the types defined here are matched by corresponding types with the first letter capitalized, for instance, *dev_t* and *Dev_t*. The capitalized variants are all equivalent to type *int* to the compiler; these are provided to be used in function prototypes which must use types compatible with the *int* type to support K&R compilers. The comments in *types.h* explain this in more detail.

One other item worth mention is the section of conditional code that starts with

```
#if _EM_WSIZE == 2
```

(lines 1502 to 1516). As noted earlier, most conditional code has been removed from the source as discussed in the text. This example was retained so we could point out one way that conditional definitions can be used. The macro used, *_EM_WSIZE*, is another example of a compiler-defined feature test macro. It tells the word size for the target system in bytes. The #if ... #else ... #endif sequence is a way of getting some definitions right once and for all, to make subsequent code compile correctly whether a 16-bit or 32-bit system is in use.

Several other files in *include/sys/* are widely used in the MINIX 3 system. The file *sys/sigcontext.h* (line 1600) defines structures used to preserve and restore normal system operation before and after execution of a signal handling routine and is used both in the kernel and the process manager. *Sys/stat.h* (line 1700) defines the structure which we saw in Fig. 1-12, returned by the stat and fstat system calls, as well as the prototypes of the functions *stat* and *fstat* and other functions used to manipulate file properties. It is referenced in several parts of the file system and the process manager.

Other files we will discuss in this section are not as widely referenced as the ones discussed above. *Sys/dir.h* (line 1800) defines the structure of a MINIX 3 directory entry. It is only referenced directly once, but this reference includes it in another header that is widely used in the file system. It is important because, among other things, it tells how many characters a file name may contain (60). The *sys/wait.h* (line 1900) header defines macros used by the wait and waitpid system calls, which are implemented in the process manager.

Several other files in *include/sys/* should be mentioned, although they are not listed in Appendix B. MINIX 3 supports tracing executables and analyzing core dumps with a debugger program, and *sys/ptrace.h* defines the various operations possible with the ptrace system call. *Sys/svrctl.h* defines data structures and macros used by svrctl, which is not really a system call, but is used like one. Svrctl is used to coordinate server-level processes as the system starts up. The select system call permits waiting for input on multiple channels—for instance, pseudo terminals waiting for network connections. Definitions needed by this call are in *sys/select.h*.

We have deliberately left discussion of *sys/ioctl.h* and related files until last, because they cannot be fully understood without also looking at a file in the next directory, *minix/ioctl.h.* The ioctl system call is used for device control operations. The number of devices which can be interfaced with a modern computer system is ever increasing. All need various kinds of control. Indeed, the main difference between MINIX 3 as described in this book and other versions is that for purposes of the book we describe MINIX 3 with relatively few input/output devices. Many others, such as network interfaces, SCSI controllers, and sound cards, can be added.

To make things more manageable, a number of small files, each containing one group of definitions, are used. They are all included by *sys/ioctl.h* (line 2000), which functions similarly to the master header of Fig. 2-32. We have listed only one of these included files, *sys/ioc_disk.h* (line 2100), in Appendix B. This and the other files included by *sys_ioctl.h* are located in the *include/sys/* directory because they are considered part of the "published interface," meaning a programmer can use them in writing any program to be run in the MINIX 3 environment. However, they all depend upon additional macro definitions provided in *minix/ioctl.h* (line 2200), which is included by each. *Minix/ioctl.h* should not be used by itself in writing programs, which is why it is in *include/minix/* rather than *include/sys/*.

The macros defined together by these files define how the various elements needed for each possible function are packed into a 32 bit integer to be passed to ioctl. For instance, disk devices need five types of operations, as can be seen in *sys/ioc_disk.h* at lines 2110 to 2114. The alphabetic 'd' parameter tells ioctl that the operation is for a disk device, an integer from 3 through 7 codes for the operation, and the third parameter for a write or read operation tells the size of the structure in which data is to be passed. In *minix/ioctl.h* lines 2225 to 2231 show that 8 bits of the alphabetic code are shifted 8 bits to the left, the 13 least significant bits of the size of the structure are shifted 16 bits to the left, and these are then logically ANDed with the small integer operation code. Another code in the most significant 3 bits of a 32-bit number encodes the type of return value.

Although this looks like a lot of work, this work is done at compile time and makes for a much more efficient interface to the system call at run time, since the parameter actually passed is the most natural data type for the host machine CPU. It does, however, bring to mind a famous comment Ken Thompson put into the source code of an early version of UNIX:

```
/* You are not expected to understand this */
```

Minix/ioctl.h also contains the prototype for the ioctl system call at line 2241. This call is not directly invoked by programmers in many cases, since the POSIX-defined functions prototyped in *include/termios.h* have replaced many uses of the old *ioctl* library function for dealing with terminals, consoles, and similar devices. Nevertheless, it is still necessary. In fact, the POSIX functions for control of terminal devices are converted into ioctl system calls by the library.

2.6.4 The MINIX 3 Header Files

The subdirectories *include/minix/* and *include/ibm/* contain header files specific to MINIX 3. Files in *include/minix/* are needed for an implementation of MINIX 3 on any platform, although there are platform-specific alternative definitions within some of them. We have already discussed one file here, *ioctl.h*. The files in *include/ibm/* define structures and macros that are specific to MINIX 3 as implemented on IBM-type machines.

We will start with the *minix/* directory. In the previous section, it was noted that *config.h* (line 2300) is included in the master headers for all parts of the MINIX 3 system, and is thus the first file actually processed by the compiler. On many occasions, when differences in hardware or the way the operating system is intended to be used require changes in the configuration of MINIX 3, editing this file and recompiling the system is all that must be done. We suggest that if you modify this file you should also modify the comment on line 2303 to help identify the purpose of the modifications.

The user-settable parameters are all in the first part of the file, but some of these parameters are not intended to be edited here. On line 2326 another header file, *minix/sys_config.h* is included, and definitions of some parameters are inherited from this file. The programmers thought this was a good idea because a few files in the system need the basic definitions in *sys_config.h* without the rest of those in *config.h*. In fact, there are many names in *config.h* which do not begin with an underscore that are likely to conflict with names in common usage, such as *CHIP* or *INTEL* that would be likely to be found in software ported to MINIX 3 from another operating system. All of the names in *sys_config.h* begin with underscores, and conflicts are less likely.

MACHINE is actually configured as *_MACHINE_IBM_PC* in *sys_config.h*; lines 2330 to 2334 lists short alternatives for all possible values for *MACHINE*. Earlier versions of MINIX were ported to Sun, Atari, and MacIntosh platforms, and the full source code contains alternatives for alternative hardware. Most of the MINIX 3 source code is independent of the type of machine, but an operating system always has some system-dependent code. Also, it should be noted that, because MINIX 3 is so new, as of this writing additional work is needed to complete porting MINIX 3 to non-Intel platforms.

Other definitions in *config.h* allow customization for other needs in a particular installation. For instance, the number of buffers used by the file system for the disk cache should generally be as large as possible, but a large number of buffers requires lots of memory. Caching 128 blocks, as configured on line 2345, is considered minimal and satisfactory only for a MINIX 3 installation on a system with less than 16 MB of RAM; for systems with ample memory a much larger number can be put here. If it is desired to use a modem or log in over a network connection the *NR_RS_LINES* and *NR_PTYS* definitions (lines 2379 and 2380) should be increased and the system recompiled. The last part of *config.h* contains definitions that are necessary, but which should not be changed. Many definitions here just define alternate names for constants defined in *sys_config.h*.

Sys_config.h (line 2500) contains definitions that are likely to be needed by a system programmer, for instance someone writing a new device driver. You are not likely to need to change very much in this file, with the possible exception of *_NR_PROCS* (line 2522). This controls the size of the process table. If you want to use a MINIX 3 system as a network server with many remote users or many server processes running simultaneously, you might need to increase this constant.

The next file is *const.h* (line 2600), which illustrates another common use of header files. Here we find a variety of constant definitions that are not likely to be changed when compiling a new kernel but that are used in a number of places. Defining them here helps to prevent errors that could be hard to track down if inconsistent definitions were made in multiple places. Other files named *const.h* can be found elsewhere in the MINIX 3 source tree, but they are for more limited use. Similarly, definitions that are used only in the kernel are included in *src/kernel/const.h*. Definitions that are used only in the file system are included in *src/servers/fs/const.h*. The process manager uses *src/servers/pm/const.h* for its local definitions. Only those definitions that are used in more than one part of the MINIX 3 system are included in *include/minix/const.h.*

A few of the definitions in *const.h* are noteworthy. *EXTERN* is defined as a macro expanding into *extern* (line 2608). Global variables that are declared in header files and included in two or more files are declared *EXTERN*, as in

```
EXTERN int who;
```

If the variable were declared just as

```
int who;
```

and included in two or more files, some linkers would complain about a multiply defined variable. Furthermore, the C reference manual explicitly forbids this construction (Kernighan and Ritchie, 1988).

To avoid this problem, it is necessary to have the declaration read

```
extern int who;
```

in all places but one. Using *EXTERN* prevents this problem by having it expand into *extern* everywhere that *const.h* is included, except following an explicit redefinition of *EXTERN* as the null string. This is done in each part of MINIX 3 by putting global definitions in a special file called *glo.h*, for instance, *src/kernel/glo.h*, which is indirectly included in every compilation. Within each *glo.h* there is a sequence

```
#ifdef _TABLE
#undef EXTERN
#define EXTERN
#endif
```

and in the *table.c* files of each part of MINIX 3 there is a line

```
#define _TABLE
```

preceding the #include section. Thus when the header files are included and expanded as part of the compilation of *table.c*, *extern* is not inserted anywhere (because *EXTERN* is defined as the null string within *table.c*) and storage for the global variables is reserved only in one place, in the object file *table.o*.

If you are new to C programming and do not quite understand what is going on here, fear not; the details are really not important. This is a polite way of rephrasing Ken Thompson's famous comment cited earlier. Multiple inclusion of header files can cause problems for some linkers because it can lead to multiple declarations for included variables. The *EXTERN* business is simply a way to make MINIX 3 more portable so it can be linked on machines whose linkers do not accept multiply defined variables.

PRIVATE is defined as a synonym for *static*. Procedures and data that are not referenced outside the file in which they are declared are always declared as *PRIVATE* to prevent their names from being visible outside the file in which they are declared. As a general rule, all variables and procedures should be declared with a local scope, if possible. *PUBLIC* is defined as the null string. An example from *kernel/proc.c* may help make this clear. The declaration

```
PUBLIC void lock_dequeue(rp)
```

comes out of the C preprocessor as

```
void lock_dequeue(rp)
```

which, according to the C language scope rules, means that the function name *lock_dequeue* is exported from the file and the function can be called from anywhere in any file linked into the same binary, in this case, anywhere in the kernel. Another function declared in the same file is

```
PRIVATE void dequeue(rp)
```

which is preprocessed to become

```
static void dequeue(rp)
```

This function can only be called from code in the same source file. *PRIVATE* and *PUBLIC* are not necessary in any sense but are attempts to undo the damage caused by the C scope rules (the default is that names are exported outside the file; it should be just the reverse).

The rest of *const.h* defines numerical constants used throughout the system. A section of *const.h* is devoted to machine or configuration-dependent definitions. For instance, throughout the source code the basic unit of memory allocation is the **click**. Different values for the click size may be chosen for different processor architectures. For Intel platforms it is 1024 bytes. Alternatives for Intel, Motorola 68000, and Sun SPARC architectures are defined on lines 2673 to 2681. This file also contains the macros *MAX* and *MIN*, so we can say

```
z = MAX(x, y);
```

to assign the larger of *x* and *y* to *z*.

Type.h (line 2800) is another file that is included in every compilation by means of the master headers. It contains a number of key type definitions, along with related numerical values.

The first two structs define two different types of memory map, one for local memory regions (within the data space of a process) and one for remote memory areas, such as a RAM disk (lines 2828 to 2840). This is a good place to mention the concepts used in referring to memory. As we just mentioned, the click is the basic unit of measurement of memory; in MINIX 3 for Intel processors a click is 1024 bytes. Memory is measured as **phys_clicks**, which can be used by the kernel to access any memory element anywhere in the system, or as **vir_clicks**, used by processes other than the kernel. A *vir_clicks* memory reference is always with respect to the base of a segment of memory assigned to a particular process, and the kernel often has to make translations between virtual (i.e. process-based) and physical (RAM-based) addresses. The inconvenience of this is offset by the fact that a process can do all its own memory references in *vir_clicks*.

One might suppose that the same unit could be used to specify the size of either type of memory, but there is an advantage to using *vir_clicks* to specify the size of a unit of memory allocated to a process, since when this unit is used a check is done to be sure that no memory is accessed outside of what has been specifically assigned to the current process. This is a major feature of the **protected mode** of modern Intel processors, such as the Pentium family. Its absence in the early 8086 and 8088 processors caused some headaches in the design of earlier versions of MINIX.

Another important structure defined here is *sigmsg* (lines 2866 to 2872). When a signal is caught the kernel has to arrange that the next time the signaled process gets to run it will run the signal handler, rather than continuing execution where it was interrupted. The process manager does most of the work of managing signals; it passes a structure like this to the kernel when a signal is caught.

The *kinfo* structure (lines 2875 to 2893) is used to convey information about the kernel to other parts of the system. The process manager uses this information when it sets up its part of the process table.

Data structures and function prototypes for **interprocess communication** are defined in *ipc.h* (line 3000). The most important definition in this file is *message* on lines 3020 to 3032. While we could have defined *message* to be an array of some number of bytes, it is better programming practice to have it be a structure containing a union of the various message types that are possible. Seven message formats, *mess_1* through *mess_8*, are defined (type *mess_6* is obsolete). A message is a structure containing a field *m_source*, telling who sent the message, a field *m_type*, telling what the message type is (e.g., *SYS_EXEC* to the system task) and the data fields.

The seven message types are shown in Fig. 2-34. In the figure four message types, the first two and the last two, seem identical. Just in terms of size of the data elements they are identical, but many of the data types are different. It happens that on an Intel CPU with a 32-bit word size the *int*, *long*, and pointer data types are all 32-bit types, but this would not necessarily be the case on another kind of hardware. Defining seven distinct formats makes it easier to recompile MINIX 3 for a different architecture.

When it is necessary to send a message containing, say, three integers and three pointers (or three integers and two pointers), then the first format in Fig. 2-34 is the one to use. The same applies to the other formats. How does one assign a value to the first integer in the first format? Suppose that the message is called *x*. Then *x.m_u* refers to the union portion of the message struct. To refer to the first of the six alternatives in the union, we use *x.m_u.m_m1*. Finally, to get at the first integer in this struct we say *x.m_u.m_m1.m1i1*. This is quite a mouthful, so somewhat shorter field names are defined as macros after the definition of message itself. Thus *x.m1_i1* can be used instead of *x.m_u.m_m1.m1i1*. The short names all have the form of the letter m, the format number, an underscore, one or two letters indicating whether the field is an integer, pointer, long, character, character array, or function, and a sequence number to distinguish multiple instances of the same type within a message.

While discussing message formats, this is a good place to note that an operating system and its compiler often have an “understanding” about things like the layout of structures, and this can make the implementer’s life easier. In MINIX 3, the *int* fields in messages are sometimes used to hold *unsigned* data types. In some cases this could cause overflow, but the code was written using the knowledge that the MINIX 3 compiler copies *unsigned* types to *int*s and *vice versa* without changing the data or generating code to detect overflow. A more compulsive approach would be to replace each *int* field with a *union* of an *int* and an *unsigned*. The same applies to the *long* fields in the messages; some of them may be used to pass *unsigned long* data. Are we cheating here? Perhaps a little bit, one might say, but if you wish to port MINIX 3 to a new platform, quite clearly the exact format of the messages is something to which you must pay a great deal of attention, and now you have been alerted that the behavior of the compiler is another factor that needs attention.

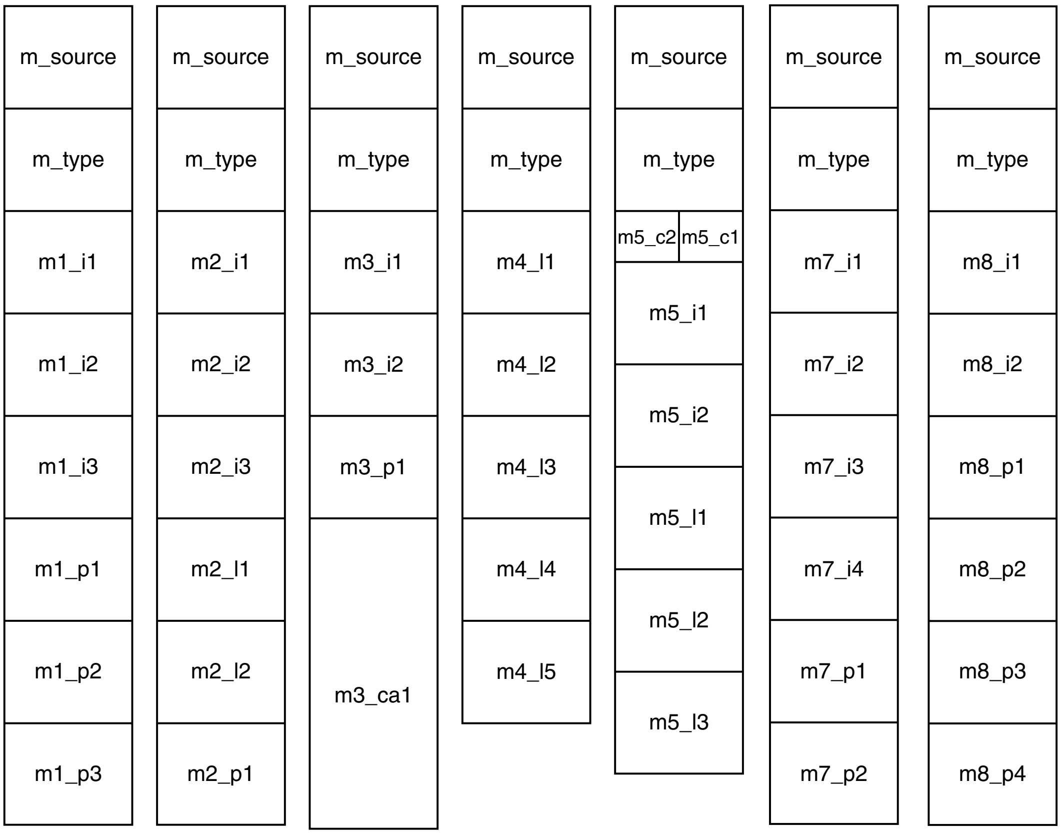

Figure 2-34. The seven message types used in MINIX 3. The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on CPUs with 32-bit pointers, such as those of Pentium family members.

Also defined in *ipc.h* are prototypes for the message passing primitives described earlier (lines 3095 to 3101). In addition to the important send, receive, sendrec, and notify primitives, several others are defined. None of these are much used; in fact one could say that they are relicts of earlier stages of development of MINIX 3. Old computer programs make good archaeological digs. They might disappear in a future release. Nevertheless, if we do not explain them now some readers undoubtedly will worry about them. The nonblocking nb_send and nb_receive calls have mostly been replaced by notify, which was implemented later and considered a better solution to the problem of sending or checking for a message without blocking. The prototype for echo has no source or destination field. This primitive serves no useful purpose in production code, but was useful during development to test the time it took to send and receive a message.

One other file in *include/minix/*, *syslib.h* (line 3200), is almost universally used by means of inclusion in the master headers of all of the user-space components of MINIX 3. This file not included in the kernel's master header file, *src/kernel/kernel.h*, because the kernel does not need library functions to access itself. *Syslib.h* contains prototypes for C library functions called from within the operating system to access other operating system services.

We do not describe details of C libraries in this text, but many library functions are standard and will be available for any C compiler. However, the C functions referenced by *syslib.h* are of course quite specific to MINIX 3 and a port of MINIX 3 to a new system with a different compiler requires porting these library functions. Fortunately this is not difficult, since most of these functions simply extract the parameters of the function call and insert them into a message structure, then send the message and extract the results from the reply message. Many of these library functions are defined in a dozen or fewer lines of C code.

Noteworthy in this file are four macros for accessing I/O ports for input or output using byte or word data types and the prototype of the *sys_sdevio* function to which all four macros refer (lines 3241 to 3250). Providing a way for device drivers to request reading and writing of I/O ports by the kernel is an essential part of the MINIX 3 project to move all such drivers to user space.

A few functions which could have been defined in *syslib.h* are in a separate file, *sysutil.h* (line 3400), because their object code is compiled into a separate library. Two functions prototyped here need a little more explanation. The first is *printf* (line 3442). If you have experience programming in C you will recognize that *printf* is a standard library function, referenced in almost all programs.

This is not the *printf* function you think it is, however. The version of *printf* in the standard library cannot be used within system components. Among other things, the standard *printf* is intended to write to standard output, and must be able to format floating point numbers. Using standard output would require going through the file system, but for printing messages when there is a problem and a system component needs to display an error message, it is desirable to be able to do this without the assistance of any other system components. Also, support for the full range of format specifications usable with the standard *printf* would bloat the code for no useful purpose. So a simplified version of *printf* that does only what is needed by operating system components is compiled into the system utilities library. This is found by the compiler in a place that will depend upon the platform; for 32-bit Intel systems it is */usr/lib/i386/libsysutil.a*. When the file system, the process manager, or another part of the operating system is linked to library functions this version is found before the standard library is searched.

On the next line is a prototype for *kputc*. This is called by the system version of *printf* to do the work of displaying characters on the console. However, more tricky business is involved here. *Kputc* is defined in several places. There is a copy in the system utilities library, which will be the one used by default. But several parts of the system define their own versions. We will see one when we study the console interface in the next chapter. The log driver (which is not described in detail here) also defines its own version. There is even a definition of *kputc* in the kernel itself, but this is a special case. The kernel does not use *printf*. A special printing function, *kprintf*, is defined as part of the kernel and is used when the kernel needs to print.

When a process needs to execute a MINIX 3 system call, it sends a message to the process manager (PM for short) or the file system (FS for short). Each message contains the number of the system call desired. These numbers are defined in the next file, *callnr.h* (line 3500). Some numbers are not used, these are reserved for calls not yet implemented or represent calls implemented in other versions which are now handled by library functions. Near the end of the file some call numbers are defined that do not correspond to calls shown in Fig 1-9. Svrctl (mentioned earlier), ksig, unpause, revive, and task_reply are used only within the operating system itself. The system call mechanism is a convenient way to implement these. In fact, because they will not be used by external programs, these "system calls," may be modified in new versions of MINIX 3 without fear of breaking user programs.

The next file is *com.h* (line 3600). One interpretation of the file name is that is stands for common, another is that it stands for communication. This file provides common definitions used for communication between servers and device drivers. On lines 3623 to 3626 task numbers are defined. To distinguish them from process numbers, task numbers are negative. On lines 3633 to 3640 process numbers are defined for the processes that are loaded in the boot image. Note these are slot numbers in the process table; they should not be confused with process id (PID) numbers.

The next section of *com.h* defines how messages are constructed to carry out a notify operation. The process numbers are used in generating the value that is passed in the *m_type* field of the message. The message types for notifications and other messages defined in this file are built by combining a base value that signifies a type category with a small number that indicates the specific type. The rest of this file is a compendium of macros that translate meaningful identifiers into the cryptic numbers that identify message types and field names.

A few other files in *include/minix/* are listed in Appendix B. *Devio.h* (line 4100) defines types and constants that support user-space access to I/O ports, as well as some macros that make it easier to write code that specifies ports and values. *Dmap.h* (line 4200) defines a struct and an array of that struct, both named *dmap*. This table is used to relate major device numbers to the functions that support them. Major and minor device numbers for the *memory* device driver and major device numbers for other important device drivers are also defined.

Include/minix/ contains several additional specialized headers that are not listed in Appendix B, but which must be present to compile the system. One is *u64.h* which provides support for 64-bit integer arithmetic operations, necessary to manipulate disk addresses on high capacity disk drives. These were not even dreamed of when UNIX, the C language, Pentium-class processors, and MINIX were first conceived. A future version of MINIX 3 may be written in a language that has built-in support for 64-bit integers on CPUs with 64-bit registers; until then, the definitions in *u64.h* provide a work-around.

