PLEXUS

Sys5 UNIX Administrator's Guide

98-05076.1 Ver. C May, 1986

Sys5 UNIX Administrator's Guide

98-05076.1 Ver. C May, 1986

PLEXUS COMPUTERS, INC.
3833 North First Street
San Jose, CA 95134
408/943-9433

Printed in the United States of America

CONTENTS

7. LP SPOOLING

8. VIRTUAL PROTOCOL MACHINE

9. REMOTE JOB ENTRY

10. SYSTEM ACTIVITY PACKAGE

11. UUCP ADMINISTRATION

1. INTRODUCTION

The *Administrator Guide* is a reference volume for those who administer a UNIX system. The guide should be used to supplement the information contained in the *Sys5 UNIX User Reference Manual*, the *Sys5 UNIX Programmer Reference Manual*, and the *Sys5 UNIX Administrator Reference Manual*. The following paragraphs contain a brief description of each chapter of the guide.

The chapter "ADMINISTRATIVE ADVICE" contains helpful advice and suggestions for administrators of the UNIX system.

The chapter "SETTING UP THE UNIX SYSTEM" describes the setup procedures for installing the Plexus Sys5 UNIX operating system.

The chapter "AUTO CALL FACILITY INSTALLATION" outlines the installation procedures for a properly installed (software) automatic call-up facility.

The chapter "UNIX SYSTEM ACCOUNTING" descibes the structure, implementation, and management of the accounting system.

The chapter "FILE SYSTEM CHECKING" describes the file system check program (**fsck**) of the UNIX system. **Fsck** audits and interactively repairs inconsistency in the file system.

The chapter "LP SPOOLING SYSTEM" defines the **LP** program and describes the role of the LP administrator in performing restricted functions and overseeing the smooth operation of **LP**.

The chapter "VIRTUAL PROTOCOL MACHINE" defines the Plexus virtual protocol machine (VPM) and describes the implementation and the administrative duties.

The chapter "UNIX SYSTEM REMOTE JOB ENTRY" defines the UNIX system remote job entry (RJE) and describes the administrative duties of an RJE administrator.

The chapter "UNIX SYSTEM ACTIVITY PACKAGE" describes the design and implementation of the UNIX system activity package. The package reports UNIX system-wide statistics.

The chapter "UUCP ADMINISTRATION" describes how a **uucp** network is set up, the format of the control files, and administrative procedures.

2. ADMINISTRATOR'S ROAD MAP

This chapter contains administrative advice based on the experience and suggestions of many system administrators. Other reasonable approaches may be taken to solve many of the problem areas described.

Getting started as a UNIX system administrator is hard work. There are no real shortcuts to a working knowledge of the system. The system administrator will need time for reading, studying, and hands-on experimenting. The system administrator should not go "live" with the system until he/she have had several weeks to learn the job and get the initial hardware quirks ironed out.

The administrator should be familiar with a lot of the distributed documentation. The "Introduction" and "How to Get Started" sections of the *Sys5 UNIX User Reference Manual* as well as all of the sections of the *Sys5 UNIX Administrator Reference Manual* should be studied.

Throughout this chapter, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX Administrator Reference Manual.* References to entries of the form **name**(N), where "N" is the number 1 or 6 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX User Reference Manual.* If "N" is a number 2 through 5 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX Programmer Reference Manual.*

In these manuals, pay special attention to: **acct**(1M), **checkall**(1M), **chmod**(1), **chown**(1), **config**(1M), **cpio**(1), **date**(1), **dcopy**(1M), **df**(1M), **don**(1M), **du**(1), **ed**(1), **env**(1), **errpt**(1M), **find**(1), **format**(1M), **fsck**(1M), **fuser**(1M), **kill**(1), **mail**(1), **mkdir**(1), **mkfs**(1M), **ncheck**(1M), **ps**(1), **rm**(1), **rmdir**(1), **shutdown**(1M), **stty**(1), **su**(1), **sync**(1M), **time**(1), **vcf**(1M), **volcopy**(1M), **wall**(1M), **who**(1), and **write**(1); **acct**(4); all of section 7; and **crash**(8).

2.1 DISK FREE SPACE

Making files is easy under the UNIX operating system. Therefore, users tend to create numerous files using large amounts of file space. It has been said that the only standard thing about all UNIX systems is the message-of-the-day telling users to clean up their files. Administratively, both free disk blocks and free inodes (UNIX system talk for file headers) can be a problem. If the free inode count falls below 100, the system spends most of its time rebuilding the free inode array. If a file system runs out of space, the system prints "nospace" messages and does little else. To avoid problems, the following start-of-day free counts should be maintained:

- The file system containing */tmp* (temporary files):
  - 16-user system: 1500 free kilobytes (KB).
  - 40-user system: 3000 free KB.
- The file system containing */usr*:
  - 3000 to 6000 free KB depending on load.
- Other user file systems:
  - 6 to 10 percent free depending on user habits (3000 KB minimum).

This brings up an associated problem: how big should file systems be? The preference is to set aside space on each drive for a copy of root/swap and use the rest of the pack for a single file system. However, if you have user groups that fight over disk space, it may be better to split them up arbitrarily (i.e., divide a pack into more than one file system). If different disk drives are set up with differing cylinder partitions between file systems, it will eventually lead to an operational blunder.

2.2 A FEW WORDS ABOUT SYSTEM TUNING

A file system reorganization can help throughput but at the expense of down time. If the reorganization is done during nonprime time, it can help.

If normal shutdown and filesave procedures are used, the file system check program [**fsck**(1M), **–S** option] will help keep the disk free list in reasonable order. Try to keep disk drive usage balanced. If there are more than 20 users, performance will be improved if the root file system (*/bin*, */tmp*, and */etc*) has a drive of its own. If there is a noisy modem (poorly executed do-it-yourself null-modem) or a disconnected modem cable, the UNIX system will spend a lot of CPU time trying to get it logged in. A random check of systems uncovers a lot of this going on.

2.3 WHY A SPARE DISK DRIVE IS NEEDED

Without a spare disk drive, the system will be *down* when a drive is *down*. Also, without the spare drive, it is difficult to reorganize file systems or to save and restore user files.

2.4 PROTECTING USER FILES

Users, especially inexperienced ones, occasionally remove their own files. Open files are sometimes lost when the system crashes. Once in a great while, an entire file system will be destroyed (picture a disk controller that goes bad and writes when it should read). Here is a suggested file backup procedure:

- Each day copy all user file systems to backup tapes. Keep these tapes 3 to 5 days before reusing them.
- Once a week copy each file system to tape. Keep weekly tapes for 8 weeks.
- Keep bimonthly tapes "forever" (they should be recopied once a year).

The most recent weekly tapes should be kept off premises. The other tapes should be in a fireproof safe if available and not too expensive.

When the UNIX system goes down, active files can get scrambled. Your users will not want to start the day over every time the system fails. In addition to good backup, you *must* have file system patching expertise available (on-site or on-call). If the system is ever rebooted for general use without first checking the file systems, terrible things will happen. Study **checkall**(1M), **fsck**(1M), and **crash**(8) as well as the "File System Checking" chapter for more information.

2.5 FILE SYSTEM BACKUP PROGRAMS

The following backup programs are distributed:

- **Find/cpio**: The UNIX system is distributed in **cpio** format. The **–cpio** option of the **find** command can be used for saving only those files that have changed or been created over a definite period.
- **Volcopy**: Physical file system copying to disk or tape. For those with a spare drive, **volcopy** to disk provides convenient file restore and quick recovery from disk disasters. Tape **volcopy** provides good long-term backup because the file system can be read-in fairly quickly, mounted, and browsed over. Disk and tape **volcopy** are generally used together for short- and long-term backup. Note that a **volcopy** from a mounted file system may result in an inconsistent copy (files being written at the time can contain invalid data).

Figure 2-1 summarizes attributes of these programs. In the figure, the file system size is 65,500 KB in all cases; times are in minutes and are relative; judgments are subjective.

	FIND/CPIO	VOLCOPY (DISK)	VOLCOPY (TAPE)
Full dump time	40	2	15
Incremental dump time	7	-	-
Full restore time	80	2	15
Incremental restore time	10	-	-
Ease of restoring:			
one file	fair	good	fair
a directory	fair	good	good
scattered files	poor	good	good
full restore	fair	very good	good
Needs tape drive	yes	no	yes
Needs spare file system (two CPUs can share)	-	yes	-
Maintains pack/tape labels	no	yes	-
Handles multireel tape	yes	-	yes
512 KB per record	1.10	88	10
Interactive (i.e., ties up console)	yes	yes	yes
May require separate I/D space	no	no*	no

* KB per record are cut to 22 without separate I/D space.

Figure 2-1. File System Backup Programs

The spare disk drive is strongly recommended. The speed and convenience of **volcopy** are by no means the only advantage of a spare drive. It is strongly recommended that the administrator modify the */etc/filesave* and */etc/checklist* files to meet the operational needs and update the local operator's manual accordingly. Remember, the more the administrator automates and documents operational procedures the less downtime will be encountered.

2.6 CONTROLLING DISK USAGE

Try to maintain the start-of-day counts recommended. Watch usage during the day by executing the **df**(1) command regularly.

The **du**(1) command should be executed (after hours) regularly (e.g., daily), and the output kept in an accessible file for later comparison. In this way, users rapidly increasing their disk usage may be spotted. This can also be accomplished by running the accounting system's **acctdusg** program [see **acct**(1M)] as shown in "The Sys5 UNIX Accounting" chapter.

The **find**(1) command can be used to locate inactive (or large) files. For example:

find / –mtime +90 –atime +90 –print >somefile

records in *somefile* the names of files neither written nor accessed in the last 90 days.

The administrator will also have to balance usage between file systems. To do this, user directories must be moved. Users should be taught to accept file system name changes (and to program around them—preferably ahead of time). The user's login directory name (available in the shell variable **HOME**) should be utilized to minimize pathname dependencies. User groups with more extensive file system structures should set up a shell variable to refer to the file system name (e.g., *FS*).

The **find**(1) and **cpio**(1) commands can be used to move user directories and to manipulate the file system tree. The following sequence is useful (it moves the directory trees *userx* and *usery* from file system *filesys1* to file system *filesys2* where, presumably, more space is available):

```
cd /filesys1
find userx usery –print | cpio –pdm /filesys2
#  Make sure new copy is OK.
#  Change userx and usery login directories
#    in the /etc/passwd file.
#  Notify userx and usery via mail(1) that
#    they have been moved and that pathname
#    dependencies in their .profile and shell
#    procedures may need changed.  See the
#    discussion on $HOME above.
rm –rf /filesys1/userx /filesys1/usery
```

When moving more than one user in this way, keep users with common interests in the same file system (these users may have linked files) and move groups of users who may have linked files with a single **cpio** command (otherwise linked files will be unlinked and duplicated).

2.7 REORGANIZING FILE SYSTEMS

There is a new file system reorganization utility called **dcopy**(1M). On an otherwise idle system, a reorganized file system has almost twice the I/O throughput of a randomly organized file system. This applies to file copying, **finds**, **fscks**, etc. **Dcopy** can take up to 2.5 hours to initially reorganize (copy) a large file system. During reorganization the system can be up, but the file system being copied must be unmounted.

For those who can afford the operator time, root reorganization once a week (requires system reboot) and user file system reorganization once a month will improve system performance. **Dcopy** is an interim step.

2.8 KEEPING DIRECTORY FILES SMALL

Directories larger than 5K bytes (320 entries) are very inefficient because of file system indirection. A UNIX system user once complained that it took the system 10 minutes to complete the login process; it turned out that his login directory was 25K bytes long, and the login program spent that time fruitlessly looking for a nonexistent *.profile* file. A large */usr/mail* or */usr/spool/uucp* directory can also really slow the system down. The following will ferret out such directories:

```
find / –type d –size +10 –print
```

Removing files from directories does not make the directories get smaller (the empty directory entries are available for reuse). The following will "compact" */usr/mail* (or any other directory):

```
mv /usr/mail /usr/omail
mkdir /usr/mail
chmod 777 /usr/mail
cd /usr/omail
find . –print | cpio –plm ../mail
cd ..
rm –rf omail
```

2.9 ADMINISTRATIVE USE OF "CRON"

The program **cron**(1M) is useful in the administration of the system; it can be used to:

- Turn off the programs in directory */usr/games* during prime time.
- Run programs off-hours:
  - accounting;
  - file system administration;
  - long-running, user-written shell procedures.

2.10 WATCH OUT FOR FILES AND DIRECTORIES THAT GROW

Most of the below files are restarted automatically by entries in */etc/rc* at system reboot.

- Accounting files:
  - */etc/wtmp*—login information; grows extremely fast with terminal line difficulties; use **acctcon**(1M) to determine the offending line(s).
  - */usr/adm/pacct*—per process accounting records; gets big quickly; monitored automatically by **ckpacct** from **cron**(1M).
  - */usr/lib/cron/log*—status log of commands executed by **cron**(1M); also watch this file for error messages from the programs being executed in */usr/spool/cron/crontab/**.
  - */usr/adm/errfile*—hardware error logging info; also read login **adm**'s mail periodically.
  - */usr/adm/ctlog*—a log of the people who use **ct**(1C) command.
  - */usr/adm/sulog*—a log of those who execute the superuser command.
  - */usr/adm/Spacct*—process accounting files left over from an accounting failure; remove these files unless the accounting files that failed are to be rerun.
- Other files:
  - */usr/spool*—spooling directory for line printers, **uucp**(1C), etc., and whose subdirectories should be compacted as described above.

2.11 ALLOCATING RESOURCES TO USERS

A prospective user should first obtain authorization to use the system and then apply for a login by providing the following information to the "system administrator":

- User's name.
- Suggested login name (not more than eight characters, beginning with a lowercase letter and not containing special or uppercase letters).
- Relationships to other users (this influences the choice of the file system).

- Estimate of required file space (this also influences the choice of the file system) and connect hours. This aids in hardware growth planning.

Users must have passwords with at least six characters. (Only the first eight characters are significant.) Also, every password must have at least two alphabetic characters and one numeric or special character. The password must differ from the user's login name and any reverse or circular shift of it. Refer to **passwd**(1) and **passwd**(4) for more information on password selection and password aging.

2.12 THE MATTER OF ACCOUNTING AND USAGE

You should run the accounting programs even if there is not a "bill" for service. Otherwise, users' habits (especially *bad* habits) will be a mystery to you. Accounting information can also help you find performance bottlenecks, unused logins, bad phone lines, etc.

2.13 DIAL-LINE UTILIZATION

If prime-time dial-line utilization gets much over 70 percent, users will start to encounter busy signals when dialing in. This, in turn, will lead to "line hogging". The only solutions are to acquire more dial-up ports, get a larger (another) machine, or lessen the number of users. Manual policing will help some, but "automatic" policing will be *invariably* subverted by users.

2.14 "BIRD-DOGGING"

When the system is busy (lines busy and/or slow response), someone should determine why this is so. The **who**(1) command lists the people logged in. The **ps**(1) command shows what they are doing. Unfortunately, **ps** operates from heuristics that can consistently fail to report certain processes in a busy system. That is, one must be careful about hanging up an apparently inactive line. The **acctcom**(1M) command can read the process accounting file */usr/adm/pacct* backwards from the most recent entry. It will print entries for selected lines or login names.

2.15 TERMINALS

Do not use uppercase only terminals. Use full-duplex, full-ASCII asynchronous terminals. Hardware horizontal tabbing is very desirable because it increases output speed and lowers system overhead. A fair proportion of the terminals should provide for correspondence-quality hard copy output to take advantage of the UNIX system word processing capabilities; see **term**(5).

2.16 SECURITY

The current UNIX operating system is not tamperproof. The system administrator cannot keep people from "breaking" the system but can usually detect that they have done so. The following command will mail (to root) a list of all "set user ID" programs owned by *root* (superuser):

find / –user root –perm –4100 –exec ls –l {} \; | mail root

Any surprises in *root*'s mail should be investigated. In dealing with security,

- Change the superuser password regularly. Do not pick obvious passwords (choose 6-to-8 character nonsense strings that combine alphabetics with digits or special characters).
- Dial ports that do not *require* passwords usually cause trouble.
- The **chroot**(1M) and **su**(1) commands are inherently dangerous as are *group* passwords.
- Login directories, *.profile* files, and files in */bin*, */usr/bin*, */lbin*, and */etc* that are writable by others than their respective owners are security weak spots; police the system regularly against them.
- Remember, no time-sharing system with dial ports is really secure. **Do not keep top secret information on the system.**

2.17 COMMUNICATING WITH THE USERS

The directory */usr/news* and the **news**(1) command are provided as a way to get "brief" announcements to your users. More pressing items (one-liners) can be entered in the */etc/motd* (message of the day) file; *motd* and (new to the user) *news* are announced at login time.

To reach users who are already logged in, use the **wall**(1M) (write all) command. Do not use **wall** while logged-in as superuser, except in emergencies.

The */usr/news* directory should be cleaned out once a week by removing everything older than 2 months. It has been found that on most systems a file in */usr/news* will reach 50 percent of the users within a day and over 80 percent within a week; *motd* should be cleaned out daily.

2.18 TROUBLESHOOTING

It would be easy to write a book on troubleshooting. The following is some effective advice in dealing with troubles. In dealing with hardware support service personnel,

- Be sure that the contractor agrees to get along with the UNIX software before you take out a hardware service contract ("It's the hardware," says you; "It's the software," says the hardware service contractor).
- Keep on top of problems. Find out about any such scheme that your contractor may have and make them prove that it is being followed. Remember that an unreported problem is getting no priority at all. If a problem persists, escalate it through the contractor's local management chain; it may also be effective to complain to the contractor's sales representative.
- A support service agreement should be arranged, to allow for timely and personal assistance on all technical questions and problems. Arrange for preventive maintenance, noncritical repair, and add-on installation work to be done before or after prime time.
- Know the details of the support service offering applicable to the installation. In particular, make sure that preventive maintenance is scheduled in advance and that it is completed.
- A "site log" should be maintained for the hardware. All troubles should be recorded in the log by the support service personnel and/or the operating personnel.
- Run error logging and maintain console sheets. Make sure error messages are shown to support service personnel.
- Take core dumps after system crashes and have them available for support service personnel.
- Keep records of downtime and make sure that support service personnel know about them.

Telephone problems are most apt to occur when rearranging or adding equipment. Occasionally, central office, trunking, or modem failures occur. In dealing with the telephone services vendor,

- Be specific with repair operators. Tell the operators that the trouble involves *data* equipment.
- If the first call fails to get results, ask for the "supervisor" on the second call, and if necessary, escalate further to get the problem solved.

Some of the obvious problem areas are:

- Disk Drives—Remember that preventive maintenance of disk drives is very important. Make sure that the support service personnel who service the hardware see the error-logging printouts and console error messages produced by the UNIX system (and that they understand them). Disk failure can ruin a file system. The *only* defense is to make a complete, daily file backup! (See the part "Protecting User Files".)
- Dial Ports—In the dial-in interface area, there is room for finger-pointing among all involved vendors. Check for obvious things such as is the system in "multiuser" mode, is the */etc/inittab* file OK, or are any cables loose (*both* ends)? In some telephone offices, trunk hunting is based on 10-number groups. Hunting *between* such groups can fail independently of anything else. The possibilities for trouble are many. Figure 2-2 attempts to describe some alternatives; it is meant primarily for users of the ICP and ACP devices used in asynchronous mode. As an example of the format, (vertical) Rule 3 reads: "If line rings *and* ring light shows *and* computer does *not* answer *and* switching the modem solves the problem, then it is likely to be a telephone company problem; also, busy out that line."

Rules:	1	2	3	4	5	6	7	8	9	0
Condition:										
Line rings	N	Y	Y	Y	Y	Y	Y	Y	Y	Y
Ring light shows on telephone console	–	N	Y	Y	Y	Y	Y	Y	Y	Y
Computer answers	–	–	N	N	Y	Y	Y	Y	Y	Y
Login message received on terminal	–	–	–	–	N	N	Y	Y	Y	Y
Switching modem solves problem	–	–	Y	N	Y	N	–	–	–	–
User can login	–	–	–	–	–	–	N	N	N	Y
Telephone console shows data received	–	–	–	–	–	–	Y	Y	N	–
Problem affects whole ICP or ACP	–	–	–	–	–	–	Y	N	–	–
Diagnosis and/or Action:										
No problem	–	–	–	–	–	–	–	–	–	X
Processor hardware problem likely	–	–	–	X	–	X	X	–	–	–
Telephone problem likely	X	X	X	–	X	–	–	–	X	–
May be a problem with user's terminal	–	–	–	–	–	–	–	X	–	–
Busy out bad line(s)	X	X	X	X	X	X	X	–	X	–

Figure 2-2. Asynchronous Line Problems

- Synchronous Ports—The following is a list of potential trouble spots:

  —The UNIX system software.

  —Interface device (e.g., KMC11B).

  —Cable to the modem.

  —The modem.

  —The communications line.

  —Other modem.

  —Other cable.

  —Other interface device.

  —Other system's software.

2.19 NULL MODEM WIRING

Improperly wired null modems can cause spurious interrupts, especially at higher baud rates. A single bad modem on a 9600-baud line can waste 15 percent of your CPU power. The following (symmetrical) wiring plan will prevent such problems:

pin 1 to 1
pin 2 to 3
pin 3 to 2
strap pin 4 to 5 in the same plug
pin 6 to 20
pin 7 to 7
pin 8 to 20
pin 20 to 6 and 8
ground unused pins

3. SETTING UP UNIX

This chapter describes the load and upgrade procedures for the Plexus implementation of the UNIX Sys5 operating system. The Plexus UNIX Sys5 consists of:

- a release tape (cartridge or 9-track),
- this release document.

The Release Tape comprises 22 files. Files 0-19 are blocked at 1024 bytes per record; file 20 is blocked at 10240 bytes per record; files 21 through the end of the tape are blocked at 5120 bytes per record. Most of the tape files and standalone programs are on the UNIX operating system release tape. These are for backup and emergency purposes, in case the disk copies of the standalones become inaccessible and you need to run the standalone programs from tape. File 20 is a **dump** of the files that make up the bootstrap. Files 21, 22, and 23 are **cpio** format files comprising the full release. File 21 contains everything except the **/usr/catman** and **/usr/man** directories. These directories are in files 22 and 23.

4. ACU

For information about Automatic Calling Unit (ACU), please reference the "UUCP ADMINISTRATION" chapter of this guide. ACU is discussed in the section on Lines File, under the listing *call-device*.

5. UNIX SYSTEM ACCOUNTING

The UNIX system accounting provides methods to collect per-process resource utilization data, record connect sessions, monitor disk utilization, and charge fees to specific logins. A set of C language programs and shell procedures is provided to reduce this accounting data into summary files and reports. This chapter describes the structure, implementation, and management of this accounting system, as well as a discussion of the reports generated and the meaning of the columnar data.

