UniPlus+®
System V, Release 2

Volume 6

Administrator Guide

Adapted to UniPlus^{+} by Heather Allen of UniSoft Systems.

PREFACE

This guide is a reference for those who administer and operate the UniPlus$^+$ system. It contains a description of console operations and general instructions for normal operator and administrator functions as they apply to the family of microprocessors running the UniPlus$^+$ operating system. This guide should be used to supplement the information contained in the *UniPlus$^+$ User Manual* and the *UniPlus$^+$ Administrator Manual* .

This guide contains 16 chapters:

- INTRODUCTION
- ADMINISTRATIVE ADVICE
- OPERATIONS
- START-UP PROCEDURES
- SINGLE USER AND MULTIUSER MODE
- DUTIES
- SYSTEM ACCOUNTING
- FSCK: FILE SYSTEM CHECKING
- LP SPOOLING SYSTEM
- SYSTEM ACTIVITY PACKAGE
- UUCP ADMINISTRATION
- TAKE/PUT: FILE TRANSFER SYSTEM
- UNIX I/O
- UNIX IMPLEMENTATION

- ERROR MESSAGES
- VIRTUAL TUNING

Chapter 1, INTRODUCTION, gives an overview of the system operator and administrator responsibilities.

Chapter 2, ADMINISTRATIVE ADVICE, contains helpful advice and suggestions for system administrators of UniPlus$^+$.

Chapter 3, OPERATIONS, explains some basic operations.

Chapter 4, START-UP PROCEDURES, explains how to start up your UniPlus$^+$ system.

Chapter 5, SINGLE USER AND MULTIUSER MODE, describes the two modes of operation of the UniPlus$^+$ operating system and the commands necessary to set the mode.

Chapter 6, DUTIES, gives specific examples of duties performed by either a computer operator or a system administrator.

Chapter 7, SYSTEM ACCOUNTING, describes the structure, implementation, and management of the accounting system.

Chapter 8, FSCK: FILE SYSTEM CHECKING, describes the file system check program (**fsck**) of the UniPlus$^+$ system. **Fsck** audits and interactively repairs inconsistency in the file system.

Chapter 9, 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**.

Chapter 10, SYSTEM ACTIVITY PACKAGE, describes the design and implementation of the UniPlus$^+$ system activity package. The package reports UniPlus$^+$ system-wide statistics.

Chapter 11, UUCP ADMINISTRATION, describes how a **uucp** network is set up, the format of the control files, and administrative procedures.

Chapter 12, TAKE/PUT: FILE TRANSFER SYSTEM, describes a rudimentary file transfer system.

Chapter 13, UNIX I/O SYSTEM, provides an overview of the UNIX I/O system.

Chapter 14, UNIX IMPLEMENTATION, describes the implementation of the resident UNIX kernel.

Chapter 15, ERROR MESSAGES, describes the UniPlus$^+$ error messages.

Chapter 16, VIRTUAL TUNING, describes the special provisions required by the paging environment.

Throughout this guide, each reference of the form *name*(1M), *name*(7), or *name*(8) refers to entries in the *UniPlus$^+$ Administrator Manual*. All other references of the form *name*(*N*), where *N* is a number, possibly followed by a letter, refer to entries in section *N* of the *UniPlus$^+$ User Manual*.

CONTENTS

Chapter 1: INTRODUCTION

CONTENTS

Chapter 1
INTRODUCTION

1. General

In this guide, procedures and examples are given for starting up your system (booting and powering), changing run levels (that is, single user and multiuser), saving and restoring files, bringing down the system in an orderly manner, and restoring the system after a crash. You should always consult documentation for your processor before performing any of the procedures in this guide.

2. System Console

Most of the operations you do will involve the system console. All messages to the operator and input from the operator are via the system console. You will be using the system console in one of three modes:

- Monitor/Boot — The UniPlus$^+$ operating system is halted. In this optional mode, a monitor or stand alone operating system may be available to operate the processor and load in the boot program, or the boot program may be already running. See the software and hardware reference manuals for your computer for initial procedures and monitor commands.

- Single user — The UniPlus$^+$ operating system is executing. The commands you enter on the system console are UniPlus$^+$ system commands. In single-user mode you are always super-user. When the system is halted or in single-user mode, the console is the only interface to the system, unless you specifically change the configuration so that another terminal acts as a console.

- Multiuser — The UniPlus$^+$ operating system is executing. The system console (and any other configured terminal) is treated as a normal user terminal.

In halt mode or single-user mode, the console will not be treated as a **login** terminal (therefore, you are super-user). When you change the system to multiuser mode, a **login** message will appear on the console. You must provide a **login** and password at this point in order to use the console. Normally you should log in as **root**. Here, it must be mentioned that the **login** you use is a local decision. In fact, the system administrator may configure your system so that it is not even necessary for you to log in after changing to multiuser.

Normal daily maintenance requirements are described and examples provided of normal operations (not including local procedures). For more information on the console (for example, set-up procedures), consult your console terminal owner's manual.

3. Input/Output Notations

Throughout this guide, the following notation is used for computer input/output:

1. Special characters are in all caps (for example, when you see CONTROL read this as the "control" or "CTRL" keyboard character and RETURN as the "carriage return" key).
2. Items within [ ]s are optional.
3. You should type in literally any indented command field that appears boldface (a keyword).
4. You should substitute with the appropriate information any command field that appears in italics.
5. All commands (system or console commands) should be terminated with a carriage return.

4. Local Needs

Because this guide is intended to be as general as possible, no machine-specific or installation-specific information has been included. Also, some operations may vary according to local procedures. It is suggested that you add specific information about:

- Hardware configuration
- Software configuration of administrative files
- Data set configuration
- Specific logging and record-keeping practices
- Contacts for hardware and software problems
- Site-dependent diagnostic procedures.

Chapter 2: ADMINISTRATIVE ADVICE

CONTENTS

LIST OF FIGURES

Chapter 2
ADMINISTRATIVE ADVICE

1. Introduction

This chapter describes administrating the UniPlus$^+$ operating system.

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 most of the distributed documentation. All of the sections of the *UniPlus$^+$ Administrator Manual* should be studied.

Pay special attention to the following in the *UniPlus$^+$ Administrator Manual* and *UniPlus$^+$ User Manual*:

chmod(1)	**mail**(1)
chown(1)	**mkdir**(1)
cpio(1)	**ps**(1)
date(1)	**rm**(1)
du(1)	**rmdir**(1)
ed(1)	**su**(1)
env(1)	**time**(1)
find(1)	**who**(1)
kill(1)	**write**(1)
acct(1M)	**mkfs**(1M)
checkall(1M)	**ncheck**(1M)
dcopy(1M)	**shutdown**(1M)
df(1M)	**sync**(1M)
errpt(1M)	**volcopy**(1M)
fsck(1M)	**wall**(1M)
fuser(1M)	

acct(4)

all of section 7

crash(8)

3. 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 over 20 users, the root file system (/**bin**, /**tmp**, and /**etc**) deserves a drive of its own. If there is a noisy modem (poorly executed do-it-yourself null-modem) or a disconnected modem cable, the UniPlus$^+$ 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.