Three files remain to be mentioned. *Keymap.h* defines the structures used to implement specialized keyboard layouts for the character sets needed for different languages. It is also needed by programs which generate and load these tables. *Bitmap.h* provides a few macros to make operations like setting, resetting, and testing bits easier. Finally, *partition.h* defines the information needed by MINIX 3 to define a disk partition, either by its absolute byte offset and size on the disk, or by a cylinder, head, sector address. The *u64_t* type is used for the offset and size, to allow use of large disks. This file does not describe the layout of a partition table on a disk, the file that does that is in the next directory.

The last specialized header directory we will consider, *include/ibm/*, contains several files which provide definitions related to the IBM PC family of computers. Since the C language knows only memory addresses, and has no provision for accessing I/O port addresses, the library contains routines written in assembly language to read and write from ports. The various routines available are declared in *ibm/portio.h* (line 4300). All possible input and output routines for byte, integer, and long data types, singly or as strings, are available, from *inb* (input one byte) to *outsl* (output a string of longs). Low-level routines in the kernel may also need to disable or reenable CPU interrupts, which are also actions that C cannot handle. The library provides assembly code to do this, and *intr_disable* and *intr_enable* are declared on lines 4325 and 4326.

The next file in this directory is *interrupt.h* (line 4400), which defines port address and memory locations used by the interrupt controller chip and the BIOS of PC-compatible systems. Finally, more I/O ports are defined in *ports.h* (line 4500). This file provides addresses needed to access the keyboard interface and the timer chip used by the clock chip.

Several additional files in *include/ibm/* with IBM-specific data are not listed in Appendix B, but are essential and should be mentioned. *Bios.h*, *memory.h*, and *partition.h* are copiously commented and are worth reading if you would like to know more about memory use or disk partition tables. *Cmos.h*, *cpu.h*, and *int86.h* provide additional information on ports, CPU flag bits, and calling BIOS and DOS services in 16-bit mode. Finally, *diskparm.h* defines a data structure needed for formatting a floppy disk.

2.6.5 Process Data Structures and Header Files

Now let us dive in and see what the code in *src/kernel/* looks like. In the previous two sections we structured our discussion around an excerpt from a typical master header; we will look first at the real master header for the kernel, *kernel.h* (line 4600). It begins by defining three macros. The first, *_POSIX_SOURCE*, is a **feature test macro** defined by the POSIX standard itself. All such macros are required to begin with the underscore character, "_". The effect of defining the *_POSIX_SOURCE* macro is to ensure that all symbols required by the standard and any that are explicitly permitted, but not required, will be visible, while hiding any additional symbols that are unofficial extensions to POSIX. We have already mentioned the next two definitions: the *_MINIX* macro overrides the effect of *_POSIX_SOURCE* for extensions defined by MINIX 3, and *_SYSTEM* can be tested wherever it is important to do something differently when compiling system code, as opposed to user code, such as changing the sign of error codes. *Kernel.h* then includes other header files from *include/* and its subdirectories *include/sys/ include/minix/*, and *include/ibm/* including all those referred to in Fig. 2-32. We have discussed all of these files in the previous two sections. Finally, six additional headers from the local directory, *src/kernel/*, are included, their names included in quote characters.

Kernel.h makes it possible to guarantee that all source files share a large number of important definitions by writing the single line

```
#include "kernel.h"
```

in each of the other kernel source files. Since the order of inclusion of header files is sometimes important, *kernel.h* also ensures that this ordering is done correctly, once and forever. This carries to a higher level the "get it right once, then forget the details" technique embodied in the header file concept. Similar master headers are provided in source directories for other system components, such as the file system and the process manager.

Now let us proceed to look at the local header files included in *kernel.h*. First we have yet another file named *config.h*, which, analogous to the system-wide file *include/minix/config.h*, must be included before any of the other local include files. Just as we have files *const.h* and *type.h* in the common header directory *include/minix/*, we also have files *const.h*. and *type.h* in the kernel source directory, *src/kernel/*. The files in *include/minix/* are placed there because they are needed by many parts of the system, including programs that run under the control of the system. The files in *src/kernel/* provide definitions needed only for compilation of the kernel. The FS, PM, and other system source directories also contain *const.h* and *type.h* files to define constants and types needed only for those parts of the system. Two of the other files included in the master header, *proto.h glo.h*, have no counterparts in the main *include/* directories, but we will find that they, too, have counterparts used in compiling the file system and the process manager. The last local header included in *kernel.h* is another *ipc.h*.

Since this is the first time it has come up in our discussion, note at the beginning of *kernel/config.h* there is a #ifndef ... #define sequence to prevent trouble if the file is included multiple times. We have seen the general idea before. But note here that the macro defined here is *CONFIG_H* without an underscore. Thus it is distinct from the macro *_CONFIG_H* defined in *include/minix/config.h*.

The kernel's version of *config.h* gathers in one place a number of definitions that are unlikely to need changes if your interest in MINIX 3 is studying how an operating system works, or using this operating system in a conventional general-purpose computer. However, suppose you want to make a really tiny version of MINIX 3 for controlling a scientific instrument or a home-made cellular telephone. The definitions on lines 4717 to 4743 allow selective disabling of kernel calls. Eliminating unneeded functionality also reduces memory requirements because the code needed to handle each kernel call is conditionally compiled using the definitions on lines 4717 to 4743. If some function is disabled, the code needed to execute it is omitted from the system binary. For example, a cellular telephone might not need to fork off new processes, so the code for doing so could be omitted from the executable file, resulting in a smaller memory footprint. Most other constants defined in this file control basic parameters. For instance, while handling interrupts a special stack of size *K_STACK_BYTES* is used. This value is set on line 4772. The space for this stack is reserved within *mpx386.s*, an assembly language file.

In *const.h* (line 4800) a macro for converting virtual addresses relative to the base of the kernel's memory space to physical addresses is defined on line 4814. A C function, *umap_local*, is defined elsewhere in the kernel code so the kernel can do this conversion on behalf of other components of the system, but for use within the kernel the macro is more efficient. Several other useful macros are defined here, including several for manipulating bitmaps. An important security mechanism built into the Intel hardware is activated by two macro definition lines here. The **processor status word** (**PSW**) is a CPU register, and **I/O Protection Level** (**IOPL**) bits within it define whether access to the interrupt system and I/O ports is allowed or denied. On lines 4850 and 4851 different PSW values are defined that determine this access for ordinary and privileged processes. These values are put on the stack as part of putting a new process in execution.

In the next file we will consider, *type.h* (line 4900), the *memory* structure (lines 4925 to 4928) uses two quantities, base address and size, to uniquely specify an area of memory.

Type.h defines several other prototypes and structures used in any implementation of MINIX 3. For instance, two structures, *kmessages*, used for diagnostic messages from the kernel, and *randomness*, used by the random number generator, are defined. *Type.h* also contains several machine-dependent type definitions. To make the code shorter and more readable we have removed conditional code and definitions for other CPU types. But you should recognize that definitions like the *stackframe_s* structure (lines 4955 to 4974), which defines how machine registers are saved on the stack, is specific to Intel 32-bit processors. For another platform the *stackframe_s* structure would be defined in terms of the register structure of the CPU to be used. Another example is the *segdesc_s* structure (lines 4976 to 4983), which is part of the protection mechanism that keeps processes from accessing memory regions outside those assigned to them. For another

CPU the *segdesc_s* structure might not exist at all, depending upon the mechanism used to implement memory protection.

Another point to make about structures like these is that making sure all the required data is present is necessary, but possibly not sufficient for optimal performance. The *stackframe_s* must be manipulated by assembly language code. Defining it in a form that can be efficiently read or written by assembly language code reduces the time required for a context switch.

The next file, *proto.h* (line 5100), provides prototypes of all functions that must be known outside of the file in which they are defined. All are written using the *_PROTOTYPE* macro discussed in the previous section, and thus the MINIX 3 kernel can be compiled either with a classic C (Kernighan and Ritchie) compiler, such as the original MINIX 3 C compiler, or a modern ANSI Standard C compiler, such as the one which is part of the MINIX 3 distribution. A number of these prototypes are system-dependent, including interrupt and exception handlers and functions that are written in assembly language.

In *glo.h* (line 5300) we find the kernel's global variables. The purpose of the macro *EXTERN* was described in the discussion of *include/minix/const.h*. It normally expands into *extern*. Note that many definitions in *glo.h* are preceded by this macro. The symbol *EXTERN* is forced to be undefined when this file is included in *table.c*, where the macro *_TABLE* is defined. Thus the actual storage space for the variables defined this way is reserved when *glo.h* is included in the compilation of *table.c*. Including *glo.h* in other C source files makes the variables in *table.c* known to the other modules in the kernel.

Some of the kernel information structures here are used at startup. *Aout* (line 5321) will hold the address of an array of the headers of all of the MINIX 3 system image components. Note that these are **physical addresses**, that is, addresses relative to the entire address space of the processor. As we will see later, the physical address of *aout* will be passed from the boot monitor to the kernel when MINIX 3 starts up, so the startup routines of the kernel can get the addresses of all MINIX 3 components from the monitor's memory space. *Kinfo* (line 5322) is also an important piece of information. Recall that the structure was defined in *include/minix/type.h*. Just as the boot monitor uses *aout* to pass information about all processes in the boot image to the kernel, the kernel fills in the fields of *kinfo* with information about itself that other components of the system may need to know about.

The next section of *glo.h* contains variables related to control of process and kernel execution. *Prev_ptr*, *proc_ptr*, and *next_ptr* point to the process table entries of the previous, current, and next processes to run. *Bill_ptr* also points to a process table entry; it shows which process is currently being billed for clock ticks used. When a user process calls the file system, and the file system is running, *proc_ptr* points to the file system process. However, *bill_ptr* will point to the user making the call, since CPU time used by the file system is charged as system time to the caller. We have not actually heard of a MINIX system whose owner charges others for their use of CPU time, but it could be done. The next variable, *k_reenter*, is used to count nested executions of kernel code, such as when an interrupt occurs when the kernel itself, rather than a user process, is running. This is important, because switching context from a user process to the kernel or vice versa is different (and more costly) than reentering the kernel. When an interrupt service complete it is important for it to determine whether control should remain with the kernel or if a user-space process should be restarted. This variable is also tested by some functions which disable and reenable interrupts, such as *lock_enqueue*. If such a function is executed when interrupts are disabled already, the interrupts should not be reenabled when reenabling is not wanted. Finally, in this section there is a counter for lost clock ticks. How a clock tick can be lost and what is done about it will be discussed when we discuss the clock task.

The last few variables defined in *glo.h*, are declared here because they must be known throughout the kernel code, but they are declared as *extern* rather than as *EXTERN* because they are **initialized variables**, a feature of the C language. The use of the *EXTERN* macro is not compatible with C-style initialization, since a variable can only be initialized once.

Tasks that run in kernel space, currently just the clock task and the system task, have their own stacks within *t_stack*. During interrupt handling, the kernel uses a separate stack, but it is not declared here, since it is only accessed by the assembly language level routine that handles interrupt processing, and does not need to be known globally. The last file included in *kernel.h*, and thus used in every compilation, is *ipc.h* (line 5400). It defines various constants used in interprocess communication. We will discuss these later when we get to the file where they are used, *kernel/proc.c*.

Several more kernel header files are widely used, although not so much that they are included in *kernel.h*. The first of these is *proc.h* (line 5500), which defines the kernel's process table. The complete state of a process is defined by the process' data in memory, plus the information in its process table slot. The contents of the CPU registers are stored here when a process is not executing and then are restored when execution resumes. This is what makes possible the illusion that multiple processes are executing simultaneously and interacting, although at any instant a single CPU can be executing instructions of only one process. The time spent by the kernel saving and restoring the process state during each **context switch** is necessary, but obviously this is time during which the work of the processes themselves is suspended. For this reason these structures are designed for efficiency. As noted in the comment at the beginning of *proc.h*, many routines written in assembly language also access these structures, and another header, *sconst.h*, defines offsets to fields in the process table for use by the assembly code. Thus changing a definition in *proc.h* may necessitate a change in *sconst.h*.

Before going further we should mention that, because of MINIX 3's microkernel structure, the process table we will discuss is here is paralleled by tables in PM and FS which contain per-process entries relevant to the function of these parts of MINIX 3. Together, all three of these tables are equivalent to the process table of an operating system with a monolithic structure, but for the moment when we speak of the process table we will be talking about only the kernel's process table. The others will be discussed in later chapters.

Each slot in the process table is defined as a struct *proc* (lines 5516 to 5545). Each entry contains storage for the process' registers, stack pointer, state, memory map, stack limit, process id, accounting, alarm time, and message info. The first part of each process table entry is a *stackframe_s* structure. A process that is already in memory is put into execution by loading its stack pointer with the address of its process table entry and popping all the CPU registers from this struct.

There is more to the state of a process than just the CPU registers and the data in memory, however. In MINIX 3, each process has a pointer to a *priv* structure in its process table slot (line 5522). This structure defines allowed sources and destinations of messages for the process and many other privileges. We will look at details later. For the moment, note that each system process has a pointer to a unique copy of this structure, but user privileges are all equal—the pointers of all user processes point to the same copy of the structure. There is also a byte-sized field for a set of bit flags, *p_rts_flags* (line 5523). The meanings of the bits will be described below. Setting any bit to 1 means a process is not runnable, so a zero in this field indicates a process is ready.

Each slot in the process table provides space for information that may be needed by the kernel. For instance, the *p_max_priority* field (line 5526), tells which scheduling queue the process should be queued on when it is ready to run for the first time. Because the priority of a process may be reduced if it prevents other processes from running, there is also a *p_priority* field which is initially set equal to *p_max_priority*. *P_priority* is the field that actually determines the queue used each time the process is ready.

The time used by each process is recorded in the two *clock_t* variables at lines 5532 and 5533. This information must be accessed by the kernel and it would be inefficient to store this in a process' own memory space, although logically that could be done. *P_nextready* (line 5535), is used to link processes together on the scheduler queues.

The next few fields hold information related to messages between processes. When a process cannot complete a send because the destination is not waiting, the sender is put onto a queue pointed to by the destination's *p_caller_q* pointer (line 5536). That way, when the destination finally does a receive, it is easy to find all the processes wanting to send to it. The *p_q_link* field (line 5537) is used to link the members of the queue together.

The rendezvous method of passing messages is made possible by the storage space reserved at lines 5538 to 5540. When a process does a receive and there is no message waiting for it, it blocks and the number of the process it wants to receive from is stored in *p_getfrom*. Similarly, *p_sendto* holds the process number of the destination when a process does a send and the recipient is not waiting. The address of the message buffer is stored in *p_messbuf*. The penultimate field in each process table slot is *p_pending* (line 5542), a bitmap used to keep track of signals that have not yet been passed to the process manager (because the process manager is not waiting for a message).

Finally, the last field in a process table entry is a character array, *p_name*, for holding the name of the process. This field is not needed for process management by the kernel. MINIX 3 provides various **debug dumps** triggered by pressing a special key on the console keyboard. Some of these allow viewing information about all processes, with the name of each process printed along with other data. Having a meaningful name associated with each process makes understanding and debugging kernel operation easier.

Following the definition of a process table slot come definitions of various constants used in its elements. The various flag bits that can be set in *p_rts_flags* are defined and described on lines 5548 to 5555. If the slot is not in use, *SLOT_FREE* is set. After a fork, *NO_MAP* is set to prevent the child process from running until its memory map has been set up. *SENDING* and *RECEIVING* indicate that the process is blocked trying to send or receive a message. *SIGNALED* and *SIG_PENDING* indicate that signals have been received, and *P_STOP* provides support for tracing. *NO_PRIV* is used to temporarily prevent a new system process from executing until its setup is complete.

The number of scheduling queues and allowable values for the *p_priority* field are defined next (lines 5562 to 5567). In the current version of this file user processes are allowed to be given access to the highest priority queue; this is probably a carry-over from the early days of testing drivers in user space and *MAX_USER_Q* should probably adjusted to a lower priority (larger number).

Next come several macros that allow addresses of important parts of the process table to be defined as constants at compilation time, to provide faster access at run time, and then more macros for run time calculations and tests. The macro *proc_addr* (line 5577) is provided because it is not possible to have negative subscripts in C. Logically, the array *proc* should go from *−NR_TASKS* to *+NR_PROCS*. Unfortunately, in C it must start at 0, so *proc*[0] refers to the most negative task, and so forth. To make it easier to keep track of which slot goes with which process, we can write

```
rp = proc_addr(n);
```

to assign to *rp* the address of the process slot for process n, either positive or negative.

The process table itself is defined here as an array of *proc* structures, *proc*[*NR_TASKS* + *NR_PROCS*] (line 5593). Note that *NR_TASKS* is defined in *include/minix/com.h* (line 3630) and the constant *NR_PROCS* is defined in *include/minix/config.h* (line 2522). Together these set the size of the kernel's process table. *NR_PROCS* can be changed to create a system capable of handling a larger number of processes, if that is necessary (e.g., on a large server).

Finally, several macros are defined to speed access. The process table is accessed frequently, and calculating an address in an array requires slow multiplication operations, so an array of pointers to the process table elements, *pproc_addr* (line 5594), is provided. The two arrays *rdy_head* and *rdy_tail* are used to maintain the scheduling queues. For example, the first process on the default user queue is pointed to by *rdy_head*[*USER_Q*].

As we mentioned at the beginning of the discussion of *proc.h* there is another file *sconst.h* (line 5600), which must be synchronized with *proc.h* if there are changes in the structure of the process table. *Sconst.h* defines constants used by assembler code, expressed in a form usable by the assembler. All of these are offsets into the *stackframe_s* structure portion of a process table entry. Since assembler code is not processed by the C compiler, it is simpler to have such definitions in a separate file. Also, since these definitions are all machine dependent, isolating them here simplifies the process of porting MINIX 3 to another processor which will need a different version of *sconst.h*. Note that many offsets are expressed as the previous value plus *W*, which is set equal to the word size at line 5601. This allows the same file to serve for compiling a 16-bit or 32-bit version of MINIX 3.

Duplicate definitions create a potential problem. Header files are supposed to allow one to provide a single correct set of definitions and then proceed to use them in many places without devoting a lot of further attention to the details. Obviously, duplicate definitions, like those in *proc.h* and *sconst.h*, violate that principle. This is a special case, of course, but as such, special attention is required if changes are made to either of these files to ensure the two files remain consistent.

The system privileges structure, *priv*, that was mentioned briefly in the discussion of the process table is fully defined in *priv.h*, on lines 5718 to 5735. First there is a set of flag bits, *s_flags*, and then come the *s_trap_mask*, *s_ipc_from*, *s_ipc_to*, and *s_call_mask* fields which define which system calls may be initiated, which processes messages may be received from or sent to, and which kernel calls are allowed.

The *priv* structure is not part of the process table, rather each process table slot has a pointer to an instance of it. Only system processes have private copies; user processes all point to the same copy. Thus, for a user process the remaining fields of the structure are not relevant, as sharing them does not make sense. These fields are bitmaps of pending notifications, hardware interrupts, and signals, and a timer. It makes sense to provide these here for system processes, however. User processes have notifications, signals, and timers managed on their behalf by the process manager.

The organization of *priv.h* is similar to that of *proc.h*. After the definition of the *priv* structure come macros definitions for the flag bits, some important addresses known at compile time, and some macros for address calculations at run time. Then the table of *priv* structures, *priv[NR_SYS_PROCS]*, is defined, followed by an array of pointers, *ppriv_addr[NR_SYS_PROCS]* (lines 5762 and 5763). The pointer array provides fast access, analogous to the array of pointers that provides fast access to process table slots. The value of *STACK_GUARD* defined on line 5738 is a pattern that is easily recognizable. Its use will be seen later; the reader is invited to search the Internet to learn about the history of this value.

The last item in *priv.h* is a test to make sure that *NR_SYS_PROCS* has been defined to be larger than the number of processes in the boot image. The #error line will print a message if the test condition tests true. Although behavior may be different with other C compilers, with the standard MINIX 3 compiler this will also abort the compilation.

The F4 key triggers a debug dump that shows some of the information in the privilege table. Figure 2-35 shows a few lines of this table for some representative processes. The flags entries mean P: preemptable, B: billable, S: system. The traps mean E: echo, S: send, R: receive, B: both, N: notification. The bitmap has a bit for each of the *NR_SYS_PROCS* (32) system processes allowed, the order corresponds to the id field. (In the figure only 16 bits are shown, to make it fit the page better.) All user processes share id 0, which is the left-most bit position. The bitmap shows that user processes such as *init* can send messages only to the process manager, file system, and reincarnation server, and must use sendrec. The servers and drivers shown in the figure can use any of the ipc primitives and all but *memory* can send to any other process.

```
--nr-   -id-   -name-    -flags-   -traps-   -ipc_to mask------
 (-4)   (01)   IDLE      P-BS-     -----     00000000 00001111
 [-3]   (02)   CLOCK     ---S-     --R--     00000000 00001111
 [-2]   (03)   SYSTEM    ---S-     --R--     00000000 00001111
 [-1]   (04)   KERNEL    ---S-     -----     00000000 00001111
    0   (05)   pm        P--S-     ESRBN     11111111 11111111
    1   (06)   fs        P--S-     ESRBN     11111111 11111111
    2   (07)   rs        P--S-     ESRBN     11111111 11111111
    3   (09)   memory    P--S-     ESRBN     00110111 01101111
    4   (10)   log       P--S-     ESRBN     11111111 11111111
    5   (08)   tty       P--S-     ESRBN     11111111 11111111
    6   (11)   driver    P--S-     ESRBN     11111111 11111111
    7   (00)   init      P-B--     E--B-     00000111 00000000
```

Figure 2-35. Part of a debug dump of the privilege table. The clock task, file server, tty, and init processes privileges are typical of tasks, servers, device drivers, and user processes, respectively. The bitmap is truncated to 16 bits.

Another header that is included in a number of different source files is *protect.h* (line 5800). Almost everything in this file deals with architecture details of the Intel processors that support protected mode (the 80286, 80386, 80486, and the Pentium series). A detailed description of these chips is beyond the scope of this book. Suffice it to say that they contain internal registers that point to **descriptor tables** in memory. Descriptor tables define how system resources are used and prevent processes from accessing memory assigned to other processes.

The architecture of 32-bit Intel processors also provides for four **privilege levels**, of which MINIX 3 takes advantage of three. These are defined symbolically on lines 5843 to 5845. The most central parts of the kernel, the parts that run during interrupts and that manage context switches, always run with *INTR_PRIVILEGE*. Every address in the memory and every register in the CPU can be accessed by a process with this privilege level. The tasks run at *TASK_PRIVILEGE* level, which allows them to access I/O but not to use instructions that modify special registers, like those that point to descriptor tables. Servers and user processes run at *USER_PRIVILEGE* level. Processes executing at this level are unable to execute certain instructions, for instance those that access I/O ports, change memory assignments, or change privilege levels themselves.

The concept of privilege levels will be familiar to those who are familiar with the architecture of modern CPUs, but those who have learned computer architecture through study of the assembly language of low-end microprocessors may not have encountered such features.

One header file in *kernel/* has not yet been described: *system.h*, and we will postpone discussing it until later in this chapter when we describe the system task, which runs as an independent process, although it is compiled with the kernel. For now we are through with header files and are ready to dig into the **.c* C language source files. The first of these that we will look at is *table.c* (line 6000). Compilation of this produces no executable code, but the compiled object file *table.o* will contain all the kernel data structures. We have already seen many of these data structures defined, in *glo.h* and other headers. On line 6028 the macro *_TABLE* is defined, immediately before the `#include` statements. As explained earlier, this definition causes *EXTERN* to become defined as the null string, and storage space to be allocated for all the data declarations preceded by *EXTERN*.