Throughout this chapter, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX Administrator Reference Manual.* References to entries of the form **name**(N), where "N" is the number 1 or 6 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX User Reference Manual.* If "N" is a number (2 through 5) possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX Programmer Reference Manual.*

The following list is a synopsis of the actions of the accounting system:

- At process termination, the UNIX system kernel writes one record per process in */usr/adm/pacct* in the form of *acct.h*.
- The **login** and **init** programs record connect sessions by writing records into */etc/wtmp*. Date changes, reboots, and shutdowns (via **acctwtmp**) are also recorded in this file.
- The disk utilization program **acctdusg** and **diskusg** break down disk usage by login.
- Fees for file restores, etc., can be charged to specific logins with the **chargefee** shell procedure.
- Each day the **runacct** shell procedure is executed via **cron** to reduce accounting data and produce summary files and reports.
- The **monacct** procedure can be executed on a monthly or fiscal period basis. It saves and restarts summary files, generates a report, and cleans up the *sum* directory. These saved summary files could be used to charge users for UNIX system usage.

5.1 Files and Directories

The */usr/lib/acct* directory contains all of the C language programs and shell procedures necessary to run the accounting system. The *adm* login (currently user ID of 4) is used by the accounting system and has the login directory structure shown in Figure 5-1.

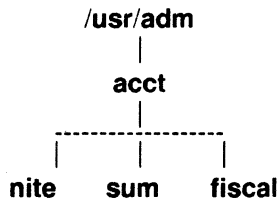


Figure 5-1. Directory Structure of the "adm" Login

The */usr/adm* directory contains the active data collection files. (For a complete explanation of the files used by the accounting system, see Figure 5-2 at the end of this section.) The *nite* directory contains files that are reused daily by the **runacct** procedure. The *sum* directory contains the cumulative summary files updated by **runacct**. The *fiscal* directory contains periodic summary files created by **monacct**.

5.2 Daily Operation

When the UNIX system is switched into multiuser mode, */usr/lib/acct/startup* is executed which does the following:

1. The **acctwtmp** program adds a "boot" record to */etc/wtmp*. This record is signified by using the system name as the login name in the *wtmp* record.
2. Process accounting is started via **turnacct**. **Turnacct on** executes the **accton** program with the argument */usr/adm/pacct*.
3. The **remove** shell procedure is executed to clean up the saved *pacct* and *wtmp* files left in the *sum* directory by **runacct**.

The **ckpacct** procedure is run via **cron** every hour of the day to check the size of */usr/adm/pacct*. If the file grows past 1000 blocks (default), **turnacct switch** is executed. The advantage of having several smaller *pacct* files becomes apparent when trying to restart **runacct** after a failure processing these records.

The **chargefee** program can be used to bill users for file restores, etc. It adds records to */usr/adm/fee* which are picked up and processed by the next execution of **runacct** and merged into the total accounting records.

Runacct is executed via **cron** each night. It processes the active accounting files, */usr/adm/pacct*, */etc/wtmp*, */usr/adm/acct/nite/disktacct*, and */usr/adm/fee*. It produces command summaries and usage summaries by login.

When the system is shut down using **shutdown**, the **shutacct** shell procedure is executed. It writes a shutdown reason record into */etc/wtmp*

and turns process accounting off.

After the first reboot each morning, the computer operator should execute */usr/lib/acct/prdaily* to print the previous day's accounting report.

5.3 Setting Up the Accoutnting System

In order to automate the operation of this accounting system, several things need to be done:

1. If not already present, add this line to the */etc/rc* file in the state 2 section:

   /bin/su – adm –c /usr/lib/acct/startup

2. If not already present, add this line to */etc/shutdown* to turn off the accounting before the system is brought down:

   /usr/lib/acct/shutacct

3. For most installations, the following three entries should be made in */usr/spool/cron/crontab/adm* so that **cron** will automatically run the daily accounting.

   ```
   0 4 * * 1-6 /usr/lib/acct/runacct 2> /usr/adm/acct/nite/fd2log
   0 2 * * 4 /usr/lib/acct/dodisk
   5 * * * * /usr/lib/acct/ckpacct
   ```

4. To facilitate monthly merging of accounting data, the following entry in */usr/spool/cron/crontab/adm* will allow **monacct** to clean up all daily reports and daily total accounting files and deposit one monthly total report and one monthly total accounting file in the *fiscal* directory.

   15 5 1 * * /usr/lib/acct/monacct

   The above entry takes advantage of the default action of **monacct** that uses the current month's date as the suffix for the file names. Notice that the entry is executed at such a time as to allow **runacct** sufficient time to complete. This will, on the first day of each month, create monthly accounting files with the entire month's data.

5. The *PATH* shell variable should be set in */usr/adm/.profile* to:

   PATH=/usr/lib/acct:/bin:/usr/bin

5.4 Runacct

Runacct is the main daily accounting shell procedure. It is normally initiated via **cron** during nonprime time hours. **Runacct** processes connect, fee, disk, and process accounting files. It also prepares daily and cumulative summary files for use by **prdaily** or for billing purposes. The following files produced by **runacct** are of particular interest:

nite/lineuse	Produced by **acctcon**, reads the *wtmp* file, and produces usage statistics for each terminal line on the system. This report is especially useful for detecting bad lines. If the ratio between the number of logoffs to logins exceeds about 3/1, there is a good possibility that the line is failing.
nite/daytacct	This file is the total accounting file for the previous day in **tacct.h** format.
sum/tacct	This file is the accumulation of each day's *nite/daytacct* and can be used for billing purposes. It is restarted each month or fiscal period by the **monacct** procedure.
sum/daycms	Produced by the **acctcms** program. It contains the daily command summary. The ASCII version of this file is *nite/daycms*.
sum/cms	The accumulation of each day's command summaries. It is restarted by the execution of **monacct**. The ASCII version is *nite/cms*.
sum/loginlog	Produced by the **lastlogin** shell procedure. It maintains a record of the last time each login was used.
sum/rprtMMDD	Each execution of **runacct** saves a copy of the daily report that can be printed by **prdaily**.

Runacct takes care not to damage files in the event of errors. A series of protection mechanisms are used that attempt to recognize an error, provide intelligent diagnostics, and terminate processing in such a way that **runacct** can be restarted with minimal intervention. It records its progress by writing descriptive messages into the file *active*. (Files used by **runacct** are assumed to be in the *nite* directory unless otherwise noted.) All diagnostics output during the execution of **runacct** is written into *fd2log*. **Runacct** will complain if the files *lock* and *lock1* exist when invoked. The *lastdate* file contains the month and day **runacct** was last invoked and is used to prevent more than one execution per day. If **runacct** detects an error, a message is written to */dev/console*, mail is sent to *root* and *adm*, locks are removed, diagnostic files are saved, and execution is terminated.

In order to allow **runacct** to be restartable, processing is broken down into separate reentrant states. A file is used to remember the last state completed. When each state completes, *statefile* is updated to reflect the next state. After processing for the state is complete, *statefile* is read and the next state is processed. When **runacct** reaches the **CLEANUP** state, it

removes the locks and terminates. States are executed as follows:

SETUP	The command **turnacct switch** is executed. The process accounting files, */usr/adm/pacct?*, are moved to */usr/adm/Spacct?.MMDD*. The */etc/wtmp* file is moved to */usr/adm/acct/nite/wtmp.MMDD* with the current time added on the end.
WTMPFIX	The *wtmp* file in the *nite* directory is checked for correctness by the **wtmpfix** program. Some date changes will cause **acctcon1** to fail, so **wtmpfix** attempts to adjust the time stamps in the *wtmp* file if a date change record appears.
CONNECT1	Connect session records are written to *ctmp* in the form of **ctmp.h**. The *lineuse* file is created, and the *reboots* file is created showing all of the boot records found in the *wtmp* file.
CONNECT2	*Ctmp* is converted to *ctacct.MMDD* which are connect accounting records. (Accounting records are in **tacct.h** format.)
PROCESS	The **acctprc1** and **acctprc2** programs are used to convert the process accounting files, */usr/adm/Spacct?.MMDD*, into total accounting records in *ptacct?.MMDD*. The *Spacct* and *ptacct* files are correlated by number so that if **runacct** fails the unnecessary reprocessing of *Spacct* files will not occur. One precaution should be noted; when restarting **runacct** in this state, remove the last *ptacct* file because it will not be complete.
MERGE	Merge the process accounting records with the connect accounting records to form *daytacct*.
FEES	Merge in any ASCII *tacct* records from the file *fee* into *daytacct*.
DISK	On the day after the **dodisk** procedure runs, merge *disktacct* with *daytacct*.
MERGETACCT	Merge *daytacct* with *sum/tacct*, the cumulative total accounting file. Each day, *daytacct* is saved in *sum/tacctMMDD*, so that *sum/tacct* can be recreated in the event it becomes corrupted or lost.
CMS	Merge in today's command summary with the cumulative command summary file *sum/cms*.

	Produce ASCII and internal format command summary files.
USEREXIT	Any installation dependent (local) accounting programs can be included here.
CLEANUP	Clean up temporary files, run **prdaily** and save its output in *sum/rprtMMDD*, remove the locks, then exit.

5.5 Recovering From Failure

The **runacct** procedure can fail for a variety of reasons; usually due to a system crash, */usr* running out of space, or a corrupted *wtmp* file. If the *activeMMDD* file exists, check it first for error messages. If the *active* file and lock files exist, check *fd2log* for any mysterious messages. The following are error messages produced by **runacct** and the recommended recovery actions:

ERROR: locks found, run aborted

The files *lock* and *lock1* were found. These files must be removed before **runacct** can restart.

ERROR: acctg already run for *date* : check /usr/adm/acct/nite/lastdate

The date in *lastdate* and today's date are the same. Remove *lastdate*.

ERROR: turnacct switch returned rc=*?*

Check the integrity of **turnacct** and **accton**. The **accton** program must be owned by *root* and have the *setuid* bit set.

ERROR: Spacct*?.MMDD* already exists

File setups probably already run.
Check status of files, then run setups manually.

ERROR: /usr/adm/acct/nite/wtmp.*MMDD* already exists, run setup manually

Self-explanatory.

ERROR: wtmpfix errors see /usr/adm/acct/nite/wtmperror

Wtmpfix detected a corrupted *wtmp* file. Use **fwtmp** to correct the corrupted file.

ERROR: connect acctg failed: check /usr/adm/acct/nite/log

The **acctcon1** program encountered a bad *wtmp* file. Use **fwtmp** to correct the bad file.

ERROR: Invalid state, check /usr/adm/acct/nite/active

The file *statefile* is probably corrupted. Check *statefile* and read *active* before restarting.

5.6 Restarting Runacct

Runacct called without arguments assumes that this is the first invocation of the day. The argument *MMDD* is necessary if **runacct** is being restarted and specifies the month and day for which **runacct** will rerun the accounting. The entry point for processing is based on the contents of *statefile*. To override *statefile*, include the desired state on the command line. For example:

To start **runacct**:

```
nohup runacct 2> /usr/adm/acct/nite/fd2log&
```

To restart **runacct**:

```
nohup runacct 0601 2> /usr/adm/acct/nite/fd2log&
```

To restart **runacct** at a specific state:

```
nohup runacct 0601 WTMPFIX 2> /usr/adm/acct/nite/fd2log&
```

5.7 Fixing Corrupted Files

Unfortunately, this accounting system is not entirely foolproof. Occasionally, a file will become corrupted or lost. Some of the files can simply be ignored or restored from the file save backup. However, certain files must be fixed in order to maintain the integrity of the accounting system.

5.7.1 Fixing WTMP Errors

The *wtmp* files seem to cause the most problems in the day-to-day operation of the accounting system. When the date is changed and the UNIX system is in multiuser mode, a set of date change records is written into */etc/wtmp*. The **wtmpfix** program is designed to adjust the time stamps in the *wtmp* records when a date change is encountered. However, some combinations of date changes and reboots will slip through **wtmpfix** and cause **acctcon1** to fail. The following steps show how to patch up a *wtmp* file.

```
cd /usr/adm/acct/nite
fwtmp < wtmp.MMDD > xwtmp
ed xwtmp
  delete corrupted records or
  delete all records from beginning up to the date change
fwtmp –ic < xwtmp > wtmp.MMDD
```

If the *wtmp* file is beyond repair, create a null *wtmp* file. This will prevent any charging of connect time. **Acctprc1** will not be able to determine which login owned a particular process, but it will be charged to the login that is first in the password file for that user id.

5.7.2 Fixing TACCT Errors

If the installation is using the accounting system to charge users for system resources, the integrity of *sum/tacct* is quite important. Occasionally, mysterious *tacct* records will appear with negative numbers, duplicate user IDs, or a user ID of 65,535. First check *sum/tacctprev* with **prtacct**. If it looks all right, the latest *sum/tacct.MMDD* should be patched up, then *sum/tacct* recreated. A simple patchup procedure would be:

```
cd /usr/adm/acct/sum
acctmerg –v < tacct.MMDD > xtacct
ed xtacct
  remove the bad records
  write duplicate uid records to another file
acctmerg –i < xtacct > tacct.MMDD
acctmerg tacctprev < tacct.MMDD > tacct
```

Remember that the **monacct** procedure removes all the *tacct.MMDD* files; therefore, *sum/tacct* can be recreated by merging these files together.

5.8 Updating Holidays

The file */usr/lib/acct/holidays* contains the prime/nonprime table for the accounting system. The table should be edited to reflect your location's holiday schedule for the year. The format is composed of three types of entries:

1. *Comment Lines*: Comment lines may appear anywhere in the file as long as the first character in the line is an asterisk.
2. *Year Designation Line*: This line should be the first data line (noncomment line) in the file and must appear only once. The line consists of three fields of four digits each (leading white space is ignored). For example, to specify the year as 1982, prime time at 9:00 a.m., and nonprime time at 4:30 p.m., the following entry would be appropriate:

1982 0900 1630

A special condition allowed for in the time field is that the time 2400 is automatically converted to 0000.

3. *Company Holidays Lines*: These entries follow the year designation line and have the following general format:

   day-of-year Month Day Description of Holiday

   The day-of-year field is a number in the range of 1 through 366 indicating the day for the corresponding holiday (leading white space is ignored). The other three fields are actually commentary and are not currently used by other programs.

5.9 Daily Reports

Runacct generates five basic reports upon each invocation. They cover the areas of connect accounting, usage by person on a daily basis, command usage reported by daily and monthly totals, and a report of the last time users were logged in.

The following paragraphs describe the reports and the meanings of their tabulated data.

5.9.1 Daily Report

In the first part of the report, the **from/to** banner should alert the administrator to the period reported on. The times are the time the last accounting report was generated until the time the current accounting report was generated. It is followed by a log of system reboots, shutdowns, power fail recoveries, and any other record dumped into */etc/wtmp* by the **acctwtmp** program [see **acct**(1M) in the *Sys5 UNIX Administrator Reference Manual*].

The second part of the report is a breakdown of line utilization. The TOTAL DURATION tells how long the system was in multiuser state (able to be accessed through the terminal lines). The columns are:

LINE	The terminal line or access port.
MINUTES	The total number of minutes that line was in use during the accounting period.
PERCENT	The total number of MINUTES the line was in use divided into the TOTAL DURATION.
# SESS	The number of times this port was accessed for a **login**(1) session.
# ON	This column does not have much meaning anymore. It used to give the number of times that the port was

used to log a user on; but since **login**(1) can no longer be executed explicitly to log in a new user, this column should be identical with SESS.

OFF — This column reflects not just the number of times a user logged off but also any interrupts that occur on that line. Generally, interrupts occur on a port when the **getty**(1M) is first invoked when the system is brought to multiuser state. Where this column does come into play is when the # OFF exceeds the # ON by a large factor. This usually indicates that the multiplexer, modem, or cable is going bad, or there is a bad connection somewhere. The most common cause of this is an unconnected cable dangling from the multiplexer.

During real time, */etc/wtmp* should be monitored as this is the file that the connect accounting is geared from. If it grows rapidly, execute **acctcon1** to see which tty line is the noisest. If the interrupting is occurring at a furious rate, general system performance will be effected.

5.9.2 Daily Usage Report

This report gives a by-user breakdown of system resource utilization. Its data consists of:

UID — The user ID.

LOGIN NAME — The login name of the user; there can be more than one login name for a single user ID, this identifies which one.

CPU (MINS) — This represents the amount of time the user's process used the central processing unit. This category is broken down into PRIME and NPRIME (nonprime) utilization. The accounting system's idea of this breakdown is located in the */usr/lib/acct/holidays* file. As delivered, prime time is defined to be 0900 through 1700 hours.

KCORE-MINS — This represents a cumulative measure of the amount of memory a process uses while running. The amount shown reflects kilobyte segments of memory used per minute. This measurement is also broken down into PRIME and NPRIME amounts.

CONNECT (MINS) This identifies "Real Time" used. What this column really identifies is the amount of time that a user was logged into the system. If this time is rather high and the column "# OF PROCS" is low, this user is what is called a "line hog". That is, this person logs in first thing in the morning and does not hardly touch the terminal the rest of the day. Watch out for these kinds of users. This column is also subdivided into PRIME and NPRIME utilization.

DISK BLOCKS When the disk accounting programs have been run, the output is merged into the total accounting record (*tacct.h*) and shows up in this column. This disk accounting is accomplished by the program **acctdusg**.

OF PROCS This column reflects the number of processes that was invoked by the user. This is a good column to watch for large numbers indicating that a user may have a shell procedure that runs amock.

OF SESS This is how many times the user logged onto the system.

DISK SAMPLES This indicates how many times the disk accounting was run to obtain the average number of DISK BLOCKS listed earlier.

FEE An often unused field in the total accounting record, the FEE field represents the total accumulation of widgets charged against the user by the **chargefee** shell procedure [see **acctsh**(1M)]. The **chargefee** procedure is used to levy charges against a user for special services performed such as file restores, etc.

5.9.3 Daily Command and Monthly Total Command Summaries

These two reports are virtually the same except that the Daily Command Summary only reports on the current accounting period while the Monthly Total Command Summary tells the story for the start of the fiscal period to the current date. In other words, the monthly report reflects the data accumulated since the last invocation of **monacct**.

The data included in these reports gives an administrator an idea as to the heaviest used commands and, based on those commands' characteristics of

system resource utilization, a hint as to what to weigh more heavily when system tuning.

These reports are sorted by TOTAL KCOREMIN, which is an arbitrary yardstick but often a good one for calculating "drain" on a system.

COMMAND NAME — This is the name of the command. Unfortunately, all shell procedures are lumped together under the name **sh** since only object modules are reported by the process accounting system. The administrator should monitor the frequency of programs called **a.out** or **core** or any other name that does not seem quite right. Often people like to work on their favorite version of backgammon only they do not want everyone to know about it. **Acctcom** is also a good tool to use for determining who executed a suspiciously named command and also if superuser privileges were used.

NUMBER CMDS — This is the total number of invocations of this particular command.

TOTAL KCOREMIN — The total cumulative measurement of the amount of kilobyte segments of memory used by a process per minute of run time.

TOTAL CPU-MIN — The total processing time this program has accumulated.

TOTAL REAL-MIN — The total real-time (wall-clock) minutes this program has accumulated. This total is the actual "waited for" time as opposed to kicking off a process in the background.

MEAN SIZE-K — This is the mean of the TOTAL KCOREMIN over the number of invocations reflected by NUMBER CMDS.

MEAN CPU-MIN — This is the mean derived between the NUMBER CMDS and TOTAL CPU-MIN.

HOG FACTOR — This is a relative measurement of the ratio of system availability to system utilization. It is computed by the formula

(total CPU time) / (elapsed time)

This gives a relative measure of the total

	available CPU time consumed by the process during its execution.
CHARS TRNSFD	This column, which may go negative, is a total count of the number of characters pushed around by the **read**(2) and **write**(2) system calls.
BLOCKS READ	A total count of the physical block reads and writes that a process performed.

5.9.4 Last Login

This report simply gives the date when a particular login was last used. This could be a good source for finding likely candidates for the archives or getting rid of unused logins and login directories.

5.10 Summary

The UNIX system accounting was designed from a UNIX system administrator's point of view. Every possible precaution has been taken to ensure that the system will run smoothly and without error. It is important to become familiar with the C programs and shell procedures. The manual pages should be studied, and it is advisable to keep a printed copy of the shell procedures handy. The accounting system should be easy to maintain, provide valuable information for the administrator, and provide accurate breakdowns of the usage of system resources for charging purposes.

5.10.1 Files in the /usr/adm directory

diskdiag	diagnostic output during the execution of disk accounting programs
dtmp	output from the *acctdusg* program
fee	output from the *chargefee* program, ASCII tacct records
pacct	active process accounting file
pacct?	process accounting files switched via *turnacct*
Spacct?.MMDD	process accounting files for *MMDD* during execution of *runacct*

5.10.2 Files in the /usr/adm/acct/nite directory

active	used by *runacct* to record progress and print warning and error messages. **active***MMDD* same as **active** after *runacct* detects an error

cms	ASCII total command summary used by *prdaily*
ctacct.MMDD	connect accounting records in *tacct.h* format
ctmp	output of *acctcon1* program, connect session records in *ctmp.h* format
daycms	ASCII daily command summary used by *prdaily*
daytacct	total accounting records for 1 day in *tacct.h* format
disktacct	disk accounting records in *tacct.h* format, created by *dodisk* procedure
fd2log	diagnostic output during execution of runacct (see *cron* entry)
lastdate	last day *runacct* executed in *date +%m%d* format
lock lock1	used to control serial use of *runacct*
lineuse	tty line usage report used by *prdaily*
log	diagnostic output from *acctcon1*
logMMDD	same as **log** after *runacct* detects an error
reboots	contains beginning and ending dates from **wtmp**, and a listing of reboots
statefile	used to record current state during execution of *runacct*
tmpwtmp	wtmp file corrected by *wtmpfix*
wtmperror	place for *wtmpfix* error messages
wtmperrorMMDD	same as **wtmperror** after *runacct* detects an error
wtmp.MMDD	previous day's wtmp file

5.10.3 Files in the /usr/adm/acct/sum directory

cms	total command summary file for current fiscal in internal summary format
cmsprev	command summary file without latest update
daycms	command summary file for yesterday in internal summary format
loginlog	created by *lastlogin*

pacct.MMDD	concatenated version of all pacct files for *MMDD*, removed after reboot by *remove* procedure
rprtMMDD	saved output of *prdaily* program
tacct	cumulative total accounting file for current fiscal
tacctprev	same as **tacct** without latest update
tacctMMDD	total accounting file for *MMDD*
wtmp.MMDD	saved copy of wtmp file for *MMDD*, removed after reboot by *remove* procedure

5.10.4 Files in the /usr/adm/acct/fiscal directory

cms?	total command summary file for fiscal ? in internal summary format
fiscrpt?	report similar to *prdaily* for fiscal ?
tacct?	total accounting file for fiscal ?