4. File System Backup Programs

The following backup programs are distributed:

- **Find/cpio**: The UniPlus$^+$ 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; judgements are subjective.

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.

5. Controlling Disk Usage

Once the UniPlus$^+$ system is a success, disk space will soon become limited. During the long delay before more drives become available, usage should be controlled. Try to maintain the start-of-day counts recommended. Watch usage during the

	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

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.

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 to be 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).

6. 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.

7. 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
```

8. 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.

9. Watch Out For Files and Directories That Grow

Most of the files below 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.

10. 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.

11. 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.

12. 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 to get rid of users. Manual policing will help some, but "automatic" policing will be *invariably* subverted by users.

13. "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.

14. 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 UniPlus$^+$ system word processing capabilities; see **term**(5).

15. Line Printers

Most line printers are troublesome and impose considerable overhead on the system. Most also lack hardware tabs, character overstrike capability, etc. A printer that will work over an asynchronous link (DC1/DC3 protocol required) is the best bet.

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.**

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.

18. 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

Chapter 3: OPERATIONS

CONTENTS

Chapter 3
OPERATIONS

1. Introduction

Information on system operations should be obtained from the manufacturer of your box. Console commands and start-up procedures vary, depending on hardware configurations.

2. Booting

In general, a **boot** program is used to start up UniPlus$^+$. This **boot** program can reside in PROM, or on a floppy, or in the beginning of a hard disk. The **boot** program must first find out where UniPlus$^+$ resides either by looking at a specific place on the disk, or prompting the user for this information. Once UniPlus$^+$ is located on the file system, the **boot** program will load it from disk to memory. For specific booting instructions, refer to the manual from the manufacturer of your box.

Once loaded, the UniPlus$^+$ operating system is ready to come up. The system will scan the */etc/inittab* file to determine among other things, which run level will be entered. If this file specifies a run level (or a default level is found), the system will enter the run level specified. Otherwise, do the following steps:

1. This message should appear on the console:

   ENTER RUN LEVEL (0-6, s or S):

   Enter **2<cr>** to go to multiuser state, or **s<cr>** to go to single user state.

2. If you requested multiuser in step 1, the system will ask you to verify the date. Then you will be asked if the file systems are to be checked. Finally, the following message

will be printed on the console:

Console Login:

If you requested single user in step 1, the # prompt will be printed. In this case, typing **telinit 2** will change the operating system state to multiuser.

3. Shutting Down

The shutdown procedure is designed to gracefully turn off all processes and bring the system back to single user state with all buffers flushed. To do this you should execute **shutdown** as described in Chapter 6. If **shutdown** is not successful, use the following sequence of commands:

```
killall
sync
init S
fsck      This is optional
```

4. Powering Down

The shutdown sequence should always be run **before** powering down. Disk drives, where they require separate powering, should be powered down **before** powering down the processor. Refer to instructions from the manufacturer for any other specific procedures.

Chapter 4
START-UP PROCEDURES

Below is a description of how to start up your UniPlus$^+$ system. A variety of procedures may be necessary to start the system. The processor and peripherals (such as disk drives) may need to be powered up. Additionally, a combination of hardware and software resets and monitor commands may be required. The final step in starting up the system is generally the **boot**. The **boot** procedure loads a copy of the UniPlus$^+$ operating system from disk, floppy, tape, or some other media into memory and executes it.

You will need to reboot the UniPlus$^+$ operating system when one of the following conditions occur:

- system crash or restart;
- loading of a new software release; or
- updating of the software release.

Once loaded, the UniPlus$^+$ operating system will typically enter the single-user "run level" awaiting your commands. When properly configured by the system administrator, the UniPlus$^+$ operating system uses **init** to automatically enter the final run level. Run levels are discussed in the "Single User and Multiuser Mode" chapter of this guide. Normally, run level s indicates single user and 2 indicates multiuser. For more information on **init** refer to **init**(1M) in the *UniPlus$^+$ Administrator Manual*, **inittab**(4) in the *UniPlus$^+$ User Manual*, or, if you are an operator, consult the local system administrator.

START-UP

See the relevant software or hardware reference manual for your computer for detailed powering and booting procedures.

Chapter 5: SINGLE USER AND MULTIUSER MODE

CONTENTS

Chapter 5

SINGLE USER AND MULTIUSER MODE

1. Introduction

There are two main modes of operation of the UniPlus$^+$ operating system: single user (level **S**) and multiuser (level **2**). The run level has eight possible values: **0-6** and **S** (or **s**). Single user is always **S** or **s**. Although multiuser is normally level **2**, the system administrator can configure the */etc/inittab* file to run multiuser at any level from **0** to **6**.

The */etc/inittab* file can also be configured so that certain procedures are followed automatically only the first time that a certain run level is entered. For example, normally you will be asked to verify date and file systems the first time you change your system to multiuser. This is caused by an entry in the **inittab** file. Subsequent changes in run level will not perform this procedure automatically unless you specifically change the **inittab** file. For more information on **init** refer to **init**(1M) in the *UniPlus$^+$ Administrator Manual*, **inittab**(4) in the *UniPlus$^+$ User Manual*, or, if you are an operator, consult your local system administrator.

When in single-user mode, all dial-up ports and hard-wired terminals are disabled and only the console terminal may interact with the processor. This mode of operation allows you to make necessary changes to the system without any other processing taking place. However, you will normally run the UniPlus$^+$ operating system in multiuser mode. Consult the documentation for your particular processor before proceeding with any of these procedures.

2. Single-User Environment

In single-user mode, you may type any available system command (followed by a **RETURN**). When the system has completed execution of the command, it will prompt with the "#" again on the next line. You use the single-user environment primarily to do *filesaves*, system maintenance, modification, or repair operations. The typical sequence of commands to change the system to multiuser mode is:

1. **fsck**
2. **telinit 2**

2.1 The Fsck Command

The command **fsck** will interactively repair any damaged file systems that result from a crash of the operating system. You should also use it to ensure that the file systems are not damaged before going into multiuser mode or taking filesaves. Usually, you will want to respond "yes" to all the prompts; however, in the event of a system crash, the damage may be extensive enough to warrant recovery from a backup pack. The procedure for this is discussed under "FILESAVES" in Chapter 6. See **fsck** in the *UniPlus$^+$ Administrator Manual* for details on the various options available and Chapter 8 in this guide for a description of all the different errors that can occur.

An example of a check of a consistent file system is illustrated below:

```
# fsck /dev/rsmd1
/dev/rsmd1
File System: usr Volume: p0603
** Phase 1 — Check Blocks and Sizes
** Phase 2 — Check Pathnames
** Phase 3 — Check Connectivity
** Phase 4 — Check Reference Counts
** Phase 5 — Check Free List
2441 files 16547 blocks 31889 free
#
```

A file system that has been damaged can be repaired as shown below. The **y** is your response. When checking a file system, you can avoid the questions asked by **fsck** concerning inconsistencies found by using the **y** option. This option will automatically attempt repairs as though you answered "yes" to the questions. Use this with caution—the corrections usually involve some data loss. If you decide to interactively repair the file system, then follow the example below:

```
# fsck /dev/rsmd2
```

The UniPlus$^+$ operating system responds:

```
/dev/rsmd2
File System: fs1 Volume: p0603
** Phase 1 — Check Blocks and Sizes
POSSIBLE FILE SIZE ERROR I=2500
** Phase 2 — Check Pathnames
** Phase 3 — Check Connectivity
** Phase 4 — Check Reference Counts
UNREF FILE I=2500  OWNER=255 MODE=100755
SIZE=0 MTIME=Dec 31 19830 1983
CLEAR? y
** Phase 5 — Check Free List
2441 files 16547 blocks 889 free
***** FILE SYSTEM WAS MODIFIED *****
#
```

All mountable file systems should be listed in the file **/etc/checklist** which **fsck** uses, and you should check these file systems each time the system is rebooted.

A faster alternative to using **fsck** is **checkall**. The **checkall** command uses **dfsck** (a front end for **fsck**) to simultaneously check two file systems in different disk drives. Included in **checkall** are the file system names that normally appear in **/etc/checklist** (see **checkall** in the *UniPlus$^+$ User Manual*).

WARNING: Never execute **fsck** on a mounted file system; it will have a bad effect since you are repairing only the physical disk. The only exception to this is the **root** file system which is always mounted.

An example of repairing the **root** file system follows:

```
# fsck /dev/smd0
/dev/smd0
File System: root Volume: p0001
** Phase 1 — Check Blocks and Sizes
POSSIBLE FILE SIZE ERROR I=416
POSSIBLE FILE SIZE ERROR I=610
POSSIBLE FILE SIZE ERROR I=614
POSSIBLE FILE SIZE ERROR I=618
POSSIBLE FILE SIZE ERROR I=625
** Phase 2 — Check Pathnames
** Phase 3 — Check Connectivity
** Phase 4 — Check Reference Counts
UNREF FILE I=416  OWNER=uucp MODE=100400
SIZE=0 MTIME=Nov 20 16:23 1983
CLEAR? y
UNREF FILE I=610  OWNER=csw MODE=100400
SIZE=0 MTIME=Nov 20 16:26 1983
CLEAR? y
UNREF FILE I=625  OWNER=cath MODE=100400
SIZE=0 MTIME=Nov 20 16:26 1983
CLEAR? y
FREE INODE COUNT WRONG IN SUPERBLK
FIX? y
** Phase 5 — Check Free List
1 DUP BLKS IN FREE LIST
BAD FREE LIST
SALVAGE? y
** Phase 6 — Salvage Free List
585 files 5463 blocks 4223 free
***** BOOT UNIX (NO SYNC !) *****
#
```

At this time you must immediately halt the processor and then reboot the system (see the relevant software or hardware reference manual for your computer for start-up procedures.)

2.2 The Telinit 2 Command

After you have checked the file systems, you may change the UniPlus$^+$ operating system to multiuser. Do this by entering the command **telinit 2**. This command activates processes that allow users to log in to the system, turn on the accounting and error logging, mount any indicated file systems, and start the **cron** and any indicated daemons. Depending upon the type of data set your site has, you may have to manually flip the toggles or pop the buttons on the data sets to allow users to log in.

3. Multiuser Environment

There are two ways to get to this level: by typing **telinit 2**; or, specifying a run level of 2 after the boot. Users are permitted to access all mounted file systems and execute all available commands. In this mode, you can perform file restore procedures and take periodic status checks of the system. Some of these periodic status checks can include:

- A check of free blocks (**df**) remaining on all mounted file systems to ensure that a file system does not run out of space.
- A check on **mail** to **root** or whatever **login** receives requests for file restores.
- A check on the number of users on the system (**who**).
- A check of all running processes ("**ps -eaf**" or **whodo**) to determine if there is some process using an abnormally large amount of CPU time.

If your site has other run levels defined, you can use the **telinit** command to change to those run levels. Finally, to change a multiuser system to single user, refer to "SYSTEM SHUTDOWN" in Chapter 6.

Chapter 6: DUTIES

CONTENTS

Chapter 6
DUTIES

1. Introduction

This chapter is a guide for the normal duties of a computer operator or system administrator. These descriptions do not represent what specific job duties are; they merely outline the general procedures to ensure that the system operates properly. Consult instructions for your processor before proceeding with any of these procedures.

2. Filesaves

Unless you make frequent copies of the file systems, a major system crash could devastate your user community. The user files could be destroyed or become inaccessible.

You should take daily *filesaves*. Should the system crash and lose files, then, at most, only a day's work will be lost. If your last filesave (or backup) was a week ago, then even after restoring the file any changes made since that backup will be lost.

There are two ways you can do filesaves: by disk and by tape. Most sites use **volcopy** to save files. See **volcopy** in the *UniPlus$^+$ Administrator Manual* for more information on the available options and use this command. You should normally do your file saving while in single-user mode, with the file system unmounted, to preclude any file system activity and subsequent damage on the saved copy. Also, to ensure system buffers are flushed and file systems are up to date, execute the **sync** command before filesaves.

Normally the filesave procedure is automated by the system administrator. You or your administrator may have created a *shell script* to perform the filesave as part of your site's local operation. Daily filesaves usually are made on disk; whereas, a weekly filesave would be more efficiently made on tape. Tape saves are necessary for long-term storage or for regular saves if you do not have a spare disk. Tapes may be previously labeled, or may be labeled by the **volcopy** command. You or your administrator may have created separate shell scripts for disk and tape saves (incorporating the procedures that follow).

You must have at least two disks, one of them a spare, for the following procedures. For ease of mapping, file systems are normally saved in the same partitions on the backup disk as they exist on the working disk. This is imperative if you ever need to boot from a backup version of **root**. The **root** file system must reside on partition **a** of the disk.

2.1 Saving the Root File System on Disk

In this example, the **root** file system on disk **0** will be saved on disk **1**, volume *S3B002* (whatever volume name you use should match the external label sticker).

1. Connect the disk to contain the filesave as disk **1**.
2. Enter the commands:

   ```
   # sync
   # fsck /dev/w0a
   # volcopy root /dev/rw0a S3B001 /dev/rw1a S3B002
   ```

   to copy the root file system from disk **0** partition **a** to disk **1** partition **a**. The following messages should appear:

   ```
   From: /dev/rw0a, to: /dev/rw1a? (DEL if wrong)
   END: 23000 blocks.
   #
   ```

   If the **from** and file systems are correct, wait for the prompt; otherwise, press the **DELETE** key to abort the copy.

3. Do step 2 for all the partitions of the disk to copy.
4. Disconnect and remove disk **1**.

In the above procedure, **fsck** in step 3 asks you to concur with any repairs necessary before attempting them. If you respond **no**, no action will be taken and **fsck** will continue. Also, **volcopy** verifies the label information on the **to** and **from** file system (for example, pack number, file system name, date last modified). You will be asked to override inconsistencies before the copy proceeds. For example:

```
# volcopy root /dev/rw0a p0001 /dev/rw1a p0105
arg.(p0105) doesn't agree with to vol.()
Type 'y' to override: y
warning! from fs(root) differs from to fs()
Type 'y' to override: y
From: /dev/rw0a, to: /dev/rw1a? (DEL if wrong)
END: 23000 blocks.
#
```

Note: In this example, the **to** partition is unlabeled, as indicated by the null volume and file system fields. For more information see **volcopy** in the *UniPlus⁺ Administrator Manual.*

2.2 Saving the User File System on Disk

In this next example, the **usr** file system, on partition **b** of disk **0**, will be saved on disk **1**, volume *p0603.*

1. Connect the disk to contain the file-save on disk **1**.
2. Enter the commands:

```
# sync
# umount /dev/w0b
# fsck /dev/rw0b
# volcopy usr /dev/rw0b p0001 /dev/rw1b p0603
```

to copy the **usr** file system from disk **0** partition **b** to disk **1** partition **b**. The following messages should appear:

```
From: /dev/rw0b, to: /dev/rw1b? (DEL if wrong)
END: 23000 blocks.
#
```

If the **from** and **to** file systems are correct, wait for the prompt; otherwise, press the **DELETE** key to abort the copy.

3. Do step 2 for all the partitions of the disk to copy.
4. Disconnect and remove the disk.

2.3 Saving the User File System on Tape

In this example, the **usr** file system is saved on tape volume *t0001*, mounted on transport **0**. The **labelit** command is used to label the tape before the copy. You should place an external paper label on the outside of the reel carrying the same information as is written in the tape header label. The external label should also indicate the sequence number of the tape if it is from a set (multi-reel volume) for the file system. Note the use of the **−n** option to **labelit**. Unless this option is used on an unlabeled tape, the program will scan the entire reel looking for a label to change before it rewinds and labels the beginning. This can be very time-consuming on 2400-foot reels.

You can store approximately 65,000 blocks of a file system on a 2400-foot tape using **volcopy** and recording at 1600 bpi. You may specify the size and type of tape in the **volcopy** command, or you can let the system prompt for the information as shown. In the example that follows, the file system requires two reels. Although this example uses only one drive, you can have both reels mounted on different drives. In that case, when the first has finished, you would simply enter the name of the second drive when asked.

1. Load the tape in transport **0**, and label it:

```
# labelit /dev/rmt0 usr t0001 −n
Skipping label check!
NEW fsname = usr, NEW volume = t0001 -- DEL if wrong!!
#
```

2. Enter the following commands:

```
# sync
# umount /dev/w0b
# fsck −y /dev/rw0b
# volcopy usr /dev/rw0b p0001 /dev/rmt0 t0001
Enter size of reel in feet for <t0001>:  2400

Reel t0001, 2400 feet, 1600 BPI
You will need 2 reels.
(The same size and density is expected for all reels)
From: /dev/rw0b, to: /dev/rmt0? (DEL if wrong)

Writing REEL 1 of 2, VOL = t0001
Changing drives? (RETURN if no, /dev/rmt_ if yes): RETURN

Mount tape 2
Type volume-ID when ready:  t0002
Cannot read header      (This tape has not been labeled!)
Type y to override: y
Volume is <garbage>, not <t0002>.
 Want to override? y

Writing REEL 2 of 2, VOL = t0002
END: 90000 blocks.
#
```

3. File Restores

3.1 Restoring from Disk

When a request is made to restore a file from a backup disk, you should first locate that disk and determine on which partition the requested file system resides. Then at the console terminal, log in to the system as **root** and proceed as the example illustrates. Following is the procedure for restoring the file **/usr/adm/acct/sum/tacct** from a previous backup disk. For this example, disk **1** is the backup disk and **/usr** is on partition **0** of the disk.

1. Connect the disk as disk **1**.
2. Enter the command:

   ```
   # mount /dev/w1b /bck −r
   ```

   This will mount the backup file system as **/bck** read-only. The following message should appear:

   ```
   WARNING!! − mounting <usr> as </bck>
   ```

3. Enter the command:

   ```
   # ls −l /bck/adm/acct/sum/tacct
   ```

   This will verify the existence of the file and the identity of the owner. The following output will appear:

   ```
   -rw-rw-r-- 1 adm  bin 1932 Aug 9 14:27 /bck/adm/acct/sum/tacct
   ```

4. Enter the command:

   ```
   # cp /bck/adm/acct/sum/tacct /usr/adm/acct/sum/tacct
   ```

   to copy the file from the backup to the specified place.

5. Enter the command:

   ```
   # chown adm /usr/adm/acct/sum/tacct
   ```

   to change the owner of the file.

6. Enter the command:

   ```
   # umount /dev/w1b
   ```

   This will unmount the backup file system.

7. Disconnect and remove the backup disk.

When you perform a file restore, it is usually a good practice to **mail** a message to the user asking for the restore when you are finished. Also, to avoid confusion, your message should refer to the file using a full pathname. The procedure for this is:

```
# mail user
I have restored the file /usr/adm/acct/sum/tacct
from Friday's backup.
your initials
#
```

3.2 Restoring from Tape

If the file does not exist on any of the backup disks or if your installation does not perform disk filesaves, then you will have to recover the file from a tape save. It is assumed that you do your tape saves in the same manner as disk saves, that is, with **volcopy**. Filesaves are discussed earlier in this chapter. To restore a file from tape, you must place the whole file system on a spare partition of the disk. The backup tape version can then be accessed in the same way as a disk save. For this example, it is assumed that there are two small file systems stored on a single tape and that the **usr** file system is the second file on the tape. Also, it is assumed that partition **e** of disk **0** is a spare partition on that disk. The tape drive is already in service.

1. Mount tape on tape drive **0**.
2. Enter the command:

   ```
   # echo < /dev/mt0
   ```

   This will space past the first file on the tape, with no rewind.
3. Enter:

```
# volcopy usr /dev/mt0 t0004 dev/rw0e S3B003
```

This will copy the file system from tape to the spare disk partition. The following messages should appear:

```
From: /dev/mt0, to: /dev/rw0e? (DEL if wrong)
END: 90000 blocks.
```

4. Enter the command:

   ```
   # mount /dev/w0e /bck -r
   ```

   This will mount the backup partition. The following message should appear on the screen:

   ```
   WARNING!! — mounting: <usr> as </bck>
   ```

5. Enter the command:

   ```
   ls -l /bck/adm/acct/sum/tacct
   ```

   This will verify the existence of the file and identify the owner. The following output will appear:

   ```
   -rw-rw-r-- 1 adm   bin  1932 Aug  9 14:27 /bck/adm/acct/sum/tacct
   ```

6. Enter:

   ```
   cp /bck/adm/acct/sum/tacct /usr/adm/acct/sum/tacct
   ```

   This will copy the file to the specified place.

7. Enter the command:

   ```
   chown adm /usr/adm/acct/sum/tacct
   ```

   to change the owner of the file.

8. Enter the command:

   ```
   umount /dev/w0a
   ```

   This will unmount the spare partition.

Sometimes a file system is so large it requires more than one tape to store the contents. In this situation, you follow the same procedure to restore a file as in the previous example.

The **volcopy** command prompts you for additional reels when necessary. In this example, the second reel has the wrong label. The **y** response overrides the inconsistency and the reel is read anyway.

1. Mount tape on tape drive 0.
2. Enter:

   ```
   volcopy -bpi1600 -feet2400 usr /dev/rmt0 t0004 dev/rw0e S3B003
   ```

   This will copy the file system from tape to the spare disk partition. The following messages should appear:

   ```
   Reel 1, 2400 feet, 1600 BPI
   From: /dev/rmt0, to: /dev/rw0e? (DEL if wrong)

   Reading REEL 1 of 3, VOL = 1
   Changing drives? (RETURN if no, /dev/rmt_ if yes): RETURN
   Mount tape 2
   Type volume-ID when ready: 2
   Volume is <1>, not <2>.
   Want to override? y

   Reading REEL 2 of 3, VOL = 1
   Fri Jul 29 12:00:02 EDT 1983

   Changing drives? (RETURN if no, /dev/rmt_ if yes): RETURN
   Mount tape 3
   Type volume-ID when ready: 3

   Reading REEL 3 of 3, VOL = 3
   END: 90000 blocks.
   ```

3. Enter the command:

   ```
   mount /dev/w0e /bck -r
   ```

   This will mount the backup partition. The following message should appear on the screen:

```
WARNING!! — mounting: <usr> as </bck>
```

4. Enter the command:

   ```
   ls -l /bck/adm/acct/sum/tacct
   ```

   This will verify the existence of the file and identify the owner. The following output will appear:

   ```
   -rw-rw-r-- 1 adm  bin 1932 Aug 9 14:27 /bck/adm/acct/sum/tacct
   ```

5. Enter:

   ```
   cp /bck/adm/acct/sum/tacct /usr/adm/acct/sum/tacct
   ```

   This will copy the file to the specified place.

6. Enter the command:

   ```
   chown adm /usr/adm/acct/sum/tacct
   ```

   to change the owner of the file.

7. Enter the command:

   ```
   umount /dev/w0e
   ```

   This will unmount the spare partition.

4. Message of the Day

When a user logs into the system, part of the **login** procedure prints out a message of the day. This message can contain several lines of useful information concerning scheduled down-time for hardware preventive maintenance (PM), clean-up messages for space-low file systems, or any other useful warnings. The trick to maintaining this file is to keep it short and to the point. A user does not want to wait ten minutes while eloquent and wordy dialogue is spewed from the terminal before he or she can begin working.

The contents of this message are stored in the file **/etc/motd**. You may change the contents of this file by using the UniPlus$^+$ system text editor. See **ed** or **vi** in the *UniPlus$^+$ User Manual.*

A sample of adding and deleting a line from this file is shown below.

```
# ed /etc/motd
26
p
9/23: Reboot at 5pm today.
d
a
9/24: Down for PM 1700-2100 on 9/30.
.
w
37
q
#
```

You can also remove the contents of the entire file (do not remove the file itself; it needs to exist so the **login** process can read it) by:

```
# cp /dev/null /etc/motd
#
```

5. System Shutdown

You will perform three distinct steps when bringing down your UniPlus$^+$ system. These steps must be performed in the indicated order, although it is not necessary to bring the system completely down for certain maintenance operations. For example, preventive maintenance (such as filesaves) must be done while in single-user mode without halting the UniPlus$^+$ system. Whereas, repairing a hard fault would necessitate removing power completely. You should never remove power from a piece of equipment that is in service, and definitely do not power down the system until the UniPlus$^+$ operating system has been halted. To bring down the system:

- Run the shutdown program (changes a multiuser system to single-user mode).
- Halt the UniPlus$^+$ program.
- Remove power.

5.1 Shutdown Program

Whenever the system must be shut down, such as for filesaves or a reboot, you should run the program **/etc/shutdown**. The shutdown procedure is designed to gracefully turn off all processes and bring the system back to single-user state with all buffers flushed.

You must be in the root directory (/) to use the shutdown program. You may specify the amount of grace period between sending a warning message out and actually shutting down. This grace period is the number of seconds of delay. For example, specifying a grace period of 300 will result in a 5-minute delay. You may also send your own message. A default message is sent to all logged-in users if you don't type your own. The following printout is an example of a typical shutdown sequence. Enter the following:

```
# cd /
# shutdown 300
```

Your shutdown procedure may vary slightly from the following, depending on how it is set up in your system. The shutdown *script* may be modified according to local procedures. A typical output is as follows:

```
SHUTDOWN PROGRAM

Thu Sep  1 18:51:58 EST 1983

Do you want to send your own message? (y or n): y
Type your message followed by <ctrl>d....

System coming down for filesaves!
Please log off.
<ctrl>d

System coming down for filesaves!
Please log off.
(waits for 5 minutes)
SYSTEM BEING BROUGHT DOWN NOW ! ! !

Busy out (push down) the appropriate
phone lines for this system.

Do you want to continue? (y or n):  y
Process accounting stopped.
Error logging stopped.
All currently running processes will now be killed.

Wait for 'INIT: SINGLE USER MODE' before halting.
```

If you executed the shutdown program while in single-user mode, (which is neither useful nor recommended) the system will not respond with the 'INIT' message above.

At the completion of this program you can either halt the system (and reboot if necessary), power down, start the filesave routine or other preventive maintenance, or bring the system back to multiuser mode. To go to multiuser, type in **telinit 2**. See the Chapter 5, SINGLE USER AND MULTIUSER MODE, for more information on changing run level.

6. System Crash Recovery

An operating system is considered to have *crashed* when it halts itself without being asked to. The reason for the halt is often unknown and can be hardware failure or software related. It is important, for obvious reasons, to determine the nature of the crash so that it will not happen again. Note any messages that appear on the console, and any pertinent information on the processing that was going on at the time the crash occurred.

Chapter 7: ACCOUNTING

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 7
SYSTEM ACCOUNTING

1. Introduction

The UniPlus$^+$ 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.

2. General

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

- At process termination, the UniPlus$^+$ 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 UniPlus$^+$ system usage.

3. 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 7.1.

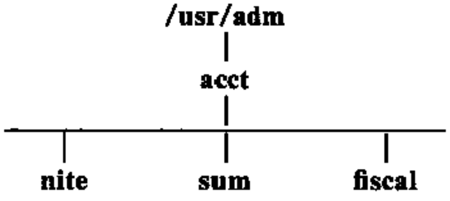

Figure 7.1. Directory Structure of the "adm" Login

The */usr/adm* directory contains the active data collection files. (For a complete explanation of the files used by the accounting system, see the table at the end of this section.) The *nite* directory contains files that are re-used 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**.

4. Daily Operation

When the UniPlus$^+$ 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. Setting up the Accounting 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
   ```

6. 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 re-entrant 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.