In addition to the variables declared in header files there are two other places where global data storage is allocated. Some definitions are made directly in *table.c*. On lines 6037 to 6041 the stack space needed by kernel components is defined, and the total amount of stack space for tasks is reserved as the array *t_stack[TOT_STACK_SPACE]* on line 6045.

The rest of *table.c* defines many constants related to properties of processes, such as the combinations of flag bits, call traps, and masks that define to whom messages and notifications can be sent that we saw in Fig. 2-35 (lines 6048 to 6071). Following this are masks to define the kernel calls allowed for various processes. The process manager and file server are all allowed unique combinations. The reincarnation server is allowed access to all kernel calls, not for its own use, but because as the parent of other system processes it can only pass to its children subsets of its own privileges. Drivers are given a common set of kernel call masks, except for the RAM disk driver which needs unusual access to memory. (Note that the comment on line 6075 that mentions the “system services manager” should say “reincarnation server”—the name was changed during development and some comments still refer to the old name.)

Finally, on lines 6095 to 6109, the *image* table is defined. It has been put here rather than in a header file because the trick with *EXTERN* used to prevent multiple declarations does not work with initialized variables; that is, you may not say

```
extern int x = 3;
```

anywhere. The *image* table provides details needed to initialize all of the processes that are loaded from the boot image. It will be used by the system at startup. As an example of the information contained here, consider the field labeled “qs” in the comment on line 6096. This shows the size of the quantum assigned to each process. Ordinary user processes, as children of init, get to run for 8 clock ticks. The CLOCK and SYSTEM tasks are allowed to run for 64 clock ticks if necessary. They are not really expected to run that long before blocking, but unlike user-space servers and drivers they cannot be demoted to a lower-priority queue if they prevent other processes from getting a chance to run.

If a new process is to be added to the boot image, a new row must be provided in the *image* table. An error in matching the size of *image* to other constants is intolerable and cannot be permitted. At the end of *table.c* tests are made for errors, using a little trick. The array *dummy* is declared here twice. In each declaration the size of *dummy* will be impossible and will trigger a compiler error if a mistake has been made. Since *dummy* is declared as *extern*, no space is allocated for it here (or anywhere). Since it is not referenced anywhere else in the code, this will not bother the compiler.

Additional global storage is allocated at the end of the assembly language file *mpx386.s*. Although it will require skipping ahead several pages in the listing to see this, it is appropriate to discuss this now, since we are on the subject of global variables. On line 6822 the assembler directive `.sect .rom` is used to put a **magic number** (to identify a valid MINIX 3 kernel) at the very beginning of the kernel’s data segment. A `.sect bss` assembler directive and the `.space` pseudoinstruction are also used here to reserve space for the kernel’s stack. The `.comm` pseudoinstruction labels several words at the top of the stack so they may be manipulated directly. We will come back to *mpx386.s* in a few pages, after we have discussed bootstrapping MINIX 3.

2.6.6 Bootstrapping MINIX 3

It is almost time to start looking at the executable code—but not quite. Before we do that, let us take a few moments to understand how MINIX 3 is loaded into memory. It is, of course, loaded from a disk, but the process is not completely trivial and the exact sequence of events depends on the kind of disk. In particular, it depends on whether the disk is partitioned or not. Figure 2-36 shows how diskettes and partitioned disks are laid out.

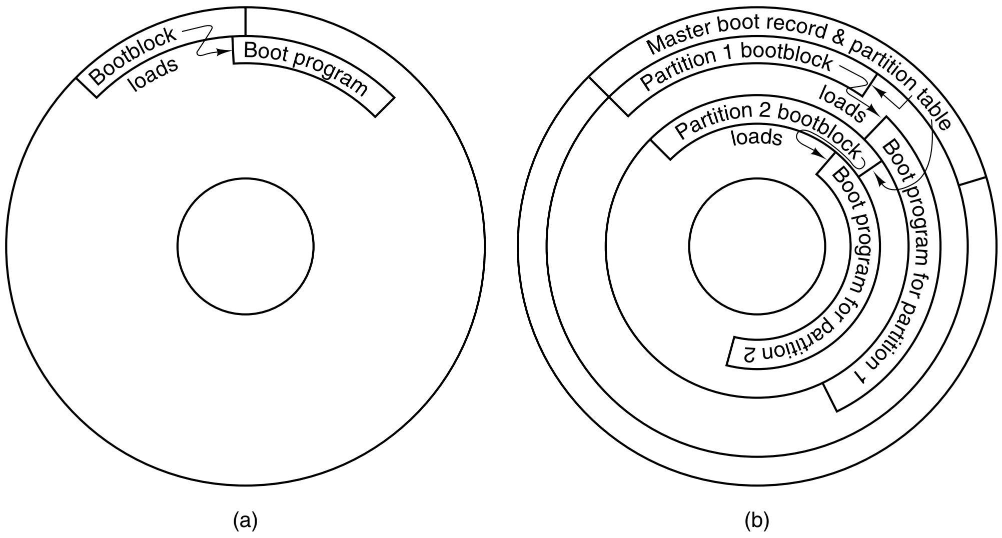
Bootblock
loads
Boot program
(a)
Master boot record & partition table
Partition 1 bootblock
loads
Partition 2 bootblock
loads
Boot program for partition 1
Boot program for partition 2
(b)

Figure 2-36. Disk structures used for bootstrapping. (a) Unpartitioned disk. The first sector is the bootblock. (b) Partitioned disk. The first sector is the master boot record, also called **masterboot**.

When the system is started, the hardware (actually, a program in ROM) reads the first sector of the boot disk, copies it to a fixed location in memory, and executes the code found there. On an unpartitioned MINIX 3 diskette, the first sector is a bootblock which loads the boot program, as in Fig. 2-36(a). Hard disks are partitioned, and the program on the first sector (called masterboot on MINIX systems) first relocates itself to a different memory region, then reads the partition table, loaded with it from the first sector. Then it loads and executes the first sector of the active partition, as shown in Fig. 2-36(b). (Normally one and only one partition is marked active). A MINIX 3 partition has the same structure as an unpartitioned MINIX 3 diskette, with a bootblock that loads the boot program. The bootblock code is the same for an unpartitioned or a partitioned disk. Since the masterboot program relocates itself the bootblock code can be written to run at the same memory address where masterboot is originally loaded.

The actual situation can be a little more complicated than the figure shows, because a partition may contain subpartitions. In this case the first sector of the partition will be another master boot record containing the partition table for the subpartitions. Eventually, however, control will be passed to a boot sector, the first sector on a device that is not further subdivided. On a diskette the first sector is always a boot sector. MINIX 3 does allow a form of partitioning of a diskette, but only the first partition may be booted; there is no separate master boot record, and subpartitions are not possible. This makes it possible for partitioned and non-partitioned diskettes to be mounted in exactly the same way. The main use for a partitioned floppy disk is that it provides a convenient way to divide an installation disk into a root image to be copied to a RAM disk and a mounted portion that can be dismounted when no longer needed, in order to free the diskette drive for continuing the installation process.

The MINIX 3 boot sector is modified at the time it is written to the disk by a special program called *installboot* which writes the boot sector and patches into it the disk address of a file named *boot* on its partition or subpartition. In MINIX 3, the standard location for the *boot* program is in a directory of the same name, that is, */boot/boot*. But it could be anywhere—the patching of the boot sector just mentioned locates the disk sectors from which it is to be loaded. This is necessary because previous to loading *boot* there is no way to use directory and file names to find a file.

Boot is the secondary loader for MINIX 3. It can do more than just load the operating system however, as it is a **monitor program** that allows the user to change, set, and save various parameters. *Boot* looks in the second sector of its partition to find a set of parameters to use. MINIX 3, like standard UNIX, reserves the first 1K block of every disk device as a **bootblock**, but only one 512-byte sector is loaded by the ROM boot loader or the master boot sector, so 512 bytes are available for saving settings. These control the boot operation, and are also passed to the operating system itself. The default settings present a menu with one choice, to start MINIX 3, but the settings can be modified to present a more complex menu allowing other operating systems to be started (by loading and executing boot sectors from other partitions), or to start MINIX 3 with various options. The default settings can also be modified to bypass the menu and start MINIX 3 immediately.

Boot is not a part of the operating system, but it is smart enough to use the file system data structures to find the actual operating system image. *Boot* looks for a file with the name specified in the *image=* boot parameter, which by default is */boot/image*. If there is an ordinary file with this name it is loaded, but if this is the name of a directory the newest file within it is loaded. Many operating systems have a predefined file name for the boot image. But MINIX 3 users are encouraged to modify it and to create new versions. It is useful to be able to select from multiple versions, in order to return to an older version if an experiment is unsuccessful.

We do not have space here to go into more detail about the boot monitor. It is a sophisticated program, almost a miniature operating system in itself. It works together with MINIX 3, and when MINIX 3 is properly shut down, the boot monitor regains control. If you would like to know more, the MINIX 3 Web site provides a link to a detailed description of the boot monitor source code.

The MINIX 3 **boot image** (also called **system image**) is a concatenation of several program files: the kernel, process manager, file system, reincarnation server, several device drivers, and *init*, as shown in Fig 2-30. Note that MINIX 3 as described here is configured with just one disk driver in the boot image, but several may be present, with the active one selected by a label. Like all binary programs, each file in the boot image includes a header that tells how much space to reserve for uninitialized data and stack after loading the executable code and initialized data, so the next program can be loaded at the proper address.

The memory regions available for loading the boot monitor and the component programs of MINIX 3 will depend upon the hardware. Also, some architectures may require adjustment of internal addresses within executable code to correct them for the actual address where a program is loaded. The segmented architecture of Intel processors makes this unnecessary.

Details of the loading process differ with machine type. The important thing is that by one means or another the operating system is loaded into memory. Following this, a small amount of preparation is required before MINIX 3 can be started. First, while loading the image, *boot* reads a few bytes from the image that tell boot some of its properties, most importantly whether it was compiled to run in 16-bit or 32-bit mode. Then some additional information needed to start the system is made available to the kernel. The *a.out* headers of the components of the MINIX 3 image are extracted into an array within *boot*'s memory space, and the base address of this array is passed to the kernel. MINIX 3 can return control to the boot monitor when it terminates, so the location where execution should resume in the monitor is also passed on. These items are passed on the stack, as we shall see later.

Several other pieces of information, the **boot parameters**, must be communicated from the boot monitor to the operating system. Some are needed by the kernel and some are not needed but are passed along for information, for instance, the name of the boot image that was loaded. These items can all be represented as *string=value* pairs, and the address of a table of these pairs is passed on the stack. Fig. 2-37 shows a typical set of boot parameters as displayed by the `sysenv` command from the MINIX 3 command line.

In this example, an important item we will see again soon is the *memory* parameter; in this case it indicates that the boot monitor has determined that there are two segments of memory available for MINIX 3 to use. One begins at hexadecimal address 800 (decimal 2048) and has a size of hexadecimal 0x92540 (decimal 599,360) bytes; the other begins at 100000 (1,048,576) and contains 0x3df00000 (64,946,176) bytes. This is typical of all but the most elderly PC-compatible computers. The design of the original IBM PC placed read-only memory at the top of the usable range of memory, which is limited to 1 MB on an 8088 CPU. Modern PC-compatible machines always have more memory than the original PC, but for compatibility they still have read-only memory at the same addresses as the older machines. Thus, the read-write memory is discontinuous,

```
rootdev=904
ramimagedev=904
ramsize=0
processor=686
bus=at
video=vga
chrome=color
memory=800:92540,100000:3DF0000
label=AT
controller=c0
image=boot/image
```

Figure 2-37. Boot parameters passed to the kernel at boot time in a typical MINIX 3 system.

with a block of ROM between the lower 640 KB and the upper range above 1 MB. The boot monitor loads the kernel into the low memory range and the servers, drivers, and *init* into the memory range above the ROM if possible. This is primarily for the benefit of the file system, so a large block cache can be used without bumping into the read-only memory.

We should also mention here that operating systems are not universally loaded from local disks. **Diskless workstations** may load their operating systems from a remote disk, over a network connection. This requires network software in ROM, of course. Although details vary from what we have described here, the elements of the process are likely to be similar. The ROM code must be just smart enough to get an executable file over the net that can then obtain the complete operating system. If MINIX 3 were loaded this way, very little would need to be changed in the initialization process that occurs once the operating system code is loaded into memory. It would, of course, need a network server and a modified file system that could access files via the network.

2.6.7 System Initialization

Earlier versions of MINIX could be compiled in 16-bit mode if compatibility with older processor chips were required, and MINIX 3 retains some source code for 16-bit mode. However, the version described here and distributed on the CD-ROM is usable only on 32-bit machines with 80386 or better processors. It does not work in 16-bit mode, and creation of a 16-bit version may require removing some features. Among other things, 32-bit binaries are larger than 16-bit ones, and independent user-space drivers cannot share code the way it could be done when drivers were compiled into a single binary. Nevertheless, a common base of C source code is used and the compiler generates the appropriate output depending upon whether the compiler itself is the 16-bit or 32-bit version of the compiler.

A macro defined by the compiler itself determines the definition of the *_WORD_SIZE* macro in the file *include/minix/sys_config.h*.

The first part of MINIX 3 to execute is written in assembly language, and different source code files must be used for the 16-bit or 32-bit compiler. The 32-bit version of the initialization code is in *mpx386.s*. The alternative, for 16-bit systems, is in *mpx88.s*. Both of these also include assembly language support for other low-level kernel operations. The selection is made automatically in *mpx.s*. This file is so short that the entire file can be presented in Fig. 2-38.

```
#include <minix/config.h>
#if _WORD_SIZE == 2
#include "mpx88.s"
#else
#include "mpx386.s"
#endif
```

Figure 2-38. How alternative assembly language source files are selected.

Mpx.s shows an unusual use of the C preprocessor #include statement. Customarily the #include preprocessor directive is used to include header files, but it can also be used to select an alternate section of source code. Using #if statements to do this would require putting all the code in both of the large files *mpx88.s* and *mpx386.s* into a single file. Not only would this be unwieldy; it would also be wasteful of disk space, since in a particular installation it is likely that one or the other of these two files will not be used at all and can be archived or deleted. In the following discussion we will use the 32-bit *mpx386.s*.

Since this is almost our first look at executable code, let us start with a few words about how we will do this throughout the book. The multiple source files used in compiling a large C program can be hard to follow. In general, we will keep discussions confined to a single file at a time. The order of inclusion of the files in Appendix B is the order in which we discuss them in the text. We will start with the entry point for each part of the MINIX 3 system, and we will follow the main line of execution. When a call to a supporting function is encountered, we will say a few words about the purpose of the call, but normally we will not go into a detailed description of the internals of the function at that point, leaving that until we arrive at the definition of the called function. Important subordinate functions are usually defined in the same file in which they are called, following the higher-level calling functions, but small or general-purpose functions are sometimes collected in separate files. We do not attempt to discuss the internals of every function, and files that contain such functions may not be listed in Appendix B.

To facilitate portability to other platforms, separate files are frequently used for machine-dependent and machine-independent code. To make code easier to understand and reduce the overall size of the listings, most conditional code for platforms other than Intel 32-bit systems has been stripped from the printed files in Appendix B. Complete versions of all files are in the source directories on the CD-ROM and are also available on the MINIX 3 Web site.

A substantial amount of effort has been made to make the code readable by humans. But a large program has many branches, and sometimes understanding a main function requires reading the functions it calls, so having a few slips of paper to use as bookmarks and deviating from our order of discussion to look at things in a different order may be helpful at times.

Having laid out our intended way of organizing the discussion of the code, we start by an exception. Startup of MINIX 3 involves several transfers of control between the assembly language routines in *mpx386.s* and C language routines in the files *start.c* and *main.c*. We will describe these routines in the order that they are executed, even though that involves jumping from one file to another.

Once the bootstrap process has loaded the operating system into memory, control is transferred to the label *MINIX* (in *mpx386.s*, line 6420). The first instruction is a jump over a few bytes of data; this includes the boot monitor flags (line 6423) mentioned earlier. At this point the flags have already served their purpose; they were read by the monitor when it loaded the kernel into memory. They are located here because it is an easily specified address. They are used by the boot monitor to identify various characteristics of the kernel, most importantly, whether it is a 16-bit or 32-bit system. The boot monitor always starts in 16-bit mode, but switches the CPU to 32-bit mode if necessary. This happens before control passes to the label *MINIX*.

Understanding the state of the stack at this point will help make sense of the following code. The monitor passes several parameters to MINIX 3, by putting them on the stack. First the monitor pushes the address of the variable *aout*, which holds the address of an array of the header information of the component programs of the boot image. Next it pushes the size and then the address of the boot parameters. These are all 32-bit quantities. Next come the monitor's code segment address and the location to return to within the monitor when MINIX 3 terminates. These are both 16-bit quantities, since the monitor operates in 16-bit protected mode. The first few instructions in *mpx386.s* convert the 16-bit stack pointer used by the monitor into a 32-bit value for use in protected mode. Then the instruction

```
mov     ebp, esp
```

(line 6436) copies the stack pointer value to the `ebp` register, so it can be used with offsets to retrieve from the stack the values placed there by the monitor, as is done at lines 6464 to 6467. Note that because the stack grows downward with Intel processors, `8(ebp)` refers to a value pushed subsequent to pushing the value located at `12(ebp)`.

The assembly language code must do a substantial amount of work, setting up a stack frame to provide the proper environment for code compiled by the C compiler, copying tables used by the processor to define memory segments, and setting up various processor registers. As soon as this work is complete, the initialization process continues by calling (at line 6481) the C function *cstart* (in *start.c*, which we will consider next). Note that it is referred to as *_cstart* in the assembly language code. This is because all functions compiled by the C compiler have an underscore prepended to their names in the symbol tables, and the linker looks for such names when separately compiled modules are linked. Since the assembler does not add underscores, the writer of an assembly language program must explicitly add one in order for the linker to be able to find a corresponding name in the object file compiled by the C compiler.

Cstart calls another routine to initialize the **Global Descriptor Table**, the central data structure used by Intel 32-bit processors to oversee memory protection, and the **Interrupt Descriptor Table**, used to select the code to be executed for each possible interrupt type. Upon returning from *cstart* the lgdt and lidt instructions (lines 6487 and 6488) make these tables effective by loading the dedicated registers by which they are addressed. The instruction

```
jmpf    CS_SELECTOR:csinit
```

looks at first glance like a no-operation, since it transfers control to exactly where control would be if there were a series of nop instructions in its place. But this is an important part of the initialization process. This jump forces use of the structures just initialized. After some more manipulation of the processor registers, *MINIX* terminates with a jump (not a call) at line 6503 to the kernel's *main* entry point (in *main.c*). At this point the initialization code in *mpx386.s* is complete. The rest of the file contains code to start or restart a task or process, interrupt handlers, and other support routines that had to be written in assembly language for efficiency. We will return to these in the next section.

We will now look at the top-level C initialization functions. The general strategy is to do as much as possible using high-level C code. As we have seen, there are already two versions of the *mpx* code. One chunk of C code can eliminate two chunks of assembler code. Almost the first thing done by *cstart* (in *start.c*, line 6920) is to set up the CPU's protection mechanisms and the interrupt tables, by calling *prot_init*. Then it copies the boot parameters to the kernel's memory, and it scans them, using the function *get_value* (line 6997) to search for parameter names and return corresponding value strings. This process determines the type of video display, processor type, bus type, and, if in 16-bit mode, the processor operating mode (real or protected). All this information is stored in global variables, for access when needed by any part of the kernel code.

Main (in *main.c*, line 7130), completes initialization and then starts normal execution of the system. It configures the interrupt control hardware by calling *intr_init*. This is done here because it cannot be done until the machine type is known. (Because *intr_init* is very dependent upon the hardware the procedure is in a separate file which we will describe later.) The parameter (1) in the call tells *intr_init* that it is initializing for MINIX 3. With a parameter (0) it can be called to reinitialize the hardware to the original state when MINIX 3 terminates and returns control to the boot monitor. *Intr_init* ensures that any interrupts that occur before initialization is complete have no effect. How this is done will be described later.

The largest part of *main*'s code is devoted to setup of the process table and the privilege table, so that when the first tasks and processes are scheduled, their memory maps, registers, and privilege information will be set correctly. All slots in the process table are marked as free and the *pproc_addr* array that speeds access to the process table is initialized by the loop on lines 7150 to 7154. The loop on lines 7155 to 7159 clears the privilege table and the *ppriv_addr* array similarly to the process table and its access array. For both the process and privilege tables, putting a specific value in one field is adequate to mark the slot as not in use. But for each table every slot, whether in use or not, needs to be initialized with an index number.

An aside on a minor characteristic of the C language: the code on line 7153

```
(pproc_addr + NR_TASKS)[i] = rp;
```

could just as well have been written as

```
pproc_addr[i + NR_TASKS] = rp;
```

In the C language *a*[*i*] is just another way of writing *(*a*+*i*). So it does not make much difference if you add a constant to *a* or to *i*. Some C compilers generate slightly better code if you add a constant to the array instead of the index. Whether it really makes a difference here, we cannot say.

Now we come to the long loop on lines 7172 to 7242, which initializes the process table with the necessary information to run all of the processes in the boot image. (Note that there is another outdated comment on line 7161 which mentions only tasks and servers.) All of these processes must be present at startup time and none of them will terminate during normal operation. At the start of the loop, *ip* is assigned the address of an entry in the *image* table created in *table.c* (line 7173). Since *ip* is a pointer to a structure, the elements of the structure can be accessed using notation like *ip−>proc_nr*, as is done on line 7174. This notation is used extensively in the MINIX 3 source code. In a similar way, *rp* is a pointer to a slot of the process table, and *priv(rp)* points to a slot of the privilege table. Much of the initialization of the process and privilege tables in the long loop consists of reading a value from the image table and storing it in the process table or the privilege table.

On line 7185 a test is made for processes that are part of the kernel, and if this is true the special *STACK_GUARD* pattern is stored in the base of the task's stack area. This can be checked later on to be sure the stack has not overflowed. Then the initial stack pointer for each task is set up. Each task needs its own private stack pointer. Since the stack grows toward lower addresses in memory, the initial stack pointer is calculated by adding the size of the task's stack to the current base address (lines 7190 and 7191). There is one exception: the *KERNEL* process (also identified as *HARDWARE* in some places) is never considered ready, never runs as an ordinary process, and thus has no need of a stack pointer.

The binaries of boot image components are compiled like any other MINIX 3 programs, and the compiler creates a header, as defined in *include/a.out.h*, at the beginning of each of the files. The boot loader copies each of these headers into its own memory space before MINIX 3 starts, and when the monitor transfers control to the *MINIX:* entry point in *mpx386.s* the physical address of the header area is passed to the assembly code in the stack, as we have seen. At line 7202, one of these headers is copied to a local *exec* structure, *ehdr*, using *hdrindex* as the index into the array of headers. Then the data and text segment addresses are converted to clicks and entered into the memory map for this process (lines 7205 to 7214).

Before continuing, we should mention a few points. First, for kernel processes *hdrindex* is always assigned a value of zero at line 7178. These processes are all compiled into the same file as the kernel, and the information about their stack requirements is in the *image* table. Since a task compiled into the kernel can call code and access data located anywhere in the kernel's space, the size of an individual task is not meaningful. Thus the same element of the array at *aout* is accessed for the kernel and for each task, and the size fields for a task is filled with the sizes for the kernel itself. The tasks get their stack information from the *image* table, initialized during compilation of *table.c*. After all kernel processes have been processed, *hdrindex* is incremented on each pass through the loop (line 7196), so all the user-space system processes get the proper data from their own headers.

Another point to mention here is that functions that copy data are not necessarily consistent in the order in which the source and destination are specified. In reading this loop, beware of potential confusion. The arguments to *strncpy*, a function from the standard C library, are ordered such that the destination comes first: `strncpy(to, from, count)`. This is analogous to an assignment operation, in which the left hand side specifies the variable being assigned to and the right hand side is the expression specifying the value to be assigned. This function is used at line 7179 to copy a process name into each process table slot for debugging and other purposes. In contrast, the *phys_copy* function uses an opposite convention, `phys_copy(from, to, quantity)`. *Phys_copy* is used at line 7202 to copy program headers of user-space processes.