6. FILE SYSTEM CHECKING

The File System Check Program (**fsck**) is an interactive file system check and repair program. **Fsck** uses the redundant structural information in the UNIX system file system to perform several consistency checks. If an inconsistency is detected, it is reported to the operator, who may elect to fix or ignore each inconsistency. These inconsistencies result from the permanent interruption of the file system updates, which are performed every time a file is modified. **Fsck** is frequently able to repair corrupted file systems using procedures based upon the order in which the UNIX system honors these file system update requests.

The purpose of this chapter is to describe the normal updating of the file system, to discuss the possible causes of file system corruption, and to present the corrective actions implemented by **fsck**. Both the program and the interaction between the program and the operator are described.

Appendix 6-1 contains the **fsck** error conditions. The meanings of the various error conditions, possible responses, and related error conditions are explained.

When a UNIX operating system is brought up, a consistency check of the file systems should always be performed. This precautionary measure helps to ensure a reliable environment for file storage on disk. If an inconsistency is discovered, corrective action must be taken.

The updating of the file system and file system corruption is described in this chapter. Finally, the set of heuristically sound corrective actions used by **fsck** are presented.

6.0.1 System Administrator Advice

Remember that system buffers are 1024 bytes. When configuring the operating system, take into consideration that the same number of buffers as before will use more main memory. Weigh this against reducing the number of buffers, which reduces the cache hit ratio and degrades performance.

6.1 Update of the File System

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UNIX operating system performs a series of file system updates. These updates, when written on disk, yield a consistent file system. To understand what happens in the event of a

permanent interruption in this sequence, it is important to understand the order in which the update requests were probably being honored. Knowing which pieces of information were probably written to the file system first, heuristic procedures can be developed to repair a corrupted file system.

There are five types of file system updates. These involve the superblock, inodes, indirect blocks, data blocks (directories and files), and free-list blocks.

6.1.1 Superblock

The superblock contains information about the size of the file system, the size of the inode list, part of the free-block list, the count of free blocks, the count of free inodes, and part of the free-inode list.

The superblock of a mounted file system (the root file system is always mounted) is written to the file system whenever the file system is unmounted or a **sync** command is issued.

6.1.2 Inodes

An inode contains information about the type of inode (directory, data, or special), the number of directory entries linked to the inode, the list of blocks claimed by the inode, and the size of the inode.

An inode is written to the file system upon closure of the file associated with the inode. (All "in" core blocks are also written to the file system upon issue of a **sync** system call.)

6.1.3 Indirect Blocks

There are three types of indirect blocks—single-indirect, double-indirect, and triple-indirect. A single-indirect block contains a list of some of the block numbers claimed by an inode. Each one of the 128 entries in an indirect block is a data-block number. A double-indirect block contains a list of single-indirect block numbers. A triple-indirect block contains a list of double-indirect block numbers.

Indirect blocks are written to the file system whenever they have been modified and released by the operating system. More precisely, they are queued for eventual writing. Physical I/O is deferred until the buffer is needed by the UNIX system or a **sync** command is issued.

6.1.4 Data Blocks

A data block may contain file information or directory entries. Each directory entry consists of a file name and an inode number.

Data blocks are written to the file system whenever they have been modified and released by the operating system.

6.1.5 First Free-List Block

The superblock contains the first free-list block. The free-list blocks are a list of all blocks that are not allocated to the superblock, inodes, indirect blocks, or data blocks. Each free-list block contains a count of the number of entries in this free-list block, a pointer to the next free-list block, and a partial list of free blocks in the file system.

Free-list blocks are written to the file system whenever they have been modified and released by the operating system.

6.2 Corruption of the File System

A file system can become corrupted in a variety of ways. Improper shutdown procedures and hardware failures are the most common.

6.2.1 Improper System Shutdown and Startup

File systems may become corrupted when proper shutdown procedures are not observed, e.g., forgetting to **sync** the system prior to halting the CPU, physically write-protecting a mounted file system, or taking a mounted file system off-line.

File systems may also become further corrupted by allowing a corrupted file system to be used (and, thus, to be modified further) can be disastrous.

6.2.2 Hardware Failure

Any piece of hardware can fail at any time. Failures can be as subtle as a bad block on a disk platter or as blatant as a nonfunctional disk controller.

6.3 Detection and Correction of Corruption

A quiescent file system (an unmounted system and not being written on) may be checked for structural integrity by performing consistency checks on the redundant data intrinsic to a file system. The redundant data is either read from the file system or computed from other known values. A quiescent state is important during the checking of a file system because of the multipass nature of the **fsck** program.

When an inconsistency is discovered, **fsck** reports the inconsistency for the operator to chose a corrective action.

Discussed in this part are how to discover inconsistencies (and possible corrective actions) for the superblock, the inodes, the indirect blocks, the data blocks containing directory entries, and the free-list blocks. These corrective actions can be performed interactively by the **fsck** command under control of the operator.

6.3.1 Superblock

One of the most common corrupted items is the superblock. The superblock is prone to corruption because every change to the file system's blocks or inodes modifies the superblock.

The superblock and its associated parts are most often corrupted when the computer is halted and the last command involving output to the file system was not a **sync** command.

The superblock can be checked for inconsistencies involving file system size, inode-list size, free-block list, free-block count, and the free-inode count.

6.3.1.1 File System Size and Inode-List Size

The file system size must be larger than the number of blocks used by the superblock and the number of blocks used by the list of inodes. The number of inodes must be less than 65,535. The file system size and inode-list size are critical pieces of information to the **fsck** program. While there is no way to actually check these sizes, **fsck** can check for them being within reasonable bounds. All other checks of the file system depend on the correctness of these sizes.

6.3.1.2 Free-Block List

The free-block list starts in the superblock and continues through the free-list blocks of the file system. Each free-list block can be checked for a list count out of range, for block numbers out of range, and for blocks already

allocated within the file system. A check is made to see that all the blocks in the file system were found.

The first free-block list is in the superblock. **Fsck** checks the list count for a value of less than 0 or greater than 50. It also checks each block number for a value of less than the first data block in the file system or greater than the last block in the file system. Then it compares each block number to a list of already allocated blocks. If the free-list block pointer is nonzero, the next free-list block is read in and the process is repeated.

When all the blocks have been accounted for, a check is made to see if the number of blocks used by the free-block list plus the number of blocks claimed by the inodes equals the total number of blocks in the file system.

If anything is wrong with the free-block list, then **fsck** may rebuild the list, excluding all blocks in the list of allocated blocks.

6.3.1.3 Free-Block Count

The superblock contains a count of the total number of free blocks within the file system. **Fsck** compares this count to the number of blocks it found free within the file system. If the counts do not agree, then **fsck** may replace the count in the superblock by the actual free-block count.

6.3.1.4 Free-Inode Count

The superblock contains a count of the total number of free inodes within the file system. **Fsck** compares this count to the number of inodes it found free within the file system. If the counts do not agree, then **fsck** may replace the count in the superblock by the actual free-inode count.

6.3.2 Inodes

An individual inode is not as likely to be corrupted as the superblock. However, because of the great number of active inodes, there is almost as likely a chance for corruption in the inode list as in the superblock.

The list of inodes is checked sequentially starting with inode 1 (there is no inode 0) and going to the last inode in the file system. Each inode can be checked for inconsistencies involving format and type, link count, duplicate blocks, bad blocks, and inode size.

6.3.2.1 Format and Type

Each inode contains a mode word. This mode word describes the type and

state of the inode. Inodes may be one of four types:

- Regular
- Directory
- Special block
- Special character.

If an inode is not one of these types, then the inode has an illegal type. Inodes may be found in one of three states—unallocated, allocated, and neither unallocated nor allocated. This last state indicates an incorrectly formatted inode. An inode can get in this state if bad data is written into the inode list through, for example, a hardware failure. The only possible corrective action is for **fsck** to clear the inode.

6.3.2.2 Link Count

Contained in each inode is a count of the total number of directory entries linked to the inode. **Fsck** verifies the link count of each inode by traversing down the total directory structure, starting from the root directory, and calculating an actual link count for each inode.

If the stored link count is nonzero and the actual link count is zero, it means that no directory entry appears for the inode. If the stored and actual link counts are nonzero and unequal, a directory entry may have been added or removed without the inode being updated.

If the stored link count is nonzero and the actual link count is zero, **fsck** can, under operator control, link the disconnected file to the *lost+found* directory. If the stored and actual link counts are nonzero and unequal, **fsck** can replace the stored link count by the actual link count.

6.3.2.3 Duplicate Blocks

Contained in each inode is a list or pointers to lists (indirect blocks) of all the blocks claimed by the inode. **Fsck** compares each block number claimed by an inode to a list of already allocated blocks. If a block number is already claimed by another inode, the block number is added to a list of duplicate blocks. Otherwise, the list of allocated blocks is updated to include the block number. If there are any duplicate blocks, **fsck** will make a partial second pass of the inode list to find the inode of the duplicated block. This is necessary because without examining the files associated with these inodes for correct content there is not enough information available to decide which inode is corrupted and should be cleared. Most of the time, the inode with the earliest modify time is incorrect and should be cleared. This condition can occur by using a file system with blocks claimed by both

the free-block list and by other parts of the file system.

A large number of duplicate blocks in an inode may be due to an indirect block not being written to the file system. **Fsck** will prompt the operator to clear both inodes.

6.3.2.4 Bad Blocks

Contained in each inode is a list or pointer to lists of all the blocks claimed by the inode. **Fsck** checks each block number claimed by an inode for a value lower than that of the first data block or greater than the last block in the file system. If the block number is outside this range, the block number is a bad block number.

If there is a large number of bad blocks in an inode, this may be due to an indirect block not being written to the file system. **Fsck** will prompt the operator to clear both inodes.

6.3.2.5 Size Checks

Each inode contains a 32-bit (4-byte) size field. This size indicates the number of characters in the file associated with the inode. This size can be checked for inconsistencies, e.g., directory sizes that are not a multiple of 16 characters or the number of blocks actually used not matching that indicated by the inode size.

A directory inode within the file system has the directory bit on in the inode mode word. The directory size must be a multiple of 16 because a directory entry contains 16 bytes (2 bytes for the inode number and 14 bytes for the file or directory name).

Fsck will warn of such directory misalignment. This is only a warning because not enough information can be gathered to correct the misalignment.

A rough check of the consistency of the size field of an inode can be performed by computing from the size field the number of blocks that should be associated with the inode and comparing it to the actual number of blocks claimed by the inode.

Fsck calculates the number of blocks that there should be in an inode by dividing the number of characters in an inode by the number of characters per block and rounding up. **Fsck** adds one block for each indirect block associated with the inode. If the actual number of blocks does not match the computed number of blocks, **fsck** will warn of a possible file-size error. This is only a warning because the UNIX system does not fill in blocks in files created in random order.

6.3.3 Indirect Blocks

Indirect blocks are owned by an inode. Therefore, inconsistencies in indirect blocks directly affect the inode that owns it.

Inconsistencies that can be checked are blocks already claimed by another inode and block numbers outside the range of the file system.

For a discussion of detection and correction of the inconsistencies associated with indirect blocks, see parts "Duplicate Blocks" and "Bad Blocks".

6.3.4 Data Blocks

The two types of data blocks are plain data blocks and directory data blocks. Plain data blocks contain the information stored in a file. Directory data blocks contain directory entries. **Fsck** does not attempt to check the validity of the contents of a plain data block.

Each directory data block can be checked for inconsistencies involving directory inode numbers pointing to unallocated inodes, directory inode numbers greater than the number of inodes in the file system, incorrect directory inode numbers for "." and "..", and directories disconnected from the file system. In addition, the validity of the contents of a directory's data block is checked.

If a directory entry inode number points to an unallocated inode, then **fsck** may remove that directory entry. This condition probably occurred because the data blocks containing the directory entries were modified and written out while the inode was not yet written out.

If a directory entry inode number is pointing beyond the end of the inode list, **fsck** may remove that directory entry. This condition occurs if bad data is written into a directory data block.

The directory inode number entry for "." should be the first entry in the directory data block. Its value should be equal to the inode number for the directory data block.

The directory inode number entry for ".." should be the second entry in the directory data block. Its value should be equal to the inode number for the parent of the directory entry (or the inode number of the directory data block if the directory is the root directory).

If the directory inode numbers are incorrect, **fsck** may replace them with the correct values.

Fsck checks the general connectivity of the file system. If directories are found not to be linked into the file system, **fsck** will link the directory back into the file system in the *lost+found* directory. This condition can be caused by inodes being written to the file system with the corresponding directory data blocks not being written to the file system.

6.3.5 Free-List Blocks

Free-list blocks are owned by the superblock. Therefore, inconsistencies in free-list blocks directly affect the superblock.

Inconsistencies that can be checked are a list count outside of range, block numbers outside of range, and blocks already associated with the file system.

For a discussion of detection and correction of the inconsistencies associated with free-list blocks, see part "Free-Block List".

6.4 Appendix 6-1 (FSCK Error Conditions)

6.4.1 Conventions

Fsck is a multipass file system check program. Each file system pass invokes a different phase of the **fsck** program. After the initial setup, **fsck** performs successive phases over each file system performing cleanup, checking blocks and sizes, pathnames, connectivity, reference counts, and the free-block list (possibly rebuilding it).

When an inconsistency is detected, **fsck** reports the error condition to the operator. If a response is required, **fsck** prints a prompt message and waits for a response. This appendix explains the meaning of each error condition, the possible responses, and the related error conditions.

The error conditions are organized by the "Phase" of the **fsck** program in which they can occur. The error conditions that may occur in more than one phase will be discussed under Part B.

6.4.2 Initialization

Before a file system check can be performed, certain tables have to be set up and certain files opened. This section describes the opening of files and the initialization of tables. Error conditions resulting from command line options, memory requests, opening of files, status of files, file system size checks, and creation of the scratch file are listed below.

6.4.2.1 C option?

C is not a legal option to **fsck**; legal options are **–y**, **–n**, **–s**, **–S**, **–t**, **–r**, **–q**, and **–D**. **Fsck** terminates on this error condition. See the **fsck**(1M) entry in the *UNIX System V Administrator Reference Manual* for further details.

6.4.2.2 Bad –t option

The **–t** option is not followed by a file name. **Fsck** terminates on this error condition. See the **fsck**(1M) entry in the *UNIX System V Administrator Reference Manual* for further details.

6.4.2.3 Invalid –s argument, defaults assumed

The **–s** option is not suffixed by 3, 4, or blocks-per-cylinder:blocks-to-skip. **Fsck** assumes a default value of 400 blocks-per-cylinder and 9 blocks-to-

skip. See the **fsck**(1M) entry in the *UNIX System V Administrator Reference Manual* for further details.

6.4.2.4 Incompatible options: –n and –s

It is not possible to salvage the free-block list without modifying the file system. **Fsck** terminates on this error condition. See the **fsck**(1M) entry in the *UNIX System V Administrator Reference Manual* for further details.

6.4.2.5 Can not fstat standard input

Fsck's attempt to **fstat** standard input failed. The occurrence of this error condition indicates a serious problem which may require additional assistance. **Fsck** terminates on this error condition.

6.4.2.6 Can not get memory

Fsck's request for memory for its virtual memory tables failed. The occurrence of this error condition indicates a serious problem which may require additional assistance. **Fsck** terminates on this error condition.

6.4.2.7 Can not open checkall file: F

The default file system checkall file *F* (usually */etc/checkall*) cannot be opened for reading. **Fsck** terminates on this error condition. Check access modes of *F*.

6.4.2.8 Can not stat root

Fsck's request for statistics about the root directory "/" failed. The occurrence of this error condition indicates a serious problem which may require additional assistance. **Fsck** terminates on this error condition.

6.4.2.9 Can not stat F

Fsck's request for statistics about the file system *F* failed. It ignores this file system and continues checking the next file system given. Check access modes of *F*.

6.4.2.10 F is not a block or character device

Fsck has been given a regular file name by mistake. It ignores this file system and continues checking the next file system given. Check file type of *F*.

6.4.2.11 Can not open F

The file system *F* cannot be opened for reading. It ignores this file system and continues checking the next file system given. Check access modes of *F*.

6.4.2.12 Size check: fsize X isize Y

More blocks are used for the inode list *Y* than there are blocks in the file system *X*, or there are more than 65,535 inodes in the file system. It ignores this file system and continues checking the next file system given.

6.4.2.13 Can not create F

Fsck's request to create a scratch file *F* failed. It ignores this file system and continues checking the next file system given. Check access modes of *F*.

6.4.2.14 CAN NOT SEEK: BLK B (CONTINUE)

Fsck's request for moving to a specified block number *B* in the file system failed. The occurrence of this error condition indicates a serious problem which may require additional assistance.

Possible responses to CONTINUE prompt are:

YES	Attempt to continue to run file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If block was part of the virtual memory buffer cache, **fsck** will terminate with the message "Fatal I/O error".
NO	Terminate program.

6.4.2.15 CAN NOT READ: BLK B (CONTINUE)

Fsck's request for reading a specified block number *B* in the file system failed. The occurrence of this error condition indicates a serious problem which may require additional assistance.

Possible responses to CONTINUE prompt are:

YES	Attempt to continue to run file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A

second run of **fsck** should be made to recheck this file system. If block was part of the virtual memory buffer cache, **fsck** will terminate with the message "Fatal I/O error".

NO — Terminate program.

6.4.2.16 CAN NOT WRITE: BLK B (CONTINUE)

Fsck's request for writing a specified block number *B* in the file system failed. The disk is write-protected.

Possible responses to CONTINUE prompt are:

YES — Attempt to continue to run file system check. Often, however, the problem will persist. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If block was part of the virtual memory buffer cache, **fsck** will terminate with the message "Fatal I/O error".

NO — Terminate program.

6.4.3 PHASE 1: CHECK BLOCKS AND SIZES

This phase concerns itself with the inode list. This part lists error conditions resulting from checking inode types, setting up the zero-link-count table, examining inode block numbers for bad or duplicate blocks, checking inode size, and checking inode format.

6.4.3.1 UNKNOWN FILE TYPE I=I (CLEAR)

The mode word of the inode *I* indicates that the inode is not a special character inode, regular inode, or directory inode.

Possible responses to CLEAR prompt are:

YES — Deallocate inode *I* by zeroing its contents. This will always invoke the UNALLOCATED error condition in Phase 2 for each directory entry pointing to this inode.

NO — Ignore this error condition.

6.4.3.2 LINK COUNT TABLE OVERFLOW (CONTINUE)

An internal table for **fsck** containing allocated inodes with a link count of zero has no more room. Recompile **fsck** with a larger value of MAXLNCNT.

Possible responses to CONTINUE prompt are:

YES	Continue with program. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If another allocated inode with a zero link count is found, this error condition is repeated.
NO	Terminate program.

6.4.3.3 B BAD I=I

Inode *I* contains block number *B* with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system. This error condition may invoke the EXCESSIVE BAD BLKS error condition in Phase 1 if inode *I* has too many block numbers outside the file system range. This error condition will always invoke the BAD/DUP error condition in Phase 2 and Phase 4.

6.4.3.4 EXCESSIVE BAD BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks with a number lower than the number of the first data block in the file system or greater than the number of the last block in the file system associated with inode *I*.

Possible responses to CONTINUE prompt are:

YES	Ignore the rest of the blocks in this inode and continue checking with next inode in the file system. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system.
NO	Terminate program.

6.4.3.5 B DUP I=I

Inode *I* contains block number *B* which is already claimed by another inode. This error condition may invoke the EXCESSIVE DUP BLKS error condition in Phase 1 if inode *I* has too many block numbers claimed by other inodes. This error condition will always invoke Phase 1b and the BAD/DUP error condition in Phase 2 and Phase 4.

6.4.3.6 EXCESSIVE DUP BLKS I=I (CONTINUE)

There is more than a tolerable number (usually 10) of blocks claimed by other inodes.

Possible responses to CONTINUE prompt are:

YES	Ignore the rest of the blocks in this inode and continue checking with next inode in the file system. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system.
NO	Terminate program.

6.4.3.7 DUP TABLE OVERFLOW (CONTINUE)

An internal table in **fsck** containing duplicate block numbers has no more room. Recompile **fsck** with a larger value of DUPTBLSIZE.

Possible responses to CONTINUE prompt are:

YES	Continue with program. This error condition will not allow a complete check of the file system. A second run of **fsck** should be made to recheck this file system. If another duplicate block is found, this error condition will repeat.
NO	Terminate program.

6.4.3.8 POSSIBLE FILE SIZE ERROR I=I

The inode *I* size does not match the actual number of blocks used by the inode. This is only a warning. If the **–q** option is used, this message is not printed.

6.4.3.9 DIRECTORY MISALIGNED I=I

The size of a directory inode is not a multiple of the size of a directory entry (usually 16). This is only a warning. If the **–q** option is used, this message is not printed.

6.4.3.10 PARTIALLY ALLOCATED INODE I=I (CLEAR)

Inode *I* is neither allocated nor unallocated.

Possible responses to CLEAR prompt are:

YES	Deallocate inode I by zeroing its contents.
NO	Ignore this error condition.

6.4.4 PHASE 1B: RESCAN FOR MORE DUPS

When a duplicate block is found in the file system, the file system is rescanned to find the inode which previously claimed that block. This part lists the error condition when the duplicate block is found.

6.4.4.1 B DUP I=I

Inode *I* contains block number *B* which is already claimed by another inode. This error condition will always invoke the BAD/DUP error condition in Phase 2. Inodes with overlapping blocks may be determined by examining this error condition and the DUP error condition in Phase 1.

6.4.5 PHASE 2: CHECK PATHNAMES

This phase concerns itself with removing directory entries pointing to error conditioned inodes from Phase 1 and Phase 1b. This part lists error conditions resulting from root inode mode and status, directory inode pointers in range, and directory entries pointing to bad inodes.

6.4.5.1 ROOT INODE UNALLOCATED. TERMINATING

The root inode (always inode number 2) has no allocate mode bits. The occurrence of this error condition indicates a serious problem which may require additional assistance. The program will terminate.

6.4.5.2 ROOT INODE NOT DIRECTORY (FIX)

The root inode (usually inode number 2) is not directory inode type.

Possible responses to FIX prompt are:

YES	Replace the root inode's type to be a directory. If the root inode's data blocks are not directory blocks, a *very* large number of error conditions will be produced.
NO	Terminate program.

6.4.5.3 DUPS/BAD IN ROOT INODE (CONTINUE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks in the root inode (usually inode number 2) for the file system.

Possible responses to CONTINUE prompt are:

YES	Ignore DUPS/BAD error condition in root inode and attempt to continue to run the file system check. If root inode is not correct, then this may result in a large number of other error conditions.
NO	Terminate program.

6.4.5.4 I OUT OF RANGE I=I NAME=F (REMOVE)

A directory entry *F* has an inode number *I* which is greater than the end of the inode list.

Possible responses to REMOVE prompt are:

YES	The directory entry *F* is removed.
NO	Ignore this error condition.