7. 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.

8. 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&
```

9. 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.

9.1 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 UniPlus$^+$ 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.

9.2 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.

10. 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 1985, prime time at 9:00 a.m., and nonprime time at 4:30 p.m., the following entry would be appropriate:

   ```
   1985  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.

11. 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.

11.1 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 *UniPlus*$^+$ *Administrator 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 any more. 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 affected.

11.2 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 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 amok.
# 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.

11.3 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 and 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.

11.4 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.

12. Summary

The UniPlus$^+$ system accounting was designed from a 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.

TABLE 7.1. 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**

TABLE 7.2. 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 *?*

TABLE 7.3. Files in the /usr/adm/acct/nite directory (Page 1 of 2)

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)

TABLE 7.3. Files in the /usr/adm/acct/nite directory (Page 2 of 2)

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

TABLE 7.4. 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

Chapter 8: FSCK: FILE SYSTEM CHECKING

CONTENTS

Chapter 8

FSCK: FILE SYSTEM CHECKING

1. Introduction

The File System Check Program (**fsck**) is an interactive file system check and repair program. **Fsck** uses the redundant structural information in the UniPlus$^+$ 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 UniPlus$^+$ 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.

The **fsck** error conditions are listed in the last section of this chapter. The meanings of the various error conditions, possible responses, and related error conditions are explained.

2. General

When a UniPlus$^+$ 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.

2.1 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.

3. Update of the File System

Every working day hundreds of files are created, modified, and removed. Every time a file is modified, the UniPlus$^+$ 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.

3.1 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.

3.2 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.)

3.3 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.

3.4 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.

3.5 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.

4. 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.

4.1 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).

4.2 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 non-functional disk controller.

5. 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.

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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.

5.1.4 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.

5.2 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.

5.2.1 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:

1. Regular
2. Directory
3. Special block
4. 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.

5.2.2 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.

5.2.3 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.

5.2.4 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.

5.2.5 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 system does not fill in blocks in files created in random order.

5.3 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".

5.4 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.

5.5 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. Fsck Error Conditions

6.1 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 in the next section.

6.2 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.

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 *UniPlus*$^+$ *Administrator Manual* for further details.

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 *UniPlus*$^+$ *Administrator Manual* for further details.

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 *UniPlus*$^+$ *Administrator Manual* for further details.

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 *UniPlus*$^+$ *Administrator Manual* for further details.

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.

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.

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*.

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.

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*.

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*.

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*.

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.

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*.

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.

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.

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.

6.3 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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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 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.

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.

6.5 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.

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.

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.

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.

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.

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.

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.

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.

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.

6.6 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.

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.

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 *UniPlus$^+$ Administrator Manual* for further details.

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 *UniPlus$^+$ Administrator Manual* for further details.

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.

6.7 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.

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.

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.*

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.*

(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.

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.

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.

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.

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 *MR, 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.

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.

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.

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.

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.

6.8 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.

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.

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.

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.

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.

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.

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.

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.

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.

6.9 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.

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 *UniPlus*$^+$ *Administrator Manual* for further details.

6.10 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.

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.

******* 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 UniPlus$^+$ system is not rebooted immediately without **sync**, the work done by **fsck** may be undone by the in-core copies of tables the UniPlus$^+$ system keeps.

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

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

Chapter 9: LP SPOOLING SYSTEM

CONTENTS

Chapter 9
LP SPOOLING SYSTEM

1. Introduction

The line printer (LP) program is a series of commands that perform diverse spooling functions under UniPlus$^+$. Since the primary LP application is off-line printing, this document focuses mainly on spooling to line printers. LP allows administrators to spool to a collection of line printers of any type and to group printers into logical classes to maximize the throughput of the devices. Users can:

- Queue and cancel print requests.
- Prevent and allow queuing to devices.
- Start and stop LP from processing requests.
- Change printer configuration.
- Find status of the LP system.

This chapter describes the role of an LP administrator.

2. Overview of LP Features

2.1 Definitions

We define several terms before presenting a brief summary of LP commands. The LP was designed to meet the needs of users on different UniPlus$^+$ 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 UniPlus$^+$ system pathname. A *printer* is a logical name that represents a device. At different times, 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 directs 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.

2.2 Commands

2.2.1 Commands for General Use

The **lp**(1) command is used to request printing 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 cancels 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.

2.3 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 UniPlus$^+$ system as **lp** qualifies as an LP administrator. All LP files and commands are owned by **lp**. The following commands are 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.

3. Installing LP

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 UniPlus$^+$ is restarted.

PRECAUTIONS

1. The files under the /usr/spool/lp directory should be modified **only by LP commands.**
2. All LP commands require set-user-id permission. If this is removed, the commands will fail.

4. 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.

4.1 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
```

4.2 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
   ```

4.3 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
   ```

4.4 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. Removing a class never implies removing 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.

5. 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 print copies of the file, 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, 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)].

6. Finding LP Status—"Lpstat"

The **lpstat** command finds 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
   ```

* Registered trademark of Xerox Corporation

7. Cancelling Requests—"Cancel"

You can cancel LP requests with 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.

8. Allowing and Refusing Requests—Accept and Reject

When a new destination is created, **lp** rejects 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 you may want to have **lp** 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
   ```

9. 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 **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 re-enabled. The **−c** option cancels the currently printing requests on busy printers 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, re-enable xyz:

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

10. 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 moving 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.

11. Stopping and Starting the Scheduler—"Lpshut" and "Lpsched"

Lpsched is the program that routes the output requests (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. If 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 UniPlus$^+$ 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 to prevent more than one scheduler from running at the same time.

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