Continuing our discussion of the initialization of the process table, at lines 7220 and 7221 the initial value of the program counter and the processor status word are set. The processor status word for the tasks is different from that for device drivers and servers, because tasks have a higher privilege level that allows them to access I/O ports. Following this, if the process is a user-space one, its stack pointer is initialized.

One entry in the process table does not need to be (and cannot be) scheduled. The *HARDWARE* process exists only for bookkeeping purposes—it is credited with the time used while servicing an interrupt. All other processes are put on the appropriate queues by the code in lines 7234 and 7235. The function called *lock_enqueue* disables interrupts before modifying the queues and then reenables them when the queue has been modified. This is not required at this point when nothing is running yet, but it is the standard method, and there is no point in creating extra code to be used just once.

The last step in initializing each slot in the process table is to call the function *alloc_segments* at line 7241. This machine-dependent routine sets into the proper fields the locations, sizes, and permission levels for the memory segments used by each process. For older Intel processors that do not support protected mode, it defines only the segment locations. It would have to be rewritten to handle a processor type with a different method of allocating memory.

Once the process table has been initialized for all the tasks, the servers, and *init*, the system is almost ready to roll. The variable *bill_ptr* tells which process gets billed for processor time; it needs to have an initial value set at line 7250, and *IDLE* is clearly an appropriate choice. Now the kernel is ready to begin its normal work of controlling and scheduling the execution of processes, as illustrated in Fig. 2-2.

Not all of the other parts of the system are ready for normal operation yet, but all of these other parts run as independent processes and have been marked ready and queued to run. They will initialize themselves when they run. All that is left is for the kernel to call *announce* to announce it is ready and then to call *restart* (lines 7251 and 7252). In many C programs *main* is a loop, but in the MINIX 3 kernel its job is done once the initialization is complete. The call to *restart* on line 7252 starts the first queued process. Control never returns to *main*.

_Restart is an assembly language routine in *mpx386.s*. In fact, *_restart* is not a complete function; it is an intermediate entry point in a larger procedure. We will discuss it in detail in the next section; for now we will just say that *_restart* causes a context switch, so the process pointed to by *proc_ptr* will run. When *_restart* has executed for the first time we can say that MINIX 3 is running—it is executing a process. *_Restart* is executed again and again as tasks, servers, and user processes are given their opportunities to run and then are suspended, either to wait for input or to give other processes their turns.

Of course, the first time *_restart* is executed, initialization is only complete for the kernel. Recall that there are three parts to the MINIX 3 process table. You might ask how can any processes run when major parts of the process table have not been set up yet. The full answer to this will be seen in later chapters. The short answer is that the instruction pointers of all processes in the boot image initially point to initialization code for each process, and all will block fairly soon. Eventually, the process manager and the file system will get to run their initialization code, and their parts of the process table will be completed. Eventually *init* will fork off a *getty* process for each terminal. These processes will block until input is typed at some terminal, at which point the first user can log in.

We have now traced the startup of MINIX 3 through three files, two written in C and one in assembly language. The assembly language file, *mpx386.s*, contains additional code used in handling interrupts, which we will look at in the next section. However, before we go on let us wrap up with a brief description of the remaining routines in the two C files. The remaining function in *start.c* is *get_value* (line 6997). It is used to find entries in the kernel environment, which is a copy of the boot parameters. It is a simplified version of a standard library function which is rewritten here in order to keep the kernel simple.

There are three additional procedures in *main.c*. *Announce* displays a copyright notice and tells whether MINIX 3 is running in real mode or 16-bit or 32-bit protected mode, like this:

```
MINIX 3.1 Copyright 2006 Vrije Universiteit, Amsterdam, The Netherlands
Executing in 32-bit protected mode
```

When you see this message you know initialization of the kernel is complete. *Prepare_shutdown* (line 7272) signals all system processes with a *SIGKSTOP* signal (system processes cannot be signaled in the same way as user processes). Then it sets a timer to allow all the system process time to clean up before it calls the final procedure here, *shutdown. Shutdown* will normally return control to the MINIX 3 boot monitor. To do so the interrupt controllers are restored to the BIOS settings by the *intr_init(0)* call on line 7338.

2.6.8 Interrupt Handling in MINIX

Details of interrupt hardware are system dependent, but any system must have elements functionally equivalent to those to be described for systems with 32-bit Intel CPUs. Interrupts generated by hardware devices are electrical signals and are handled in the first place by an interrupt controller, an integrated circuit that can sense a number of such signals and for each one generate a unique data pattern on the processor's data bus. This is necessary because the processor itself has only one input for sensing all these devices, and thus cannot differentiate which device needs service. PCs using Intel 32-bit processors are normally equipped with two such controller chips. Each can handle eight inputs, but one is a slave which feeds its output to one of the inputs of the master, so fifteen distinct external devices can be sensed by the combination, as shown in Fig. 2-39. Some of the fifteen inputs are dedicated; the clock input, IRQ 0, for instance, does not have a connection to any socket into which a new adapter can be plugged. Others are connected to sockets and can be used for whatever device is plugged in.

In the figure, interrupt signals arrive on the various *IRQ n* lines shown at the right. The connection to the CPU's INT pin tells the processor that an interrupt has occurred. The INTA (interrupt acknowledge) signal from the CPU causes the controller responsible for the interrupt to put data on the system data bus telling the processor which service routine to execute. The interrupt controller chips are

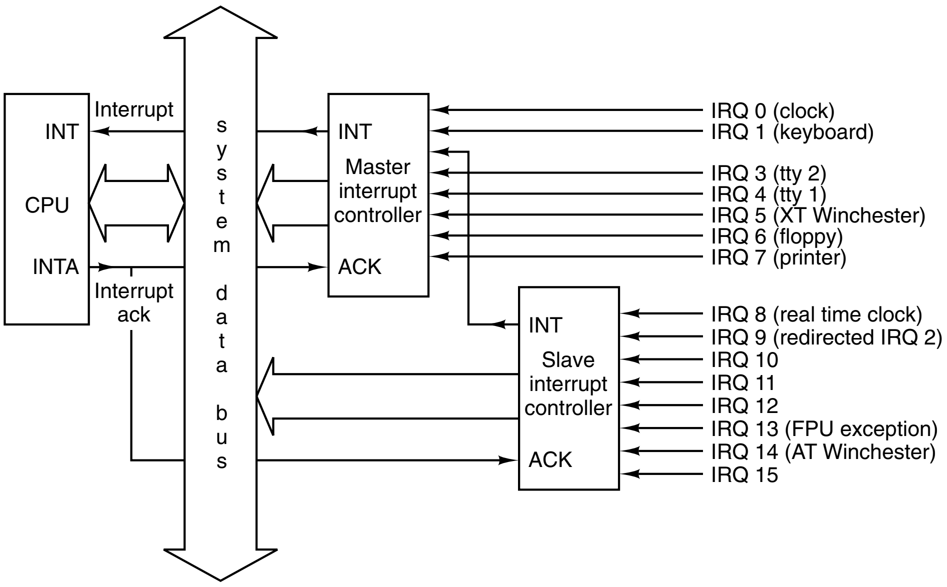

Figure 2-39. Interrupt processing hardware on a 32-bit Intel PC.

programmed during system initialization, when *main* calls *intr_init*. The programming determines the output sent to the CPU for a signal received on each of the input lines, as well as various other parameters of the controller's operation. The data put on the bus is an 8-bit number, used to index into a table of up to 256 elements. The MINIX 3 table has 56 elements. Of these, 35 are actually used; the others are reserved for use with future Intel processors or for future enhancements to MINIX 3. On 32-bit Intel processors this table contains interrupt gate descriptors, each of which is an 8-byte structure with several fields.

Several modes of response to interrupts are possible; in the one used by MINIX 3, the fields of most concern to us in each of the interrupt gate descriptors point to the service routine's executable code segment and the starting address within it. The CPU executes the code pointed to by the selected descriptor. The result is exactly the same as execution of an

```
int     <nnn>
```

assembly language instruction. The only difference is that in the case of a hardware interrupt the <nnn> originates from a register in the interrupt controller chip, rather than from an instruction in program memory.

The task-switching mechanism of a 32-bit Intel processor that is called into play in response to an interrupt is complex, and changing the program counter to execute another function is only a part of it. When the CPU receives an interrupt while running a process it sets up a new stack for use during the interrupt service.

The location of this stack is determined by an entry in the **Task State Segment** (TSS). One such structure exists for the entire system, initialized by *cstart*'s call to *prot_init*, and modified as each process is started. The effect is that the new stack created by an interrupt always starts at the end of the *stackframe_s* structure within the process table entry of the interrupted process. The CPU automatically pushes several key registers onto this new stack, including those necessary to reinstate the interrupted process' own stack and restore its program counter. When the interrupt handler code starts running, it uses this area in the process table as its stack, and much of the information needed to return to the interrupted process will have already been stored. The interrupt handler pushes the contents of additional registers, filling the stackframe, and then switches to a stack provided by the kernel while it does whatever must be done to service the interrupt.

Termination of an interrupt service routine is done by switching the stack from the kernel stack back to a stackframe in the process table (but not necessarily the same one that was created by the last interrupt), explicitly popping the additional registers, and executing an iretd (return from interrupt) instruction. Iretd restores the state that existed before an interrupt, restoring the registers that were pushed by the hardware and switching back to a stack that was in use before an interrupt. Thus an interrupt stops a process, and completion of the interrupt service restarts a process, possibly a different one from the one that was most recently stopped. Unlike the simpler interrupt mechanisms that are the usual subject of assembly language programming texts, nothing is stored on the interrupted process' working stack when a user process is interrupted. Furthermore, because the stack is created anew in a known location (determined by the TSS) after an interrupt, control of multiple processes is simplified. To start a different process all that is necessary is to point the stack pointer to the stackframe of another process, pop the registers that were explicitly pushed, and execute an iretd instruction.

The CPU disables all interrupts when it receives an interrupt. This guarantees that nothing can occur to cause the stackframe within a process table entry to overflow. This is automatic, but assembly-level instructions exist to disable and enable interrupts, as well. Interrupts remain disabled while the kernel stack, located outside the process table, is in use. A mechanism exists to allow an exception handler (a response to an error detected by the CPU) to run when the kernel stack is in use. An exception is similar to an interrupt and exceptions cannot be disabled. Thus, for the sake of exceptions there must be a way to deal with what are essentially nested interrupts. In this case a new stack is not created. Instead, the CPU pushes the essential registers needed for resumption of the interrupted code onto the existing stack. An exception is not supposed to occur while the kernel is running, however, and will result in a panic.

When an iretd is encountered while executing kernel code, a the return mechanism is simpler than the one used when a user process is interrupted. The processor can determine how to handle the iretd by examining the code segment selector that is popped from the stack as part of the iretd's action.

The privilege levels mentioned earlier control the different responses to interrupts received while a process is running and while kernel code (including interrupt service routines) is executing. The simpler mechanism is used when the privilege level of the interrupted code is the same as the privilege level of the code to be executed in response to the interrupt. The usual case, however, is that the interrupted code is less privileged than the interrupt service code, and in this case the more elaborate mechanism, using the TSS and a new stack, is employed. The privilege level of a code segment is recorded in the code segment selector, and as this is one of the items stacked during an interrupt, it can be examined upon return from the interrupt to determine what the iretd instruction must do.

Another service is provided by the hardware when a new stack is created to use while servicing an interrupt. The hardware checks to make sure the new stack is big enough for at least the minimum quantity of information that must be placed on it. This protects the more privileged kernel code from being accidentally (or maliciously) crashed by a user process making a system call with an inadequate stack. These mechanisms are built into the processor specifically for use in the implementation of operating systems that support multiple processes.

This behavior may be confusing if you are unfamiliar with the internal working of 32-bit Intel CPUs. Ordinarily we try to avoid describing such details, but understanding what happens when an interrupt occurs and when an iretd instruction is executed is essential to understanding how the kernel controls the transitions to and from the "running" state of Fig. 2-2. The fact that the hardware handles much of the work makes life much easier for the programmer, and presumably makes the resulting system more efficient. All this help from the hardware does, however, make it hard to understand what is happening just by reading the software.

Having now described the interrupt mechanism, we will return to *mpx386.s* and look at the tiny part of the MINIX 3 kernel that actually sees hardware interrupts. An entry point exists for each interrupt. The source code at each entry point, *_hwint00* to *_hwint07*, (lines 6531 to 6560) looks like a call to *hwint_master* (line 6515), and the entry points *_hwint08* to *_hwint15* (lines 6583 to 6612) look like calls to *hwint_slave* (line 6566). Each entry point appears to pass a parameter in the call, indicating which device needs service. In fact, these are really not calls, but macros, and eight separate copies of the code defined by the macro definition of *hwint_master* are assembled, with only the *irq* parameter different. Similarly, eight copies of the *hwint_slave* macro are assembled. This may seem extravagant, but assembled code is very compact. The object code for each expanded macro occupies fewer than 40 bytes. In servicing an interrupt, speed is important, and doing it this way eliminates the overhead of executing code to load a parameter, call a subroutine, and retrieve the parameter.

We will continue the discussion of *hwint_master* as if it really were a single function, rather than a macro that is expanded in eight different places. Recall that before *hwint_master* begins to execute, the CPU has created a new stack in the *stackframe_s* of the interrupted process, within its process table slot. Several key registers have already been saved there, and all interrupts are disabled. The first action of *hwint_master* is to call *save* (line 6516). This subroutine pushes all the other registers necessary to restart the interrupted process. *Save* could have been written inline as part of the macro to increase speed, but this would have more than doubled the size of the macro, and in any case *save* is needed for calls by other functions. As we shall see, *save* plays tricks with the stack. Upon returning to *hwint_master*, the kernel stack, not a stackframe in the process table, is in use.

Two tables declared in *glo.h* are now used. *_Irq_handlers* contains the hook information, including addresses of handler routines. The number of the interrupt being serviced is converted to an address within *_irq_handlers*. This address is then pushed onto the stack as the argument to *_intr_handle*, and *_intr_handle* is called, We will look at the code of *_intr_handle* later. For the moment, we will just say that not only does it call the service routine for the interrupt that was called, it sets or resets a flag in the *_irq_actids* array to indicate whether this attempt to service the interrupt succeeded, and it gives other entries on the queue another chance to run and be removed from the list. Depending upon exactly what was required of the handler, the IRQ may or may not be available to receive another interrupt upon the return from the call to *_intr_handle*. This is determined by checking the corresponding entry in *_irq_actids*.

A nonzero value in *_irq_actids* shows that interrupt service for this IRQ is not complete. If so, the interrupt controller is manipulated to prevent it from responding to another interrupt from the same IRQ line. (lines 6722 to 6724). This operation masks the ability of the controller chip to respond to a particular input; the CPU's ability to respond to all interrupts is inhibited internally when it first receives the interrupt signal and has not yet been restored at this point.

A few words about the assembly language code used may be helpful to readers unfamiliar with assembly language programming. The instruction

```
jz 0f
```

on line 6521 does not specify a number of bytes to jump over. The 0f is not a hexadecimal number, nor is it a normal label. Ordinary label names are not permitted to begin with numeric characters. This is the way the MINIX 3 assembler specifies a **local label**; the 0f means a jump **forward** to the next numeric label 0, on line 6525. The byte written on line 6526 allows the interrupt controller to resume normal operation, possibly with the line for the current interrupt disabled.

An interesting and possibly confusing point is that the 0: label on line 6525 occurs elsewhere in the same file, on line 6576 in *hwint_slave*. The situation is even more complicated than it looks at first glance since these labels are within macros and the macros are expanded before the assembler sees this code. Thus there are actually sixteen 0: labels in the code seen by the assembler. The possible proliferation of labels declared within macros is the reason why the assembly language provides local labels; when resolving a local label, the assembler uses the nearest one that matches in the specified direction, and additional occurrences of a local label are ignored.

_Intr_handle is hardware dependent, and details of its code will be discussed when we get to the file *i8259.c.* However, a few word about how it functions are in order now. *_Intr_handle* scans a linked list of structures that hold, among other things, addresses of functions to be called to handle an interrupt for a device, and the process numbers of the device drivers. It is a linked list because a single IRQ line may be shared with several devices. The handler for each device is supposed to test whether its device actually needs service. Of course, this step is not necessary for an IRQ such as the clock interrupt, IRQ 0, which is hard wired to the chip that generates clock signals with no possibility of any other device triggering this IRQ.

The handler code is intended to be written so it can return quickly. If there is no work to be done or the interrupt service is completed immediately, the handler returns *TRUE.* A handler may perform an operation like reading data from an input device and transferring the data to a buffer where it can be accessed when the corresponding driver has its next chance to run. The handler may then cause a message to be sent to its device driver, which in turn causes the device driver to be scheduled to run as a normal process. If the work is not complete, the handler returns *FALSE.* An element of the *_irq_act_ids* array is a bitmap that records the results for all the handlers on the list in such a way that the result will be zero if and only if every one of the handlers returned *TRUE.* If that is not the case, the code on lines 6522 to 6524 disables the IRQ before the interrupt controller as a whole is reenabled on line 6536.

This mechanism ensures that none of the handlers on the chain belonging to an IRQ will be activated until all of the device drivers to which these handlers belong have completed their work. Obviously, there needs to be another way to reenable an IRQ. That is provided in a function *enable_irq* which we will see later. Suffice it to say, each device driver must be sure that *enable_irq* is called when its work is done. It also is obvious that *enable_irq* first should reset its own bit in the element of *_irq_act_ids* that corresponds to the IRQ of the driver, and then should test whether all bits have been reset. Only then should the IRQ be reenabled on the interrupt controller chip.

What we have just described applies in its simplest form only to the clock driver, because the clock is the only interrupt-driven device that is compiled into the kernel binary. The address of an interrupt handler in another process is not meaningful in the context of the kernel, and the *enable_irq* function in the kernel cannot be called by a separate process in its own memory space. For user-space device drivers, which means all device drivers that respond to hardware-initiated interrupts except for the clock driver, the address of a common handler, *generic_handler*, is stored in the linked list of hooks. The source code for this function is in the system task files, but since the system task is compiled together with the kernel and since this code is executed in response to an interrupt it cannot really be considered part of the system task. The other information in each element of the list of hooks includes the process number of the associated device driver. When *generic_handler* is called it sends a message to the correct device driver which causes the specific handler functions of the driver to run. The system task supports the other end of the chain of events described above as well. When a user-space device driver completes its work it makes a sys_irqctl kernel call, which causes the system task to call *enable_irq* on behalf of that driver to prepare for the next interrupt.

Returning our attention to *hwint_master*, note that it terminates with a ret instruction (line 6527). It is not obvious that something tricky happens here. If a process has been interrupted, the stack in use at this point is the kernel stack, and not the stack within a process table that was set up by the hardware before *hwint_master* was started. In this case, manipulation of the stack by *save* will have left the address of *_restart* on the kernel stack. This results in a task, driver, server, or user process once again executing. It may not be, and in fact very likely is not, the same process as was executing when the interrupt occurred. This depends upon whether the processing of the message created by the device-specific interrupt service routine caused a change in the process scheduling queues. In the case of a hardware interrupt this will almost always be the case. Interrupt handlers usually result in messages to device drivers, and device drivers generally are queued on higher priority queues than user processes. This, then, is the heart of the mechanism which creates the illusion of multiple processes executing simultaneously.

To be complete, let us mention that if an interrupt could occur while kernel code were executing, the kernel stack would already be in use, and *save* would leave the address of *restart1* on the kernel stack. In this case, whatever the kernel was doing previously would continue after the ret at the end of *hwint_master*. This is a description of handling of nested interrupts, and these are not allowed to occur in MINIX 3— interrupts are not enabled while kernel code is running. However, as mentioned previously, the mechanism is necessary in order to handle exceptions. When all the kernel routines involved in responding to an exception are complete *_restart* will finally execute. In response to an exception while executing kernel code it will almost certainly be true that a process different from the one that was interrupted last will be put into execution. The response to an exception in the kernel is a panic, and what happens will be an attempt to shut down the system with as little damage as possible.

Hwint_slave (line 6566) is similar to *hwint_master*, except that it must reenable both the master and slave controllers, since both of them are disabled by receipt of an interrupt by the slave.

Now let us move on to look at *save* (line 6622), which we have already mentioned. Its name describes one of its functions, which is to save the context of the interrupted process on the stack provided by the CPU, which is a stackframe within the process table. *Save* uses the variable *_k_reenter* to count and determine the level of nesting of interrupts. If a process was executing when the current interrupt occurred, the

```
mov    esp, k_stktop
```

instruction on line 6635 switches to the kernel stack, and the following instruction pushes the address of *_restart*. If an interrupt could occur while the kernel stack were already in use the address of *restart1* would be pushed instead (line 6642). Of course, an interrupt is not allowed here, but the mechanism is here to handle exceptions. In either case, with a possibly different stack in use from the one that was in effect upon entry, and with the return address in the routine that called it buried beneath the registers that have just been pushed, an ordinary return instruction is not adequate for returning to the caller. The

```
jmp    RETADR-P_STACKBASE(eax)
```

instructions that terminate the two exit points of *save*, at line 6638 and line 6643 use the address that was pushed when *save* was called.

Reentrancy in the kernel causes many problems, and eliminating it resulted in simplification of code in several places. In MINIX 3 the *_k_reenter* variable still has a purpose—although ordinary interrupts cannot occur while kernel code is executing exceptions are still possible. For now, the thing to keep in mind is that the jump on line 6634 will never occur in normal operation. It is, however, necessary for dealing with exceptions.

As an aside, we must admit that the elimination of reentrancy is a case where programming got ahead of documentation in the development of MINIX 3. In some ways documentation is harder than programming—the compiler or the program will eventually reveal errors in a program. There is no such mechanism to correct comments in source code. There is a rather long comment at the start of *mpx386.s* which is, unfortunately, incorrect. The part of the comment on lines 6310 to 6315 should say that a kernel reentry can occur only when an exception is detected.

The next procedure in *mpx386.s* is *_s_call*, which begins on line 6649. Before looking at its internal details, look at how it ends. It does not end with a ret or jmp instruction. In fact, execution continues at *_restart* (line 6681). *_S_call* is the system call counterpart of the interrupt-handling mechanism. Control arrives at *_s_call* following a software interrupt, that is, execution of an int <nnn> instruction. Software interrupts are treated like hardware interrupts, except of course the index into the Interrupt Descriptor Table is encoded into the nnn part of an int <nnn> instruction, rather than being supplied by an interrupt controller chip. Thus, when *_s_call* is entered, the CPU has already switched to a stack inside the process table (supplied by the Task State Segment), and several registers have already been pushed onto this stack. By falling through to *_restart*, the call to *_s_call* ultimately terminates with an iretd instruction, and, just as with a hardware interrupt, this instruction will start whatever process is pointed to by *proc_ptr* at that point. Figure 2-40 compares the handling of a hardware interrupt and a system call using the software interrupt mechanism.

