20

System Configuration

This chapter concludes our discussion of operating system design by answering two practical questions: how can the code built in earlier chapters be transformed to make it suitable for a system with different peripheral devices, and how can the code be collected into a so-called *kernel* that is isolated from user processes by the hardware's *system call* mechanism?

20.1 The Need For Multiple Configurations

When the system code is transformed to make it suitable for a particular set of peripheral devices, we say it has been *configured*; the result is often referred to as a *configuration*. The preceding chapters described the design process as if operating systems were built for one particular machine with a specific set of I/O devices. In reality, of course, a given operating system must run on a variety of hardware configurations, each with its own set of device and interrupt vector addresses. More importantly, devices are often combined on boards that limit the choice of addresses, so an operating system designer does not have complete freedom over them, even if the hardware can be changed. For example, a system that contains 4 serial line units can have them on a single board, in which case the device addresses are contiguous; or they can be located on several boards, in which case it may not be possible to make the addresses contiguous.

Handcrafting a new system configuration for each machine would be costly, time-consuming, and error-prone. PC-Xinu, like most systems, relies on a *configuration program* that helps reduce this cost by automating some of the chores. The program, called *config*, is not part of the system proper, and it will not be shown in detail. While the implementation is not important, understanding what *config* does is important: it takes as input a *specification* of the devices and parameters, and it produces PC-Xinu source files that generate the final object programs when compiled. Before discussing the input *config* expects and the general method it uses, the next section considers the entire configuration process.

20.2 Static vs. Dynamic Configuration

System configuration can be performed: when the system is generated (i.e. compiled and linked together), when the system is started (i.e., during bootstrap), or while the system is running. In general, postponing configuration makes the system more flexible but increases the overhead. The most advanced systems can reconfigure themselves dynamically as new devices are connected or old ones removed; the least advanced must be recompiled to accommodate even minor changes like switching the baud rate of a terminal.

The chief advantage of configuring at generation time (early) is that the memory image contains only those device drivers needed for devices that exist. Another advantage is that the system spends less time trying to identify hardware details during the bootstrap process, because these details are bound into the code. The chief disadvantage of early configuration is that a system configured for one machine cannot run on any other (unless they are identical, including such small details as device and interrupt vector addresses).

Deferring configuration until system startup allows the designer to build more robust code, because a single system can operate on several hardware configurations. During startup, the system locates devices, initializes interrupt vectors, establishes appropriate entries in the device table, and somehow correlates the devices with their drivers. It should be noted that the hardware on many machines limits the amount of configuration that can be deferred until startup. For example, devices on the PC must be known by the system, because it cannot determine their type or the location of their interrupt vectors at run-time.

Reconfiguring at run-time allows a system to adapt to changes in the hardware without stopping. Some systems keep all device drivers resident in main memory, so they can associate new devices with an appropriate driver immediately. Others permit device drivers to be loaded on demand (and unloaded on request). In such systems, the hardware cooperates closely with the software to inform the system whenever a device or processor becomes ready, or a ready device becomes disabled.

20.3 The Details Of Configuration In PC-Xinu

Like many systems, PC-Xinu uses a mixture of configuration strategies. Because it is a small system designed to run on primitive hardware, much of the configuration occurs early. For example, devices and associated drivers must be specified when a system is generated. Even in a small system like PC-Xinu, some of the configuration can be postponed until system startup. For example, the size of memory is tested after PC-Xinu begins. Interrupt vector initialization also occurs dynamically, when the system calls the driver initialization routines.

Most of the differences between configurations of PC-Xinu concern details of device addresses, interrupt vectors, and driver routines. Thus, the program *config*, responsible for producing a system given its specification, deals primarily with device configura-