```
/usr/lib/lpshut
```

causes **lpsched** to stop running and terminates all printing. 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.

12. 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. When 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.

13. Setting up Hard-Wired Devices and Login Terminals as LP Printers

13.1 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 UniPlus$^+$ 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 print 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.

13.2 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 UniPlus$^+$ is rebooted or when **lpsched** is restarted, it will be disabled automatically.

14. 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.

Chapter 10: SYSTEM ACTIVITY PACKAGE

CONTENTS

Chapter 10
SYSTEM ACTIVITY PACKAGE

1. Introduction

This chapter describes the design and implementation of the UniPlus$^+$ System Activity Package. UniPlus$^+$ contains several counters that are incremented as system actions occur. The system activity package reports UniPlus$^+$ 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.

The package has 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 use.
- **sag**(1G) command—displays system activity in a graphical form.
- **sadp**(1) 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—provides procedures 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 section "System Activity Counters." The section "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 the section "File Descriptions."

2. System Activity Counters

UniPlus$^+$ manages several 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*. The system table overflow counters are kept in the *_syserr* structure. The device activity counters are extracted from the device status tables. In this version, the I/O activity of the following devices is recorded: RP06, RM05, RS04, RF11, RK05, RP03, RL02, TM03, and TM11.

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), one of the *cpu[]* counters is incremented.

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

Bread and bwrite—The *bread* and *bwrite* counters 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 in to 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 occurred.

Iget, namei, and dirblk—These counters apply to file-access operations. *Iget* and *namei*, in particular, are the names of UniPlus$^+$ 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 inumber of a file corresponding to a special path. *Iget* is a routine called to locate the inode entry of a file (inumber). 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. 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 record queue activities. They are implemented in the *clock.c* routine. At every one-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—As to the I/O activity for a disk or tape device, 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.

3. 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.

3.1 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, you will want 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 use 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. See the section "Reporting Items," which gives the formula for deriving each reported item.

3.2 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**.

3.3 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).

3.4 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. This command is described in manual entry **sadp**(1).

4. 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.

4.1 Facilities

- **sadc**—The executable module of **sadc.c** (see "File Descriptions") which reads system counters from */dev/kmem* and records them to a file. In addition to the file argument, 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**.

4.2 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.

5. File Descriptions

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 use. 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.

6. The "Sysinfo" Structure

```
struct sysinfo   {
                  time_t        cpu[4];
#define          CPU_IDLE       0
#define          CPU_USER       1
#define          CPU_KERNEL     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;
    };
```

7. Reporting Items

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

7.1 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.

7.2 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.

7.3 Disk or Tape RgO 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-actswo] / I/O-ops;
avg-service-time = I/O-active / I/O-ops.
```

7.4 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.

7.5 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.

Chapter 11: UUCP ADMINISTRATION

CONTENTS

LIST OF FIGURES

Chapter 11
UUCP ADMINISTRATION

1. Introduction

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.

2. 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.

2.1 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.

2.2 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 hardwired 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.

2.3 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.

3. UUCP Software

Figure 11.1 (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 11.2 (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.

4. Installation

4.1 Object Modules

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

1. **uucp**—The file transfer command (bin/uucp).
2. **uux**—The remote execution command (bin/uux).
3. **uucico**—The **uucp** network daemon (usr/lib/uucp/...).
4. **uustat**—Network status command (bin/uustat).
5. **uuto**—Sends source files to destination (bin/uuto).
6. **uulog**—Queries a summary log of **uucp** and **uux** transactions (bin/uulog).

7. **uuname**—lists the **uucp** names of known systems (bin/uuname).
8. **uuclean**—Cleanup command (usr/lib/uucp/...).
9. **uusub**—The command for monitoring and creating a subnetwork (bin/uusub).
10. **uuxqt**—The remote execution daemon (usr/lib/uucp/...).
11. **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 (usr/lib/uucp/...).

4.2 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 a 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**.

4.3 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.
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.

4.3.1 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).

4.4 System File—"L.sys"

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	This is a string that indicates the days-of-week and times-of-day when the system should 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 of times (e.g., 0800−1230). If no time portion is specified, any time of day is assumed to be allowed for the call. 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.

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.

4.5 Dialing Prefixes—"L-dialcodes"

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.

4.6 Userfile

The *USERFILE* 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.)

4.7 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*.

5. 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.

5.1 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).