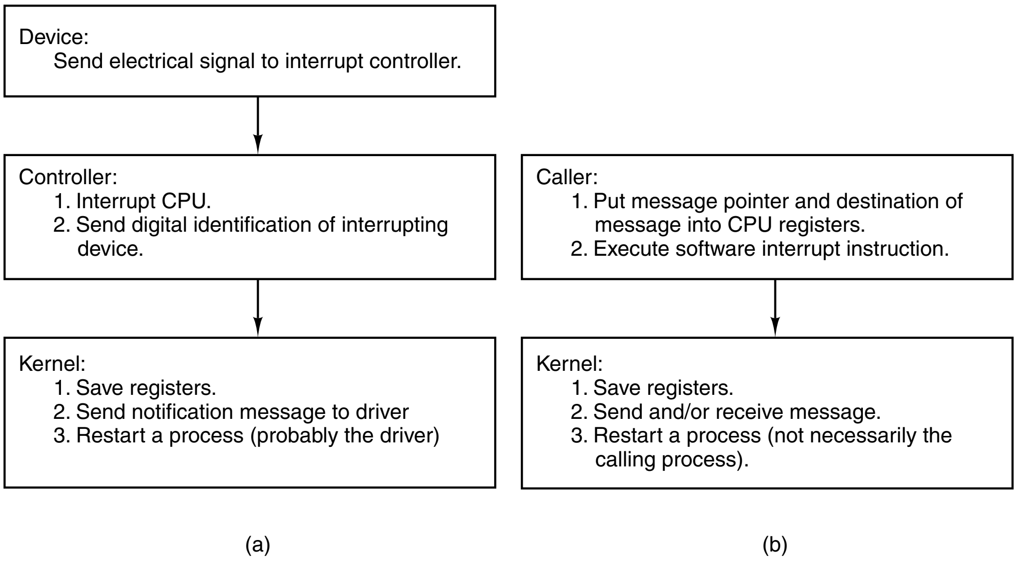
Device:
Send electrical signal to interrupt controller.
Controller:
1. Interrupt CPU.
2. Send digital identification of interrupting device.
Kernel:
1. Save registers.
2. Send notification message to driver
3. Restart a process (probably the driver)
Caller:
1. Put message pointer and destination of message into CPU registers.
2. Execute software interrupt instruction.
Kernel:
1. Save registers.
2. Send and/or receive message.
3. Restart a process (not necessarily the calling process).
(a)
(b)

Figure 2-40. (a) How a hardware interrupt is processed. (b) How a system call is made.

Let us now look at some details of *_s_call*. The alternate label, *_p_s_call*, is a vestige of the 16-bit version of MINIX 3, which has separate routines for protected mode and real mode operation. In the 32-bit version all calls to either label end up here. A programmer invoking a MINIX 3 system call writes a function call in C that looks like any other function call, whether to a locally defined function or to a routine in the C library. The library code supporting a system call sets up a message, loads the address of the message and the process id of the destination into CPU registers, and then invokes an `int SYS386_VECTOR` instruction. As described above, the result is that control passes to the start of *_s_call*, and several registers have already been pushed onto a stack inside the process table. All interrupts are disabled, too, as with a hardware interrupt.

The first part of the *_s_call* code resembles an inline expansion of *save* and saves the additional registers that must be preserved. Just as in *save*, a

```
mov    esp, k_stktop
```

instruction then switches to the kernel stack. (The similarity of a software interrupt to a hardware interrupt extends to both disabling all interrupts). Following this comes a call to *_sys_call* (line 6672), which we will discuss in the next section. For now we just say that it causes a message to be delivered, and that this in turn causes the scheduler to run. Thus, when *_sys_call* returns, it is probable that *proc_ptr* will be pointing to a different process from the one that initiated the system call. Then execution falls through to *restart*.

We have seen that *_restart* (line 6681) is reached in several ways:

1. By a call from *main* when the system starts.
2. By a jump from *hwint_master* or *hwint_slave* after a hardware interrupt.
3. By falling through from *_s_call* after a system call.

Fig. 2-41 is a simplified summary of how control passes back and forth between processes and the kernel via *_restart*.

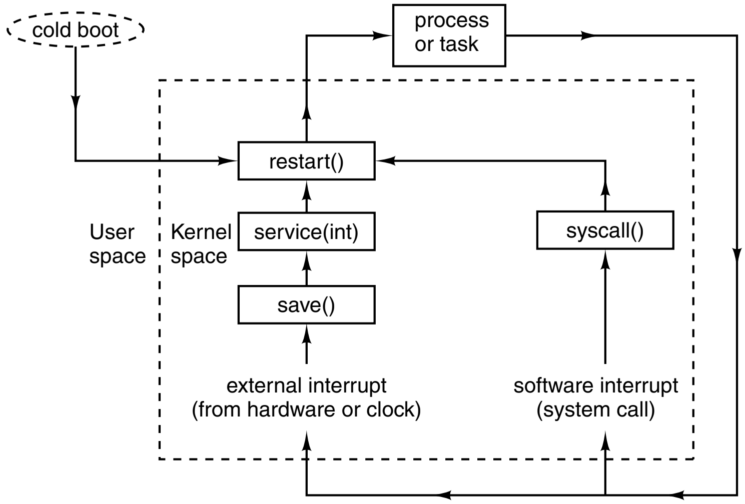

Figure 2-41. *Restart* is the common point reached after system startup, interrupts, or system calls. The most deserving process (which may be and often is a different process from the last one interrupted) runs next. Not shown in this diagram are interrupts that occur while the kernel itself is running.

In every case interrupts are disabled when *_restart* is reached. By line 6690 the next process to run has been definitively chosen, and with interrupts disabled it cannot be changed. The process table was carefully constructed so it begins with a stack frame, and the instruction on this line,

```
mov   esp, (_proc_ptr)
```

points the CPU's stack pointer at the stack frame. The

```
lldt    P_LDT_SEL(esp)
```

instruction then loads the processor's local descriptor table register from the stack frame. This prepares the processor to use the memory segments belonging to the next process to be run. The following instruction sets the address in the next process' process table entry to that where the stack for the next interrupt will be set up, and the following instruction stores this address into the TSS.

The first part of *_restart* would not be necessary if an interrupt occured when kernel code (including interrupt service code) were executing, since the kernel stack would be in use and termination of the interrupt service would allow the kernel code to continue. But, in fact, the kernel is not reentrant in MINIX 3, and ordinary interrupts cannot occur this way. However, disabling interrupts does not disable the ability of the processor to detect exceptions. The label *restart1* (line 6694) marks the point where execution resumes if an exception occurs while executing kernel code (something we hope will never happen). At this point *k_reenter* is decremented to record that one level of possibly nested interrupts has been disposed of, and the remaining instructions restore the processor to the state it was in when the next process executed last. The penultimate instruction modifies the stack pointer so the return address that was pushed when *save* was called is ignored. If the last interrupt occurred when a process was executing, the final instruction, iretd, completes the return to execution of whatever process is being allowed to run next, restoring its remaining registers, including its stack segment and stack pointer. If, however, this encounter with the iretd came via *restart1*, the kernel stack in use is not a stackframe, but the kernel stack, and this is not a return to an interrupted process, but the completion of handling an exception that occurred while kernel code was executing. The CPU detects this when the code segment descriptor is popped from the stack during execution of the iretd, and the complete action of the iretd in this case is to retain the kernel stack in use.

Now it is time to say something more about exceptions. An **exception** is caused by various error conditions internal to the CPU. Exceptions are not always bad. They can be used to stimulate the operating system to provide a service, such as providing more memory for a process to use, or swapping in a currently swapped-out memory page, although such services are not implemented in MINIX 3. They also can be caused by programming errors. Within the kernel an exception is very serious, and grounds to panic. When an exception occurs in a user program the program may need to be terminated, but the operating system should be able to continue. Exceptions are handled by the same mechanism as interrupts, using descriptors in the interrupt descriptor table. These entries in the table point to the sixteen exception handler entry points, beginning with *_divide_error* and ending with *_copr_error*, found near the end of *mpx386.s*, on lines 6707 to 6769. These all jump to *exception* (line 6774) or *errexception* (line 6785) depending upon whether the condition pushes an error code onto the stack or not. The handling here in the assembly code is similar to what we have already seen, registers are pushed and the C routine *_exception* (note the underscore) is called to handle the event. The consequences of exceptions vary. Some are ignored, some cause panics, and some result in sending signals to processes. We will examine *_exception* in a later section.

One other entry point is handled like an interrupt: *_level0_call* (line 6714). It is used when code must be run with privilege level 0, the most privileged level. The entry point is here in *mpx386.s* with the interrupt and exception entry points because it too is invoked by execution of an int <nnn> instruction. Like the exception routines, it calls *save*, and thus the code that is jumped to eventually will terminate with a ret that leads to *_restart*. Its usage will be described in a later section, when we encounter some code that needs privileges normally not available, even to the kernel.

Finally, some data storage space is reserved at the end of the assembly language file. Two different data segments are defined here. The

```
.sect .rom
```

declaration at line 6822 ensures that this storage space is allocated at the very beginning of the kernel's data segment and that it is the start of a read-only section of memory. The compiler puts a magic number here so *boot* can verify that the file it loads is a valid kernel image. When compiling the complete system various string constants will be stored following this. The other data storage area defined at the

```
.sect .bss
```

(line 6825) declaration reserves space in the kernel's normal uninitialized variable area for the kernel stack, and above that some space is reserved for variables used by the exception handlers. Servers and ordinary processes have stack space reserved when an executable file is linked and depend upon the kernel to properly set the stack segment descriptor and the stack pointer when they are executed. The kernel has to do this for itself.

2.6.9 Interprocess Communication in MINIX 3

Processes in MINIX 3 communicate by messages, using the rendezvous principle. When a process does a send, the lowest layer of the kernel checks to see if the destination is waiting for a message from the sender (or from ANY sender). If so, the message is copied from the sender's buffer to the receiver's buffer, and both processes are marked as runnable. If the destination is not waiting for a message from the sender, the sender is marked as blocked and put onto a queue of processes waiting to send to the receiver.

When a process does a receive, the kernel checks to see if any process is queued trying to send to it. If so, the message is copied from the blocked sender to the receiver, and both are marked as runnable. If no process is queued trying to send to it, the receiver blocks until a message arrives.

In MINIX 3, with components of the operating system running as totally separate processes, sometimes the rendezvous method is not quite good enough. The notify primitive is provided for precisely these occasions. A notify sends a barebones message. The sender is not blocked if the destination is not waiting for a message. The notify is not lost, however. The next time the destination does a receive pending notifications are delivered before ordinary messages. Notifications can be used in situations where using ordinary messages could cause deadlocks. Earlier we pointed out that a situation where process *A* blocks sending a message to process *B* and process *B* blocks sending a message to process *A* must be avoided. But if one of the messages is a nonblocking notification there is no problem.

In most cases a notification informs the recipient of its origin, and little more. Sometimes that is all that is needed, but there are two special cases where a notification conveys some additional information. In any case, the destination process can send a message to the source of the notification to request more information.

The high-level code for interprocess communication is found in *proc.c*. The kernel's job is to translate either a hardware interrupt or a software interrupt into a message. The former are generated by hardware and the latter are the way a request for system services, that is, a system call, is communicated to the kernel. These cases are similar enough that they could have been handled by a single function, but it was more efficient to create specialized functions.

One comment and two macro definitions near the beginning of this file deserve mention. For manipulating lists, pointers to pointers are used extensively, and a comment on lines 7420 to 7436 explains their advantages and use. Two useful macros are defined. *BuildMess* (lines 7458 to 7471), although its name implies more generality, is used only for constructing the messages used by notify. The only function call is to *get_uptime*, which reads a variable maintained by the clock task so the notification can include a timestamp. The apparent calls to a function named *priv* are expansions of another macro, defined in *priv.h*,

```
#define priv(rp)        ((rp)->p_priv)
```

The other macro, *CopyMess*, is a programmer-friendly interface to the assembly language routine *cp_mess* in *klib386.s*.

More should be said about *BuildMess*. The *priv* macro is used for two special cases. If the origin of a notification is *HARDWARE*, it carries a payload, a copy of the destination process' bitmap of pending interrupts. If the origin is *SYSTEM*, the payload is the bitmap of pending signals. Because these bitmaps are available in the *priv* table slot of the destination process, they can be accessed at any time. Notifications can be delivered later if the destination process is not blocked waiting for them at the time they are sent. For ordinary messages this would require some kind of buffer in which an undelivered message could be stored. To store a notification all that is required is a bitmap in which each bit corresponds to a process that can send a notification. When a notification cannot be sent the bit corresponding to the sender is set in the recipient's bitmap. When a receive is done the bitmap is checked and if a bit is found to have been set the message is regenerated. The bit tells the origin of the message, and if the origin is *HARDWARE* or

SYSTEM, the additional content is added. The only other item needed is the timestamp, which is added when the message is regenerated. For the purposes for which they are used, timestamps do not need to show when a notification was first attempted, the time of delivery is sufficient.

The first function in *proc.c* is *sys_call* (line 7480). It converts a software interrupt (the int SYS386_VECTOR instruction by which a system call is initiated) into a message. There are a wide range of possible sources and destinations, and the call may require either sending or receiving or both sending and receiving a message. A number of tests must be made. On lines 7480 and 7481 the function code *SEND*), *RECEIVE*, etc.,) and the flags are extracted from the first argument of the call. The first test is to see if the calling process is allowed to make the call. *Iskerneln*, used on line 7501, is a macro defined in proc.h (line 5584). The next test is to see that the specified source or destination is a valid process. Then a check is made that the message pointer points to a valid area of memory. MINIX 3 privileges define which other processes any given process is allowed to send to, and this is tested next (lines 7537 to 7541). Finally, a test is made to verify that the destination process is running and has not initiated a shutdown (lines 7543 to 7547). After all the tests have been passed one of the functions *mini_send*, *mini_receive*, or *mini_notify* is called to do the real work. If the function was *ECHO* the *CopyMess* macro is used, with identical source and destination. *ECHO* is meant only for testing, as mentioned earlier.

The errors tested for in *sys_call* are unlikely, but the tests are easily done, as ultimately they compile into code to perform comparisons of small integers. At this most basic level of the operating system testing for even the most unlikely errors is advisable. This code is likely to be executed many times each second during every second that the computer system on which it runs is active.

The functions *mini_send*, *mini_rec*, and *mini_notify* are the heart of the normal message passing mechanism of MINIX 3 and deserve careful study.

Mini_send (line 7591) has three parameters: the caller, the process to be sent to, and a pointer to the buffer where the message is. After all the tests performed by *sys_call*, only one more is necessary, which is to detect a send deadlock. The test on lines 7606 to 7610 verifies that the caller and destination are not trying to send to each other. The key test in *mini_send* is on lines 7615 and 7616. Here a check is made to see if the destination is blocked on a receive, as shown by the *RECEIVING* bit in the *p_rts_flags* field of its process table entry. If it is waiting, then the next question is: "Who is it waiting for?" If it is waiting for the sender, or for ANY, the *CopyMess* macro is used to copy the message and the receiver is unblocked by resetting its *RECEIVING* bit. Then *enqueue* is called to give the receiver an opportunity to run (line 7620).

If, on the other hand, the receiver is not blocked, or is blocked but waiting for a message from someone else, the code on lines 7623 to 7632 is executed to block and dequeue the sender. All processes wanting to send to a given destination are strung together on a linked list, with the destination's *p_callerq* field pointing to the process table entry of the process at the head of the queue. The example of Fig. 2-42(a) shows what happens when process 3 is unable to send to process 0. If process 4 is subsequently also unable to send to process 0, we get the situation of Fig. 2-42(b).

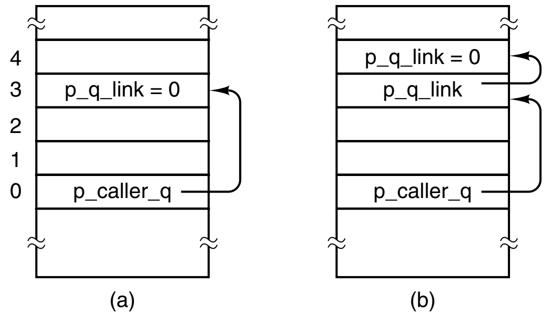

Figure 2-42. Queueing of processes trying to send to process 0.

Mini_receive (line 7642) is called by *sys_call* when its *function* parameter is *RECEIVE* or *BOTH*. As we mentioned earlier, notifications have a higher priority than ordinary messages. However, a notification will never be the right reply to a send, so the bitmaps are checked to see if there are pending notifications only if the *SENDREC_BUSY* flag is not set. If a notification is found it is marked as no longer pending and delivered (lines 7670 to 7685). Delivery uses both the *Build-Mess* and *CopyMess* macros defined near the top of *proc.c*.

One might have thought that, because a timestamp is part of a notify message, it would convey useful information, for instance, if the recipient had been unable to do a receive for a while the timestamp would tell how long it had been undelivered. But the notification message is generated (and timestamped) at the time it is delivered, not at the time it was sent. There is a purpose behind constructing the notification messages at the time of delivery, however. The code is unnecessary to save notification messages that cannot be delivered immediately. All that is necessary is to set a bit to remember that a notification should be generated when delivery becomes possible. You cannot get more economical storage than that: one bit per pending notification.

It is also the case that the current time is usually what is needed. For instance, notification is used to deliver a *SYN_ALARM* message to the process manager, and if the timestamp were not generated when the message was delivered the PM would need to ask the kernel for the correct time before checking its timer queue.

Note that only one notification is delivered at a time, *mini_send* returns on line 7684 after delivery of a notification. But the caller is not blocked, so it is free to do another receive immediately after getting the notification. If there are no notifications, the caller queues are checked to see if a message of any other type is pending (lines 7690 to 7699. If such a message is found it is delivered by the

CopyMess macro and the originator of the message is then unblocked by the call to *enqueue* on line 7694. The caller is not blocked in this case.

If no notifications or other messages were available, the caller will be blocked by the call to *dequeue* on line 7708.

Mini_notify (line 7719) is used to effectuate a notification. It is similar to *mini_send*, and can be discussed quickly. If the recipient of a message is blocked and waiting to receive, the notification is generated by *BuildMess* and delivered. The recipient's *RECEIVING* flag is turned off and it is then *enqueue*-ed (lines 7738 to 7743). If the recipient is not waiting a bit is set in its *s_notify_pending* map, which indicates that a notification is pending and identifies the sender. The sender then continues its own work, and if another notification to the same recipient is needed before an earlier one has been received, the bit in the recipient's bitmap is overwritten—effectively, multiple notifications from the same sender are merged into a single notification message. This design eliminates the need for buffer management while offering asynchronous message passing.

When *mini_notify* is called because of a software interrupt and a subsequent call to *sys_call*, interrupts will be disabled at the time. But the clock or system task, or some other task that might be added to MINIX 3 in the future might need to send a notification at a time when interrupts are not disabled. *Lock_notify* (line 7758) is a safe gateway to *mini_notify*. It checks *k_reenter* to see if interrupts are already disabled, and if they are, it just calls *mini_notify* right away. If interrupts are enabled they are disabled by a call to *lock*, *mini_notify* is called, and then interrupts are reenabled by a call to *unlock*.

2.6.10 Scheduling in MINIX 3

MINIX 3 uses a multilevel scheduling algorithm. Processes are given initial priorities that are related to the structure shown in Fig. 2-29, but there are more layers and the priority of a process may change during its execution. The clock and system tasks in layer 1 of Fig. 2-29 receive the highest priority. The device drivers of layer 2 get lower priority, but they are not all equal. Server processes in layer 3 get lower priorities than drivers, but some less than others. User processes start with less priority than any of the system processes, and initially are all equal, but the *nice* command can raise or lower the priority of a user process.

The scheduler maintains 16 queues of runnable processes, although not all of them may be used at a particular moment. Fig. 2-43 shows the queues and the processes that are in place at the instant the kernel completes initialization and begins to run, that is, at the call to *restart* at line 7252 in *main.c*. The array *rdy_head* has one entry for each queue, with that entry pointing to the process at the head of the queue. Similarly, *rdy_tail* is an array whose entries point to the last process on each queue. Both of these arrays are defined with the *EXTERN* macro in *proc.h* (lines 5595 and 5596). The initial queueing of processes during system startup is determined by the *image* table in *table.c* (lines 6095 to 6109).

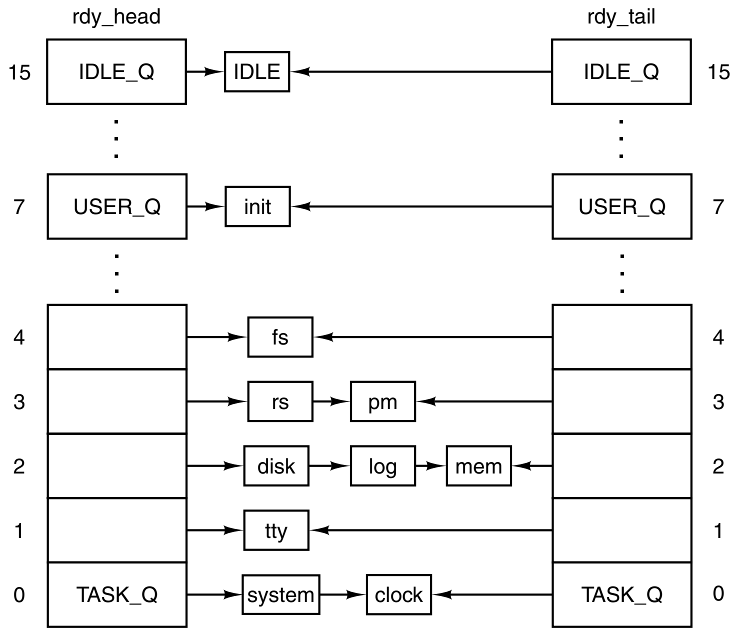

Figure 2-43. The scheduler maintains sixteen queues, one per priority level. Shown here is the initial queuing of processes as MINIX 3 starts up.

Scheduling is round robin in each queue. If a running process uses up its quantum it is moved to the tail of its queue and given a new quantum. However, when a blocked process is awakened, it is put at the head of its queue if it had any part of its quantum left when it blocked. It is not given a complete new quantum, however; it gets only what it had left when it blocked. The existence of the array *rdy_tail* makes adding a process to the end of a queue efficient. Whenever a running process becomes blocked, or a runnable process is killed by a signal, that process is removed from the scheduler's queues. Only runnable processes are queued.

Given the queue structures just described, the scheduling algorithm is simple: find the highest priority queue that is not empty and pick the process at the head of that queue. The *IDLE* process is always ready, and is in the lowest priority queue. If all the higher priority queues are empty, *IDLE* is run.

We saw a number of references to *enqueue* and *dequeue* in the last section. Now let us look at them. *Enqueue* is called with a pointer to a process table entry as its argument (line 7787). It calls another function, *sched*, with pointers to variables that determine which queue the process should be on and whether it is to be added to the head or the tail of that queue. Now there are three possibilities. These are classic data structures examples. If the chosen queue is empty, both *rdy_head* and *rdy_tail* are made to point to the process being added, and the link field, *p_nextready*, gets the special pointer value that indicates nothing follows, *NIL_PROC*. If the process is being added to the head of a queue, its *p_nextready* gets the current value of *rdy_head*, and then *rdy_head* is pointed to the new process. If the process is being added to the tail of a queue, the *p_nextready* of the current occupant of the tail is pointed to the new process, as is *rdy_tail*. The *p_nextready* of the newly-ready process then is pointed to *NIL_PROC*. Finally, *pick_proc* is called to determine which process will run next.

When a process must be made unready *dequeue* line 7823 is called. A process must be running in order to block, so the process to be removed is likely to be at the head of its queue. However, a signal could have been sent to a process that was not running. So the queue is traversed to find the victim, with a high likelihood it will be found at the head. When it is found all pointers are adjusted appropriately to take it out of the chain. If it was running, *pick_proc* must also be called.

One other point of interest is found in this function. Because tasks that run in the kernel share a common hardware-defined stack area, it is a good idea to check the integrity of their stack areas occasionally. At the beginning of *dequeue* a test is made to see if the process being removed from the queue is one that operates in kernel space. If it is, a check is made to see that the distinctive pattern written at the end of its stack area has not been overwritten (lines 7835 to 7838).

Now we come to *sched*, which picks which queue to put a newly-ready process on, and whether to put it on the head or the tail of that queue. Recorded in the process table for each process are its quantum, the time left on its quantum, its priority, and the maximum priority it is allowed. On lines 7880 to 7885 a check is made to see if the entire quantum was used. If not, it will be restarted with whatever it had left from its last turn. If the quantum was used up, then a check is made to see if the process had two turns in a row, with no other process having run. This is taken as a sign of a possible infinite, or at least, excessively long, loop, and a penalty of +1 is assigned. However, if the entire quantum was used but other processes have had a chance to run, the penalty value becomes −1. Of course, this does not help if two or more processes are executing in a loop together. How to detect that is an open problem.