6.4.5.5 UNALLOCATED I=I OWNER=O MODE=M SIZE=S MTIME=T NAME=F (REMOVE)

A directory entry *F* has an inode *I* without allocate mode bits. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed. If the file system is not mounted and the **–n** option was not specified, the entry will be removed automatically if the inode it points to is character size 0.

Possible responses to REMOVE prompt are:

YES	The directory entry *F* is removed.
NO	Ignore this error condition.

6.4.5.6 DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T DIR=F (REMOVE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, directory inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and directory name *F* are printed.

Possible responses to REMOVE prompt are:

YES	The directory entry *F* is removed.

NO Ignore this error condition.

6.4.5.7 DUP/BAD I=I OWNER=O MODE=M SIZE=S MTIME=T FILE=F (REMOVE)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory entry *F*, inode *I*. The owner *O*, mode *M*, size *S*, modify time *T*, and file name *F* are printed.

Possible responses to REMOVE prompt are:

YES The directory entry *F* is removed.

NO Ignore this error condition.

6.4.5.8 BAD BLK B IN DIR I=I OWNER=O MODE=M SIZE=S MTIME=T

This message only occurs when the **–q** option is used. A bad block was found in DIR inode *I*. Error conditions looked for in directory blocks are nonzero padded entries, inconsistent "." and ".." entries, and imbedded slashes in the name field. This error message indicates that the user should at a later time either remove the directory inode if the entire block looks bad or change (or remove) those directory entries that look bad.

6.4.6 PHASE 3: CHECK CONNECTIVITY

This phase concerns itself with the directory connectivity seen in Phase 2. This part lists error conditions resulting from unreferenced directories and missing or full *lost+found* directories.

6.4.6.1 UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)

The directory inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed. **Fsck** will force the reconnection of a nonempty directory.

Possible responses to RECONNECT prompt are:

YES Reconnect directory inode *I* to the file system in directory for lost files (usually *lost+found*). This may invoke *lost+found* error condition in Phase 3 if there are problems connecting directory inode *I* to *lost+found*. This may also invoke CONNECTED

error condition in Phase 3 if link was successful.

NO — Ignore this error condition. This will always invoke UNREF error condition in Phase 4.

6.4.6.2 SORRY. NO lost+found DIRECTORY

There is no *lost+found* directory in the root directory of the file system; **fsck** ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Check access modes of *lost+found*. See **fsck**(1M) in the *UNIX System V Administrator Reference Manual* for further details.

6.4.6.3 SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; **fsck** ignores the request to link a directory in *lost+found*. This will always invoke the UNREF error condition in Phase 4. Clean out unnecessary entries in *lost+found* or make *lost+found* larger. See **fsck**(1M) in the *UNIX System V Administrator Reference Manual* for further details.

6.4.6.4 DIR I=I1 CONNECTED. PARENT WAS I=I2

This is an advisory message indicating a directory inode *I1* was successfully connected to the *lost+found* directory. The parent inode *I2* of the directory inode *I1* is replaced by the inode number of the *lost+found* directory.

6.4.7 PHASE 4: CHECK REFERENCE COUNTS

This phase concerns itself with the link count information seen in Phase 2 and Phase 3. This part lists error conditions resulting from unreferenced files; missing or full *lost+found* directory; incorrect link counts for files, directories, or special files; unreferenced files and directories; bad and duplicate blocks in files and directories; and incorrect total free-inode counts.

6.4.7.1 UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (RECONNECT)

Inode *I* was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. If the **–n** option is not set and the file system is not mounted, empty files will not be reconnected and will be cleared automatically.

Possible responses to RECONNECT prompt are:

YES	Reconnect inode *I* to file system in the directory for lost files (usually *lost+found*). This may invoke *lost+found* error condition in Phase 4 if there are problems connecting inode *I* to *lost+found*.
NO	Ignore this error condition. This will always invoke CLEAR error condition in Phase 4.

6.4.7.2 SORRY. NO lost+found DIRECTORY

There is no *lost+found* directory in the root directory of the file system; **fsck** ignores the request to link a file in *lost+found*. This will always invoke CLEAR error condition in Phase 4. Check access modes of *lost+found*.

6.4.7.3 SORRY. NO SPACE IN lost+found DIRECTORY

There is no space to add another entry to the *lost+found* directory in the root directory of the file system; **fsck** ignores the request to link a file in *lost+found*. This will always invoke the CLEAR error condition in Phase 4. Check size and contents of *lost+found*.

6.4.7.4 (CLEAR)

The inode mentioned in the immediately previous error condition cannot be reconnected.

Possible responses to CLEAR prompt are:

YES	Deallocate inode mentioned in the immediately previous error condition by zeroing its contents.
NO	Ignore this error condition.

6.4.7.5 LINK COUNT FILE I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for inode *I*, which is a file, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* are printed.

Possible responses to ADJUST prompt are:

YES	Replace link count of file inode *I* with *Y*.
NO	Ignore this error condition.

6.4.7.6 LINK COUNT DIR I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for inode *I*, which is a directory, is *X* but should be *Y*. The owner *O*, mode *M*, size *S*, and modify time *T* of directory inode *I* are printed.

Possible responses to ADJUST prompt are:

YES	Replace link count of directory inode *I* with *Y*.
NO	Ignore this error condition.

6.4.7.7 LINK COUNT F I=I OWNER=O MODE=M SIZE=S MTIME=T COUNT=X SHOULD BE Y (ADJUST)

The link count for *F* inode *I* is *X* but should be *Y*. The file name *F*, owner *O*, mode *M*, size *S*, and modify time *T* are printed.

Possible responses to ADJUST prompt are:

YES	Replace link count of inode *I* with *Y*.
NO	Ignore this error condition.

6.4.7.8 UNREF FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Inode *I*, which is a file, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. If the **–n** option is not set and the file system is not mounted, empty files will be cleared automatically.

Possible responses to CLEAR prompt are:

YES	Deallocate inode *I* by zeroing its contents.
NO	Ignore this error condition.

6.4.7.9 UNREF DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Inode *I*, which is a directory, was not connected to a directory entry when the file system was traversed. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed. If the **–n** option is not set and the file system is not mounted, empty directories will be cleared automatically. Nonempty directories will not be cleared.

Possible responses to CLEAR prompt are:

YES	Deallocate inode *I* by zeroing its contents.

NO	Ignore this error condition.

6.4.7.10 BAD/DUP FILE I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with file inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed.

Possible responses to CLEAR prompt are:

YES	Deallocate inode *I* by zeroing its contents.
NO	Ignore this error condition.

6.4.7.11 BAD/DUP DIR I=I OWNER=O MODE=M SIZE=S MTIME=T (CLEAR)

Phase 1 or Phase 1b have found duplicate blocks or bad blocks associated with directory inode *I*. The owner *O*, mode *M*, size *S*, and modify time *T* of inode *I* are printed.

Possible responses to CLEAR prompt are:

YES	Deallocate inode *I* by zeroing its contents.
NO	Ignore this error condition.

6.4.7.12 FREE INODE COUNT WRONG IN SUPERBLK (FIX)

The actual count of the free inodes does not match the count in the superblock of the file system. If the **–q** option is specified, the count will be fixed automatically in the superblock.

Possible responses to FIX prompt are:

YES	Replace count in superblock by actual count.
NO	Ignore this error condition.

6.4.8 PHASE 5: CHECK FREE LIST

This phase concerns itself with the free-block list. This part lists error conditions resulting from bad blocks in the free-block list, bad free-blocks count, duplicate blocks in the free-block list, unused blocks from the file system not in the free-block list, and the total free-block count incorrect.

6.4.8.1 EXCESSIVE BAD BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks with a value less than the first data block in the file system or greater than the last block in the file system.

Possible responses to CONTINUE prompt are:

YES	Ignore rest of the free-block list and continue execution of **fsck**. This error condition will always invoke "BAD BLKS IN FREE LIST" error condition in Phase 5.
NO	Terminate program.

6.4.8.2 EXCESSIVE DUP BLKS IN FREE LIST (CONTINUE)

The free-block list contains more than a tolerable number (usually 10) of blocks claimed by inodes or earlier parts of the free-block list.

Possible responses to CONTINUE prompt are:

YES	Ignore the rest of the free-block list and continue execution of **fsck**. This error condition will always invoke "DUP BLKS IN FREE LIST" error condition in Phase 5.
NO	Terminate program.

6.4.8.3 BAD FREEBLK COUNT

The count of free blocks in a free-list block is greater than 50 or less than 0. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5.

6.4.8.4 X BAD BLKS IN FREE LIST

X blocks in the free-block list have a block number lower than the first data block in the file system or greater than the last block in the file system. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5.

6.4.8.5 X DUP BLKS IN FREE LIST

X blocks claimed by inodes or earlier parts of the free-list block were found in the free-block list. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5.

6.4.8.6 X BLK(S) MISSING

X blocks unused by the file system were not found in the free-block list. This error condition will always invoke the "BAD FREE LIST" condition in Phase 5.

6.4.8.7 FREE BLK COUNT WRONG IN SUPERBLOCK (FIX)

The actual count of free blocks does not match the count in the superblock of the file system.

Possible responses to FIX prompt are:

YES	Replace count in superblock by actual count.
NO	Ignore this error condition.

6.4.8.8 BAD FREE LIST (SALVAGE)

Phase 5 has found bad blocks in the free-block list, duplicate blocks in the free-block list, or blocks missing from the file system. If the **–q** option is specified, the free-block list will be salvaged automatically.

Possible responses to SALVAGE prompt are:

YES	Replace actual free-block list with a new free-block list. The new free-block list will be ordered to reduce time spent by the disk waiting for the disk to rotate into position.
NO	Ignore this error condition.

6.4.9 PHASE 6: SALVAGE FREE LIST

This phase concerns itself with the free-block list reconstruction. This part lists error conditions resulting from the blocks-to-skip and blocks-per-cylinder values.

6.4.9.1 Default free-block list spacing assumed

This is an advisory message indicating the blocks-to-skip is greater than the blocks-per-cylinder, the blocks-to-skip is less than 1, the blocks-per-cylinder is less than 1, or the blocks-per-cylinder is greater than 500. The default values of 9 blocks-to-skip and 400 blocks-per-cylinder are used. See **fsck**(1M) in the *UNIX System V Administrator Reference Manual* for further details.

6.4.10 CLEANUP

Once a file system has been checked, a few cleanup functions are performed. This part lists advisory messages about the file system and modify status of the file system.

6.4.10.1 X files Y blocks Z free

This is an advisory message indicating that the file system checked contained *X* files using *Y* blocks leaving *Z* blocks free in the file system.

6.4.10.2 ***** BOOT UNIX (NO SYNC!) *****

This is an advisory message indicating that a mounted file system or the root file system has been modified by **fsck**. If the UNIX system is not rebooted immediately without **sync**, the work done by **fsck** may be undone by the in-core copies of tables the UNIX system keeps.

6.4.10.3 ***** FILE SYSTEM WAS MODIFIED *****

This is an advisory message indicating that the current file system was modified by **fsck**.

7. LP SPOOLING

The line printer (LP) program is a series of commands that perform diverse spooling functions under the UNIX operating system. Since the primary LP application is off-line printing, this document focuses mainly on spooling to line printers. LP allows administrators to customize the system to spool to a collection of line printers of any type and to group printers into logical classes in order to maximize the throughput of the devices. Users are provided the capabilities of:

- Queuing and canceling print requests
- Preventing and allowing queuing to devices
- Starting and stopping LP from processing requests
- Changing configuration of printers
- Finding status of the LP system.

This chapter describes the role of an LP administrator in performing restricted functions and overseeing the smooth operation of LP.

Throughout this chapter, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX Administrator Reference Manual.* References to entries of the form **name**(N), where "N" is the number 1 or 6 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX User Reference Manual.* If "N" is a number 2 through 5 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX Programmer Reference Manual.*

7.1 Overview of LP Features

7.1.1 Definitions

Several terms must be defined before presenting a brief summary of LP commands. The LP was designed with the flexibility to meet the needs of users on different UNIX systems. Changes to the LP configuration are performed by the **lpadmin**(1M) command.

LP makes a distinction between printers and printing devices. A *device* is a physical peripheral device or a file and is represented by a full UNIX system pathname. A *printer* is a logical name that represents a device. At different points in time, a printer may be associated with different devices. A *class* is a name given to an ordered list of printers. Every class must contain at least one printer. Each printer may be a member of zero or more classes. A *destination* is a printer or a class. One destination may be designated as the *system default destination.* The **lp**(1) command will direct all output to this destination unless the user specifies otherwise. Output that is routed to

a printer will be printed only by that printer, whereas output directed to a class will be printed by the first available class member.

Each invocation of **lp** creates an output request that consists of the files to be printed and options from the lp command line. An interface program which formats requests must be supplied for each printer. The LP scheduler, **lpsched**(1M), services requests for all destinations by routing requests to interface programs to do the printing on devices. An LP configuration for a system consists of devices, destinations, and interface programs.

7.1.2 Commands

7.1.2.1 Commands for General Use

The **lp**(1) command is used to request the printing of files. It creates an output request and returns a request id of the form

dest–seqno

to the user, where *seqno* is a unique sequence number across the entire LP system and *dest* is the destination where the request was routed.

Cancel is used to cancel output requests. The user supplies request ids as returned by **lp** or printer names, in which case the currently printing requests on those printers are canceled.

Disable prevents **lpsched** from routing output requests to printers.

Enable(1) allows **lpsched** to route output requests to printers.

7.1.2.2 Commands for LP Administrators

Each LP system must designate a person or persons as LP administrator to perform the restricted functions listed below. Either the superuser or any user who is logged into the UNIX system as **lp** qualifies as an LP administrator. All LP files and commands are owned by **lp** except for **lpadmin** and **lpsched** which are owned by root. The following commands will be described in more detail later in this chapter.

lpadmin(1M)	Modifies LP configuration. Many features of this command cannot be used when **lpsched** is running.
lpsched(1M)	Routes output requests to interface programs which do the printing on devices.
lpshut	Stops **lpsched** from running. All printing activity is halted, but other LP commands may still be used.
accept(1M)	Allows **lp** to accept output requests for destinations.

reject	Prevents **lp** from accepting requests for destinations.
lpmove	Moves output requests from one destination to another. Whole destinations may be moved at one time. This command cannot be used when **lpsched** is running.

7.2 Building LP

All LP commands are built from source code that resides in the */usr/src/cmd/lp* directory including the make file, *lp.mk*. Unless some of the definitions in *lp.mk* are changed, LP may be installed only by the superuser. Before installing a new LP system, make sure there is a login called *lp* on your system and that the spool directory, */usr/spool/lp*, does not exist. To install LP, perform the following:

```
cd /usr/src/cmd/lp
make –f lp.mk install
```

This builds all LP commands and creates an initial LP configuration consisting of no printers, classes, or default destination. LP must be configured by an LP administrator using the **lpadmin** command in order to create a useful spooler.

In addition, add the following code to */etc/rc*:

```
rm –f /usr/spool/lp/SCHEDLOCK
/usr/lib/lpsched
echo "LP scheduler started"
```

This starts the LP scheduler each time that the UNIX system is restarted.

Several variables in *lp.mk* may be changed before installing LP to customize the system:

Variable	*Default Value*	*Meaning*
SPOOL	*/usr/spool/lp*	spool directory
ADMIN	*lp*	logname of LP Administrator
GROUP	*bin*	group owning LP commands/data
ADMDIR	*/usr/lib*	commands of administrator
USRDIR	*/usr/bin*	user commands reside here

If an existing LP spool directory is corrupted (but not the LP programs) or if it needs to be rebuilt from scratch, make sure that **lpsched** is not running and perform the following as superuser:

1. Make copies of any interface programs that are not standard LP software. **DO NOT** make these copies underneath the spool directory. The pathname for printer "p" is */usr/spool/lp/interface/p*.
2. rm –fr /usr/spool/lp
3. Make *–f lp.mk* new. (This recreates the bare LP configuration described above.)

PRECAUTIONS

1. Some LP commands invoke other LP commands. Moving them after they are built will cause some commands to fail.
2. The files under the SPOOL directory should be modified **only by LP commands**.
3. All LP commands require set-user-id permission. If this is removed, the commands will fail.

7.3 Configuring LP—the "lpadmin" Command

Changes to the LP configuration should be made by using the **lpadmin** command and not by hand. **Lpadmin** will not attempt to alter the LP configuration when **lpsched** is running, except where explicitly noted below.

7.3.1 Introducing New Destinations

The following information must be supplied to **lpadmin** when introducing a new printer:

1. The printer name (**–p** printer) is an arbitrary name which must conform to the following rules:
   - It must be no longer than 14 characters.
   - It must consist solely of alphanumeric characters and underscores.
   - It must not be the name of an existing LP destination (printer or class).
2. The device associated with the printer (**–v** device). This is the pathname of a hard-wired printer, a login terminal, or other file that is writable by **lp**.
3. The printer interface program. This may be specified in one of three ways:
   - It may be selected from a list of model interfaces supplied with LP (**–m** model).

- It may be the same interface that an existing printer uses (**–e** printer).
- It may be a program supplied by the LP administrator (**–i** interface).

Information which need not always be supplied when creating a new printer includes:

1. The user may specify **–h** to indicate that the device for the printer is hardwired or the device is the name of a file (this is assumed by default). If, on the other hand, the device is the pathname of a login terminal, then **–l** must be included on the command line. This indicates to *lpsched* that it must automatically disable this printer each time *lpsched* starts running. This fact is reported by *lpstat* when it indicates printer status:

   ```
   $ lpstat –pa
   printer a (login terminal) disabled Oct 31 11:15 –
         disabled by scheduler: login terminal
   ```

   This is done because device names for login terminals can be (and usually are) associated with different physical devices from day to day. If the scheduler did not take this action, somebody might log in and be surprised that LP is spooling to his/her terminal!

2. The new printer may be added to an existing class or added to a new class (**–c**class). New class names must conform to the same rules for new printer names.

EXAMPLES

The following examples will be referenced by further examples in later sections.

1. Create a printer called pr1 whose device is */dev/printer* and whose interface program is the model hp interface:

   ```
   $ /usr/lib/lpadmin –ppr1 –v/dev/printer –mhp
   ```

2. Add a printer called pr2 whose device is */dev/tty22* and whose interface is a variation of the model prx interface. It is also a login terminal:

   ```
   $ cp /usr/spool/lp/model/prx xxx
     < edit xxx >
   $ /usr/lib/lpadmin –ppr2 –v/dev/tty22 –ixxx –l
   ```

3. Create a printer called pr3 whose device is */dev/tty23*. The pr3 will be added to a new class called cl1 and will use the same interface as printer pr2:

```
$ /usr/lib/lpadmin –ppr3 –v/dev/tty23 –epr2 –ccl1
```

7.3.2 Modifying Existing Destinations

Modifications to existing destinations must always be made with respect to a printer name (**–p**printer). The modifications may be one or more of the following:

1. The device for the printer may be changed (**–v**device). If this is the only modification, then this may be done even while *lpsched* is running. This facilitates changing devices for login terminals.
2. The printer interface program may be changed (**–m**model, **–e**printer, **–i**interface).
3. The printer may be specified as hardwired (**–h**) or as a login terminal (**–l**).
4. The printer may be added to a new or existing class (**–c**class).
5. The printer may be removed from an existing class (**–r**class). Removing the last remaining member of a class causes the class to be deleted. No destination may be removed if it has pending requests. In that case, **lpmove** or **cancel** should be used to move or delete the pending requests.

EXAMPLES

These examples are based on the LP configuration created by those in the previous section.

1. Add printer pr2 to class cl1:

   ```
   $ /usr/lib/lpadmin –ppr2 –ccl1
   ```

2. Change pr2's interface program to the model prx interface, change its device to */dev/tty24*, and add it to a new class called cl2:

   ```
   $ /usr/lib/lpadmin –ppr2 –mprx –v/dev/tty24 –ccl2
   ```

   Note that printers pr2 and pr3 now use different interface programs even though pr3 was originally created with the same interface as pr2. Printer pr2 is now a member of two classes.

3. Specify printer pr2 as a hard-wired printer:

   ```
   $ /usr/lib/lpadmin –ppr2 –h
   ```

4. Add printer pr1 to class cl2:

   ```
   $ /usr/lib/lpadmin –ppr1 –ccl2
   ```

   The members of class cl2 are now pr2 and pr1, in that order. Requests routed to class cl2 will be serviced by pr2 if both pr2 and pr1

are ready to print; otherwise, they will be printed by the one which is next ready to print.

5. Remove printers pr2 and pr3 from class cl1:

   $ /usr/lib/lpadmin –ppr2 –rcl1
   $ /usr/lib/lpadmin –ppr3 –rcl1

   Since pr3 was the last remaining member of class cl1, the class is removed.

6. Add pr3 to a new class called cl3.

   $ /usr/lib/lpadmin –ppr3 –ccl3

7.3.3 Specifying the System Default Destination

The system default destination may be changed even when **lpsched** is running.

EXAMPLES

1. Establish class cl1 as the system default destination:

   $ /usr/lib/lpadmin –dcl1

2. Establish no default destination:

   $ /usr/lib/lpadmin –d

7.3.4 Removing Destinations

Classes and printers may be removed only if there are no pending requests that were routed to them. Pending requests must either be canceled using **cancel** or moved to other destinations using **lpmove** before destinations may be removed. If the removed destination is the system default destination, then the system will have no default destination until the default destination is respecified. When the last remaining member of a class is removed, then the class is also removed. The removal of a class never implies the removal of printers.

EXAMPLES

1. Make printer pr1 the system default destination:

   $ /usr/lib/lpadmin –dpr1

   Remove printer pr1:

   $ /usr/lib/lpadmin –xpr1

   Now there is no system default destination.

2. Remove printer pr2:

$ /usr/lib/lpadmin –xpr2

Class cl2 is also removed since pr2 was its only member.

3. Remove class cl3:

   $ /usr/lib/lpadmin –xcl3

   Class cl3 is removed, but printer pr3 remains.

7.4 Making an Output Request—the "lp" Command

Once LP destinations have been created, users may request output by using the **lp** command. The request id that is returned may be used to see if the request has been printed or to cancel the request.

The LP program determines the destination of a request by checking the following list in order:

- If the user specifies **–d***dest* on the command line, then the request is routed to *dest*.
- If the environment variable **LPDEST** is set, the request is routed to the value of *LPDEST*.
- If there is a system default destination, then the request is routed there.
- The request is rejected.

EXAMPLES

1. There are at least four ways to print the password file on the system default destination:

   lp /etc/passwd
   lp < /etc/passwd
   cat /etc/passwd | lp
   lp –c /etc/passwd

   The last three ways cause copies of the file to be printed, whereas the first way prints the file directly. Thus, if the file is modified between the time the request is made and the time it is actually printed, then the changes will be reflected in the output.