tion. It reads a specification file that describes the types of device drivers available, as well as a list of specific devices. It produces output files that contain the definition of a device switch table and code to initialize it. We have already seen the output files, *conf.h* and *conf.c*, in Chapter 11. *Config* reads input specifications from the file named *pcxconf*. Its contents are shown below.

```
/* pcxconf - PC-Xinu system configuration specifications */

tty:    on BIOS
                -i ttyinit -o ttyopen -c ioerr
                -r ttyread -w ttywrite -s ioerr
                -g ttygetc -p ttyputc -n ttycntl
                -iint ttyiin

        on WINDOW
                -i lwinit -o ionull -c lwclose
                -r lwread -w lwwrite -s ioerr
                -g lwgetc -p lwputc -n lwcntl

dsk:    on BIOS
                -i dsinit -o dsopen -c ioerr
                -r dsread -w dswrite -s dsseek
                -g ioerr -p ioerr -n dscntl

df:     on DSK
                -i lfinit -o ioerr -c lfclose
                -r lfread -w lfwrite -s lfseek
                -g lfgetc -p lfputc -n ioerr

dos:    on MSDOS
                -i ionull -o msopen -c ioerr
                -r ioerr -w ioerr -s ioerr
                -g ioerr -p ioerr -n mscntl

mf:     on DOS
                -i mfinit -o ioerr -c mfclose
                -r mfread -w mfwrite -s mfseek
                -g mfgetc -p mfputc -n ioerr

%

#include <bios.h>

/* console + windows */
```

```
CONSOLE is tty  on BIOS         name="tty" ivec = "KBDVEC| BIOSFLG"
GENERIC is tty  on WINDOW
GENERIC is tty  on WINDOW
GENERIC is tty  on WINDOW
GENERIC is tty  on WINDOW

/* disk device + logical files */

DS0     is dsk  name="ds0"
GENERIC is df
GENERIC is df
GENERIC is df
GENERIC is df
GENERIC is df

/* MS-DOS file interface + logical files */

DOS     is dos  name="dos"
GENERIC is mf
GENERIC is mf
GENERIC is mf
GENERIC is mf

%

/* Configuration and size constants */

#define MEMMARK                         /* enable memory marking        */
#define NPROC   30                      /* number of user processes     */
#define NSEM    100                     /* total number of semaphores   */

#define VERSION "6pc (1-Dec-87)"        /* label printed at startup     */
```

20.3.1 Input To Config

The input to *config* is divided into three sections by occurrences of the separator character '%'. Section 1 contains device *class and type declarations* that identify major device classes (*e.g.*, *tty*) and device types (*e.g.*, *WINDOW*). Section 2 contains *device definitions* that define device names (*e.g.*, *CONSOLE*) and pseudo-devices. Section 3 contains *constant definitions* that select system options (e.g., whether to use memory marking) or override default sizes and parameters (e.g., the number of processes). We will now consider the first two parts of the specification file in detail.

Device class and type declarations define device class names, list the possible device types belonging to a class, and give the default device drivers associated with each type. For example, the declaration:

```
tty
        on BIOS         -i ttyinit   -o ttyopen   -c ioerr
                        -r ttyread   -w ttywrite  -s ioerr
                        -p ttyputc   -g ttygetc   -n ttycntl
                        -iint ttyiin

        on WINDOW       -i lwinit    -o ionull    -c lwclose
                        -r lwread    -w lwwrite   -s ioerr
                        -g lwgetc    -p lwputc    -n lwcntl
```

defines device class *tty* and two associated device types. One type, *BIOS*, is associated with the PC hardware and the ROM BIOS software; the other, *WINDOW*, is a logical device type. If the device class has only one type, the phrase ‘‘*on* unit-name’’ can be omitted. Keywords *-i*, *-o*, *-c*, *-r*, *-w*, *-s*, *-g*, *-p*, *-n*, are abbreviations for the device driver functions *init*, *open*, *close*, *read*, *write*, *seek*, *getc*, *putc*, and *control*; they are used to associate a set of device driver routines with the type. Similarly, keywords *-iint* and *-oint* specify the name of the input and output interrupt drive routines. A driver routine may be omitted; in this case *config* assigns *ioerr* as a default.

Declaring a device type serves two purposes. First, it provides an abbreviation so individual devices can be declared without writing down the set of driver routines again and again. Second it informs *config* that all the devices of a given type belong to the device class, distinguished only by minor device numbers. Devices within the same class receive minor numbers in sequence, starting at zero. Notice that the concept of ‘‘device class’’ has been separated from the concept of ‘‘hardware type,’’ so an arbitrary set of devices can be thought of as a single class even if their hardware or software interfaces differ. Thus, there may be a general device class like *dsk* and only one array of control blocks for *dsk* devices, even though specific disks use slightly different lower-level driver routines.