Next the queue to use is determined. Queue 0 is highest priority; queue 15 is lowest. One could argue it should be the other way around, but this way is consistent with the traditional "nice" values used by UNIX, where a positive "nice" means a process runs with lower priority. Kernel processes (the clock and system tasks) are immune, but all other processes may have their priority reduced, that is, be moved to a higher-numbered queue, by adding a positive penalty. All processes start with their maximum priority, so a negative penalty does not change anything until positive penalties have been assigned. There is also a lower bound on priority, ordinary processes never can be put on the same queue as *IDLE*.

Now we come to *pick_proc* (line 7910). This function's major job is to set *next_ptr*. Any change to the queues that might affect the choice of which process to run next requires *pick_proc* to be called again. Whenever the current process blocks, *pick_proc* is called to reschedule the CPU. In essence, *pick_proc* is the scheduler.

Pick_proc is simple. Each queue is tested. *TASK_Q* is tested first, and if a process on this queue is ready, *pick_proc* sets *proc_ptr* and returns immediately. Otherwise, the next lower priority queue is tested, all the way down to *IDLE_Q*. The pointer *bill_ptr* is changed to charge the user process for the CPU time it is about to be given (line 7694). This assures that the last user process to run is charged for work done on its behalf by the system.

The remaining procedures in *proc.c* are *lock_send*, *lock_enqueue*, and *lock_dequeue*. These all provide access to their basic functions using *lock* and *unlock*, in the same way we discussed for *lock_notify*.

In summary, the scheduling algorithm maintains multiple priority queues. The first process on the highest priority queue is always run next. The clock task monitors the time used by all processes. If a user process uses up its quantum, it is put at the end of its queue, thus achieving a simple round-robin scheduling among the competing user processes. Tasks, drivers, and servers are expected to run until they block, and are given large quanta, but if they run too long they may also be preempted. This is not expected to happen very often, but it is a mechanism to prevent a high-priority process with a problem from locking up the system. A process that prevents other processes from running may also be moved to a lower priority queue temporarily.

2.6.11 Hardware-Dependent Kernel Support

Several functions written in C are nevertheless hardware specific. To facilitate porting MINIX 3 to other systems these functions are segregated in the files to be discussed in this section, *exception.c*, *i8259.c*, and *protect.c*, rather than being included in the same files with the higher-level code they support.

Exception.c contains the exception handler, *exception* (line 8012), which is called (as *_exception*) by the assembly language part of the exception handling code in *mpx386.s*. Exceptions that originate from user processes are converted to signals. Users are expected to make mistakes in their own programs, but an exception originating in the operating system indicates something is seriously wrong and causes a panic. The array *ex_data* (lines 8022 to 8040) determines the error message to be printed in case of panic, or the signal to be sent to a user process for each exception. Earlier Intel processors do not generate all the exceptions, and the third field in each entry indicates the minimum processor model that is capable of generating each one. This array provides an interesting summary of the evolution of the Intel family of processors upon which MINIX 3 has been implemented. On line 8065 an alternate message is printed if a panic results from an interrupt that would not be expected from the processor in use.

Hardware-Dependent Interrupt Support

The three functions in *i8259.c* are used during system initialization to initialize the Intel 8259 interrupt controller chips. The macro on line 8119 defines a dummy function (the real one is needed only when MINIX 3 is compiled for a 16-bit Intel platform). *Intr_init* (line 8124) initializes the controllers. Two steps ensure that no interrupts will occur before all the initialization is complete. First *intr_disable* is called at line 8134. This is a C language call to an assembly language function in the library that executes a single instruction, cli, which disables the CPU's response to interrupts. Then a sequence of bytes is written to registers on each interrupt controller, the effect of which is to inhibit response of the controllers to external input. The byte written at line 8145 is all ones, except for a zero at the bit that controls the cascade input from the slave controller to the master controller (see Fig. 2-39). A zero enables an input, a one disables. The byte written to the secondary controller at line 8151 is all ones.

A table stored in the i8259 interrupt controller chip generates an 8-bit index that the CPU uses to find the correct interrupt gate descriptor for each possible interrupt input (the signals on the right-hand side of Fig. 2-39). This is initialized by the BIOS when the computer starts up, and these values can almost all be left in place. As drivers that need interrupts start up, changes can be made where necessary. Each driver can then request that a bit be reset in the interrupt controller chip to enable its own interrupt input. The argument *mine* to *intr_init* is used to determine whether MINIX 3 is starting up or shutting down. This function can be used both to initialize at startup and to restore the BIOS settings when MINIX 3 shuts down.

After initialization of the hardware is complete, the last step in *intr_init* is to copy the BIOS interrupt vectors to the MINIX 3 vector table.

The second function in *8259.c* is *put_irq_handler* (line 8162). At initialization *put_irq_handler* is called for each process that must respond to an interrupt. This puts the address of the handler routine into the interrupt table, *irq_handlers*, defined as *EXTERN* in *glo.h*. With modern computers 15 interrupt lines is not always enough (because there may be more than 15 I/O devices) so two I/O devices may need to share an interrupt line. This will not occur with any of the basic devices supported by MINIX 3 as described in this text, but when network interfaces, sound cards, or more esoteric I/O devices must be supported they may need to share interrupt lines. To allow for this, the interrupt table is not just a table of addresses. *Irq_handlers[NR_IRQ_VECTORS]* is an array of pointers to *irq_hook* structs, a type defined in *kernel/type.h*. These structures contain a field which is a pointer to another structure of the same type, so a linked list can be built, starting with one of the elements of *irq_handlers*. *Put_irq_handler* adds an entry to one of these lists. The most important element of such an entry is a pointer to an **interrupt handler**, the function to be executed when an interrupt is generated, for example, when requested I/O has completed.

Some details of *put_irq_handler* deserve mention. Note the variable *id* which is set to 1 just before the beginning of the while loop that scans through the linked list (lines 8176 to 8180). Each time through the loop *id* is shifted left 1 bit. The test on line 8181 limits the length of the chain to the size of *id*, or 32 handlers for a 32-bit system. In the normal case the scan will result in finding the end of the chain, where a new handler can be linked. When this is done, *id* is also stored in the field of the same name in the new item on the chain. *Put_irq_handler* also sets a bit in the global variable *irq_use*, to record that a handler exists for this IRQ.

If you fully understand the MINIX 3 design goal of putting device drivers in user-space, the preceding discussion of how interrupt handlers are called will have left you slightly confused. The interrupt handler addresses stored in the hook structures cannot be useful unless they point to functions within the kernel's address space. The only interrupt-driven device in the kernel's address space is the clock. What about device drivers that have their own address spaces?

The answer is, the system task handles it. Indeed, that is the answer to most questions regarding communication between the kernel and processes in user-space. A user space device driver that is to be interrupt driven makes a sys_irqctl call to the system task when it needs to register as an interrupt handler. The system task then calls *put_irq_handler*, but instead of the address of an interrupt handler in the driver's address space, the address of *generic_handler*, part of the system task, is stored in the interrupt handler field. The process number field in the hook structure is used by *generic_handler* to locate the *priv* table entry for the driver, and the bit in the driver's pending interrupts bitmap corresponding to the interrupt is set. Then *generic_handler* sends a notification to the driver. The notification is identified as being from *HARDWARE*, and the pending interrupts bitmap for the driver is included in the message. Thus, if a driver must respond to interrupts from more than one source, it can learn which one is responsible for the current notification. In fact, since the bitmap is sent, one notification provides information on all pending interrupts for the driver. Another field in the hook structure is a policy field, which determines whether the interrupt is to be reenabled immediately, or whether it should remain disabled. In the latter case, it will be up to the driver to make a sys_irqenable kernel call when service of the current interrupt is complete.

One of the goals of MINIX 3 design is to support run-time reconfiguration of I/O devices. The next function, *rm_irq_handler*, removes a handler, a necessary step if a device driver is to be removed and possibly replaced by another. Its action is just the opposite of *put_irq_handler*.

The last function in this file, *intr_handle* (line 8221), is called from the *hwint_master* and *hwint_slave* macros we saw in *mpx386.s*. The element of the array of bitmaps *irq_actids* which corresponds the interrupt being serviced is used to keep track of the current status of each handler in a list. For each function in the list, *intr_handle* sets the corresponding bit in *irq_actids*, and calls the handler.

If a handler has nothing to do or if it completes its work immediately, it returns "true" and the corresponding bit in *irq_actids* is cleared. The entire bitmap for an interrupt, considered as an integer, is tested near the end of the *hwint_master* and *hwint_slave* macros to determine if that interrupt can be reenabled before another process is restarted.

Intel Protected Mode Support

Protect.c contains routines related to protected mode operation of Intel processors. The **Global Descriptor Table** (GDT), **Local Descriptor Tables** (LDTs), and the **Interrupt Descriptor Table**, all located in memory, provide protected access to system resources. The **GDT** and **IDT** are pointed to by special registers within the CPU, and GDT entries point to **LDT**s. The GDT is available to all processes and holds segment descriptors for memory regions used by the operating system. Normally, there is one LDT for each process, holding segment descriptors for the memory regions used by the process. Descriptors are 8-byte structures with a number of components, but the most important parts of a segment descriptor are the fields that describe the base address and the limit of a memory region. The IDT is also composed of 8-byte descriptors, with the most important part being the address of the code to be executed when the corresponding interrupt is activated.

Cstart in *start.c* calls *prot_init* (line 8368), which sets up the GDT on lines 8421 to 8438. The IBM PC BIOS requires that it be ordered in a certain way, and all the indices into it are defined in *protect.h*. Space for an LDT for each process is allocated in the process table. Each contains two descriptors, for a code segment and a data segment—recall we are discussing here segments as defined by the hardware; these are not the same as the segments managed by the operating system, which considers the hardware-defined data segment to be further divided into data and stack segments. On lines 8444 to 8450 descriptors for each LDT are built in the GDT. The functions *init_dataseg* and *init_codeseg* build these descriptors. The entries in the LDTs themselves are initialized when a process' memory map is changed (i.e., when an exec system call is made).

Another processor data structure that needs initialization is the **Task State Segment** (**TSS**). The structure is defined at the start of this file (lines 8325 to 8354) and provides space for storage of processor registers and other information that must be saved when a task switch is made. MINIX 3 uses only the fields that define where a new stack is to be built when an interrupt occurs. The call to *init_dataseg* on line 8460 ensures that it can be located using the GDT.

To understand how MINIX 3 works at the lowest level, perhaps the most important thing is to understand how exceptions, hardware interrupts, or int <nnn> instructions lead to the execution of the various pieces of code that has been written to service them. These events are processed by means of the interrupt gate descriptor table. The array *gate_table* (lines 8383 to 8418), is initialized by the compiler with the addresses of the routines that handle exceptions and hardware interrupts and then is used in the loop at lines 8464 to 8468 to initialize this table, using calls to the *int_gate* function.

There are good reasons for the way the data are structured in the descriptors, based on details of the hardware and the need to maintain compatibility between advanced processors and the 16-bit 286 processor. Fortunately, we can usually leave these details to Intel's processor designers. For the most part, the C language allows us to avoid the details. However, in implementing a real operating system the details must be faced at some point. Figure 2-44 shows the internal structure of one kind of segment descriptor. Note that the base address, which C programs can refer to as a simple 32-bit unsigned integer, is split into three parts, two of which are separated by a number of 1-, 2-, and 4-bit quantities. The limit is a 20-bit quantity stored as separate 16-bit and 4-bit chunks. The limit is interpreted as either a number of bytes or a number of 4096-byte pages, based on the value of the *G* (granularity) bit. Other descriptors, such as those used to specify how interrupts are handled, have different, but equally complex structures. We discuss these structures in more detail in Chap. 4.

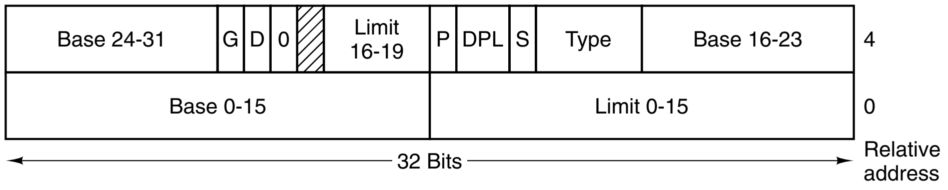

Figure 2-44. The format of an Intel segment descriptor.

Most of the other functions defined in *protect.c* are devoted to converting between variables used in C programs and the rather ugly forms these data take in the machine readable descriptors such as the one in Fig. 2-44. *Init_codeseg* (line 8477) and *init_dataseg* (line 8493) are similar in operation and are used to convert the parameters passed to them into segment descriptors. They each, in turn, call the next function, *sdesc* (line 8508), to complete the job. This is where the messy details of the structure shown in Fig. 2-44 are dealt with. *Init_codeseg* and *init_data_seg* are not used just at system initialization. They are also called by the system task whenever a new process is started up, in order to allocate the proper memory segments for the process to use. *Seg2phys* (line 8533), called only from *start.c*, performs an operation which is the inverse of that of *sdesc,* extracting the base address of a segment from a segment descriptor. *Phys2seg* (line 8556), is no longer needed, the sys_segctl kernel call now handles access to remote memory segments, for instance, memory in the PC's reserved area between 640K and 1M. *Int_gate* (line 8571) performs a similar function to *init_codeseg* and *init_dataseg* in building entries for the interrupt descriptor table.

Now we come to a function in *protect.c*, *enable_iop* (line 8589), that can perform a dirty trick. It changes the privilege level for I/O operations, allowing the current process to execute instructions which read and write I/O ports. The description of the purpose of the function is more complicated than the function itself, which just sets two bits in the word in the stack frame entry of the calling process that will be loaded into the CPU status register when the process is next executed. A function to undo this is not needed, as it will apply only to the calling process. This function is not currently used and no method is provided for a user space function to activate it.

The final function in *protect.c* is *alloc_segments* (line 8603). It is called by *do_newmap*. It is also called by the *main* routine of the kernel during initialization. This definition is very hardware dependent. It takes the segment assignments that are recorded in a process table entry and manipulates the registers and descriptors the Pentium processor uses to support protected segments at the hardware level. Multiple assignments like those on lines 8629 to 8633 are a feature of the C language.

2.6.12 Utilities and the Kernel Library

Finally, the kernel has a library of support functions written in assembly language that are included by compiling *klib.s* and a few utility programs, written in C, in the file *misc.c*. Let us first look at the assembly language files. *Klib.s* (line 8700) is a short file similar to *mpx.s*, which selects the appropriate machine-specific version based upon the definition of *WORD_SIZE*. The code we will discuss is in *klib386.s* (line 8800). This contains about two dozen utility routines that are in assembly code, either for efficiency or because they cannot be written in C at all.

_Monitor (line 8844) makes it possible to return to the boot monitor. From the point of view of the boot monitor, all of MINIX 3 is just a subroutine, and when MINIX 3 is started, a return address to the monitor is left on the monitor's stack. *_Monitor* just has to restore the various segment selectors and the stack pointer that was saved when MINIX 3 was started, and then return as from any other subroutine.

Int86 (line 8864) supports BIOS calls. The BIOS is used to provide alternative disk drivers which are not described here. *Int86* transfers control to the boot monitor, which manages a transfer from protected mode to real mode to execute a BIOS call, then back to protected mode for the return to 32-bit MINIX 3. The boot monitor also returns the number of clock ticks counted during the BIOS call. How this is used will be seen in the discussion of the clock task.

Although *_phys_copy* (see below) could have been used for copying messages, *_cp_mess* (line 8952), a faster specialized procedure, has been provided for that purpose. It is called by

```
cp_mess(source, src_clicks, src_offset, dest_clicks, dest_offset);
```

where *source* is the sender's process number, which is copied into the *m_source* field of the receiver's buffer. Both the source and destination addresses are specified by giving a click number, typically the base of the segment containing the buffer, and an offset from that click. This form of specifying the source and destination is more efficient than the 32-bit addresses used by *_phys_copy*.

_Exit, *__exit*, and *___exit* (lines 9006 to 9008) are defined because some library routines that might be used in compiling MINIX 3 make calls to the standard C function *exit*. An exit from the kernel is not a meaningful concept; there is nowhere to go. Consequently, the standard *exit* cannot be used here. The solution here is to enable interrupts and enter an endless loop. Eventually, an I/O operation or the clock will cause an interrupt and normal system operation will resume. The entry point for *___main* (line 9012) is another attempt to deal with a compiler action which, while it might make sense while compiling a user program, does not have any purpose in the kernel. It points to an assembly language `ret` (return from subroutine) instruction.

_Phys_insw (line 9022), *_phys_insb* (line 9047), *_phys_outsw* (line 9072), and *_phys_outsb* (line 9098), provide access to I/O ports, which on Intel hardware occupy a separate address space from memory and use different instructions from memory reads and writes. The I/O instructions used here, `ins`, `insb`, `outs`, and `outsb`, are designed to work efficiently with arrays (strings), and either 16-bit words or 8-bit bytes. The additional instructions in each function set up all the parameters needed to move a given number of bytes or words between a buffer, addressed physically, and a port. This method provides the speed needed to service disks, which must be serviced more rapidly than could be done with simpler byte- or word-at-a-time I/O operations.

A single machine instruction can enable or disable the CPU's response to all interrupts. *_Enable_irq* (line 9126) and *_disable_irq* (line 9162) are more complicated. They work at the level of the interrupt controller chips to enable and disable individual hardware interrupts.

_Phys_copy (line 9204) is called in C by

```
phys_copy(source_address, destination_address, bytes);
```

and copies a block of data from anywhere in physical memory to anywhere else. Both addresses are absolute, that is, address 0 really means the first byte in the entire address space, and all three parameters are unsigned longs.

For security, all memory to be used by a program should be wiped clean of any data remaining from a program that previously occupied that memory. This is done by the MINIX 3 `exec` call, ultimately using the next function in *klib386.s*, *phys_memset* (line 9248).

The next two short functions are specific to Intel processors. *_Mem_rdw* (line 9291) returns a 16-bit word from anywhere in memory. The result is zero-extended into the 32-bit *eax* register. The *_reset* function (line 9307) resets the processor. It does this by loading the processor's interrupt descriptor table register with a null pointer and then executing a software interrupt. This has the same effect as a hardware reset.

The *idle_task* (line 9318) is called when there is nothing else to do. It is written as an endless loop, but it is not just a busy loop (which could have been used to have the same effect). *Idle_task* takes advantage of the availability of a hlt instruction, which puts the processor into a power-conserving mode until an interrupt is received. However, hlt is a privileged instruction and executing hlt when the current privilege level is not 0 will cause an exception. So *idle_task* pushes the address of a subroutine containing a hlt and then calls *level0* (line 9322). This function retrieves the address of the *halt* subroutine, and copies it to a reserved storage area (declared in *glo.h* and actually reserved in *table.c*).

_Level0 treats whatever address is preloaded to this area as the functional part of an interrupt service routine to be run with the most privileged permission level, level zero.

The last two functions are *read_tsc* and *read_flags*. The former reads a CPU register which executes an assembly language instruction known as rdtsc, read time stamp counter. This counts CPU cycles and is intended for benchmarking or debugging. This instruction is not supported by the MINIX 3 assembler, and is generated by coding the opcode in hexadecimal. Finally, *read_flags* reads the processor flags and returns them as a C variable. The programmer was tired and the comment about the purpose of this function is incorrect.

The last file we will consider in this chapter is *utility.c* which provides three important functions. When something goes really, really wrong in the kernel, *panic* (line 9429) is invoked. It prints a message and calls *prepare_shutdown.* When the kernel needs to print a message it cannot use the standard library *printf*, so a special *kprintf* is defined here (line 9450). The full range of formatting options available in the library version are not needed here, but much of the functionality is available. Because the kernel cannot use the file system to access a file or a device, it passes each character to another function, *kputc* (line 9525), which appends each character to a buffer. Later, when *kputc* receives the *END_OF_KMESS* code it informs the process which handles such messages. This is defined in *include/minix/config.h*, and can be either the log driver or the console driver. If it is the log driver the message will be passed on to the console as well.

2.7 THE SYSTEM TASK IN MINIX 3

A consequence of making major system components independent processes outside the kernel is that they are forbidden from doing actual I/O, manipulating kernel tables and doing other things operating system functions normally do. For example, the fork system call is handled by the process manager. When a new process is created, the kernel must know about it, in order to schedule it. How can the process manager tell the kernel?

The solution to this problem is to have a kernel offer a set of services to the drivers and servers. These services, which are not available to ordinary user processes, allow the drivers and servers to do actual I/O, access kernel tables, and do other things they need to, all without being inside the kernel.

These special services are handled by the **system task**, which is shown in layer 1 in Fig. 2-29. Although it is compiled into the kernel binary program, it is really a separate process and is scheduled as such. The job of the system task is to accept all the requests for special kernel services from the drivers and servers and carry them out. Since the system task is part of the kernel's address space, it makes sense to study it here.

Earlier in this chapter we saw an example of a service provided by the system task. In the discussion of interrupt handling we described how a user-space device driver uses sys_irqctl to send a message to the system task to ask for installation of an interrupt handler. A user-space driver cannot access the kernel data structure where addresses of interrupt service routines are placed, but the system task is able to do this. Furthermore, since the interrupt service routine must also be in the kernel's address space, the address stored is the address of a function provided by the system task, *generic_handler*. This function responds to an interrupt by sending a notification message to the device driver.

This is a good place to clarify some terminology. In a conventional operating system with a monolithic kernel, the term **system call** is used to refer to all calls for services provided by the kernel. In a modern UNIX-like operating system the POSIX standard describes the system calls available to processes. There may be some nonstandard extensions to POSIX, of course, and a programmer taking advantage of a system call will generally reference a function defined in the C libraries, which may provide an easy-to-use programming interface. Also, sometimes separate library functions that appear to the programmer to be distinct "system calls" actually use the same access to the kernel.

In MINIX 3 the landscape is different; components of the operating system run in user space, although they have special privileges as system processes. We will still use the name "system call" for any of the POSIX-defined system calls (and a few MINIX extensions) listed in Fig. 1-9, but user processes do not request services directly of the kernel. In MINIX 3 system calls by user processes are transformed into messages to server processes. Server processes communicate with each other, with device drivers, and with the kernel by messages. The subject of this section, the system task, receives all requests for kernel services. Loosely speaking, we could call these requests system calls, but to be more exact we will refer to them as **kernel calls**. Kernel calls cannot be made by user processes. In many cases a system call that originates with a user process results in a kernel call with a similar name being made by a server. This is always because some part of the service being requested can only be dealt with by the kernel. For instance a fork system call by a user process goes to the process manager, which does some of the work. But a fork requires changes in the kernel part of the process table, and to complete the action the process manager makes a sys_fork call to the system task, which can manipulate data in kernel space. Not all kernel calls have such a clear connection to a single system call. For instance, there is a sys_devio kernel call to read or write I/O ports. This kernel call comes from a device driver. More than half of all the system calls listed in Fig. 1-9 could result in a device driver being activated and making one or more sys_devio calls.

Technically speaking, a third category of calls (besides system calls and kernel calls) should be distinguished. The **message primitives** used for interprocess communication such as send, receive, and notify can be thought of as system-call-like. We have probably called them that in various places in this book—after all, they do call the system. But they should properly be called something different from both system calls and kernel calls. Other terms may be used. **IPC primitive** is sometimes used, as well as **trap**, and both of these may be found in some comments in the source code. You can think of a message primitive as being like the carrier wave in a radio communications system. Modulation is usually needed to make a radio wave useful; the message type and other components of a message structure allow the message call to convey information. In a few cases an unmodulated radio wave is useful; for instance, a radio beacon to guide airplanes to an airport. This is analogous to the notify message primitive, which conveys little information other than its origin.

2.7.1 Overview of the System Task

The system task accepts 28 kinds of messages, shown in Fig. 2-45. Each of these can be considered a kernel call, although, as we shall see, in some cases there are multiple macros defined with different names that all result in just one of the message types shown in the figure. And in some other cases more than one of the message types in the figure are handled by a single procedure that does the work.