2. Print two copies of file abc on printer xyz and title the output "my file":

   pr abc | lp –dxyz –n2 –t"my file"

3. Print file xxx on a Diablo* 1640 printer called zoo in 12-pitch and write to the user's terminal when printing has completed:

   lp –dzoo –o12 –w xxx

   In this example, "12" is an option that is meaningful to the model Diablo 1640 interface program that prints output in 12-pitch mode [see **lpadmin**(1M)].

7.5 Finding LP Status—LPSTAT

The **lpstat** command is used to find status information about LP requests, destinations, and the scheduler.

EXAMPLES

1. List the status of all pending output requests made by this user:

   lpstat

   The status information for a request includes the request id, the logname of the user, the total number of characters to be printed, and the date and time the request was made.

2. List the status of printers p1 and p2:

   lpstat –pp1,p2

7.6 Cancleing Request—CANCEL

The LP requests may be canceled using the **cancel** command. Two kinds of arguments may be given to the command—request ids and printer names. The requests named by the request ids are canceled and requests that are currently printing on the named printers are canceled. Both types of arguments may be intermixed.

EXAMPLE

Cancel the request that is now printing on printer xyz:

cancel xyz

If the user that is canceling a request is not the same one that made the request, then mail is sent to the owner of the request. LP allows any user to cancel requests in order to eliminate the need for users to find LP administrators when unusual output should be purged from printers.

---

* Registered trademark of Xerox Corporation

7.7 Allowing and Refusing Requests—ACCEPT and REJECT

When a new destination is created, **lp** will reject requests that are routed to it. When the LP administrator is sure that it is set up correctly, he or she should allow **lp** to accept requests for that destination. The **accept** command performs this function.

Sometimes it is necessary to prevent **lp** from routing requests to destinations. If printers have been removed or are waiting to be repaired or if too many requests are building for printers, then it may be desirable to cause **lp** to reject requests for those destinations. The **reject** command performs this function. After the condition that led to the rejection of requests has been remedied, the **accept** command should be used to allow requests to be taken again.

The acceptance status of destinations is reported by the **–a** option of **lpstat**.

EXAMPLES

1. Cause **lp** to reject requests for destination xyz:

   /usr/lib/reject –r"printer xyz needs repair" xyz

   Any users that try to route requests to xyz will encounter the following:

   ```
   $ lp –dxyz file
   lp: can not accept requests for destination "xyz"
         -- printer xyz needs repair
   ```

2. Allow **lp** to accept requests routed to destination xyz:

   /usr/lib/accept xyz

7.8 Allowing and Inhibiting Printing—ENABLE and DISABLE

The **enable** command allows the LP scheduler to print requests on printers. That is, the scheduler routes requests only to the interface programs of enabled printers. Note that it is possible to enable a printer and at the same time prevent further requests from being routed to it.

The **disable** command will undo the effects of the **enable** command. It prevents the scheduler from routing requests to printers, independently of whether or not **lp** is allowing them to accept requests. Printers may be disabled for several reasons including malfunctioning hardware, paper jams, and end of day shutdowns. If a printer is busy at the time it is disabled, then the request that was printing will be reprinted in its entirety either on another printer (if the request was originally routed to a class of printers) or on the same one when the printer is reenabled. The **–c** option causes the currently printing requests on busy printers to be canceled in addition to disabling the printers. This is useful if strange output is causing a printer to behave abnormally.

EXAMPLE

Disable printer xyz because of a paper jam:

```
$ disable –r"paper jam" xyz
printer "xyz" now disabled
```

Find the status of printer xyz:

```
$ lpstat –pxyz
printer "xyz" disabled since Jan 5 10:15 –
        paper jam
```

Now, reenable xyz:

```
$ enable xyz
printer "xyz" now enabled
```

7.9 Moving Requests Between Destinations—LPMOVE

Occasionally, it is useful for LP administrators to move output requests between destinations. For instance, when a printer is down for repairs, it may be desirable to move all of its pending requests to a working printer. This is one way to use the **lpmove** command. The other use of this command is to move specific requests to a different destination. **Lpmove** will refuse to move requests while the LP scheduler is running.

EXAMPLES

1. Move all requests for printer abc to printer xyz:

   ```
   $ /usr/lib/lpmove abc xyz
   ```

   All of the moved requests are renamed from abc-nnn to xyz-nnn. As a side effect, destination abc is no longer accepting further requests.

2. Move requests zoo-543 and abc-1200 to printer xyz:

   ```
   $ /usr/lib/lpmove zoo-543 abc-1200 xyz
   ```

   The two requests are now renamed xyz-543 and xyz-1200.

7.10 Stopping and Starting the Scheduler—LPSHUT and LPSCHED

Lpsched is the program that routes the output requests that were made with **lp** through the appropriate printer interface programs to be printed on line printers. Each time the scheduler routes a request to an interface program, it records an entry in the log file, */usr/spool/lp/log*. This entry contains the logname of the user that made the request, the request id, the name of the printer that the request is being printed on, and the date and time that printing first started. In the case that a request has been restarted, more than one entry in the log file may refer to the request. The scheduler also records error messages in the log file. When **lpsched** is started, it

renames */usr/spool/lp/log* to */usr/spool/lp/oldlog* and starts a new log file.

No printing will be performed by the LP system unless **lpsched** is running. Use the command

lpstat –r

to find the status of the LP scheduler.

Lpsched is normally started by the **/etc/rc** program as described above and continues to run until the UNIX system is shut down. The scheduler operates in the */usr/spool/lp* directory. When it starts running, it will exit immediately if a file called *SCHEDLOCK* exists. Otherwise, it creates this file in order to prevent more than one scheduler from running at the same time.

Occasionally, it is necessary to shut down the scheduler in order to reconfigure LP or to rebuild the LP software. The command

/usr/lib/lpshut

causes **lpsched** to stop running and terminates all printing activity. All requests that were in the middle of printing will be reprinted in their entirety when the scheduler is restarted.

To restart the LP scheduler, use the command

/usr/lib/lpsched

Shortly after this command is entered, **lpstat** should report that the scheduler is running. If not, it is possible that a previous invocation of **lpsched** exited without removing *SCHEDLOCK*, so try the following:

rm –f /usr/spool/lp/SCHEDLOCK
/usr/lib/lpsched

The scheduler should be running now.

7.11 Printer Interface Programs

Every LP printer must have an interface program which does the actual printing on the device that is currently associated with the printer. Interface programs may be shell procedures, C programs, or any other executable program. The LP model interfaces are all written as shell procedures and can be found in the */usr/spool/lp/model* directory. At the time **lpsched** routes an output request to a printer P, the interface program for P is invoked in the directory */usr/spool/lp* as follows:

interface/P id user title copies options file ...
where
id is the request id returned by **lp**
user is logname of user who made the request
title is optional title specified by the user
copies is number of copies requested by user
options is a blank-separated list of class or
printer-dependent options specified by user
file is the full pathname of a file to be printed

EXAMPLES

The following examples are requests made by user "smith" with a system default destination of printer "xyz". Each example lists an **lp** command line followed by the corresponding command line generated for printer xyz's interface program:

1. lp /etc/passwd /etc/group
   interface/xyz xyz-52 smith "" 1 "" /etc/passwd /etc/group
2. pr /etc/passwd | lp –t"users" –n5
   interface/xyz xyz–53 smith users 5 "" /usr/spool/lp/request/xyz/d0–53
3. lp /etc/passwd –oa –ob
   interface/xyz xyz–54 smith "" 1 "a b" /etc/passwd

When the interface program is invoked, its standard input comes from /dev/null and both the standard output and standard error output are directed to the printer's device. Devices are opened for reading as well as writing when file modes permit. In the case where a device is a regular file, all output is appended to the end of the file.

Given the command line arguments and the output directed to a device, interface programs may format their output in any way they choose. Interface programs must ensure that the proper stty modes (terminal characteristics such as baud rate, output options, etc.) are in effect on the output device. This may be done in a shell interface only if the device is opened for reading:

stty mode ... <&1

That is, take the standard input for the stty command from the device.

When printing has completed, it is the responsibility of the interface program to exit with a code indicative of the success of the print job. Exit codes are interpreted by **lpsched** as follows:

CODE	MEANING TO LPSCHED

0	The print job has completed successfully.
1 to 127	A problem was encountered in printing this particular request (e.g., too many nonprintable characters). This problem will not affect future print jobs. **Lpsched** notifies users by mail that there was an error in printing the request.
greater than 127	These codes are reserved for internal use by **lpsched**. Interface programs must not exit with codes in this range.

When problems that are likely to affect future print jobs occur (e.g., a device filter program is missing), the interface programs would be wise to disable printers so that print requests are not lost. When a busy printer is disabled, the interface program will be terminated with signal 15.

7.12 Setting Up Hard-Wired Devices and Login Terminals as LP Printers

7.12.1 Hard-wired Devices

As an example of how to set up a hard-wired device for use as an LP printer, consider using tty line 15 as printer xyz. As superuser, perform the following:

1. Avoid unwanted output from non-LP processes and ensure that LP can write to the device:

   $ chown lp /dev/tty15
   $ chmod 600 /dev/tty15

2. Change */etc/inittab* so that tty15 is not a login terminal. In other words, ensure that */etc/getty* is not trying to log users in at this terminal. Change the entries for tty15 to:

   15:2:off:/etc/getty -t60 tty15 1200

   Enter the command:

   $ telinit Q

   If there is currently an invocation of */etc/getty* running on tty15, kill it. When the UNIX system is rebooted, tty15 will be initialized with default stty modes. Thus, it is up to LP interface programs to establish the proper baud rate and other stty modes for correct printing to occur.

3. Introduce printer xyz to LP using the model prx interface program:

   $ /usr/lib/lpadmin –pxyz –v/dev/tty15 –mprx

4. When xyz is created, it will initially be disabled and **lp** will be rejecting requests routed to it. If it is desired, allow **lp** to accept requests for xyz:

   /usr/lib/accept xyz

   This will allow requests to build up for xyz and to be printed when it is enabled at a later time.

5. When it is desired for printing to occur, be sure that the printer is ready to receive output. For several printers, this means that the top of form has been adjusted and that the printer is on-line. Enable printing to occur on xyz:

   enable xyz

   When requests have been routed to xyz, they will begin printing.

7.12.2 Login Terminals

Login terminals may also be used as LP printers. To do this for a Diablo 1640 terminal called abc, perform the following:

1. Introduce printer abc to LP using the model 1640 interface program:

   $ /usr/lib/lpadmin –pabc –v/dev/null –m1640 –l

   Note that */dev/null* is used as abc's device because we will specify the actual device each time that abc is enabled. This device may be different from day to day. When abc is created, it will initially be disabled; and **lp** will be rejecting requests routed to it. If it is desired, allow **lp** to accept requests for abc:

   /usr/lib/accept abc

   This will allow requests to build up for abc and to be printed when it is enabled at a later time. It is not advisable to enable abc for printing, however, until the following steps have been taken.

2. Log terminal in if this has not already been done.

3. Assuming the **tty**(1) command reports that this terminal is */dev/tty02*, associate this device with printer abc:

   $ /usr/lib/lpadmin –pabc –v/dev/tty02

   Note that **lpadmin** may be used only by an LP administrator. If it is desired for other users to routinely perform this step, then an LPA may establish a program owned by **lp** or by **root** with set-user-id permission that performs this function.

4. When it is desired for printing to occur, be sure that the printer is ready to receive output. For several printers, this means that the top of form has been adjusted. Enable printing to occur on abc:

   enable abc

   When requests have been routed to abc, they will begin printing.

5. When all printing has stopped on abc or when you want it back as a regular login terminal, you may prevent it from printing more output:

```
$ disable abc
printer "abc" now disabled
```

If abc is enabled when the UNIX system is rebooted or when **lpsched** is restarted, it will be disabled automatically.

7.13 Summary

The administrative functions of the LP administrator have been described in detail. These functions include configuring and reconfiguring LP; maintaining printer interface programs; accepting, rejecting, and moving print requests; stopping and starting the LP scheduler; and enabling and disabling printers. LP offers administrators the following advantages over other centrally supported printer packages:

- Printers may be grouped into classes.
- LP may be configured to meet the needs of each site.
- Administrators may supply interface programs to format output in any way desirable.
- LP functions are performed by simple commands and not by hand.

8. VIRTUAL PROTOCOL MACHINE

This document describes the UNIX Virtual Protocol Machine (VPM). VPM is a general-purpose UNIX interface for synchronous communications lines. VPM allows link-level protocols such as BISYNC and HDLC to be inplemented on the Plexus ICP microcomputer in a high-level language. The hardware required to support VPM is a Plexus host computer, and an ICP. The link-level communications protocol is excuted by the VPM interpreter running in the Plexus ICP. This implementation technique leads to a portable protocol representation and efficient protocol execution.

The VPM software consists of a protocol compiler, a UNIX driver, an interpreter that executes in the Plexus ICP, and several utility programs. The compiler, which executes in the host computer, translates a protocol described in a high-level language into a load module for the ICP. The load module contains the VPM interpreter and a compiled representation of the protocol. The interpreter executes the protocol, communicates with the UNIX driver in the host computer, and controls the communications line interface.

The first release of VPM supported a large class of protocols collectively known as BISYNC. These protocols are distinguished by the use of control characters to provide framing and transparency. At the frame level, these protocols operate in a half-duplex manner, although they sometimes use full-duplex communications facilities to reduce the time required to reverse the direction of transmission.

The release of VPM adds support for bit-oriented, full-duplex protocols. This class of protocols includes IBM's Synchronous Data Link Control (SDLC) and the international standard High-Level Data Link Control (HDLC). LAPB, a subset of HDLC which is the link-level protocol specified in the BX.25 Bell System Standard, has been implemented using VPM and is available with the Sys5 release. The interpreter used for bit-oriented protocols is different from that used for character-oriented (BISYNC) protocols. The appropriate interpreter is selected by means of a compiler option.

Other features of VPM include:

1. an increase in the number of transmit and receive buffers which the interpreter can accept at one time.
2. additional debugging facilities.
3. provisions for interprocess communication between the protocol script and a UNIX driver or a user process, and
4. a cleaner separation of functions in the UNIX driver to facilitate tailoring of VPM to particular applications.

8.1 Support for Bit-Oriented Protocols

The capability to use bit-oriented protocols such as HDLC is provided by a new set of communications primitives. These primitives are frame-oriented and non-blocking, whereas the BISYNC primitives are character-oriented and blocking. The new primitives are fully described in the attached manual entry for *vpmc*(1C). An overview of these primitives follows.

The VPM interpreter maintains a set of queues for transmit buffers. When a transmit buffer is passed to the ICP by the UNIX driver, the buffer is appended to the unopened-transmit-buffer queue. The protocol script in the ICP obtains a transmit buffer from the unopened-transmit-buffer queue by means of the *getxfrm* primitive; the buffer is then said to be *open*. In order to get (open) a transmit buffer, the script must provide a transmit-sequence number. This sequence number must be distinct from the sequence number currently assigned to every other currently-open transmit buffer. This sequence number is used to identify the buffer for subsequent calls to the *xmtfrm* and *rtnxfrm* primitives. The *xmtfrm* primitive initiates transmission of the specified buffer, using the control information specified by a previous *setctl* primitive. Transmission proceeds asynchronously. The script can test for completion of an output transfer by means of the *xmtbusy* primitive. Open transmit buffers can be transmitted any number of times. When the script decides that a buffer has successfully been received at the destination, it notifies the interpreter by means of the *rtnxfrm* primitive. This causes the buffer to be placed on the transmit-buffer-return queue; the buffer is then no longer considered to be open and the sequence number can be reused. The driver is notified as soon as possible that the buffer has been closed. The buffer is then removed from the transmit-buffer-return queue.

When a receive buffer is passed to the ICP by the driver, the buffer is placed on the empty-receive-buffer queue. When the first byte of a new frame arrives, an empty receive buffer is obtained from the empty-receive-buffer queue and the incoming characters are placed into the buffer as they arrive. An incoming frame will be discarded if the frame is too short (less than four bytes including CRC), if the frame is too long to fit in the receive buffer, or if the CRC is incorrect. If a frame is received successfully, the buffer is placed on the completed-receive frame queue, otherwise the buffer is returned to the empty-receive-buffer queue. When the script executes a *rcvfrm* primitive, the buffer at the head of the completed-receive-frame queue is removed from that queue and becomes the current receive buffer. If the script subsequently executes a *rtnrfrm* primitive before executing another *rcvfrm* primitive, the current receive buffer is placed on the receive-buffer-return queue. If the script executes a *rcvfrm* primitive before executing a *rtnrfrm* primitive, the current receive buffer, if any, is returned to the empty-receive-frame queue. Buffers on the receive-buffer-return queue

are returned to the driver at the first opportunity. If the empty-receive-buffer queue is empty when the first byte of a new frame is rceived, the first five bytes of the frame are retained in a staging area and the remainder of the frame is discarded. This allows a protocol script to receive a control frame (up to seven bytes including CRC) when no data buffer is available. When the next *rcvfrm* primitive is executed, the script will receive the information in the staging area along with an indication that the remainder of the frame has been discarded. If another frame arrives while the staging area is thus occupied, the new frame is discarded entirely.

A count is kept of the number of frames discarded for each reason. These counters may be read and reset from the host computer.

8.1.1 The VPM Split Driver

Since the VPM interpreter and a protocol script generally use most of the memory of the ICP any higher levels of protocol that are required must be executed by the host CPU. The purpose of the VPM split driver is to provide a framework in which higher-level protocols can be implemented conveniently using low-level routines in the VPM driver to communicate with the interpreter in the ICP.

A set of functions has been written that provides a general-purpose interface to the link-level protocol being executed by the interpreter in the ICP. Their capabilities include a means to queue transmit and empty receive buffers for use by the protocol script in the ICP, to start and stop the script, and to send commands to and receive reports from the script. A means of getting a copy of and resetting the VPM interpreter's error counters is also provided. These functions will be referred to as interface functions or collectively as the interface module. Appendix I contains a description of each of these routines.

To implement higher levels of a protocol as a UNIX device driver, a set of routines must be written to implement the standard UNIX system calls: *open, close, read, write,* and *ioctl* as well as the required protocol. These routines will be referred to as protocol functions or collectively as a protocol module. The standard VPM driver does not implement a higher-level protocol but instead provides a transparent user interface that can be used by applications that supply their own higher levels of protocol. This driver can be used as an example for those interested in writing a different protocol module. Appendix 2 contains a description of these routines.

At lease two other protocol modules have been written thus far. They are the Synchronous Terminal Interface [4, *st*(4)], and the BANCS THP Interface.

VPM allows up to four different protocol modules to be executing simultaneously. One ICP and one interface-module minor device* is required

for each protocol being executed. Any number of protocol modules may be implemented, but no more than four can be in use at any one time since no more than four ICPs are supported. In general, each protocol module can have up to 256 minor devices. The VPM protocol module, however, can have at most 16 minor devices; this restriction is due to the fact that the minor device number of the VPM protocol module is used not only to specify the VPM minor device but also to specify the interface-module minor device and the ICP minor device. The low-order four bits of the protocol-module minor device number determine the protocol-module minor device; the next two bits determine the interface-module minor device; the next two bits determine the ICP minor device.

Transmit buffers and receive buffers are passed between the VPM interpreter, the interface module, and the protocol module by means of pointers to data structures known as *buffer descriptors*. The buffer-descriptor structure is defined as follows:

```
struct vpmbd {
    short c_ct;                 /* Buffer size */
    short d_adres;              /* Low-order 16 bits of buffer address */
    char  d_hbits;              /* High-order 2 bits of buffer address  */
    char  d_sta;                /* Protocol-dependent */
    char  d_type;               /* Protocol-dependent */
    char  d_dev;                /* Protocol-dependent */
    struct buf *d_buf;          /* Pointer to system buffer descriptor */
    int   d_bos;                /* Index of next byte in buffer */
    int   d_vpmtdev;            /* Minor device number */
}
```

For empty receive buffers, *c_ct* must be equal to the buffer size in bytes; for transmit buffers, *c_ct* must be equal to the number of bytes to be transmitted. When a receive buffer is returned to the protocol module, *c_ct* is equal to the number of data bytes in the buffer. *D_adres* and *d_hbits* must contain an 18-bit MULTIBUS-mapped buffer address; the low-order 16 bits must be in *d_adres* and the high-order two bits must be in the low-order two bits of *id_hbits*. *D_type*, *d_sta*, and *d_dev* are protocol-dependent; when using the BISYNC interpreter these three bytes may be read and modified by the protocol script. See the discussion of *getxbuf*, *getrbuf*, *rtnxbuf*, and *rtnrbuf* in *vpmc*(1C). *D_buf* contains a pointer to a system buffer descripter; this is used to return the buffer to the system buffer pool. *D_bos* is the index of the first byte in the buffer not yet returned to the user. *D_vpmdev* is the minor device number of the protocol-module minor device to which the buffer is allocated.

8.1.2 The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver and the protocol script to aid in debugging new protocol modules and protocol scripts. It may also be used to debug other drivers or system code not related to the VPM driver. This driver can be configured to have a number of minor devices. Each minor device provides a means by which a user program can read data generated by functions within the operating system. This data is recorded by calls to *trsave* as described in Appendix 3. Each call to *trsave* generates a unit of data known as an *event record* which consists of a channel number (one byte), a count (one byte) and *count* bytes of data. The channel number can be used to multiplex up to l6 data streams on each minor device.

Associated with each minor device of the trace driver is a *clist* queue, which is used to save event records provided a user program has that minor device open and has enabled the channel to which the event records were written. Channels may be enabled in any combination, using the *ioctl* command VPMTRCO. See the manual entry for *trace*(4). While a minor device read queue is full, event records for that minor device are discarded. Appendix 3 contains a description of each trace-driver routine.

Minor device 0 of the trace driver is used by the VPM driver to record a variety of debugging information generated within the VPM driver and also to record the data generated by the *trace* primitive in a protocol script. Minor device 1 of the trace driver is used to record the information generated by the *snap* primitive in a protocol script. The *vpmtrace* and *vmpsnap* commands are available for reading and formatting the data passed via these two minor devices. These two commands are described in the attached manual entry for *vpmstart*(IC). Appendix 4 contains a description of the VPM driver event trace.

8.1.3 Miscellaneous Improvements

Two new primitives have been added to the protocol language to allow communication between the link-level protocol script in the ICP and a higher-level protocol implemented in a user program or a VPM protocol module. The *getcmd* primitive allows the script to receive a four-byte command from a user program or a protocol module. The standard VPM protocol module allows a user program to pass a command to the script via an *ioctl* system call. Other VPM protocol modules can pass a command to the script by calling the *vpmcmd* routine in the VPM interface module. The *rtnrpt* primitive allows the script in the ICP to send a four-byte report to a protocol module or to a user program. The standard VPM protocol module allows a user program to receive a script report by means of an *ioctl* system call. A protocol module can receive reports from the interface module by calling the *vpmrpt* routine of the VPM interface module.