The second part of the input specification file contains *device definitions*. Each device definition defines one device by giving its name, its class, and information that is not supplied by the class (*e.g.*, the print name and interrupt vector address). If the device class has two or more associated device types, the device definition must also specify the type name with the phrase ‘‘*on* type-name.’’ For example, the declaration of class *tty*, given above, included two sets of drivers, one for type ‘‘BIOS,’’ the other for type ‘‘WINDOW.’’ A device definition for a console of class *tty* and of type *BIOS* is written:

```
CONSOLE  is tty  on BIOS  name="tty"  ivec="KBDVEC | BIOSFLG"
```

The definition for CONSOLE specifies a name "tty" and an input interrupt vector address, ''*KBDVEC|BIOSFLG*.'' Similarly, an output vector address and a port address can be specified, using the forms ''*ovec*=output-vector'' and ''*port*=port-address,'' respectively. Addresses can be specified as decimal, octal, or hexadecimal numbers (in conventional C format), or – as in the example above – as quoted strings. Since the terms *KBDVEC* and *BIOSFLG* are not known to *config*, C preprocessor statements such as

```
#include <bios.h>
```

may precede the list of device specifications in the device definition section.

The information needed to fill in driver fields of the device switch entry for this device is taken from the device class *tty* and type *BIOS*. For example, *pcxconf* will specify that for device *CONSOLE* the driver routine corresponding to *read* is *ttyread*. The names of driver routines can also be specified in the device definition. Values in the device definition override those given in the class and type declaration when both have been supplied for a particular device. For example, the definition:

```
CONSOLE is tty on BIOS name="tty" ivec="KBDVEC|BIOSFLG" -g mygetc
```

specifies the CONSOLE device as before, except that name *mygetc* is used for the *getc* driver instead of the default name *ttygetc*. Such a declaration might be made to test a new version of a driver before replacing the standard version.

Device declarations beginning with ''*GENERIC*'' are used to declare pseudo-devices which are not generally accessible by their device names. The line

```
GENERIC is tty on WINDOW
```

illustrates the definition of a *tty* pseudo-device on the device type *WINDOW*. Since window pseudo-devices are allocated dynamically at run-time, it is not necessary to know their names.

20.3.2 Computation Of Minor Device Numbers

Consider the files that *config* produces. *Conf.h* contains the declaration of the device switch table, and *conf.c* contains the code that initializes it. For a given device, its *devtab* entry contains a set of pointers to the device driver routines that correspond to high-level I/O operations like *open*, *close*, *read*, and *write*. The entry also contains the device names, input and output interrupt vector addresses, and port address. All this information comes from file *pcxconf* in a straightforward way.

The device switch entry also contains a field that gives what we have referred to as the *minor device number*. Minor device numbers are nothing more than integers that distinguish among multiple real devices, all of which belong to the same device class. PC-Xinu drivers use the minor device number as an index into the array of control blocks to

associate a control block with each real device. We have assumed that the minor device numbers are assigned sequentially, starting at zero for each class of device. For example, Figure 20.1 shows how device ids and minor numbers are assigned on a system that has three *tty* devices and two *dsk* devices.

device name	device identifier	device class	minor number
CONSOLE	0	tty	0
DISK0	1	dsk	0
DISK1	2	dsk	1
TERMINAL	3	tty	1
PRINTER	4	tty	2

Figure 20.1 An example of device configuration.

Notice that the three devices with device class *tty* have minor numbers zero, one, and two, even though their device ids happen to be zero, three, and four. Program *config* assigns these minor device numbers based on the definitions in *pcxconf*. To understand how it determines which minor number to assign to which device, we need to look closely at the input.

20.4 Configuring A PC-Xinu System

To configure a PC-Xinu system, the programmer edits file *pcxconf*, adding or changing device information and symbolic constants as desired. Changing constants like *MEMMARK* that are used throughout the system will change the way system routines are compiled, because each routine includes a copy of *conf.h* at compile time.

When run, *config* first reads and records the device class and type declarations. It then reads device definitions, assigns minor device numbers, and produces the source code in *conf.c* and *conf.h*. Finally, it appends definitions of symbolic constants from the third section of the specification file onto *conf.h*, making them available to other routines.

After *config* produces a new version of *conf.c* and *conf.h*, routines that include *conf.h* must be recompiled and grouped into library files. All the PC-Xinu procedures reside in the library file, *xinu.lib*. The entire process is controlled by the utility program *make*, which automatically checks the modification dates and only recompiles procedures when necessary. When *xinu.lib* has been built, configuration is complete. To run an application program, the user compiles it and links it together with the routines in *xinu.lib* and the standard C library routines.