The main program of the system task is structured like other tasks. After doing necessary initialization it runs in a loop. It gets a message, dispatches to the appropriate service procedure, and then sends a reply. A few general support functions are found in the main file, *system.c*, but the main loop dispatches to a procedure in a separate file in the *kernel/system/* directory to process each kernel call. We will see how this works and the reason for this organization when we discuss the implementation of the system task.

First we will briefly describe the function of each kernel call. The message types in Fig. 2-45 fall into several categories. The first few are involved with process management. Sys_fork, sys_exec, sys_exit, and sys_trace are obviously closely related to standard POSIX system calls. Although *nice* is not a POSIX-required system call, the command ultimately results in a sys_nice kernel call to

Message type	From	Meaning
sys_fork	PM	A process has forked
sys_exec	PM	Set stack pointer after EXEC call
sys_exit	PM	A process has exited
sys_nice	PM	Set scheduling priority
sys_privctl	RS	Set or change privileges
sys_trace	PM	Carry out an operation of the PTRACE call
sys_kill	PM,FS, TTY	Send signal to a process after KILL call
sys_getksig	PM	PM is checking for pending signals
sys_endksig	PM	PM has finished processing signal
sys_sigsend	PM	Send a signal to a process
sys_sigreturn	PM	Cleanup after completion of a signal
sys_irqctl	Drivers	Enable, disable, or configure interrupt
sys_devio	Drivers	Read from or write to an I/O port
sys_sdevio	Drivers	Read or write string from/to I/O port
sys_vdevio	Drivers	Carry out a vector of I/O requests
sys_int86	Drivers	Do a real-mode BIOS call
sys_newmap	PM	Set up a process memory map
sys_segctl	Drivers	Add segment and get selector (far data access)
sys_memset	PM	Write char to memory area
sys_umap	Drivers	Convert virtual address to physical address
sys_vircopy	FS, Drivers	Copy using pure virtual addressing
sys_physcopy	Drivers	Copy using physical addressing
sys_virvcopy	Any	Vector of VCOPY requests
sys_physvcopy	Any	Vector of PHYSCOPY requests
sys_times	PM	Get uptime and process times
sys_setalarm	PM, FS, Drivers	Schedule a synchronous alarm
sys_abort	PM, TTY	Panic: MINIX is unable to continue
sys_getinfo	Any	Request system information

Figure 2-45. The message types accepted by the system task. "Any" means any system process; user processes cannot call the system task directly.

change the priority of a process. The only one of this group that is likely to be unfamiliar is sys_privctl. It is used by the reincarnation server (RS), the MINIX 3 component responsible for converting processes started as ordinary user processes into system processes. Sys_privctl changes the privileges of a process, for instance, to allow it to make kernel calls. Sys_privctl is used when drivers and servers that are not part of the boot image are started by the */etc/rc* script. MINIX

3 drivers also can be started (or restarted) at any time; privilege changes are needed whenever this is done.

The next group of kernel calls are related to signals. Sys_kill is related to the user-accessible (and misnamed) system call kill. The others in this group, sys_getksig, sys_endksig, sys_sigsend, and sys_sigreturn are all used by the process manager to get the kernel's help in handling signals.

The sys_irqctl, sys_devio, sys_sdevio, and sys_vdevio kernel calls are unique to MINIX 3. These provide the support needed for user-space device drivers. We mentioned sys_irqctl at the start of this section. One of its functions is to set a hardware interrupt handler and enable interrupts on behalf of a user-space driver. Sys_devio allows a user-space driver to ask the system task to read or write from an I/O port. This is obviously essential; it also should be obvious that it involves more overhead than would be the case if the driver were running in kernel space. The next two kernel calls offer a higher level of I/O device support. Sys_sdevio can be used when a sequence of bytes or words, i.e., a string, is to be read from or written to a single I/O address, as might be the case when accessing a serial port. Sys_vdevio is used to send a vector of I/O requests to the system task. By a vector is meant a series of (port, value) pairs. Earlier in this chapter, we described the *intr_init* function that initializes the Intel i8259 interrupt controllers. On lines 8140 to 8152 a series of instructions writes a series of byte values. For each of the two i8259 chips, there is a control port that sets the mode and another port that receives a sequence of four bytes in the initialization sequence. Of course, this code executes in the kernel, so no support from the system task is needed. But if this were being done by a user-space process a single message passing the address to a buffer containing 10 (port, value) pairs would be much more efficient than 10 messages each passing one port address and a value to be written.

The next three kernel calls shown in Fig. 2-45 involve memory in distinct ways. The first, sys_newmap, is called by the process manager when the memory used by a process changes, so the kernel's part of the process table can be updated. Sys_segctl and sys_memset provide a safe way to provide a process with access to memory outside its own data space. The memory area from 0xa0000 to 0xfffff is reserved for I/O devices, as we mentioned in the discussion of startup of the MINIX 3 system. Some devices use part of this memory region for I/O—for instance, video display cards expect to have data to be displayed written into memory on the card which is mapped here. Sys_segctl is used by a device driver to obtain a segment selector that will allow it to address memory in this range. The other call, sys_memset, is used when a server wants to write data into an area of memory that does not belong to it. It is used by the process manager to zero out memory when a new process is started, to prevent the new process from reading data left by another process.

The next group of kernel calls is for copying memory. Sys_umap converts virtual addresses to physical addresses. Sys_vircopy and sys_physcopy copy regions of memory, using either virtual or physical addresses. The next two calls, sys_virvcopy and sys_physvcopy are vector versions of the previous two. As with vectored I/O requests, these allow making a request to the system task for a series of memory copy operations.

Sys_times obviously has to do with time, and corresponds to the POSIX times system call. Sys_setalarm is related to the POSIX alarm system call, but the relation is a distant one. The POSIX call is mostly handled by the process manager, which maintains a queue of timers on behalf of user processes. The process manager uses a sys_setalarm kernel call when it needs to have a timer set on its behalf in the kernel. This is done only when there is a change at the head of the queue managed by the PM, and does not necessarily follow every alarm call from a user process.

The final two kernel calls listed in Fig. 2-45 are for system control. Sys_abort can originate in the process manager, after a normal request to shutdown the system or after a panic. It can also originate from the tty device driver, in response to a user pressing the Ctrl-Alt-Del key combination.

Finally, sys_getinfo is a catch-all that handles a diverse range of requests for information from the kernel. If you search through the MINIX 3 C source files you will, in fact, find very few references to this call by its own name. But if you extend your search to the header directories you will find no less than 13 macros in *include/minix/syslib.h* that give another name to Sys_getinfo. An example is

```
sys_getkinfo(dst)      sys_getinfo(GET_KINFO, dst, 0,0,0)
```

which is used to return the *kinfo* structure (defined in *include/minix/type.h* on lines 2875 to 2893) to the process manager for use during system startup. The same information may be needed at other times. For instance, the user command *ps* needs to know the location of the kernel's part of the process table to display information about the status of all processes. It asks the PM, which in turn uses the *sys_getkinfo* variant of sys_getinfo to get the information.

Before we leave this overview of kernel call types, we should mention that sys_getinfo is not the only kernel call that is invoked by a number of different names defined as macros in *include/minix/syslib.h*. For example, the sys_sdevio call is usually invoked by one of the macros sys_insb, sys_insw, sys_outsb, or sys_outsw. The names were devised to make it easy to see whether the operation is input or output, with data types byte or word. Similarly, the sys_irqctl call is usually invoked by a macro like sys_irqenable, sys_irqdisable, or one of several others. Such macros make the meaning clearer to a person reading the code. They also help the programmer by automatically generating constant arguments.

2.7.2 Implementation of the System Task

The system task is compiled from a header, *system.h*, and a C source file, *system.c*, in the main *kernel/* directory. In addition there is a specialized library built from source files in a subdirectory, *kernel/system/*. There is a reason for this organization. Although MINIX 3 as we describe it here is a general-purpose operating system, it is also potentially useful for special purposes, such as embedded support in a portable device. In such cases a stripped-down version of the operating system might be adequate. For instance, a device without a disk might not need a file system. We saw in *kernel/config.h* that compilation of kernel calls can be selectively enabled and disabled. Having the code that supports each kernel call linked from the library as the last stage of compilation makes it easier to build a customized system.

Putting support for each kernel call in a separate file simplifies maintenance of the software. But there is some redundancy between these files, and listing all of them would add 40 pages to the length of this book. Thus we will list in Appendix B and describe in the text only a few of the files in the *kernel/system/* directory. However, all the files are on the CD-ROM and the MINIX 3 Web site.

We will begin by looking at the header file, *kernel/system.h* (line 9600). It provides prototypes for functions corresponding to most of the kernel calls listed in Fig. 2-45. In addition there is a prototype for *do_unused*, the function that is invoked if an unsupported kernel call is made. Some of the message types in Fig. 2-45 correspond to macros defined here. These are on lines 9625 to 9630. These are cases where one function can handle more than one call.

Before looking at the code in *system.c*, note the declaration of the call vector *call_vec*, and the definition of the macro *map* on lines 9745 to 9749. *Call_vec* is an array of pointers to functions, which provides a mechanism for dispatching to the function needed to service a particular message by using the message type, expressed as a number, as an index into the array. This is a technique we will see used elsewhere in MINIX 3. The *map* macro is a convenient way to initialize such an array. The macro is defined in such a way that trying to expand it with an invalid argument will result in declaring an array with a negative size, which is, of course, impossible, and will cause a compiler error.

The top level of the system task is the procedure *sys_task*. After a call to initialize an array of pointers to functions, *sys_task* runs in a loop. It waits for a message, makes a few tests to validate the message, dispatches to the function that handles the call that corresponds to the message type, possibly generating a reply message, and repeats the cycle as long as MINIX 3 is running (lines 9768 to 9796). The tests consists of a check of the *priv* table entry for the caller to determine that it is allowed to make this type of call and making sure that this type of call is valid. The dispatch to the function that does the work is done on line 9783. The index into the *call_vec* array is the call number, the function called is the one whose address is in that cell of the array, the argument to the function is a pointer to the message, and the return value is a status code. A function may return a *EDONTREPLY* status, meaning no reply message is required, otherwise a reply message is sent at line 9792.

As you may have noticed in Fig. 2-43, when MINIX 3 starts up the system task is at the head of the highest priority queue, so it makes sense that the system task's *initialize* function initializes the array of interrupt hooks and the list of alarm timers (lines 9808 to 9815). In any case, as we noted earlier, the system task is used to enable interrupts on behalf of user-space drivers that need to respond to interrupts, so it makes sense to have it prepare the table. The system task is used to set up timers when synchronous alarms are requested by other system processes, so initializing the timer lists is also appropriate here.

Continuing with initialization, on lines 9822 to 9824 all slots in the *call_vec* array are filled with the address of the procedure *do_unused*, called if an unsupported kernel call is made. Then the rest of the file lines 9827 to 9867, consists of multiple expansions of the *map* macro, each one of which installs the address of a function into the proper slot in *call_vec*.

The rest of *system.c* consists of functions that are declared *PUBLIC* and that may be used by more than one of the routines that service kernel calls, or by other parts of the kernel. For instance, the first such function, *get_priv* (line 9872), is used by *do_privctl*, which supports the sys_privctl kernel call. It is also called by the kernel itself while constructing process table entries for processes in the boot image. The name is a perhaps a bit misleading. *Get_priv* does not retrieve information about privileges already assigned, it finds an available *priv* structure and assigns it to the caller. There are two cases—system processes each get their own entry in the *priv* table. If one is not available then the process cannot become a system process. User processes all share the same entry in the table.

Get_randomness (line 9899) is used to get seed numbers for the random number generator, which is a implemented as a character device in MINIX 3. The newest Pentium-class processors include an internal cycle counter and provide an assembly language instruction that can read it. This is used if available, otherwise a function is called which reads a register in the clock chip.

Send_sig generates a notification to a system process after setting a bit in the *s_sig_pending* bitmap of the process to be signaled. The bit is set on line 9942. Note that because the *s_sig_pending* bitmap is part of a *priv* structure, this mechanism can only be used to notify system processes. All user processes share a common *priv* table entry, and therefore fields like the *s_sig_pending* bitmap cannot be shared and are not used by user processes. Verification that the target is a system process is made before *send_sig* is called. The call comes either as a result of a *sys_kill* kernel call, or from the kernel when *kprintf* is sending a string of characters. In the former case the caller determines whether or not the target is a system process. In the latter case the kernel only prints to the configured output process, which is either the console driver or the log driver, both of which are system processes.

The next function, *cause_sig* (line 9949), is called to send a signal to a user process. It is used when a *sys_kill* kernel call targets a user process. It is here in *system.c* because it also may be called directly by the kernel in response to an exception triggered by the user process. As with *send_sig* a bit must be set in the recipient's bitmap for pending signals, but for user processes this is not in the *priv* table, it is in the process table. The target process must also be made not ready by a call to *lock_dequeue*, and its flags (also in the process table) updated to indicate it is going to be signaled. Then a message is sent—but not to the target process. The message is sent to the process manager, which takes care of all of the aspects of signaling a process that can be dealt with by a user-space system process.

Next come three functions which all support the sys_umap kernel call. Processes normally deal with virtual addresses, relative to the base of a particular segment. But sometimes they need to know the absolute (physical) address of a region of memory, for instance, if a request is going to be made for copying between memory regions belonging to two different segments. There are three ways a virtual memory address might be specified. The normal one for a process is relative to one of the memory segments, text, data, or stack, assigned to a process and recorded in its process table slot. Requesting conversion of virtual to physical memory in this case is done by a call to *umap_local* (line 9983).

The second kind of memory reference is to a region of memory that is outside the text, data, or stack areas allocated to a process, but for which the process has some responsibility. Examples of this are a video driver or an Ethernet driver, where the video or Ethernet card might have a region of memory mapped in the region from 0xa0000 to 0xfffff which is reserved for I/O devices. Another example is the memory driver, which manages the ramdisk and also can provide access to any part of the memory through the devices */dev/mem* and */dev/kmem*. Requests for conversion of such memory references from virtual to physical are handled by *umap_remote* (line 10025).

Finally, a memory reference may be to memory that is used by the BIOS. This is considered to include both the lowest 2 KB of memory, below where MINIX 3 is loaded, and the region from 0x90000 to 0xfffff, which includes some RAM above where MINIX 3 is loaded plus the region reserved for I/O devices. This could also be handled by *umap_remote*, but using the third function, *umap_bios* (line 10047), ensures that a check will be made that the memory being referenced is really in this region.

The last function defined in *system.c* is *virtual_copy* (line 10071). Most of this function is a C switch which uses one of the three *umap_** functions just described to convert virtual addresses to physical addresses. This is done for both the source and destination addresses. The actual copying is done (on line 10121) by a call to the assembly language routine *phys_copy* in *klib386.s*.

2.7.3 Implementation of the System Library

Each of the functions with a name of the form *do_xyz* has its source code in a file in a subdirectory, *kernel/system/do_xyz.c*. In the *kernel/* directory the *Makefile* contains a line

```
cd system && $(MAKE) –$(MAKEFLAGS) $@
```

which causes all of the files in *kernel/system/* to be compiled into a library, *system.a* in the main *kernel/* directory. When control returns to the main kernel directory another line in the *Makefile* cause this local library to be searched first when the kernel object files are linked.

We have listed two files from the *kernel/system/* directory in Appendix B. These were chosen because they represent two general classes of support that the system task provides. One category of support is access to kernel data structures on behalf of any user-space system process that needs such support. We will describe *system/do_setalarm.c* as an example of this category. The other general category is support for specific system calls that are mostly managed by user-space processes, but which need to carry out some actions in kernel space. We have chosen *system/do_exec.c* as our example.

The sys_setalarm kernel call is somewhat similar to sys_irqenable, which we mentioned in the discussion of interrupt handling in the kernel. Sys_irqenable sets up an address to an interrupt handler to be called when an IRQ is activated. The handler is a function within the system task, *generic_handler*. It generates a notify message to the device driver process that should respond to the interrupt. *System/do_setalarm.c* (line 10200) contains code to manage timers in a way similar to how interrupts are managed. A sys_setalarm kernel call initializes a timer for a user-space system process that needs to receive a synchronous alarm, and it provides a function to be called to notify the user-space process when the timer expires. It can also ask for cancellation of a previously scheduled alarm by passing zero in the expiration time field of its request message. The operation is simple—on lines 10230 to 10232 information from the message is extracted. The most important items are the time when the timer should go off and the process that needs to know about it. Every system process has its own timer structure in the *priv* table. On lines 10237 to 10239 the timer structure is located and the process number and the address of a function, *cause_alarm*, to be executed when the timer expires, are entered.

If the timer was already active, sys_setalarm returns the time remaining in its reply message. A return value of zero means the timer is not active. There are several possibilities to be considered. The timer might previously have been deactivated—a timer is marked inactive by storing a special value, *TMR_NEVER* in its *exp_time* field . As far as the C code is concerned this is just a large integer, so an explicit test for this value is made as part of checking whether the expiration time has passed. The timer might indicate a time that has already passed. This is unlikley to happen, but it is easy to check. The timer might also indicate a time in the future. In either of the first two cases the reply value is zero, otherwise the time remaining is returned (lines 10242 to 10247).

Finally, the timer is reset or set. At this level this is done putting the desired expiration time into the correct field of the timer structure and calling another function to do the work. Of course, resetting the timer does not require storing a value. We will see the functions *reset* and *set* soon, their code is in the source file for the clock task. But since the system task and the clock task are both compiled into the kernel image all functions declared *PUBLIC* are accessible.

There is one other function defined in *do_setalarm.c*. This is *cause_alarm*, the watchdog function whose address is stored in each timer, so it can be called when the timer expires. It is simplicity itself—it generates a notify message to the process whose process number is also stored in the timer structure. Thus the synchronous alarm within the kernel is converted into a message to the system process that asked for an alarm.

As an aside, note that when we talked about the initialization of timers a few pages back (and in this section as well) we referred to synchronous alarms requested by system processes. If that did not make complete sense at this point, and if you are wondering what is a synchronous alarm or what about timers for nonsystem processes, these questions will be dealt with in the next section, when we discuss the clock task. There are so many interconnected parts in an operating system that it is almost impossible to order all topics in a way that does not occasionally require a reference to a part that has not been already been explained. This is particularly true when discussing implementation. If we were not dealing with a real operating system we could probably avoid bringing up messy details like this. For that matter, a totally theoretical discussion of operating system principles would probably never mention a system task. In a theory book we could just wave our arms and ignore the problems of giving operating system components in user space limited and controlled access to privileged resources like interrupts and I/O ports.

The last file in the *kernel/system/* directory which we will discuss in detail is *do_exec.c* (line 10300). Most of the work of the exec system call is done within the process manager. The process manager sets up a stack for a new program that contains the arguments and the environment. Then it passes the resulting stack pointer to the kernel using sys_exec, which is handled by *do_exec* (line 10618). The stack pointer is set in the kernel part of the process table, and if the process being exec-ed is using an extra segment the assembly language *phys_memset* function defined in *klib386.s* is called to erase any data that might be left over from previous use of that memory region (line 10330).

An exec call causes a slight anomaly. The process invoking the call sends a message to the process manager and blocks. With other system calls, the resulting reply would unblock it. With exec there is no reply, because the newly loaded core image is not expecting a reply. Therefore, *do_exec* unblocks the process itself on line 10333 The next line makes the new image ready to run, using the *lock_enqueue* function that protects against a possible race condition. Finally, the command string is saved so the process can be identified when the user invokes the *ps* command or presses a function key to display data from the process table.

To finish our discussion of the system task, we will look at its role in handling a typical operating service, providing data in response to a read system call. When a user does a read call, the file system checks its cache to see if it has the block needed. If not, it sends a message to the appropriate disk driver to load it into the cache. Then the file system sends a message to the system task telling it to copy the block to the user process. In the worst case, eleven messages are needed to read a block; in the best case, four messages are needed. Both cases are shown in Fig. 2-46. In Fig. 2-46 (a), message 3 asks the system task to execute I/O instructions; 4 is the ACK. When a hardware interrupt occurs the system task tells the waiting driver about this event with message 5. Messages 6 and 7 are a request to copy the data to the FS cache and the reply, message 8 tells the FS the data is ready, and messages 9 and 10 are a request to copy the data from the cache to the user, and the reply. Finally message 11 is the reply to the user. In Fig. 2-46 (b), the data is already in the cache, messages 2 and 3 are the request to copy it to the user and the reply. These messages are a source of overhead in MINIX 3 and are the price paid for the highly modular design.

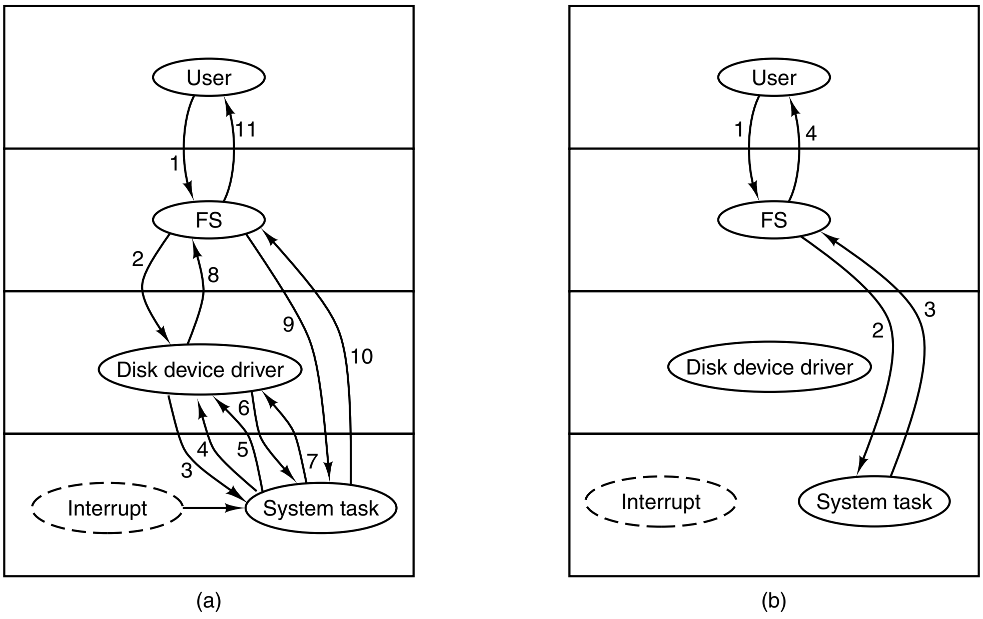

Figure 2-46. (a) Worst case for reading a block requires eleven messages. (b) Best case for reading a block requires four messages.

Kernel calls to request copying of data are probably the most heavily used ones in MINIX 3. We have already seen the part of the system task that ultimately does the work, the function *virtual_copy*. One way to deal with some of the inefficiency of the message passing mechanism is to pack multiple requests into a message. The *sys_virvcopy* and *sys_physvcopy* kernel calls do this. The content of a message that invokes one of these call is a pointer to a vector specifying multiple blocks to be copied between memory locations. Both are supported by *do_vcopy*, which executes a loop, extracting source and destination addresses and block lengths and calling *phys_copy* repeatedly until all the copies are complete. We will see in the next chapter that disk devices have a similar ability to handle multiple transfers based on a single request.

2.8 THE CLOCK TASK IN MINIX 3

Clocks (also called **timers**) are essential to the operation of any timesharing system for a variety of reasons. For example, they maintain the time of day and prevent one process from monopolizing the CPU. The MINIX 3 clock task has some resemblance to a device driver, in that it is driven by interrupts generated by a hardware device. However, the clock is neither a block device, like a disk, nor a character device, like a terminal. In fact, in MINIX 3 an interface to the clock is not provided by a file in the */dev/* directory. Furthermore, the clock task executes in kernel space and cannot be accessed directly by user-space processes. It has access to all kernel functions and data, but user-space processes can only access it via the system task. In this section we will first a look at clock hardware and software in general, and then we will see how these ideas are applied in MINIX 3.