The *trace* primitive of the protocol language has been augmented to allow two arguments. The form with one argument is still supported; if only one argument is given, the second argument is assumed to be zero. A *snap* primitive has been added. This primitive causes four bytes of data from the script followed by a four-byte time stamp to be placed on the read queue for trace driver minor device 1.

The *time* primitive that allows a script to initialize a timer or test its current value. If the argument to *timer* is non-zero, the timer is initialized with the value of the argument. The timer is decremented ten times a second until it reaches zero. If the timer primitive is called with an argument of zero, it returns the current value of the timer. This value is zero if the timer has expired, otherwise non-zero.

The interpreter would accept at most one transmit buffer and one receive buffer at any given time. In the interpreter will accept up to four transmit buffers and four receive buffers at a time. This applies to the bit-oriented (HDLC) interpreter only.

8.2 Implementation

This section has two parts: the first gives configuration guidelines for VPM and the ICPs and tells how to install and boot VPM; the second gives procedures for compiling and link-loading protocol scripts.

8.2.1 Installing and Booting VPM

Each ICP can support up to eight users. If VPM is also installed, an additional dedicated ICP is required as the VPM. Therefore, a P/60 with 32 users and a VPM requires FIVE ICPs. A P/15, P/20 or P/35 with 8 users and a VPM requires TWO ICPs.

For all systems, the lowest numbered ICP must be the VPM. Thus while VPM is operating, ports 0-7 may not be used as TTY ports; users' TTY ports must be numbered beginning with 8. For example, on a P/60 with 16 users and VPM, the VPM uses ports 0-7 and users' TTYs are numbered 8-23. The port assignments are changed by modifying the file **/etc/inittab** for use with VPM.

If you want VPM in operation only intermittently, the VPM ICP can function to a limited extent as a TTY ICP; the single wire-wrapped device and the parallel port are unavailable. In other words, seven devices remain available on the VPM ICP when VPM is not operating. You can switch back and forth by alternating between versions of **/etc/rc** that call different versions of **/etc/inittab**.

Six basic steps are required to bring up VPM. The following sections describe each step.

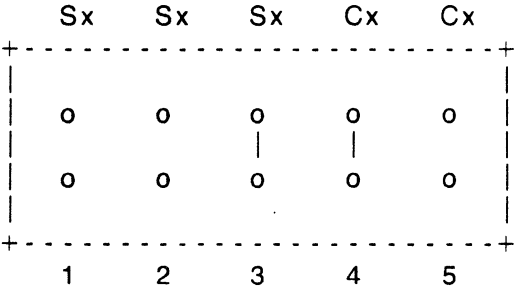
1. Perform several small hardware changes.
2. Create the VPM devices.
3. Modify **/etc/inittab**.
4. Modify **/etc/rc**.
5. Boot Sys3.
6. Modify the VPM library for switched or constant carrier.

8.2.1.1 Hardware Installation

The ICP(s) that are to be the VPM(s) require the following special hardware features.

Note that port 2 on ICP0 is the recommended VPM port. Further references to a port will be to port 2.

1. The line(s) that are to be synchronous require the Carrier Detect signal and Clear to Send signal to be strapped on the ICP. (See *Plexus User's Manual.*)
2. The pin-pairs 3 and 4 must be jumpered for synchronous transmission. Set up the 10-pin jumper network to look like the diagram below.



3. Make sure your VPM ICP is part number 60-00079 Rev Z, 60-00079-1 (any rev), 60-00085 (any rev), or 60-00091 (any rev). Earlier ICPs must be upgraded in order to be used for VPM.
4. Each port using synchronous transmission must be configured for external clock. This is accomplished via a pair of three-pin jumper networks for each port on that ICP. The jumper networks are designated Tx and Xx, where 'x' means the port number.
   a. If your VPM ICP is part number 60-00079 Rev Z, 60-00079-1 (any rev), or 60-00085 (any rev), the following describes the procedure for configuring a part for external clocking.

Normally, the center pin (A) is jumpered to either outer pin (B or C) using a two-pin female jumper block. However, configuring a port for external clocking requires wirewrapping pin B to pin C on both jumper networks for the port.

For example, to strap port 2 for external clock, do the following:

- Remove the two-pin jumper blocks from T2 and from X2.
- Using a wirewrap tool, connect T2-B to T2-C.
- Using a wirewrap tool, connect X2-B to X2-C.

b. If your VPM ICP is part number 60-00091 (any rev), the procedure for configuring a port for external clocking is simpler. On this ICP, the common pin(c) has been moved to facilitate changing the clocking mode of the port. On previous ICP's, the common was located between the other two pins (A and B), making wirewrapping necessary for external clocking. On this ICP, switching to external clocking only requires moving the jumper blocks from pin pair A-C to B-C.

For example, to strap port 2 for external clock, do the following:

- Remove the two-pin jumper blocks from T2 and from X2.
- Jumper pins X2-B and X2-C using the jumper block.
- Jumper pins T2-B and T2-C using the jumper block.

c. The connection cable between the VPM ICP port and the synchronous modem is a specially strapped Plexus RS232C modem cable. The following RS232C modem cable leads must be strapped:

VPM ICP Port RS232 Leads	to	Modem Cable RS232 Leads
2		3
3		2
4		5
5		4
6		20
7		7
8		8
15		15
17		17
20		6

d. If your ICP is part number 60-00085 rev H or later, you must set switches on switchpak D1 (may also be labeled U-51). If your

VPM ICP is port 0-3, switch 3 must be on. If your VPM ICP is port 4-7, switch 4 must be on.

8.2.2 Create the VPM Devices

The installation of Sys5.2 automatically creates the VPM devices. The following is an explanation of how they were created.

Login as root and bring your system to init state 1. Then use *mknod*(1M) to create a node for each VPM line and each ICP:

/etc/mknod /dev/vpm? c <major> <minor>
/etc/mknod /dev/ic? c <major> <minor>

where *major* is 18. *minor* is defined as follows: the two most significant bits denote the physical ICP number (0-3), the next two bits denote the VPM protocol number (0-3), and the four least significant bits denote the the physical line number on the ICP. For example, if ICPs 0 and 1 are to be used for VPM using protocol number 1 and line number 3, then the minor device numbers should be 023 and 0123, respectively. Input may be in decimal or octal.

For example, the **mknod** step might proceed as follows:

mknod /dev/vpm2 c 18 2

If TTY devices have been displaced by the new VPM ICP, you must do **mknod**s for these TTYs to link them to a different ICP.

8.2.3 Modify /etc/inittab

Change the logical device assignments in **/etc/inittab** so that only VPM devices (**/dev/vpm0 - /dev/vpm7**) are assigned ports 0-7 on ICP0. TTY devices formerly assigned these ports should receive port assignments on different ICPs. The lines for ports 0-7 on ICP0 should look like this:

2:00:off:/etc/getty tty0 b
2:01:off:/etc/getty tty1 b
2:02:off:/etc/getty tty2 b
.
.
.
2:07:off:/etc/getty tty7 b

Note that logins are disabled on VPM ports.

If port 2 on ICP0 is moved to port 2 on ICP 3 (port 26 from the system's point of view), the old line

2:02:respawn:/etc/getty tty2 b

should be changed to

2:26:respawn:/etc/getty tty26 b

8.2.4 Modify /etc/rc

Verify that your **/etc/rc** downloads your ICPs correctly when the system is brought to multi-user state. No download step is required for VPM ICPs. Since the VPM ICP must be the first (ICP0), your other ICP(s) must change their device numbers and the lines in **/etc/rc** that download these ICPs must reflect these new numbers. Find the lines that do the *dnld* command. The lines should look like this:

/etc/dnld -d -f /usr/lib/dnld/icp -o /dev/ic*n* -a 4000

Verify that **/etc/rc** contains a line like this one for each of your ICPs. If it does not, edit **/etc/rc**, adding line(s) for the missing ICP(s). Increment *n* as appropriate to reflect the addition of the VPM ICP as ICP0.

8.2.5 Reboot

Shutdown and reboot normally, following the procedures in the *Plexus User's Manual.*

8.2.6 Switched or Constant Carrier

VPM uses whatever library is in the file **/usr/src/uts/m68/icp/libvpm.a**. If you require switched carrier, do nothing; the correct file is already in place. If you require constant carrier, back up the file **/usr/src/uts/m68/icp/libvpm.a** and copy the file **/usr/src/uts/m68/icp/libvpm.a.ccar** to **/usr/src/uts/m68/icp/libvpm.a**.

8.2.7 Compiling and Loading VPM Scripts

This section gives the steps to compile and load VPM scripts.

1. You must be in the directory **/usr/src/cmd/vpm**, so issue the command

   cd /usr/src/cmd/vpm

2. Then move your protocol script into this directory, renaming it **vpmscript.r**.

   mv <your protocol script name> vpmscript.r

3. Execute the following command

   make -f vpmscript.mk

   This compiles the script in **vpmscript.r** and link-loads this compiled script with the rest of the VPM ICP kernel. The object file created in

this step is called **vpm0**; it is down-loadable into the VPM ICP.

4. To download **vpm0**, use either of the programs **dnld**(1) or **vpmstart**(1).

5. You may combine up to four scripts in one download module. Scripts to be combined in this way must be called **proto?code.s**, where "?" represents a number from 0 to 3. To combine scripts, you must modify the file **vpmscript.mk**. inserting instructions for each script (**proto?code.s**) to be compiled into a **proto?.s** file, where "?" represents a number from 0 to 3. The following lines accomplish this compilation; note that this whole series of steps must be done for each script. Therefore, you must copy these lines into the file **vpmscript.mk** once for each script, making sure you make the appropriate substitution of a number for the "?".

(2) **fgrep define sas_tempc > sas_define**

(3) **cat /usr/include/icp/opdef.h sas_tempc | /lib/cpp > tf**

(4) **/usr/lib/vpm/vratfor < tf > tg**

(5) **cp tg /usr/src/uts/m68/icp/vpmicp/proto?code.s**

(6) **cat sas_define proto?.s > th**

(7) **cp th /usr/src/uts/m68/icp/vpmicp/proto?.s**

8.3 Appendix 1 - The VPM Interface Module

The VPM interface functions provide a general-purpose interface between a higher-level protocol implemented in a VPM protocol module and the link-level protocol script executed by the VPM interpreter in the ICP. The ICP driver is used by the interface functions to pass commands to and receive reports from the VPM interpreter. When reports are received by the interface module that must be passed on to the protocol module, the protocol module's receive-interrupt routine (*vpmtrint* in the case of the standard VPM protocol module) is called.

This appendix describes each interface function. *Dev* is an argument to many of the interface functions and has the same meaning for all but two of them; the low-order four bits of the argument are not used by the interface functions; the next two bits determine the interface module minor device number; the next two bits determine the ICP minor device. Although *dev* is declared as an *int*, only the low-order eight bits are meaningful at this time. In calls to the *vpmtrace* and *vpmsnap* routines, *dev* need not be a minor device number since it is just saved as part of the event record. The definition of *dev* will not be repeated for each function.

vpmcmd (dev, cmd)
int dev;
char *cmd:

This function passes a command to the script. *Cmd* is the address of a four-byte array. The four bytes are passed to the VPM interpreter, which saves them until the protocol script executes a *getcmd* primitive. Only the most recent four bytes passed by a *vpmcmd* call are saved by the VPM interpreter.

struct vpmbd *vpmdeq (clp)
struct clist *clp:

This function removes the buffer-descriptor pointer at the head of the queue pointed to by *clp* and returns it to the caller. If the queue is empty, a null pointer is returned.

vpmemptq (dev, bdp)
int dev;
struct vpmdb *bdp;

This function is used to pass an empty receive buffer for use by the interpreter in the ICP. *Bdp* is a pointer to a buffer descriptor or null. If *bdp* is not a null pointer, the buffer decriptor is appended to the empty-receive-buffer queue for the interface module specified by *dev*. If the VPM interpreter currently has room for another empty receive buffer, the buffer at the head of the queue is removed and passed to the ICP. The sum of the

number of buffers on the empty-receive buffer queue and the number of receive buffers the VPM interpreter has in its queues is returned to the caller. If *bdp* is a null pointer, the above sum is returned and nothing else is done.

vpmenq (bdp, clp)
struct vpmbd*bdp;
struct clist *clp;

If *bdp* is a null pointer, the number of buffer-descriptor pointers on the *clist* queue pointed to by *clp* is returned. If *bdp* is a not a null pointer, the buffer descriptor pointed to by *bdp* is appended to the *clist* queue pointed to by *clp* and the number of pointers currently on that queue is passed as the return value.

char *vpmerrs (dev, n)
int dev, n;

This function is used to read and reset error counters in the VPM interpreter. The function passes a GETECMD command to the VPM interpreter and blocks until the interpreter responds; this command causes the interpreter to copy its error counters to an array in the interface module and send a completion report to the driver. After the copy operation is completed, a pointer to the error-count array is passed to the caller as the return value. The second argument is not currently used.

char *vpmrpt(dev)
int dev;

This function is used to receive a script report from the ICP. When the protocol script executes a *rtnrpt* primitive, four bytes of data are passed to the interface module. If a *rtnrpt* has been executed by the protocol script since the last call to *vpmrpt*, a pointer to the four bytes passed by the most recent *rtnrpt* primitive is returned; otherwise zero is returned.

vpmsave (type, dev, word1, word2)
char type, dev;
short word1, word2;

This function creates an event record with the following structure:

```
struct {
        short  c_sequn;         /* Sequence number */
        char   c_type;          /* Argument type */
        char   c_dev;           /* Argument dev */
        short  c_word1;         /* Argument word1 */
        short  c_word2;         /* Argument word2  */
}
```

This event record is passed to the trace driver using *trsave*.

vpmsnap (type, dev, word1, word2)
char type, dev;
short word1, word2;

This function is similar to *vpmsave*. The only difference is that a time stamp (*long s_lbolt*) is added to the event record after word2. A protocol script may generate a time-stamped event record by executing the *snap* primitive.

vpmstart (dev, type,rint)
int dev, type;
int (*rint)();

This function must be called on the first open of the protocol-module minor device associated with the interface-module minor device and ICP identified by *dev*. *Type* is a number that identifies the program running in the ICP and must agree with the value specified when the ICP load module was loaded into the ICP. For VPM interpreters, *type* is conventionally 6. *Rint* is the name of a protocol-module routine to be called by the interface module when it needs to return a transmit buffer, a receive buffer, a script report, or an error-termination code. See the description of *vpmtrint* in appendix 2 for an example of such a routine. *Vpmstart* sends a RUN command to the VPM interpreter which causes it to begin execution of the protocol script. If the interface module identified by *dev* is not configured, ENXIO is returned. If the module is already running, i.e., *vpmstart* has been called and *fpmstop* has not been called, or if the ICP is not running or was loaded using a different magic number, EACCESS is returned. A return value of zero indicates a normal completion.

vpmstop (dev)
int dev;

This routine is called to halt the execution of the protocol script by the interpreter. The routine waits until the last transmit buffer has been returned by the protocol script (or 5 seconds have elapsed), then sends a HALT command to the VPM interpreter, which causes the interpreter to stop executing the protocol script. When the interpreter acknowledges the HALT command (or 5 seconds), any transmit or receive buffers still enqueued on the interface module's transmit-and-empty-buffer queues are returned to the protocol module. This does not include buffers contained in the interpreter's queues. Generally, when the protocol script is halted normally, the interpreter will have one or more empty receive buffers. If the interpreter or protocol script terminates in error, some transmit buffers may also remain unaccounted for. The means the protocol module must keep a record of all buffers in use for each particular minor device, so that these buffers can be returned to the pool of available buffers when that minor device is closed.

8.4 Appendix 2 - The VPM Protocol Module

This appendix gives a detailed description of the functions that make up the standard VPM protocol module. The description may be useful as a guide in writing other VPM protocol modules. The *dev* argument to the following routines is declared as an *int*; however, only the low-order eight bits are meaningful at this time. The low-order four bits are used to determine the minor device of the protocol module; the next two bits determine the minor device of the interface module; the next two bits determine the ICP minor device.

vpmopen (dev, flag)
int dev, flag;

This function opens the protocol-module minor device specified by the low-order four bits of *dev*. *Flag* contains the option bits specified on the *open* system call. Exclusive or non-exclusive opens are permitted. If the driver is opened for both reading-and-writing, the *open* is exclusive, i.e., no further *opens* are permitted. If the driver is opened for both reading only or for writing only, the *open* is non-exclusive and subsequent *opens* for reading only or writing only are permitted. If this device is not open when this function is called, it obtains a number of non-addressable system buffers to be used as receive buffers and passes them to the VPM interpreter using the interface routine *vpmemptq*. *Vpmopen* also calls the interface routine *vpmstart* if the minor device was not already open.

vpmclose (dev)
int dev;

This function closes the minor device specified by the low-order four bits of *dev*. It calls the interface routine *vpmstop*, flushes the receive queue for the specified minor device, releases its buffers, and reinitializes its data structure.

vpmwrite (dev)
int dev;

This function implements the *write* system call. If the transmit queue is not full, the function obtains a non-addressable system buffer, copies up to 512 bytes of the user's write data into it, and enqueues the buffer on the level 2 transmit queue using the interface function *vpmxmtq*. These steps are repeated until all of the user's *write* data has been copied. If the transmit queue is full when this function is called or if it becomes full while the function is executing, the calling process is blocked until there is room in the queue for more transmit buffers.

vpmread (dev)
int dev;

This function implements the *read* system call. When it is called, the calling process is blocked until the receive queue is non-empty. As data is received by the VPM interpreter, it is placed into an empty receive buffer. When the protocol script decides that the data contained in a particular buffer is valid, it executes a *rtnbuf* (BISNYC) or *rtnfrm* (HDLC) primitive, which causes the buffer descriptor pointer to be passed to the interface module's interrupt routine. The interface module then passes the buffer descriptor pointer to the protocol module by calling the protocol module's interrupt routine. The protocol module enqueues the buffer descriptor pointer on the receive queue and wakes up (unblocks) the reader(s). The number of bytes requested, or the data in one buffer, whichever is less, is copied to the user process; the number of bytes copied is passed as the return value. Any bytes remaining in a buffer are used to satisfy subsequent *read* requests.

vpmioctl (dev, cmd, arg, mode)
int dev, cmd, mode;
char *arg;

This function implements the *ioctl* system call. *Cmd* determines the function to be performed as follows:

> VPMCMD - Pass a command to the protocol script. The first four bytes of the array pointed to by *arg* are passed to the VPM interpreter which saves them and passes them to the protocol script the next time it executes a *getcmd* primitive.
>
> VPMERRS - Get and reset the VPM interpreter's error counters. The eight-byte array containing the VPM interpreter's error counters is copied to the user array pointed to by *arg*. The interpreter's copy of the error counters is then set to zero.
>
> VPMRPT - Get a report from the protocol script. If the protocol script has executed a *rtnrpt* primitive since the last time this *ioctl* command was issued, the script report (four bytes) is copied to the user array pointed to by *arg* and one is passed as the return value; otherwise, zero is passed as the returned value.

The *inode* argument is not used. The values for VPMCMD, VPMERRS, and VPMRPT are defined in file */usr/include/sys/vpm.h*.

vpmtrint (dev, code, bdp)
int dev, code;
struct vpmbd *bdp;

The address of this function is passed to the protocol module using the *vpmstart* function described in Appendix 1. This routine is called from the interface module to return transmit buffers, receive buffers, script reports, or error termination codes. It is usually called at interrupt priority and therfore

must not sleep or do unnecessary work. *Code* identifies the purpose of the call and determines the meaning of *bdp* as follows:

RRTNXBUF - *Bdp* is a pointer to the buffer descriptor for a transmit buffer. This call is made when the protocol script executes a *rtnxbuf* (BISYNC) or a *rtnxfrm* (HDLC).

RRTNRBUF - *Bdp* is a pointer to the buffer descriptor for a receive buffer. This call is made when the protocol script executes a *rtnbuf* (BISYNC) or a *rtnfrm* (HDLC).

RRTNEBUF - *Bdf* is a pointer to the buffer descriptor for an empty receive buffer. This call is used to return empty receive buffers when the interface module is stopped by calling *vpmstop*.

ERRTERM - *Bdp* is the error-termination code passed to the interface module by the VPM interpreter when it halts the protocol script because of an error condition. The meaning of these error codes is given in the attached manual entry for *vmp*(4).

The values for RRTNXBUF, RRTNRBUF, RRTNEBUF, and ERRITERM are defined in the */usr/include/sys/vpm.h*.

8.5 Appendix 3 - The Trace Driver

The trace driver provides a means by which a user program can receive trace information generated by the VPM driver, a protocol script, or some other driver. See the attached manual entry for *trace*(4).

A description of each routine of the trace driver follows.

tropen (dev)
int dev;

This function opens the minor device specified by *dev* exclusively.

trclose (dev)
int dev;

This function closes the minor device specified by *dev*. It discards any data on the read queue and initializes the data structure associated with the minor device.

trread (dev)
int dev;

This function implements the *read* system call; it sleeps until at least until at least one event record is available on the read queue associated with *dev*. It then copies records to the user until the user's read count is less than the number of bytes in the next event record or until the read queue is empty. The number of bytes copied is passed as the return value.

trioctl (dev, cmd, arg, mode)
int dev, cmd, arg, mode;

This function implements the *ioctl* system call. *Cmd* indicates the operation to be performed. The driver has one command:

> VPMTRO - Enable a trace channel. In order for data to be saved on the read queue for minor device *dev*, the device must be open and the channel to which it is written must be enabled. This command enables channel *arg*, which must be in the range 0 to 15. Any combination of channels may be enabled by repeatedly calling this function with different values of *arg*. All channels are disabled when the minor device is closed.

trsave (dev, chno, buf, ct)
char dev, chno, *buf, ct;

If minor device *dev* of the trace driver is open and channel *chno* of that minor device is enabled then *chno* and *ct*, followed by *ct* bytes starting at address *buf*, are copied onto the read queue associated with *dev*, provided the read queue for that device has room for the complete event record. If not, the record is discarded.

