CHAPTER 5

SYSTEM MANAGEMENT

5.1 THE SYSTEM MANAGER

Every computer system - even a single-user system - should have a user designated as the System Manager. The System Manager has responsibility for:

a) Allocating user IDs, groups, and passwords.

b) Creating and amending the system startup procedure.

c) Formatting and maintaining the system disk (the main disk drive).

d) Maintaining backups and archives of the system disk.

The System Manager is always the super-super user - user zero of group zero. Many system functions are only available to the super user group, or even only to the super-super user. Modules created by the super-super user may only be loaded from files owned by the super-super user. This gives protection against super-user facilities being used illegally.

The initial user ID and group is 0.0. Therefore on a single user system the user is always the super-super user, and has access to all facilities. When a user is logged on, the user ID and group are taken from the password file, according to the user's name and password. So on a multi-user system users can be prevented from unauthorized access to resources, or modification of other users' filing systems.

5.2 THE FILING SYSTEM

Disk files have read, write, and execute permissions for private and public use. For historical reasons the disk file manager ('RBF') does not support separate group permissions, unlike the other components of the operating system. The private permissions relate to use by the file owner *and* other members of the same group. The public permissions relate to all other users. Therefore a file that has only private read and write permissions can only be read, modified, or deleted by members of the file owner's group. Note that the same historical reasons - compatibility with the OS-9/6809 filing system - dictate that the disk filing system can only store the user ID and group as byte values, whereas elsewhere they are stored as word values. This imposes a practical limit of 255 on user IDs and group numbers.

To modify the attributes (permissions) of a file - for example, using the **attr** utility - a user must have write permission for the file. The System Manager has the special ability to modify the attributes of any file. Therefore by doing so he can gain access to any file, and read, modify, or delete it. Note that directories are files, and have file attributes. If a user does not have permission to access a directory, he cannot access any of the files in the directory.

The **dcheck** utility checks the integrity of the filing structure on a disk, and can make simple repairs. Note that sectors found to be bad during the verify pass of disk formatting will be reported as "allocated but not in the file structure".

5.3 THE PASSWORD FILE

The password file is the file 'SYS/password' in the root directory of the "initial mass storage device" specified in the **init** module (usually '/dd' or '/h0'). It should only have permissions for private read and write - for the System Manager (group zero) - to prevent modification by other users. The password file is a simple text file, that can be created and modified by any text editor, such as the **umacs** screen editor. There is a single line entry for each user. The elements on the line are separated by commas. For example:

```
Henry,penguin,1.3,100,.,/dd/HENRY,shell -p="@Henry: "
```

`Henry`	User name.
`penguin`	User password.

`1.3`	User group number and user ID (separated by a full stop '.') – user 3 of group 1.
`100`	Initial process priority.
	Initial execution directory – no change from current.
`/dd/HENRY`	Initial data directory.
`shell -p="@Henry: "`	Initial command line to execute – fork the **shell** command line interpreter, with the '-p' option to set the prompt string.

The initial directories are relative to the directories of the **login** utility processing the password file, which in turn will have inherited the directories of the **tsmon** utility that normally monitors a terminal and forks **login** when the [CR] key is pressed.

When assigning process priorities, bear in mind that a small absolute difference in priorities has a large effect on the allocation of CPU time.

5.4 SYSTEM STARTUP

After going through its coldstart procedure, the kernel forks up the program whose name is given in the **init** configuration data module – usually **sysgo**. The **sysgo** supplied by Microware forks a **shell** to execute a text file 'startup', and then goes into an endless loop forking up a **shell** and waiting for it to die.

The source of **sysgo** (in assembly language) is supplied in the file 'SYSMODS/sysgo.a', to allow customization for special applications.

The 'startup' file – located in the root directory of the initial mass storage device specified in the **init** module – is the place to put commands to load additional modules not present in the boot file, request the date and time (if the system does not have a battery-backed clock chip), and fork any required incarnations of **tsmon**, for multi-user systems. For example:

```
-nt
* The next line is needed if the system does not have a
* battery-backed clock chip:
setime </term
* Load a 'dd' alias device descriptor for the hard disk:
```

```
load BOOTOBJS/dd.h0
chx /dd/CMDS
chd /dd
mfree
echo Starting up /t1, /t2, and /t3
echo "Hit RETURN to log on" ! tee /t1 /t2 /t3
tsmon /t1 /t2 /t3 &
* The next line asks for a logon on the system console.
* In this case the 'startup' shell never terminates:
tsmon /term
```

5.5 THE .LOGIN FILE

When **login** (forked by **tsmon** when [ENTER] is hit) forks the initial command line shown in the password file, it does so with a special parameter that causes the shell to look for a file '.login'. (Note that all files whose names start with '.' are normally hidden from **dir**, but may be viewed with the '-a' option). If the file is present in the current directory the **shell** executes the command lines in the file before giving the user the prompt.