2.8.1 Clock Hardware

Two types of clocks are used in computers, and both are quite different from the clocks and watches used by people. The simpler clocks are tied to the 110- or 220-volt power line, and cause an interrupt on every voltage cycle, at 50 or 60 Hz. These are essentially extinct in modern PCs.

The other kind of clock is built out of three components: a crystal oscillator, a counter, and a holding register, as shown in Fig. 2-47. When a piece of quartz crystal is properly cut and mounted under tension, it can be made to generate a periodic signal of very high accuracy, typically in the range of 5 to 200 MHz, depending on the crystal chosen. At least one such circuit is usually found in any computer, providing a synchronizing signal to the computer's various circuits. This signal is fed into the counter to make it count down to zero. When the counter gets to zero, it causes a CPU interrupt. Computers whose advertised clock rate is higher than 200 MHz normally use a slower clock and a clock multiplier circuit.

Programmable clocks typically have several modes of operation. In **one-shot mode**, when the clock is started, it copies the value of the holding register into the counter and then decrements the counter at each pulse from the crystal. When the counter gets to zero, it causes an interrupt and stops until it is explicitly started again by the software. In **square-wave mode**, after getting to zero and causing the

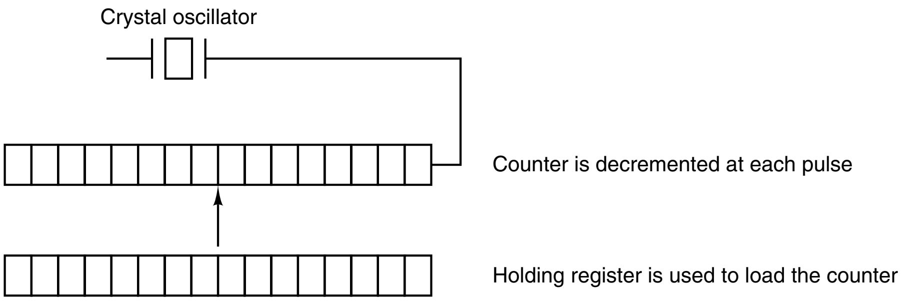
Crystal oscillator
Counter is decremented at each pulse
Holding register is used to load the counter

Figure 2-47. A programmable clock.

interrupt, the holding register is automatically copied into the counter, and the whole process is repeated again indefinitely. These periodic interrupts are called **clock ticks**.

The advantage of the programmable clock is that its interrupt frequency can be controlled by software. If a 1-MHz crystal is used, then the counter is pulsed every microsecond. With 16-bit registers, interrupts can be programmed to occur at intervals from 1 microsecond to 65.536 milliseconds. Programmable clock chips usually contain two or three independently programmable clocks and have many other options as well (e.g., counting up instead of down, interrupts disabled, and more).

To prevent the current time from being lost when the computer's power is turned off, most computers have a battery-powered backup clock, implemented with the kind of low-power circuitry used in digital watches. The battery clock can be read at startup. If the backup clock is not present, the software may ask the user for the current date and time. There is also a standard protocol for a networked system to get the current time from a remote host. In any case the time is then translated into the number of seconds since 12 A.M. **Universal Coordinated Time (UTC)** (formerly known as Greenwich Mean Time) on Jan. 1, 1970, as UNIX and MINIX 3 do, or since some other benchmark. Clock ticks are counted by the running system, and every time a full second has passed the real time is incremented by one count. MINIX 3 (and most UNIX systems) do not take into account leap seconds, of which there have been 23 since 1970. This is not considered a serious flaw. Usually, utility programs are provided to manually set the system clock and the backup clock and to synchronize the two clocks.

We should mention here that all but the earliest IBM-compatible computers have a separate clock circuit that provides timing signals for the CPU, internal data busses, and other components. This is the clock that is meant when people speak of CPU clock speeds, measured in Megahertz on the earliest personal computers, and in Gigahertz on modern systems. The basic circuitry of quartz crystals, oscillators and counters is the same, but the requirements are so different that modern computers have independent clocks for CPU control and timekeeping.

2.8.2 Clock Software

All the clock hardware does is generate interrupts at known intervals. Everything else involving time must be done by the software, the clock driver. The exact duties of the clock driver vary among operating systems, but usually include most of the following:

1. Maintaining the time of day.
2. Preventing processes from running longer than they are allowed to.
3. Accounting for CPU usage.
4. Handling the alarm system call made by user processes.
5. Providing watchdog timers for parts of the system itself.
6. Doing profiling, monitoring, and statistics gathering.

The first clock function, maintaining the time of day (also called the **real time**) is not difficult. It just requires incrementing a counter at each clock tick, as mentioned before. The only thing to watch out for is the number of bits in the time-of-day counter. With a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years. Clearly the system cannot store the real time as the number of ticks since Jan. 1, 1970 in 32 bits.

Three approaches can be taken to solve this problem. The first way is to use a 64-bit counter, although doing so makes maintaining the counter more expensive since it has to be done many times a second. The second way is to maintain the time of day in seconds, rather than in ticks, using a subsidiary counter to count ticks until a whole second has been accumulated. Because 2^{32} seconds is more than 136 years, this method will work until well into the twenty-second century.

The third approach is to count ticks, but to do that relative to the time the system was booted, rather than relative to a fixed external moment. When the backup clock is read or the user types in the real time, the system boot time is calculated from the current time-of-day value and stored in memory in any convenient form. When the time of day is requested, the stored time of day is added to the counter to get the current time of day. All three approaches are shown in Fig. 2-48.

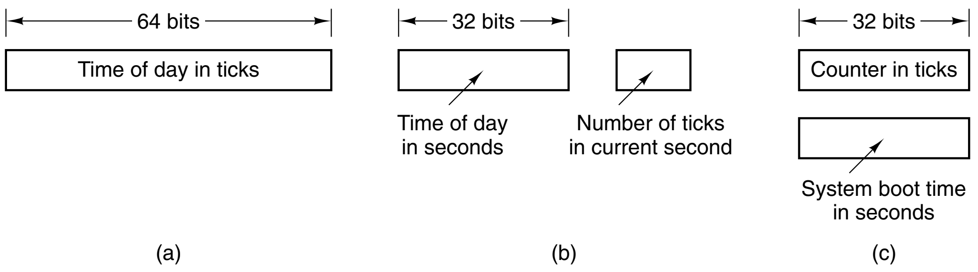

Figure 2-48. Three ways to maintain the time of day.

The second clock function is preventing processes from running too long. Whenever a process is started, the scheduler should initialize a counter to the value of that process' quantum in clock ticks. At every clock interrupt, the clock driver decrements the quantum counter by 1. When it gets to zero, the clock driver calls the scheduler to set up another process.

The third clock function is doing CPU accounting. The most accurate way to do it is to start a second timer, distinct from the main system timer, whenever a process is started. When that process is stopped, the timer can be read out to tell how long the process has run. To do things right, the second timer should be saved when an interrupt occurs and restored afterward.

A less accurate, but much simpler, way to do accounting is to maintain a pointer to the process table entry for the currently running process in a global variable. At every clock tick, a field in the current process' entry is incremented. In this way, every clock tick is "charged" to the process running at the time of the tick. A minor problem with this strategy is that if many interrupts occur during a process' run, it is still charged for a full tick, even though it did not get much work done. Properly accounting for the CPU during interrupts is too expensive and is rarely done.

In MINIX 3 and many other systems, a process can request that the operating system give it a warning after a certain interval. The warning is usually a signal, interrupt, message, or something similar. One application requiring such warnings is networking, in which a packet not acknowledged within a certain time interval must be retransmitted. Another application is computer-aided instruction, where a student not providing a response within a certain time is told the answer.

If the clock driver had enough clocks, it could set a separate clock for each request. This not being the case, it must simulate multiple virtual clocks with a single physical clock. One way is to maintain a table in which the signal time for all pending timers is kept, as well as a variable giving the time of the next one. Whenever the time of day is updated, the driver checks to see if the closest signal has occurred. If it has, the table is searched for the next one to occur.

If many signals are expected, it is more efficient to simulate multiple clocks by chaining all the pending clock requests together, sorted on time, in a linked list, as shown in Fig. 2-49. Each entry on the list tells how many clock ticks following the previous one to wait before causing a signal. In this example, signals are pending for 4203, 4207, 4213, 4215, and 4216.

In Fig. 2-49, a timer has just expired. The next interrupt occurs in 3 ticks, and 3 has just been loaded. On each tick, *Next signal* is decremented. When it gets to 0, the signal corresponding to the first item on the list is caused, and that item is removed from the list. Then *Next signal* is set to the value in the entry now at the head of the list, in this example, 4. Using absolute times rather than relative times is more convenient in many cases, and that is the approach used by MINIX 3.

Note that during a clock interrupt, the clock driver has several things to do. These things include incrementing the real time, decrementing the quantum and

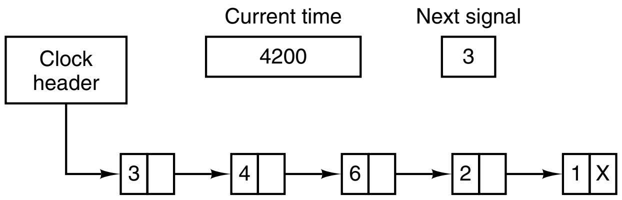

Figure 2-49. Simulating multiple timers with a single clock.

checking for 0, doing CPU accounting, and decrementing the alarm counter. However, each of these operations has been carefully arranged to be very fast because they have to be repeated many times a second.

Parts of the operating system also need to set timers. These are called **watchdog timers**. When we study the hard disk driver, we will see that a wakeup call is scheduled each time the disk controller is sent a command, so an attempt at recovery can be made if the command fails completely. Floppy disk drivers use timers to wait for the disk motor to get up to speed and to shut down the motor if no activity occurs for a while. Some printers with a movable print head can print at 120 characters/sec (8.3 msec/character) but cannot return the print head to the left margin in 8.3 msec, so the terminal driver must delay after typing a carriage return.

The mechanism used by the clock driver to handle watchdog timers is the same as for user signals. The only difference is that when a timer goes off, instead of causing a signal, the clock driver calls a procedure supplied by the caller. The procedure is part of the caller's code. This presented a problem in the design of MINIX 3, since one of the goals was to remove drivers from the kernel's address space. The short answer is that the system task, which is in kernel space, can set alarms on behalf of some user-space processes, and then notify them when a timer goes off. We will elaborate on this mechanism further on.

The last thing in our list is profiling. Some operating systems provide a mechanism by which a user program can have the system build up a histogram of its program counter, so it can see where it is spending its time. When profiling is a possibility, at every tick the driver checks to see if the current process is being profiled, and if so, computes the bin number (a range of addresses) corresponding to the current program counter. It then increments that bin by one. This mechanism can also be used to profile the system itself.

2.8.3 Overview of the Clock Driver in MINIX 3

The MINIX 3 clock driver is contained in the file *kernel/clock.c*. It can be considered to have three functional parts. First, like the device drivers that we will see in the next chapter, there is a task mechanism which runs in a loop, waiting for messages and dispatching to subroutines that perform the action requested in each message. However, this structure is almost vestigial in the clock task. The message mechanism is expensive, requiring all the overhead of a context switch. So for the clock this is used only when there is a substantial amount of work to be done. Only one kind of message is received, there is only one subroutine to service the message, and a reply message is not sent when the job is done.

The second major part of the clock software is the interrupt handler that is activated 60 times each second. It does basic timekeeping, updating a variable that counts clock ticks since the system was booted. It compares this with the time for the next timer expiration. It also updates counters that register how much of the quantum of the current process has been used and how much total time the current process has used. If the interrupt handler detects that a process has used its quantum or that a timer has expired it generates the message that goes to the main task loop. Otherwise no message is sent. The strategy here is that for each clock tick the handler does as little as necessary, as fast as possible. The costly main task is activated only when there is substantial work to do.

The third general part of the clock software is a collection of subroutines that provide general support, but which are not called in response to clock interrupts, either by the interrupt handler or by the main task loop. One of these subroutines is coded as *PRIVATE*, and is called before the main task loop is entered. It initializes the clock, which entails writing data to the clock chip to cause it to generate interrupts at the desired intervals. The initialization routine also puts the address of the interrupt handler in the right place to be found when the clock chip triggers the IRQ 8 input to the interrupt controller chip, and then enables that input to respond.

The rest of the subroutines in *clock.c* are declared *PUBLIC*, and can be called from anywhere in the kernel binary. In fact none of them are called from *clock.c* itself. They are mostly called by the system task in order to service system calls related to time. These subroutines do such things as reading the time-since-boot counter, for timing with clock-tick resolution, or reading a register in the clock chip itself, for timing that requires microsecond resolution. Other subroutines are used to set and reset timers. Finally, a subroutine is provided to be called when MINIX 3 shuts down. This one resets the hardware timer parameters to those expected by the BIOS.

The Clock Task

The main loop of the clock task accepts only a single kind of message, *HARD_INT*, which comes from the interrupt handler. Anything else is an error. Furthermore, it does not receive this message for every clock tick interrupt, although the subroutine called each time a message is received is named *do_clocktick*. A message is received, and *do_clocktick* is called only if process scheduling is needed or a timer has expired.

The Clock Interrupt Handler

The interrupt handler runs every time the counter in the clock chip reaches zero and generates an interrupt. This is where the basic timekeeping work is done. In MINIX 3 the time is kept using the method of Fig. 2-48(c). However, in *clock.c* only the counter for ticks since boot is maintained; records of the boot time are kept elsewhere. The clock software supplies only the current tick count to aid a system call for the real time. Further processing is done by one of the servers. This is consistent with the MINIX 3 strategy of moving functionality to processes that run in user space.

In the interrupt handler the local counter is updated for each interrupt received. When interrupts are disabled ticks are lost. In some cases it is possible to correct for this effect. A global variable is available for counting lost ticks, and it is added to the main counter and then reset to zero each time the handler is activated. In the implementation section we will see an example of how this is used.

The handler also affects variables in the process table, for billing and process control purposes. A message is sent to the clock task only if the current time has passed the expiration time of the next scheduled timer or if the quantum of the running process has been decremented to zero. Everything done in the interrupt service is a simple integer operation—arithmetic, comparison, logical AND/OR, or assignment—which a C compiler can translate easily into basic machine operations. At worst there are five additions or subtractions and six comparisons, plus a few logical operations and assignments in completing the interrupt service. In particular there is no subroutine call overhead.

Watchdog Timers

A few pages back we left hanging the question of how user-space processes can be provided with watchdog timers, which ordinarily are thought of as user-supplied procedures that are part of the user's code and are executed when a timer expires. Clearly, this can not be done in MINIX 3. But we can use a **synchronous alarm** to bridge the gap from the kernel to user space.

This is a good time to explain what is meant by a synchronous alarm. A signal may arrive or a conventional watchdog may be activated without any relation to what part of a process is currently executing, so these mechanisms are **asynchronous**. A synchronous alarm is delivered as a message, and thus can be received only when the recipient has executed receive. So we say it is synchronous because it will be received only when the receiver expects it. If the notify method is used to inform a recipient of an alarm, the sender does not have to block, and the recipient does not have to be concerned with missing the alarm. Messages from notify are saved if the recipient is not waiting. A bitmap is used, with each bit representing a possible source of a notification.

Watchdog timers take advantage of the *timer_t* type *s_alarm_timer* field that exists in each element of the *priv* table. Each system process has a slot in the *priv* table. To set a timer, a system process in user space makes a sys_setalarm call, which is handled by the system task. The system task is compiled in kernel space, and thus can initialize a timer on behalf of the calling process. Initialization entails putting the address of a procedure to execute when the timer expires into the correct field, and then inserting the timer into a list of timers, as in Fig. 2-49.

The procedure to execute has to be in kernel space too, of course. No problem. The system task contains a watchdog function, *cause_alarm*, which generates a notify when it goes off, causing a synchronous alarm for the user. This alarm can invoke the user-space watchdog function. Within the kernel binary this is a true watchdog, but for the process that requested the timer, it is a synchronous alarm. It is not the same as having the timer execute a procedure in the target's address space. There is a bit more overhead, but it is simpler than an interrupt.

What we wrote above was qualified: we said that the system task can set alarms on behalf of *some* user-space processes. The mechanism just described works only for system processes. Each system process has a copy of the *priv* structure, but a single copy is shared by all non-system (user) processes. The parts of the *priv* table that cannot be shared, such as the bitmap of pending notifications and the timer, are not usable by user processes. The solution is this: the process manager manages timers on behalf of user processes in a way similar to the way the system task manages timers for system processes. Every process has a *timer_t* field of its own in the process manager's part of the process table.

When a user process makes an alarm system call to ask for an alarm to be set, it is handled by the process manager, which sets up the timer and inserts it into its list of timers. The process manager asks the system task to send it a notification when the first timer in the PM's list of timers is scheduled to expire. The process manager only has to ask for help when the head of its chain of timers changes, either because the first timer has expired or has been cancelled, or because a new request has been received that must go on the chain before the current head. This is used to support the POSIX-standard alarm system call. The procedure to execute is within the address space of the process manager. When executed, the user process that requested the alarm is sent a signal, rather than a notification.

Millisecond Timing

A procedure is provided in *clock.c* that provides microsecond resolution timing. Delays as short as a few microseconds may be needed by various I/O devices. There is no practical way to do this using alarms and the message passing interface. The counter that is used for generating the clock interrupts can be read directly. It is decremented approximately every 0.8 microseconds, and reaches zero 60 times a second, or every 16.67 milliseconds. To be useful for I/O timing it would have to be polled by a procedure running in kernel-space, but much work has gone into moving drivers out of kernel-space. Currently this function is used only as a source of randomness for the random number generator. More use might be made of it on a very fast system, but this is a future project

Summary of Clock Services

Figure 2-50 summarizes the various services provided directly or indirectly by *clock.c*. There are several functions declared *PUBLIC* that can be called from the kernel or the system task. All other services are available only indirectly, by system calls ultimately handled by the system task. Other system processes can ask the system task directly, but user processes must ask the process manager, which also relies on the system task.

Service	Access	Response	Clients
get_uptime	Function call	Ticks	Kernel or system task
set_timer	Function call	None	Kernel or system task
reset_timer	Function call	None	Kernel or system task
read_clock	Function call	Count	Kernel or system task
clock_stop	Function call	None	Kernel or system task
Synchronous alarm	System call	Notification	Server or driver, via system task
POSIX alarm	System call	Signal	User process, via PM
Time	System call	Message	Any process, via PM

Figure 2-50. The time-related services supported by the clock driver.

The kernel or the system task can get the current uptime, or set or reset a timer without the overhead of a message. The kernel or the system task can also call *read_clock*, which reads the counter in the timer chip, to get time in units of approximately 0.8 microseconds. The *clock_stop* function is intended to be called only when MINIX 3 shuts down. It restores the BIOS clock rate. A system process, either a driver or a server, can request a synchronous alarm, which causes activation of a watchdog function in kernel space and a notification to the requesting process. A POSIX-alarm is requested by a user process by asking the process manager, which then asks the system task to activate a watchdog. When the timer expires, the system task notifies the process manager, and the process manager delivers a signal to the user process.

2.8.4 Implementation of the Clock Driver in MINIX 3

The clock task uses no major data structures, but several variables are used to keep track of time. The variable *realtime* (line 10462) is basic—it counts all clockticks. A global variable, *lost_ticks*, is defined in *glo.h* (line 5333). This variable is provided for the use of any function that executes in kernel space that might disable interrupts long enough that one or more clock ticks could be lost. It currently is used by the *int86* function in *klib386.s*. *Int86* uses the boot monitor to manage the transfer of control to the BIOS, and the monitor returns the number of clock ticks counted while the BIOS call was busy in the ecx register just before the return to the kernel. This works because, although the clock chip is not triggering the MINIX 3 clock interrupt handler when the BIOS request is handled, the boot monitor can keep track of the time with the help of the BIOS.

The clock driver accesses several other global variables. It uses *proc_ptr*, *prev_ptr*, and *bill_ptr* to reference the process table entry for the currently running process, the process that ran previously, and the process that gets charged for time. Within these process table entries it accesses various fields, including *p_user_time* and *p_sys_time* for accounting and *p_ticks_left* for counting down the quantum of a process.

When MINIX 3 starts up, all the drivers are called. Most of them do some initialization then try to get a message and block. The clock driver, *clock_task* (line 10468), does that too. First it calls *init_clock* to initialize the programmable clock frequency to 60 Hz. When a message is received, it calls *do_clocktick* if the message was a *HARD_INT* (line 10486). Any other kind of message is unexpected and treated as an error.

Do_clocktick (line 10497) is not called on each tick of the clock, so its name is not an exact description of its function. It is called when the interrupt handler has determined there might be something important to do. One of the conditions that results in running *do_clocktick* is the current process using up all of its quantum. If the process is preemptable (the system and clock tasks are not) a call to *lock_dequeue* followed immediately by a call to *lock_enqueue* (lines 10510 to 10512) removes the process from its queue, then makes it ready again and reschedules it. The other thing that activates *do_clocktick* is expiration of a watchdog timer. Timers and linked lists of timers are used so much in MINIX 3 that a library of functions to support them was created. The library function *tmrs_exptimers* called on line 10517 runs the watchdog functions for all expired timers and deactivates them.

Init_clock (line 10529) is called only once, when the clock task is started. There are several places one could point to and say, "This is where MINIX 3 starts running." This is a candidate; the clock is essential to a preemptive multitasking system. *Init_clock* writes three bytes to the clock chip that set its mode and set the proper count into the master register. Then it registers its process number, IRQ, and handler address so interrupts will be directed properly. Finally, it enables the interrupt controller chip to accept clock interrupts.

The next function, *clock_stop*, undoes the initialization of the clock chip. It is declared *PUBLIC* and is not called from anywhere in *clock.c*. It is placed here because of the obvious similarity to *init_clock*. It is only called by the system task when MINIX 3 is shut down and control is to be returned to the boot monitor.

As soon as (or, more accurately, 16.67 milliseconds after) *init_clock* runs, the first clock interrupt occurs, and clock interrupts repeat 60 times a second as long as MINIX 3 runs. The code in *clock_handler* (line 10556) probably runs more frequently than any other part of the MINIX 3 system. Consequently, *clock_handler* was built for speed. The only subroutine calls are on line 10586; they are only needed if running on an obsolete IBM PS/2 system. The update of the current time (in ticks) is done on lines 10589 to 10591. Then user and accounting times are updated.

Decisions were made in the design of the handler that might be questioned. Two tests are done on line 10610 and if either condition is true the clock task is notified. The *do_clocktick* function called by the clock task repeats both tests to decide what needs to be done. This is necessary because the notify call used by the handler cannot pass any information to distinguish different conditions. We leave it to the reader to consider alternatives and how they might be evaluated.

The rest of *clock.c* contains utility functions we have already mentioned. *Get_uptime* (line 10620) just returns the value of *realtime*, which is visible only to functions in *clock.c*. *Set_timer* and *reset_timer* use other functions from the timer library that take care of all the details of manipulating a chain of timers. Finally, *read_clock* reads and returns the current count in the clock chip's count-down register.

2.9 SUMMARY

To hide the effects of interrupts, operating systems provide a conceptual model consisting of sequential processes running in parallel. Processes can communicate with each other using interprocess communication primitives, such as semaphores, monitors, or messages. These primitives are used to ensure that no two processes are ever in their critical sections at the same time. A process can be running, runnable, or blocked and can change state when it or another process executes one of the interprocess communication primitives.

Interprocess communication primitives can be used to solve such problems as the producer-consumer, dining philosophers, and reader-writer. Even with these primitives, care has to be taken to avoid errors and deadlocks. Many scheduling algorithms are known, including round-robin, priority scheduling, multilevel queues, and policy-driven schedulers.

MINIX 3 supports the process concept and provides messages for interprocess communication. Messages are not buffered, so a send succeeds only when the receiver is waiting for it. Similarly, a receive succeeds only when a message is already available. If either operation does not succeed, the caller is blocked. MINIX 3 also provides a nonblocking supplement to messages with a notify primitive. An attempt to send a notify to a receiver that is not waiting results in a bit being set, which triggers notification when a receive is done later.