20.4.1 Counting Devices

How does the system know how many devices exist? How does a driver know how many devices it controls? *Config* counts devices as it processes the specification. To pass the information on to other programs, it inserts defined constants in *conf.h* that specify the number of devices of each class and the total number of devices. Constant *NDEVS* is an integer that tells the total number of devices. The device-independent I/O routines use *NDEVS* to test whether a device id corresponds to a valid device. Constants of the form N*xxx* tell the number of devices of class *xxx* (*e.g.*, *Ntty* gives the total number of *tty* devices). The device driver routines use these constants to declare the size of control block arrays.

20.5 System Calls And Procedures

Most hardware includes special instructions that programs use to call system routines. Even the 8088 includes one: *INT*. These instructions take an integer argument that specifies the desired system procedure. When executed, they trap to a dispatch routine just like an exception or device interrupt. The system call dispatch routine must examine the argument, *i*, and pass control to the system procedure that performs function *i*. The chief disadvantage of using the system call mechanism is that all system procedures must be present in memory, independent of whether they will be used at run-time. Loading procedures that are never used is a luxury that may not be viable on small systems. By defining such an interface, the designer places a distinct boundary between user code and system code, preventing users from adding more layers to the design easily.

The advantages of using special instructions to call system procedures are: the hardware mechanism may be more efficient than normal procedure calling mechanisms, and it allows a user program to be loaded into memory without knowing the exact addresses of system procedures. The latter may be important if the operating system needs to load new programs from disk dynamically. In systems that have hardware-assisted memory management, it may be necessary to use system call instructions to keep the user's address space separate from the system's address space.

How difficult is it to change PC-Xinu to use hardware system call instructions? Not difficult at all, although it will force every system procedure to be present in every memory image. To make the change, the designer first chooses a set of system procedures that correspond to system calls. Each of these is assigned a number starting at zero. Calls to system procedures in the user's code must be mapped into a special system call instruction with the appropriate integer argument. This can be done by writing special assembler language routines that perform the mapping at run-time, or by having the compiler recognize the names of system calls and generate special code for them.

Building the system call dispatcher is also quite easy. The code consists largely of a branch table and a small routine that accesses the system call argument and uses it to select the appropriate branch from the table. Control passes to the individual system procedure, which executes and returns to the dispatcher. The dispatcher then returns control to the caller. Usually, the hardware's system call instruction disables interrupts when executed and reenables them when the dispatcher returns.

20.6 Summary

This book has covered the fundamentals of operating systems design, including a look at the basic system components, the design process, and system configuration. We have seen components for: memory management, process management, process coordination, process synchronization, interprocess communication, clock management, device management, device drivers, and a file system. This chapter discussed how a subset of the procedures can be selected to give a fixed set of system calls, and how the software can be reconfigured to accommodate changes in hardware devices.

Abstractions like processes and device-independent I/O are extremely powerful notions, because they make it possible for programmers to deal with complicated computations. Knowing how to design and organize software that supports such abstractions is a skill that has many applications. Of course, some programmers will use this skill to build general-purpose operating systems for new machines, but others will apply the same skill to design such things as run-time systems for concurrent programming languages, database systems that support concurrent transactions, and special-purpose systems for microprocessors. Independent of the application, the same general design procedure applies – by now the reader should have a firm grasp of that design procedure and the system it will ultimately produce.

FOR FURTHER STUDY

Few books or papers consider the topic of hardware or operating system configuration in detail. The memo by Kridle et. al. [1983] is a good example of the informal way configuration information is often disseminated; it also assumes that readers are more interested in configuring hardware for the operating system than configuring the operating system to fit their hardware.

Much has been written on the topic of portability. Miller [1978] discusses a portable UNIX system, while Cheriton et. al. [1979] considers a system designed to be portable.

EXERCISES

20.1 Replace part of the tty driver and reconfigure PC-Xinu to use your routine.

20.2 Find out how other systems are configured. Read about IBM's SYSGEN procedure (See if you can find someone with first-hand experience).

20.3 PC-Xinu reconfiguration takes much longer than necessary if every program is recompiled whenever *pcxconf* changes. Write a *config* program that separates constants into several different include files to eliminate unnecessary recompilation.

20.4 Discuss whether a configuration program is worthwhile. Include some estimate of the extra effort required to make a system easily configurable. Remember that a programmer is likely to have little experience or knowledge about a system when it is first configured.