This is a very important mechanism, as it allows environment variables to be set, and the default directories to be changed. For example:

```
setenv TERM vt100
setenv _sh 0
setenv HOME ../PROJECTS/PENGUINS
chd
echo "You are in: " -r
pd
```

The environment variable **TERM** is used by screen-orientated programs, such as **umacs**, to determine the type of terminal that is being used. The environment variable '_sh' sets the initial "shell level", used by the **shell** when the first character of the prompt string is '@'. When a **shell** is forked, it looks for this environment variable. If it exists, the **shell** increments its value, and substitutes it for the '@' character in the prompt string (unless it is zero, in which case the '@' is simply not displayed). This helps keep track of **shell**s forked from other programs, such as **umacs** or **debug** - the prompt string appears with a leading number if the **shell** has been forked from within another program.

5.6 DISK FORMATTING

Disk formatting is achieved using the **format** utility. **format** has three phases:

1) Physical format.
2) Verify.
3) Logical format.

The physical format phase issues a format request to the device driver, to physically rewrite the sectoring information on the disk. The verify phase reads all the sectors on the disk, to determine which are faulty. The logical format builds the disk identification sector, the allocation bitmap, and the root directory. It effectively "forgets" all files previously existing on the disk.

The physical format and verify phases may be omitted. However, it is recommended that the verify pass always be performed, unless you are certain the disk has no errors, or the drive has automatic defect handling (all modern SCSI hard disk drives have).

The **format** utility '-c' option allows the specification of a "cluster size" other than the default of one (it must be a power of two). The cluster size is the number of sectors per bit in the allocation bit map, and is the minimum allocatable block of disk space. RBF uses the bulk of the File Descriptor sector of a file for the file's segmentation table. Each entry takes 5 bytes (a 24-bit start logical sector number, and a 16-bit number of sectors). For example, if the sector size is 256 bytes, the table can accommodate 48 entries, each referring to a maximum of 65535 sectors.

However, there is a further restriction on allocation of space in a file. RBF will not allocate a segment for which the allocation would cross a bit map sector boundary. This limits a segment to a maximum of "eight times the sector size" clusters. For example, if the sector size is 256 bytes, a segment cannot exceed 2048 clusters. At this sector size a file cannot have more than 98304 clusters (approximately 24Mbytes at one sector per cluster). Therefore it is recommended that larger disks be formatted with a cluster size greater than 1 - approximately "disk size in Megabytes" divided by 10, if the sector size is 256 bytes.

5.7 INSTALLING A BOOT FILE

On systems where the operating system is not in ROM, the boot program reads a file known as the boot file from disk. This file contains at least the basic operating system. To simplify the boot program, special information is put on the disk to identify the boot file. This is done using the utility **os9gen**.

The disk identification sector (sector zero) of each disk contains the start sector number and size (in bytes) of the boot file on the disk. In versions of OS-9 earlier than 2.4, to simplify the booting procedure the boot program assumes that the boot file is contiguous. It calculates the number of sectors in the boot file, from the size given in sector zero, and simply reads that many sectors starting at the sector number given in sector zero. Because the boot file size in the identification sector is a 16-bit word, the boot file size is limited to 64k bytes. (This is a historical limitation from OS-9/6809).

From OS-9 version 2.4 onwards, Microware offers the implementor an alternative set of boot ROM example source code, known as 'CBOOT' (because it is written in C). This code has the ability to read any file, using the segmentation information in the file's File Descriptor sector. This uses an alternative form of the information in sector zero. The "boot file size" field is set to zero, indicating that the alternative form is being used. The "boot file start sector number" is the sector number of the File Descriptor sector for the boot file. The boot program reads the File Descriptor sector, and from it takes the segmentation table, which allows it to read a boot file of any size, even if the file is not contiguous on disk.

If boot file sector number in sector zero is zero, the disk has no boot file installed. The **os9gen** utility is used to install a boot file on a disk. It can simply set the values in sector zero to point to a file already on the disk, or alternatively build the file as well, by merging other specified files. If the older contiguous boot file form is used, **os9gen** also checks that the file is contiguous.

The boot file consists simply of modules merged together. The kernel should always be the first module, as many boot programs assume it will be. Modules which cannot fit in the boot file (if the older contiguous form is used, limited to 64k) may be loaded in the 'startup' file. **os9gen** has the following options:

-b=<n>	Set size of memory buffer in which to build boot file - in kbytes.
-e	Generate later non-contiguous boot file (can be greater than 64k bytes in size).
-q=<path>	Don't build boot file, just set sector zero values to point to existing file on boot disk.
-r	Clear sector zero boot file fields - makes disk non-bootable.