8.6 Appendix 4 - The VPM Event Trace

Calls to the interface routine *vpmsave* have been placed strategically throughout the standard VPM protocol module (*vpmt.c*) and the VPM interface module (*vpmb.c*) to provide an event trace for debugging new protocol modules and/or protocol scripts. A protocol script may generate an event record by executing a *trace* primitive. All such event records are discarded unless some user program has opened minor device 0 of the trace driver and enabled channel 0 of that minor device. The command *vpmtrace*(1C) opens this device and enables channel 0, then reads event records and prints them on the standard output as they are received. Each kind of event record that is generated by the VPM driver will be described by giving the *vpmsave* function call as it appears in *vpmt.c* or *vpmb.c*, followed by an example of the line printed by *vpmtrace* as a result of this call. Following this, the context of the *vpmsave* call and the definition of the parameters passed will be given. The definition of a parameter that appears in more than one call will not be repeated. The first five calls to *vpmsave* occur in the source file *vpmt.c*; the remaining calls occur in *vpmb.c*.

vpmsave('p', dev, ec, 0)

243 p l00 l5 0

Called if *vpmstart* returns an error code. The first field of the printed record contains a sequence number assigned by *vpmsave*. The remaining four fields contain the four remaining arguments to *vpmsave* in the same order as they appear in the call to *vpmsave*. The first argument to *vpmsave*, in this case a 'p', identifies the record type. *Dev* is the minor device number as defined earlier; *ec* is the value returned by *vpmstart*.

vpmsave('0', dev, vp->vt_state, 0)

244 o l00 l 0

Called just before the normal return point of *vpmopen*. The variable, *vp->vt_state*, contains the state bits for the protocol module. Refer to the source file, *vpmt.c*, for the definitions of the state bits.

vpmsave ('c', dev, vp->vt_state, 0)

245 c l00 13 0

Called from *vpmclose* just before the state bits are initialized.

vpmsave ('w', dev, ct, dp)

246 w l00 l000

Called just before putting a buffer-descriptor pointer on the transmit queue in *vpmwrite*. *Ct* is the number of bytes in the buffer. When executing on a PDP11, *dp* is the pointer to the buffer descriptor; *dp* is not meaningful when executing on a VAX because pointers are four bytes on a VAX and the argument corresponding to *dp* is declared as a *short*.

vpmsave ('r', dev, ent, dp->d_bos)

247 r l00 500 500

Called from *vpmread* just after *cnt* bytes have been moved to the user's read buffer. The parameter *dp->d_bos* is the number of bytes remaining in the current receive buffer.

vpmsave ('s', dev, vp->vbstate, 0)

248 s l00 40l 0

Called just before the normal return from *vpmstart*. The parameter *vp->vb_state* contains the state bits for the interface module. For the definitions of the state bits, refer to the source file *vpmb.c*.

vpmsave ('t', dev, vp->vb_state, vp->vb_xbkmc)

249 t l00 0 0

Called just before the normal return from *vpmstop*. The parameter *vp->vb_xbkmc* is the number of transmit buffers currently held by the VPM interpreter. It can be non-zero if the protocol script or interpreter terminates in error.

vpmsave ('X', dev, vp->vb_xbkmc, 0)

250 X l00 l 0

Called from *vpmbrint*. the interface module's receive-interrupt routine, each time the VPM interpreter returns a transmit buffer.

vpmsave ('R', dev, vp->vb_vrkmc, 0)

251 R l00 l 0

Called from *vpmbrint* each time the VPM interpreter returns a receive buffer. The parameter *vp->vb_rbkmc* contains the number of receive buffers currently held by the interpreter.

vpmsave ('T', dev, sel4, sel6)

252 T l00 370 21 34

Called from *vpmbrint* when a trace report is received from the interpreter. This occurs when the protocol script executes a *trace* primitive. *Sel4* contains the value of the script location counter (plus two) at the time the *trace* primitive was executed. By referring to the assembly-language listing of the protocol script generated by the *-l* option of *vpmc*, the point in the protocol script at which the trace was executed can be determined. The value of the location counter is two greater than the location of the *trace* instruction as shown in the assembley-language listing. *Sel6* contains the byte or bytes passed by the *trace* primitive. *Vpmtrace* prints these two bytes in separate fields.

vpmsave ('E', dev, sel4, sel6)

253 E 244 21

Called from *vpmbrint* when an error-termination report is received from the interpreter. *Sel4* contains the script location counter at the time execution of the script was terminated. *Sel6* contains the termination code. For an explanation of these codes see the attached manual entry for *vpm*(4).

vpmsave ('P', dev, sel4, sel6)

254 P l00 2105 l055

Called from *vpmbrint* when a script report is received from the interpreter. This occurs when the protocol script executes a *rtnrpt* primitive. *Sel4* and *sel6* contain the four bytes transferred by this primitive.

vpmsave ('F', dev, sel4, sel6)

255 F l00 3 0

Called form *vpmbrint* when an error-count report is received from the interpreter. *Sel4* and *sel6* do not contain any meaningful data for this event type.

vpmsave ('S', dev, sel4, sel6)

256 S l00 40l 0

Called from *vpmbrint* when a start-up report is received from the interpreter. The low-order eight bits of *sel4* contain a parameter defining the maximum number of transmit buffers the interpreter can accept; the high-order eight bits contain a parameter defining the maximum number of receive buffers. *Sel6* contains the options supported by the interpreter.

vpmsave ('C', dev, vp-vb_state, bp->xbkmc)

257 C l00 l 0

Called from *vpmclean* just before the data structure associated with *dev* is initialized.

9. UNIX SYSTEM REMOTE JOB ENTRY

This chapter contains an overview of the Plexus implementation of the Sys5 UNIX Remote Job Entry (RJE) and the Plexus Batch 2780/3780. For detailed information on RJE please reference the Plexus publication *Plexus RJE/HASP Release Notice*. For detailed information on Batch, consult the Plexus publication *Plexus Batch Release Notice*.

RJE is the communal name for a collection of programs and a file organization that allows a UNIX system, equipped with the appropriate hardware and associated Virtual Protocol Machine (VPM) software, to communicate with IBM's Job Entry Subsystems by mimicking an IBM 360 remote work station.

Similarly, Batch is the communal name for a group of programs and a file organization that allows an appropriately equipped UNIX system to communicate with IBM's Job Entry Subsystems by mimicking a 2780 or 3780 remote work station.

While active, RJE and Batch run in the background and require no human supervision. They quietly transmit to the IBM system, jobs that have been queued, and operator requests. They receive from the IBM system, print and punch data sets and message output. They enter the data sets into the proper UNIX system directory and notify the appropriate user of their arrival. They store the message output and make these messages available for public inspection.

In order to use RJE or Batch you need to be familiar with a subset of basic commands. You must understand the directory structure of the file system, and you should know something about the attributes of files. You must know how to enter, edit, and examine text files, and how to communicate with other users and with the system.

RJE and Batch are designed to be autonomous facilities that do not require manual supervision. RJE and Batch may be initiated automatically by the UNIX reboot procedures and continue in execution until the sytem is shut down.

Whether you use RJE or Batch 2780/3780 depends on the protocol of the IBM system you wish to communicate with.

10. SYSTEM ACTIVITY PACKAGE

This chapter describes the design and implementation of the UNIX System Activity Package. The UNIX operating system contains a number of counters that are incremented as various system actions occur. The system activity package reports UNIX system-wide measurements including central processing unit (CPU) utilization, disk and tape input/output (I/O) activities, terminal device activity, buffer usage, system calls, system switching and swapping, file-access activity, queue activity, and message and semaphore activities.

Throughout this chapter, each reference of the form **name**(1M), **name**(7), or **name**(8) refers to entries in the *Sys5 UNIX Administrator Reference Manual.* References to entries of the form **name**(N), where "N" is the number 1 or 6 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX User Reference Manual.* If "N" is a number 2 through 5 possibly followed by a letter, refer to entry **name** in section **N** of the *Sys5 UNIX Programmer Reference Manual.*

The package provides four commands that generate various types of reports. Procedures that automatically generate daily reports are also included. The five functions of the activity package are:

- **sar**(1) command—allows a user to generate system activity reports in real-time and to save system activities in a file for later usage.
- **sag**(1G) command—displays system activity in a graphical form.
- **sadp**(1M) command—samples disk activity once every second during a specified time interval and reports disk usage and seek distance in either tabular or histogram form.
- **timex**(1)—a modified **time**(1) command that times a command and also (optionally) reports concurrent system activity and process accounting activity.
- system activity daily reports—procedures are provided for sampling and saving system activities in a data file periodically and for generating the daily report from the data file.

The system activity information reported by this package is derived from a set of system counters located in the operation system kernel. These system counters are described in the part "System Activity Counters". The part "System Activity Commands" describes the commands provided by this package. The procedure for generating daily reports is given in "Daily Report Generation". For a description of the files used by the system activity package, see Attachment 10-1 at the end of this chapter.

10.1 System Activity Counters

The UNIX operating system manages a number of counters that record various activities and provide the basis for the system activity reporting system. The data structure for most of these counters is defined in the *sysinfo* structure in */usr/include/sys/sysinfo.h* (see Attachment 10-2 at the end of this chapter). The system table overflow counters are kept in the *_syserr* structure.

The following paragraphs describe the system activity counters sampled by the system activity package.

Cpu time counters—There are four time counters that may be incremented at each clock interrupt 60 times per second. According to the mode the CPU is in at the interrupt (idle, user, kernel, and wait for I/O completion), exactly one of the *cpu[]* counters is incremented.

Lread and lwrite—The *lread* and *lwrite* counters are used to count logical read and write requests issued by the system to block devices.

Bread and bwrite—The *bread* and *bwrite* counters are used to count the number of times data is transferred between the system buffers and the block devices. These actual I/Os are triggered by logical I/Os that cannot be satisfied by the current contents of the buffers. The ratio of block I/O to logical I/O is a common measure of the effectiveness of the system buffering.

Phread and phwrite—The *phread* and *phwrite* counters count read and write requests issued by the system to raw devices.

Swapin and swapout—The *swapin* and *swapout* counters are incremented for each system request initiating a transfer from or to the swap device. More than one request is usually involved in bringing a process into or out of memory because text and data are handled separately. Frequently used programs are kept on the swap device and are swapped in rather than loaded from the file system. The *swapin* counter reflects these initial loading operations as well as resumptions of activity, while the *swapout* counter reveals the level of actual "swapping." The amount of data transferred between the swap device and memory are measured in blocks and counted by *bswapin* and *bswapout*.

Pswitch and syscall—These counters are related to the management of multiprogramming. *Syscall* is incremented every time a system call is invoked. The numbers of invocations of **read**(2), **write**(2), **fork**(2), and **exec**(2) system calls are kept in counters *sysread*, *syswrite*, *sysfork*, and *sysexec*, respectively. *Pswitch* counts the times the switcher was invoked, which occurs when:

1. A system call resulted in a road block
2. An interrupt occurred resulting in awakening a higher priority process
3. A 1 second clock interrupt occurs.

Iget, namei, and dirblk—These counters apply to file-access operations. *Iget* and *namei*, in particular, are the names of UNIX operating system routines. The counters record the number of times the respective routines are called. *Namei* is the routine that performs file system path searches. It searches the various directory files to get the associated i-number of a file corresponding to a special path. *Iget* is a routine called to locate the inode entry of a file (i-number). It first searches the in-core inode table. If the inode entry is not in the table, routine *iget* will get the inode from the file system where the file resides and make an entry in the in-core inode table for the file. *Iget* returns a pointer to this entry. *Namei* calls *iget*, but other file access routines also call *iget*. Therefore, counter *iget* is always greater than counter *namei*.

Counter *dirblk* records the number of directory block reads issued by the system. It is noted that the directory blocks read divided by the number of *namei* calls estimates the average path length of files.

Runque, runocc, swpque, and swpocc—These counters are used to record queue activities. They are implemented in the *clock.c* routine. At every 1 second interval, the clock routine examines the process table to see whether any processes are in core and in ready state. If so, the counter *runocc* is incremented and the number of such processes are added to counter *runque*. While examining the process table, the clock routine also checks whether any processes in the swap device are in ready state. The counter *swpocc* is incremented if the swap queue is occupied, and the number of processes in swap queue is added to counter *swpque*.

Readch and writech—The *readch* and *writech* counters record the total number of bytes (characters) transferred by the **read** and **write** system calls, respectively.

Monitoring terminal device activities—There are six counters monitoring terminal device activities. *Rcvint*, *xmtint*, and *mdmint* are counters measuring hardware interrupt occurrences for receiver, transmitter, and modem individually. *Rawch*, *canch*, and *outch* count number of characters in the raw queue, canonical queue, and output queue. Characters generated by devices operating in the *cooked* mode, such as terminals, are counted in both *rawch* and (as edited) in *canch*; but characters from raw devices, such as communication processors, are counted only in *rawch*.

Msg and sema counters—These counters record message sending and receiving activities and semaphore operations, respectively.

Monitoring I/O activities—Four counters are kept for each disk or tape drive in the device status table. Counter *io_ops* is incremented when an I/O operation has occurred on the device. It includes block I/O, swap I/O, and physical I/O. *Io_bcnt* counts the amount of data transferred between the device and memory in 512-byte units. *Io_act* and *io_resp* measure the active time and response time of a device in time ticks summed over all I/O requests that have completed for each device. The device active time includes the device seeking, rotating, and data transferring times, while the response time of an I/O operation is from the time the I/O request is queued to the device to the time when the I/O completes.

Inodeovf, fileovf, textovf, and procovf—These counters are extracted from *_syserr* structure. When an overflow occurs in any of the inode, file, text, and process tables, the corresponding overflow counter is incremented.

10.2 System Activity Commands

The system activity package provides three commands for generating various system activity reports and one command for profiling disk activities. These tools facilitate observation of system activity during

- A controlled stand-alone test of a large system
- An uncontrolled run of a program to observe the operating environment
- Normal production operation.

Commands **sar** and **sag** permit the user to specify a sampling interval and number of intervals for examining system activity and then to display the observed level of activity in tabular or graphical form. The **timex** command reports the amount of system activity that occurred during the precise period of execution of a timed command. The **sadp** command allows the user to establish a sampling period during which access location and seek distance on specified disks are recorded and later displayed as a tabular summary or as a histogram.

10.2.1 The "sar" Command

The **sar** command can be used in the following two ways:

- When the frequency arguments **t** and **n** are specified, it invokes the data collection program **sadc** to sample the system activity counters in the operating system every **t** seconds for **n** intervals and generates system activity reports in real-time. Generally, it is desirable to include the option to save the sampled data in a file for later examination. The format of the data file is shown in **sar**(1M). In addition to the system counters, a time stamp is also included. It gives the time at which the sample was taken.
- If no frequency arguments are supplied, it generates system activity reports for a specified time interval from an existing data file that was created by **sar** at an earlier time.

A convenient usage is to run **sar** as a background process saving its samples in a temporary file but sending its standard output to /dev/null. Then an experiment is conducted after which the system activity is extracted from the temporary file. The **sar**(1) manual entry describes the usage and lists various types of reports. Attachment 10-3 (at the end of this chapter) gives the formula for deriving each reported item.

10.2.2 The "sag" Command

Sag displays system activity data graphically. It relies on the data file produced by a prior run of **sar** after which any column of data or the combination of columns of data of the **sar** report can be plotted. A fairly simple but powerful command syntax allows the specification of cross plots or time plots. Data items are selected using the **sar** column header names. The **sar**(1G) manual entry describes its options and usage. The system activity graphical program invokes **graphics**(1G) and **tplot**(1G) commands to have the graphical output displayed on any of the terminal types supported by **tplot**.

10.2.3 The "timex" Command

The **timex** command is an extension of the **time**(1) command. Without options, **timex** behaves like **time**. In addition to giving the time information, it can also print a system activity report and a process accounting report. For all the options available, refer to the manual entry **timex**(1). It should be emphasized that the *user* and *sys* times reported in the second and third lines are for the measured process itself including all its children while the remaining data (including the *cpu user* % and *cpu sys* %) are for the entire system.

While the normal use of **timex** will probably be to measure a single command, multiple commands can also be timed either by combining them in an executable file and timing it or by typing:

timex sh −c "cmd1; cmd2; ... ;"

This establishes the necessary parent-child relationships to correctly extract the user and system times consumed by **cmd1**, **cmd2**, ... (and the shell).

10.2.4 The "sadp" Command

Sadp is a user level program that can be invoked independently by any user. It requires no storage or extra code in the operating system and allows the user to specify the disks to be monitored. The program is reawakened every second, reads system tables from */dev/kmem*, and extracts the required information. Because of the 1 second sampling, only a small fraction of disk requests are observed; however, comparative studies have shown that the statistical determination of disk locality is adequate when sufficient samples are collected.

In the operating system, there is an *iobuf* for each disk drive. It contains two pointers which are head and tail of the I/O active queue for the device. The actual requests in the queue may be found in three buffer header pools—system buffer headers for block I/O requests, physical buffer headers for physical I/O requests, and swap buffer headers for swap I/O. Each buffer header has a forward pointer that points to the next request in the I/O active queue and a backward pointer that points to the previous request.

Sadp snapshots the *iobuf* of the monitored device and the three buffer header pools once every second during the monitoring period. It then traces the requests in the I/O queue, records the disk access location, and seeks distance in buckets of 8-cylinder increments. At the end of monitoring period, it prints out the sampled data. The output of **sadp** can be used to balance load among disk drives and to rearrange the layout of a particular disk pack. The usage of this command is described in manual entry **sadp**(1M).

10.3 Daily Report Generation

The previous part described the commands available to users to initiate activity observations. It is probably desirable for each installation to routinely monitor and record system activity in a standard way for historical analysis. This part describes the steps that a system administrator may follow to automatically produce a standard daily report of system activity.

10.3.1 Facilities

- **sadc**—The executable module of **sadc.c** (see Attachment 10-1 at the end of this chapter) which reads system counters from */dev/kmem* and records them to a file. In addition, two frequency arguments are usually specified to indicate the sampling interval and number of samples to be taken. In case no frequency arguments are given, it writes a dummy record in the file to indicate a system restart.
- **sa1**—The shell procedure that invokes **sadc** to write system counters in the daily data file */usr/adm/sa***dd** where **dd** represents the day of the month. It may be invoked with sampling interval and iterations as arguments.
- **sa2**—The shell procedure that invokes the **sar** command to generate daily report */usr/adm/sa/sar***dd** from the daily data file */usr/adm/sa/sa***dd**. It also removes daily data files and report files after 7 days. The starting and ending times and all report options of **sar** are applicable to **sa2**.

10.3.2 Suggested Operational Setup

It is suggested that the **cron**(1M) control the normal data collection and report generation operations. For example, the sample entries in */usr/spool/cron/crontab/sys*:

```
0 * * * 0,6 /usr/lib/sa/sa1
0 18–7 * * 1–5 /usr/lib/sa/sa1
0 8–17 * * 1–5 /usr/lib/sa/sa1 1200 3
```

would cause the data collection program **sadc** to be invoked every hour on the hour. Moreover, depending on the arguments presented, it writes data to the data file one to three times at every 20 minutes. Therefore, under the control of **cron**(1M), the data file is written every 20 minutes between 8:00 and 18:00 on weekdays and hourly at other times.

Note that data samples are taken more frequently during prime time on weekdays to make them available for a finer and more detailed graphical display. It is suggested that **sa1** be invoked hourly rather than invoking it once every day; this ensures that if the system crashes data collection will be resumed within an hour after the system is restarted.

Because system activity counters restart from zero when the system is restarted, a special record is written on the data file to reflect this situation. This process is accomplished by invoking **sadc** with no frequency arguments within */etc/rc* when going to multiuser state:

```
su adm –c "/usr/lib/sa/sadc /usr/adm/sa/sa'date +%d'"
```

Cron(1M) also controls the invocation of **sar** to generate the daily report via shell procedure **sa2**. One may choose the time period the daily report is to cover and the groups of system activity to be reported. For instance, if:

```
0 20 * * 1–5 /usr/lib/sa/sa2 –s 8:00 –e 18:00 –i 3600 –uybd
```

is an entry in */usr/spool/cron/crontab/sys*, **cron** will execute the **sar** command to generate daily reports from the daily data file at 20:00 on weekdays. The daily report reports the CPU utilization, terminal device activity, buffer usage, and device activity every hour from 8:00 to 18:00.

In case of a shortage of the disk space or for any other reason, these data files and report files can be removed by the superuser. The manual entry **sar**(1M) describes the daily report generation procedure.

ATTACHMENT 10-1

The source files and shell programs of the system activity package are in directory */usr/src/cmd/sa*.

sa.h The system activity header file defines the structure of data file and device information for measured devices. It is included in **sadc.c**, **sar.c**, and **timex.c**.

sadc.c The data collection program that accesses */dev/kmem* to read the system activity counters and writes data either on standard output or on a binary data file. It is invoked by the **sar** command generating a real-time report. It is also invoked indirectly by entries in */usr/spool/cron/crontab/sys* to collect system activity data.

sar.c The report generation program invokes **sadc** to examine system activity data, generates reports in real-time, and saves the data to a file for later usage. It may also generate system activity reports from an existing data file. It is invoked indirectly by **cron** to generate daily reports.

saghdr.h The header file for **saga.c** and **sagb.c**. It contains data structures and variables used by **saga.c** and **sagb.c**.

saga.c & sagb.c The graph generation program that first invokes **sar** to format the data of a data file in a tabular form and then displays the **sar** data in graphical form.

sa1.sh The shell procedure that invokes **sadc** to write data file records. It is activated by entries in */usr/spool/cron/crontab/sys*.

sa2.sh The shell procedure that invokes **sar** to generate the report. It also removes the daily data files and daily report files after a week. It is activated by an entry in */usr/spool/cron/crontab/sys* on weekdays.

timex.c The program that times a command and generates a system activity or process accounting report.

sadp.c The program that samples and reports disk activities.

ATTACHMENT 10-2

```
struct sysinfo  {
                 time_t          cpu[4];
#define          CPU_IDLE        0
#define          CPU_USER        1
#define          CPU_KERNAL      2
#define          CPU_WAIT        3
                 time_t          wait[3];
#define          W_IO            0
#define          W_SWAP          1
#define          W_PIO           2
                 long            bread;
                 long            bwrite;
                 long            lread;
                 long            lwrite;
                 long            phread;
                 long            phwrite;
                 long            swapin;
                 long            swapout;
                 long            bswapin;
                 long            bswapout;
                 long            pswitch;
                 long            syscall;
                 long            sysread;
                 long            syswrite;
                 long            sysfork;
                 long            sysexec;
                 long            runque;
                 long            runocc;
                 long            swpque;
                 long            swpocc;
                 long            iget;
                 long            namei;
                 long            dirblk;
                 long            readch;
                 long            writech;
                 long            rcvint;
                 long            xmtint;
                 long            mdmint;
                 long            rawch;
                 long            canch;
                 long            outch;
                 long            msg;
                 long            sema;
};
```

ATTACHMENT 10-3

The derivation of the reported items is given in this attachment. Each item discussed below is the data difference sampled at two distinct times *t2* and *t1*.

CPU Utilization

%-of-cpu-x = cpu-x / (cpu-idle + cpu-user + cpu-kernel + cpu-wait) * 100

where cpu-x is cpu-idle, cpu-user, cpu-kernel (cpu-sys), or cpu-wait.

Cache Hit Ratio

%-of-cache-I/O = (logical-I/O − block-I/O) / logical-I/O * 100

where cache I/O is cache read or cache write.

Disk or Tape I/O Activity

%-of-busy = I/O-active / (t2 − t1) * 100;
avg-queue-length = I/O-resp / I/O-active;
avg-wait = (I/O-resp − I/O-active) / I/O-ops;
avg-service-time = I/O-active / I/O-ops.

Queue Activity

avg-x-queue-length = x-queue / x-queue-occupied-time;
%-of-x-queue-occupied-time = x-queue-occupied-time / (t2 − t1);

where x-queue is run queue or swap queue.