-x	Pathlists to files for building the boot file are relative to the current execution directory.
-z[=<path>]	Take the list of pathlists to build the boot file from standard input (or a file), rather than from command line parameters.

Use the '-b' option to inform **os9gen** of the expected maximum size of the boot file. The given value must not be less (in kbytes) than the size of the boot file, and cannot be greater than 64 unless the '-e' option is specified.

If the boot program supports non-contiguous boot files, it is much easier to use this facility ('-e' option), rather than being concerned about keeping the size of the boot file below 64k, and ensuring that it is contiguous (although **os9gen** will ensure this if at all possible). In this case the '-q' option of **os9gen** can be used to set the sector zero values to point at any file into which the boot modules have been merged.

If the earlier (contiguous) boot file form is used, **os9gen** must merge the boot file itself, as this gives a much greater likelihood that the file will be contiguous. In this case it is advisable to use the '-z' option, requesting **os9gen** to read the file names to merge from a text file you have created. This makes it much easier to create boot files at a later date, perhaps with modifications for special purposes. If such a file has not been provided with your system, use the **mdir** utility with the '-e' option to display the module directory. The modules in the boot file will be listed first, and there will be a distinct break in the sequence of addresses between the last module in the boot file and the first module loaded after booting.

os9gen and **format** require that the device descriptor for the disk drive has formatting enabled (this is one of the options flags in the device descriptor). Usually this is not the case for hard disk descriptors, to prevent unintentional or unauthorized formatting, and a special descriptor (often **h0fmt** or **fh0**) must be explicitly loaded and used. For example:

```
$ load BOOTOBJS/h0fmt
$ chd /dd/CMDS/BOOTOBJS
$ os9gen /h0fmt -z=bootlist -b=100 -e
```

The '-q' option of **os9gen** requires that the device containing the current data directory, or the device name in the pathlist if a full pathlist is given for the file, be the same as the target device name (and it is letter case sensitive). For example:

```
$ load BOOTOBJS/h0fmt
$ chd /h0fmt
$ os9gen /h0fmt -q=oldboot
```

5.8 ARCHIVING

Computer data (such as program source files) is usually very valuable, if only because of the time that went into creating it. The loss of some data can have catastrophic effects for a business. Current development projects may be set back by months, and finished products may no longer be supportable. Hard disk drives are by no means infallible, so it is obviously very important to "back up" the computer's hard disk regularly, to minimize the loss if the hard disk does fail.

Yet it is surprising how few development systems are regularly backed up onto tape or other archiving medium. Generally this is because most development systems are not originally specified with a tape drive, and it is often difficult to convince management to buy one as an afterthought. If the computer does not have a tape drive, the only alternative archiving medium usually available is floppy disk, and archiving a hard disk with perhaps 100Mbytes of files onto floppy disks is a tedious and time-consuming task that is generally only undertaken once every few months, if ever.

It is for this reason that a system manager should be appointed for the computer even before it is purchased. He will then have the incentive to ensure that the computer is purchased with a tape drive already installed. If you already have a computer, and it does not have a tape drive (or other high capacity off line storage, such as optical disk), I strongly recommend that you purchase one.

How frequently you back up the hard disk depends on the rate at which you generate valuable data - that is, how long it would take you to regenerate the data created since the last backup if the hard disk fails just before the next backup. Once a week is usually sufficient. Use at least two tapes, and cycle between them, in case the hard disk drive fails while you are generating the backup tape. It is not necessary to save all of the hard disk files to tape at every back up. Instead, you can save only the files that have changed since the last complete back up. This is known as an "incremental" back up. For example, you might do a complete back up every three months, and an incremental back up every week. It is advisable to keep the backup tapes at a different site from the computer system, in case of fire or burglary.

A tape drive is not only useful for backing up the hard disk to protect against hardware failure. Because the tape is removable (unlike most hard disks) it provides "off line storage". That is, any amount of data can be stored, by using additional tapes. To access the data the appropriate tape must be placed in the tape drive. This allows the computer to generate, save, and access unlimited amounts of data, even though it only has immediate rapid access to perhaps 200Mbytes, on the hard disk drive. The saving of data that is not presently required (but may be required in the future), so that space can be freed on the hard disk, is known as archiving. Because the archived data is no longer available on the hard disk it is important to keep a careful written record of what is on each tape. It is also useful to store a printed directory listing with the tape.

Microware provide the **fsave** and **frestore** utilities for backing up (and archiving) and retrieving files. The use of these utilities is described at length in the OS-9 User's Manual. **fsave** and **frestore** will work with any form of storage medium, and provide incremental back up and interactive retrieval facilities.