The Rest of System Activity

avg-rate-of-x = x / (t2 − t1)

where x is swap in/out, blks swapped in/out, terminal device activities, read/write characters, block read/write, logical read/write, process switch, system calls, read/write, fork/exec, iget, namei, directory blocks read, disk/tape I/O activities, message, or semaphore activities.

11. UUCP ADMINISTRATION

This chapter describes how a **uucp** network is set up, the format of control files, and administrative procedures. Administrators should be familiar with the manual pages for each of the **uucp** related commands.

11.1 Planning

In setting up a network of UNIX systems, there are several considerations that should be taken into account *before* configuring each system on the network. The following parts attempt to outline the most important considerations.

11.1.1 Extent of the Network

Some basic decisions about access to processors in the network must be made before attempting to set up the configuration files. If an administrator has control over only one processor and an existing network is being joined, then the administrator must decide what level of access should be granted to other systems. The other members of the network must make a similar decision for the new system. The UNIX system *password* mechanism is used to grant access to other systems. The file */usr/lib/uucp/USERFILE* restricts access by other systems to parts of the file system tree, and the file */usr/lib/uucp/L.sys* on the local processor determines how many other systems on the network can be reached.

When setting up more than one processor, the administrator has control of a larger portion of the network and can make more decisions about the setup. For example, the network can be set up as a private network where only those machines under the direct control of the administrator can access each other. Granting no access to machines outside the network can be done if security is paramount; however, this is usually impractical. Very limited access can be granted to outside machines by each of the systems on the private network. Alternatively, access to/from the outside world can be confined to only one processor. This is frequently done to minimize the effort in keeping access information (passwords, phone numbers, login sequences, etc.) updated and to minimize the number of security holes for the private network.

11.1.2 Hardware and Line Speeds

There are only two supported means of interconnection by **uucp**(1),

1. Direct connection using a null modem.
2. Connection over the Direct Distance Dialing (DDD) network.

In choosing hardware, the equipment used by other processors on the network must be considered. For example, if some systems on the network have only 103-type (300-baud) data sets, then communication with them is not possible unless the local system has a 300-baud data set connected to a calling unit. (Most data sets available on systems are 1200-baud.) If hard-wired connections are to be used between systems, then the distance between systems must be considered since a null modem cannot be used when the systems are separated by more than several hundred feet. The limit for communication at 9600-baud is about 800 to 1000 feet. However, the RS232 specification and Western Electric Support Groups only allow for less than 50 feet. Limited distance modems must be used beyond 50 feet as noise on the lines becomes a problem.

11.1.3 Maintenance and Administration

There is a minimum amount of maintenance that must be provided on each system to keep the access files updated, to ensure that the network is running properly, and to track down line problems. When more than one system is involved, the job becomes more difficult because there are more files to update and because users are much less patient when failures occur between machines that are under local control.

11.2 UUCP Software

Figure 10-1 (at the end of this chapter) is an illustration of the daemons used by the **uucp** network to communicate with another system. The **uucp**(1) or **uux**(1) command queues users requests and spawns the **uucico** daemon to call another system. Figure 10-2 (at the end of this chapter) illustrates the structure of **uucico** and the tasks that it performs in communicating with another system. **Uucico** initiates the call to another system and performs the file transfer. On the receiving side, **uucico** is invoked to receive the transfer. Remote execution jobs are actually done by transferring a command file to the remote system and invoking a daemon (**uuxqt**) to execute that command file and return the results.

11.3 Installation

The **uucp**(1) package is delivered as part of the standard UNIX system distribution. It resides in its own subdirectory (called *uucp*) in the commands area and has its own make file (*uucp.mk*). The **uucp** package is installed as part of the normal distribution; however, if it must be reinstalled for any reason, then the sequence

make –f uucp.mk install

should be executed.

11.3.1 Object Modules

The following object modules are installed as part of the **uucp** make procedure.

1. **uucp**—The file transfer command.
2. **uux**—The remote execution command.
3. **uucico**—The **uucp** network daemon.
4. **uustat**—Network status command.
5. **uuclean**—Cleanup command.
6. **uusub**—The command for monitoring and creating a subnetwork.
7. **uuxqt**—The remote execution daemon.
8. **uudemon.day**—A shell procedure that is invoked each day to maintain the network. Shell scripts for execution each week (**uudemon.wk**) and each hour (**uudemon.hr**) are also distributed.

11.3.2 Password File

To allow remote systems to call the local system, password entries must be made for any **uucp** logins. For example,

nuucp:zaaAA:6:1:UUCP.Admin:/usr/spool/uucppublic:/usr/lib/uucp/uucico

Note that the **uucico** daemon is used for the shell, and the spool directory is used as the working directory.

There must also be an entry in the *passwd* file for an **uucp** administrative login. This login is the owner of all the **uucp** object and spooled data files and is usually "uucp".

For example, the following is an entry in */etc/passwd* for this administrative login:

uucp:zAvLCKp:5:1:UUCP.Admin:/usr/lib/uucp:

Note that the standard shell is used instead of **uucico**. If an owner other than "uucp" is chosen, the *make* file for **uucp** (*/usr/src/cmd/uucp/uucp.mk*) must be edited. The line "OWNER=uucp" must be changed to reflect the new owner login.

11.3.3 Lines File

The file */usr/lib/uucp/L-devices* contains the list of all lines that are directly connected to other systems or are available for calling other systems. The file contains the attributes of the lines and whether the line is a permanent connection or can call via a dialer. The format of the file is

type line call-device speed protocol

where each field is

type Two keywords are used to describe whether a line is directly connected to another system (DIR) or uses an automatic calling unit (ACU). An X.25 permanent virtual circuit would use the DIR keyword.

line This is the device name for the line (e.g., *ttyab* for a direct line, *cul0* for a line connected to an ACU).

call-device If the ACU keyword is specified, this field contains the device name of the ACU. Otherwise, the field is ignored; however, a placeholder must be used in this field so that the *protocol* field can be interpreted. This "device" is actually a compiled program that handles any required control signals for your modem. The program we supply (/usr/plx/dial) supports HAYES 3451 modems only.

speed The line speed that the connection is to run at. (The speed field is currently ignored if an X.25 link is used.)

protocol This is an optional field that needs only be filled in if the connection is for a protocol other than the default terminal protocol. The X.25 protocol is the only other protocol supported and the single character *x* is used to select this protocol.

The following entries illustrate various types of connections:

```
DIR ttyab 0 9600
ACU cul0 cua0 1200
DIR x25.s0 0 300 x
```

The first entry is for a hard-wired line running at 9600-baud between two systems. Note that the *acu-device* field is zero. The second entry is for a line with a 1200-baud ACU. The last entry is for an X.25 synchronous direct connection between systems. Note that the *protocol* field is filled in and that the *acu-device* and *line speed* fields are meaningless.

11.3.3.1 Naming Conventions

It is often useful when naming lines that are directly connected between systems or which are dedicated to calling other systems to choose a naming scheme that conveys the use of the line. In the earlier examples, the name *ttyab* is used for the line that directly connects two systems named *a* and *b*. Similarly, lines associated with calling units are best given names that relate them to the calling unit (note the names *cul0* and *cua0* to specify the line and calling unit, respectively).

11.3.4 System File

Each entry in this file represents a system that can be called by the local **uucp** programs. More than one line may be present for a particular system. In this case, the additional lines represent alternative communication paths that will be tried in sequential order. The fields are described below.

system name Name of the remote system.

time String indicating days-of-week and times-of-day when the system can be called (e.g., MoTuTh0800–1730).

The day portion may be a list containing *Su*, *Mo*, *Tu*, *We*, *Th*, *Fr*, *Sa*; or it may be *Wk* for any week-day or *Any* for any day. The time should be a range (e.g., 0800–1230). If no time portion is specified, any time of day is assumed to be allowed. Note that a time range that spans 0000 is permitted; 0800-0600 means all times are allowed other than times between 6 and 8 am. An optional subfield is available to specify the minimum time (minutes) before a retry following a failed attempt. The subfield separator is a "," (e.g., *Any,9* means call any time but wait at least 9 minutes before retrying the call after a failure has occurred).

device This is either *ACU* or the hard-wired device name to be used for the call. For the hard-wired case, the last part of the special file name is used (e.g., tty0).

class This is usually the line speed for the call (e.g., 300).

phone The phone number is made up of an optional alphabetic abbreviation (dialing prefix) and a numeric part. The abbreviation should be one that appears in the *L-dialcodes* file (e.g., mh1212, boston555–1212). For the hard-wired devices, this field contains the same string as used for the *device* field.

login The login information is given as a series of fields and subfields in the format

[expect send] . . .

where *expect* is the string expected to be read and *send* is the string to be sent when the *expect* string is received.

The expect field may be made up of subfields of the form

expect[–send–expect] . . .

where the *send* is sent if the prior *expect* is *not* successfully read and the *expect* following the *send* is the next expected string. (For example, login--login will expect *login*; if it gets it, the program will go on to the next field; if it does not get *login*, it will send *null* followed by a new line, then expect *login* again.) If no characters are initially expected from the remote machine, the string "" (a null string) should be used in the first expect field.

There are two special names available to be sent during the login sequence. The string *EOT* will send an EOT character, and the string *BREAK* will try to send a *BREAK* character. (The *BREAK* character is simulated using line speed changes and null characters and may not work on all devices and/or systems.) A number from 1 to 9 may follow the *BREAK* (e.g., *BREAK1*, will send 1 null character instead of the default of 3). Note that *BREAK1* usually works best for 300-/1200-baud lines.

There are several character strings that cause specific actions when they are a part of a string sent during the login sequence.

\s Send a space character.

\d Delay one second before sending or reading more characters.

\c If at the end of a string, suppress the new-line that is normally sent. Ignored otherwise.

\N Send a null character.

These character strings are useful for making **uucp** communicate via direct lines to data switches.

A typical entry in the *L.sys* file would be

sys Any ACU 300 mh7654 login uucp ssword: word

The expect algorithm matches all or part of the input string as illustrated in the password field above.

11.3.5 Dialing Prefixes

This file contains the dial-code abbreviations used in the *L.sys* file (e.g., py, mh, boston). The entry format is

abb dial-seq

where *abb* is the abbreviation and *dial-seq* is the dial sequence to call that location.

The line

py 165–

would be set up so that entry py7777 would send 165–7777 to the dial unit.

11.3.6 Userfile

This file contains user accessibility information. It specifies four types of constraints:

1. Files that can be accessed by a normal user of the local machine.
2. Files that can be accessed from a remote computer.
3. Login name used by a particular remote computer.
4. Whether a remote computer should be called back in order to confirm its identity.

Each line in the file has the format

login,sys [c] pathname [pathname] ...

where

login is the login name for a user or the remote computer.

sys is the system name for a remote computer.

c is the optional *call-back required* flag.

pathname is a pathname prefix that is acceptable for *sys*.

The constraints are implemented as follows:

1. When the program is obeying a command stored on the local machine, the pathnames allowed are those given on the first line in the *USERFILE* that has the login name of the user who entered the command. If no such line is found, the first line with a *null* login name is used.
2. When the program is responding to a command from a remote machine, the pathnames allowed are those given on the first line in the file that has the system name that matches the remote machine. If no such line is found, the first one with a *null* system name is used.
3. When a remote computer logs in, the login name that it uses *must* appear in the *USERFILE*. There may be several lines with the same login name but one of them must either have the name of the remote system or must contain a *null* system name.
4. If the line matched in (3.) contains a "c", the remote machine is called back before any transactions take place.

The line

u,m /usr/xyz

allows machine *m* to login with name *u* and request the transfer of files whose names start with */usr/xyz*. The line

you, /usr/you

allows the ordinary user *you* to issue commands for files whose name starts with */usr/you*. (This type restriction is seldom used.)

The lines

u,m /usr/xyz /usr/spool
u, /usr/spool

allows *any* remote machine to login with name *u*. If its system name is not *m*, it can only ask to transfer files whose names start with */usr/spool*. If it is system *m*, it can send files from paths */usr/xyz* as well as */usr/spool*. The lines

root, /
, /usr

allow any user to transfer files beginning with */usr* but the user with login *root* can transfer any file. (Note that any file that is to be transferred must be readable by anybody.)

11.3.7 Forwarding File

There are two files that allow restrictions to be placed on the forwarding mechanism. The format of the entries in each file is the same,

system
or
system,user,user2,...

The file *ORIGFILE* (*/usr/lib/uucp/ORIGFILE*) restricts the access of systems that are attempting to forward through the local system. The file contains the list of systems (and users) for whom the local system is willing to forward. Each entry refers to the system that was the *source* of the original job and not the name of the last system to forward the file. The second file, *FWDFILE* (*/usr/lib/uucp/FWDFILE*), is a list of valid systems that a job can be forwarded to. (It is not necessarily the name of the destination of a job, but merely the next valid node.) This file will be a subset of the *L.sys* file and can be used to prevent forwarding to systems that are very expensive to reach but to which access by local users is allowed (e.g., links to overseas universities). If neither of these files exist, **uucp** will be perfectly happy to forward for any system. As an example, if the entry for system *australia* were in the *ORIGFILE* but not in the *FWDFILE* on system *mhtsa*, it would mean that system *australia* would be capable of forwarding jobs into the network via system *mhtsa*. However, no systems in the network could forward a job to *australia* via system *mhtsa*.

11.4 Administration

The role of the *uucp* administrator depends heavily on the amount of traffic that enters or leaves a system and the quality of the connections that can be made to and from that system. For the average system, only a modest amount of traffic (100 to 200 files per day) pass through the system and little if any intervention with the *uucp* automatic cleanup functions is necessary. Systems that pass large numbers of files (200 to 10,000) may require more attention when problems occur. The following parts describe the routine administrative tasks that must be performed by the administrator or are automatically performed by the *uucp* package. The part on problems describes what are the most frequent problems and how to effectively deal with them.

11.4.1 Cleanup

The biggest problem in a dialup network like *uucp* is dealing with the backlog of jobs that cannot be transmitted to other systems. The following cleanup activities should be routinely performed by shell scripts started from **cron**(1).

11.4.1.1 Cleanup of Undeliverable Jobs

The **uudemon.day** procedure usually contains an invocation of the **uuclean** command to purge any jobs that are older than some fixed time (usually 72 hours). A similar procedure is usually used to purge any *lock* or *status* files. An example invocation of **uuclean**(1M) to remove both job files and old status files every 48 hours is:

/usr/lib/uucp/uuclean –pST –pC –n48

11.4.1.2 Cleanup of the Public Area

In order to keep the local file system from overflowing when files are sent to the public area, the **uudemon.day** procedure is usually set up with a **find** command to remove any files that are older than 7 days. This interval may need to be shortened if there is not sufficient space to devote to the public area.

11.4.1.3 Compaction of Log Files

The files *SYSLOG* and *LOGFILE* that contain logging information are compacted daily (using the **pack** command from the shell script **uudemon.day**) and should be kept for 1 week before being overwritten.

11.4.2 Polling Other Systems

Systems that are passive members of the network must be polled by other systems in order for their files to be sent. This can be arranged by using the **uusub**(1) command as follows:

uusub –cmhtsd

which will call *mhtsd* when it is invoked.

11.4.3 Problems

The following sections list the most frequent problems that appear on systems that make heavy use of **uucp**(1).

11.4.3.1 Out of Space

The file system used to spool incoming or outgoing jobs can run out of space and prevent jobs from being spawned or received from remote systems. The inability to receive jobs is the worse of the two conditions. When file space does become available, the system will be flooded with the backlog of traffic.

11.4.3.2 Bad ACU and Modems

The ACU and incoming modems occasionally cause problems that make it difficult to contact other systems or to receive files. These problems are usually readily identifiable since *LOGFILE* entries will usually point to the bad line. If a bad line is suspected, it is useful to use the **cu**(1) command to try calling another system using the suspected line.

11.4.3.3 Administrative Problems

Some **uucp** networks have so many members that it is difficult to keep track of changing passwords, changing phone numbers, or changing logins on remote systems. This can be a very costly problem since ACU's will be tied up calling a system that cannot be reached.

11.5 Debugging

In order to verify that a system on the network can be contacted, the **uucico** daemon can be invoked from a user's terminal directly.

For example, to verify that *mhtsd* can be contacted, a job would be queued for that system as follows:

uucp –r file mhtsd!~/tom

The **–r** option forces the job to be queued but does not invoke the daemon to process the job. The **uucico** command can then be invoked directly:

/usr/lib/uucp/uucico –r1 –x4 –smhtsd

The **–r1** option is necessary to indicate that the daemon is to start up in *master* mode (i.e., it is the calling system). The **–x4** specifies the level of debugging that is to be printed. Higher levels of debugging can be printed (greater than 4) but requires familiarity with the internals of **uucico**. If several jobs are queued for the remote system, it is not possible to force **uucico** to send one particular job first. The contents of *LOGFILE* should also be monitored for any error indications that it posts. Frequently, problems can be isolated by examining the entries in *LOGFILE* associated with a particular system. The file *ERRLOG* also contains error indications.

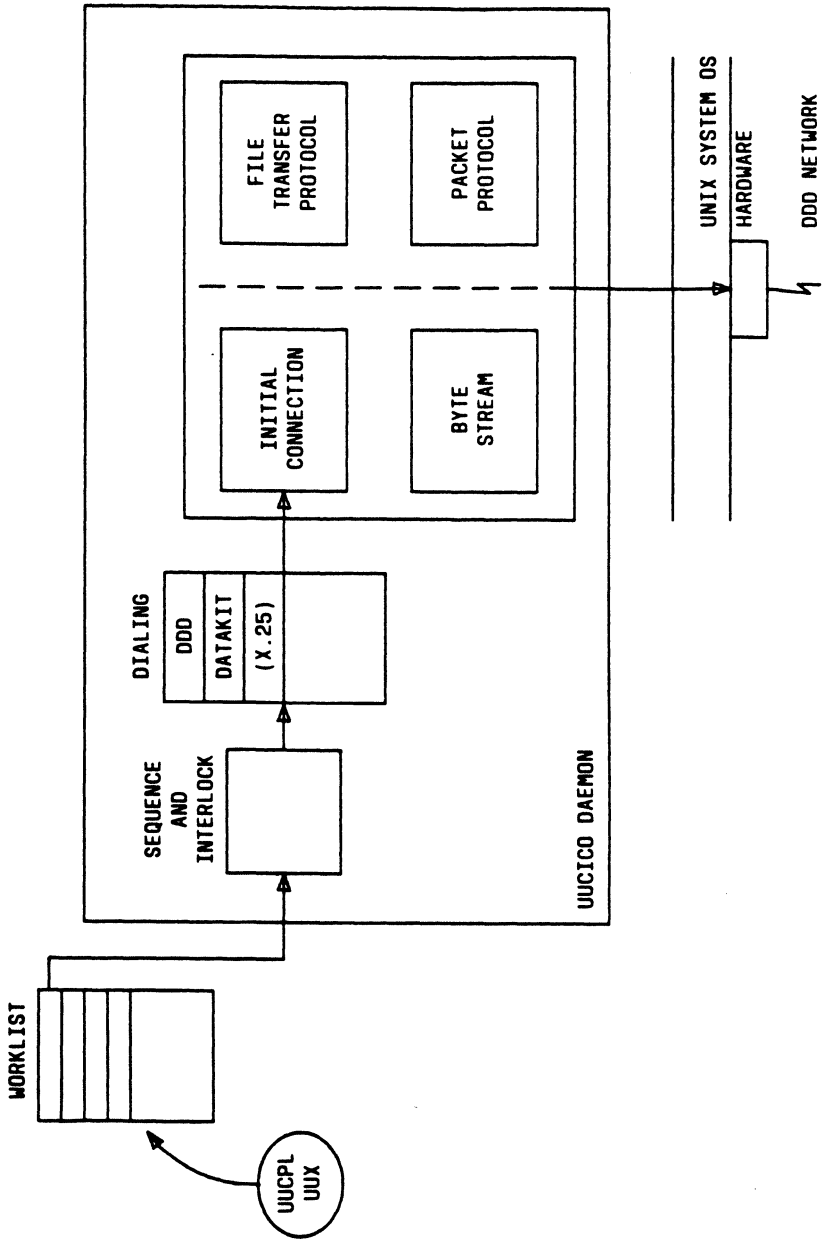


Figure 10-1. Uucico Daemon Functional Blocks

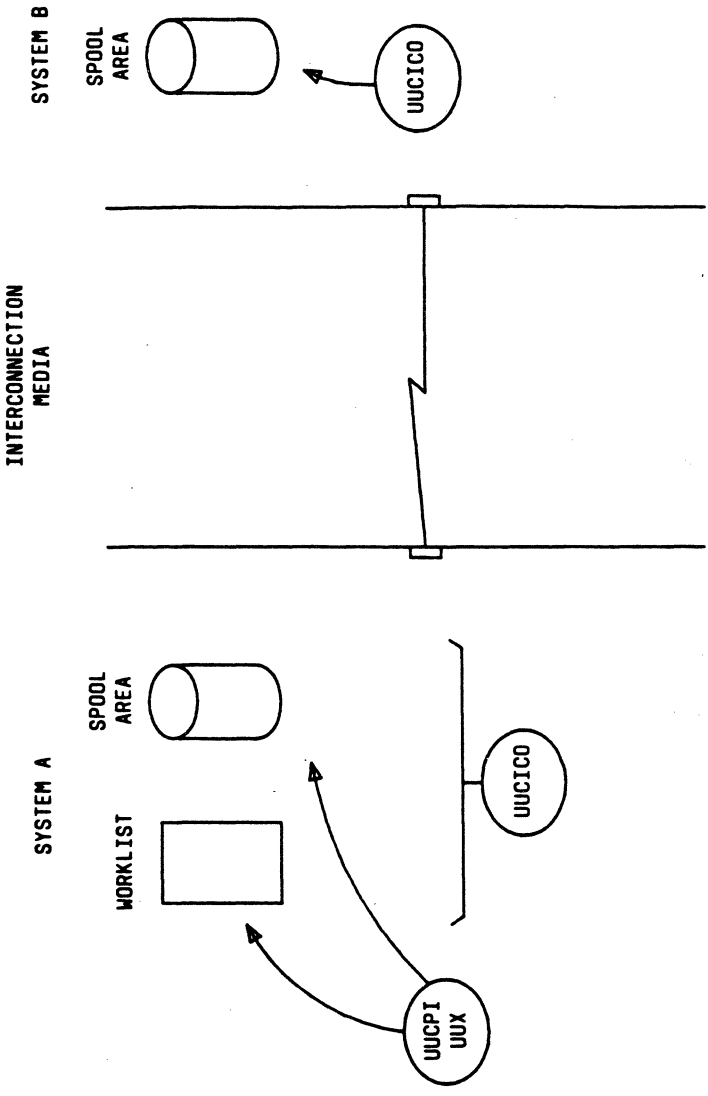


Figure 10-2. Uucp Network Daemon