5.1.1 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
```

5.1.2 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.

5.1.3 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.

5.2 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.

5.3 Problems

The following sections list the most frequent problems that appear on systems that make heavy use of **uucp**(1).

5.3.1 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.

5.3.2 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.

5.3.3 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.

6. 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.

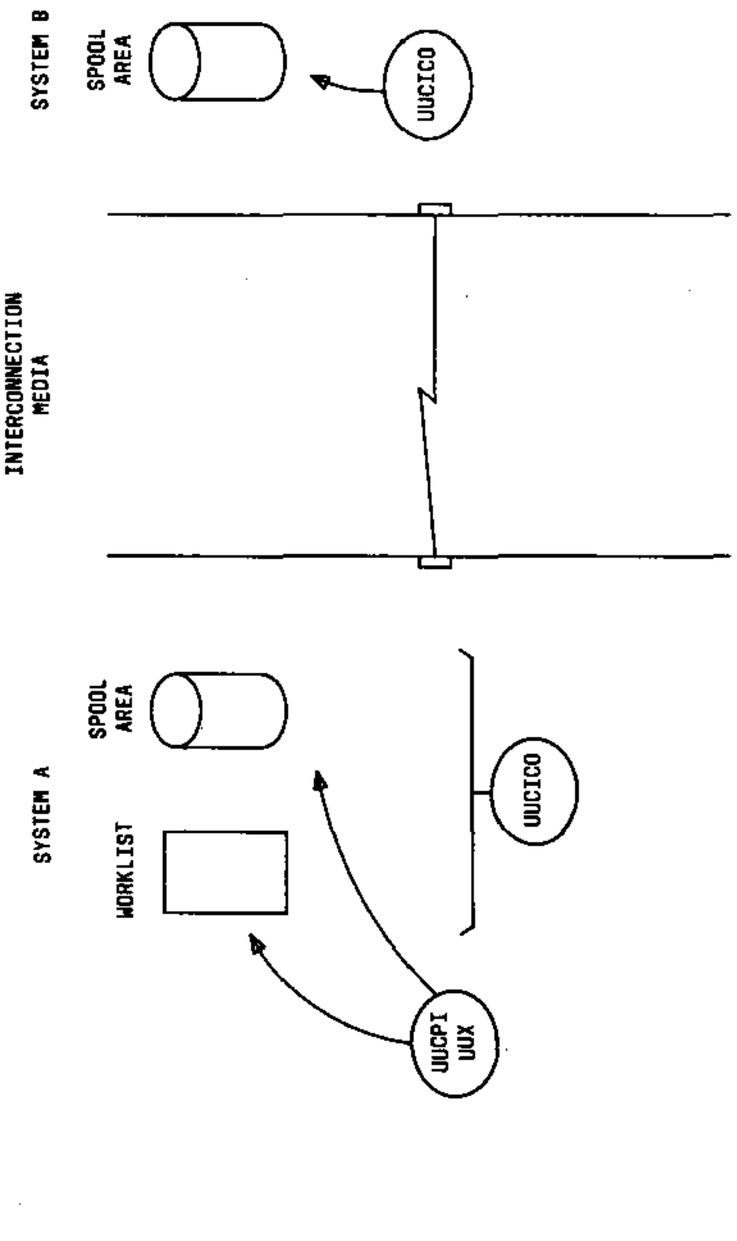

Figure 11.1 Uucp Network Daemon

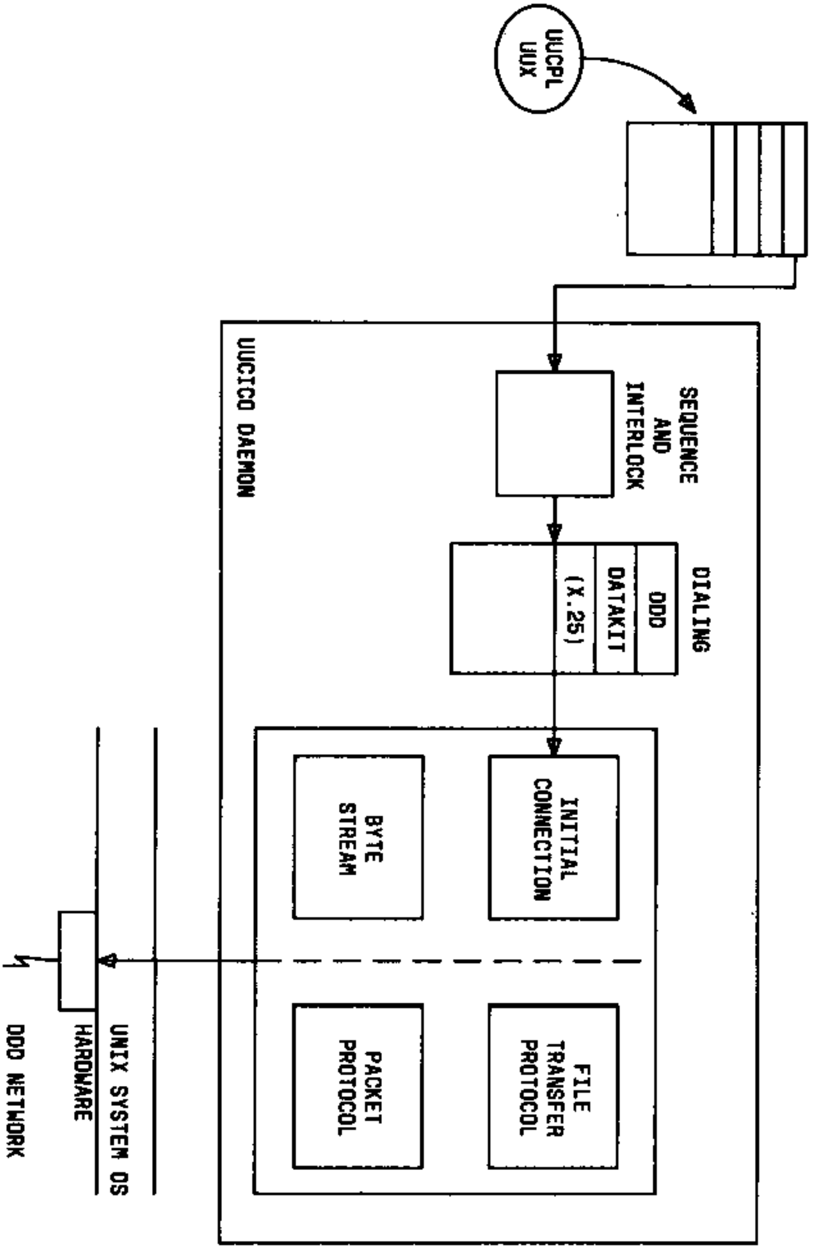

Figure 11.2 Uucico Daemon Functional Blocks

Chapter 12: TAKE/PUT: FILE TRANSFER SYSTEM

CONTENTS

Chapter 12

TAKE/PUT: FILE TRANSFER SYSTEM

1. Introduction

The **take/put** system is a group of programs and related files providing a rudimentary file transfer system for UniPlus$^+$. The system provides a mechanism for executing commands across machines as well as for transferring file and directory.

Like **uucp**(1C), the **take/put** interface assigns specific roles to each of two computers connected together. One is the master (or local machine), which initiates all transfers between computers. The other is the slave (or remote machine). The port on the master machine which connects to the slave *does*not have a login running (i.e., its line of **/etc/ttys** begins with a zero). The slave port *does* have a login running, although **take** and **put** cannot log in. The **cu**(1C) program must initially be used to log in on the slave machine.

Unlike **uucp**, the **take/put** system does not provide a mechanism for queueing transfers or for initiating transfers from the slave machine. Of course, two ports can be dedicated on both machines to provide two-way transfer initiation capabilities (discussed later), which has no mechanism for blocking access to the port while a transfer is in progress. Therefore, if a transfer is initiated while another is in progress, both will fail.

The system was designed to form a simple 'star' communication network between a more powerful computer (Vax 11/750) operating as the slave and several microcomputers acting as masters. Note that the single slave computer has one port dedicated to each microcomputer.

2. Necessary Files and Programs

2.1 Master Machine

take	Used to get files, directories, and the output of commands executed on the slave.
put	Used to transmit files, directories, and input to commands executed on the slave.

Both programs have a similar usage. If **−p** occurs as the first argument, then the second argument is the **port** to use in communicating with the slave. If no alternate port is specified, **/dev/tty0** is used.

The next argument may be **−sSPEED**, where **SPEED** is of **300**, **600**, **1200**, **2400**, **4800**, or **9600** and specifies the baud rate of the port.

The next argument may be **−c** indicating that the following argument is the name of the command to be executed on the slave machine. Any arguments following that command name are arguments to the command. If **−c** is not specified, then a **−i[id]** may be. This indicates that the specified filename should be remapped on the slave machine through a special table (described later). If the optional **id** is not specified, then the special file **/etc/sys_id** (on the master machine) is checked. If it exists, it contains a single word **id**.

Unless the **−c** option is specified, the filename to take or send (i.e., the input file) must be given. Whenever an input filename is given an output filename **may** be specified also, otherwise it defaults to the input filename. There is one special case: if the input file is a regular file and the output filename is a directory, then the output filename is formed by concatenating the last segment of the input filename (called the tail) with the directory specified as the output file.

If the input file is a directory, then **tar**(1) is used to convert it to and from a 'flat' file suitable for transmission or reception. The output file (whether explicitly or implicitly defined) must also be a directory. See the later section on Directory Transfers.

Note that when using **take**, the first filename specified (i.e., the input file) is on the slave (remote) machine and the second (optional) filename (i.e., the output file) is the local one. The situation is reversed when using **put**, since the first filename is the local file to be transmitted. A simple way to remember this is that the input file comes first, must always be specified, and must exist. The second filename (output file) is optional (defaulting to the input filename if unspecified) and may or may not exist.

2.2 Slave Machine

take7	Used to send files, directories, and the results of commands to the **take** program running on the master machine.
put7	Used to receive files, directories and the input to commands from the the **put** program running on the master machine.

No port need be specified to **take7** or **put7** since they are executed when **take** or **put** 'types' the appropriate command line on the connecting port. They simply use their standard input, standard output, and standard error to communicate with **take** or **put**.

The arguments for **take7** and **put7** are similar to those for **take** and **put**. In the case of a command, the **−c** is passed along and the command is enclosed in quotes to form a single argument. The **−i** option is also passed.

In the case of a file transaction to the slave machine (i.e., a **put**), two additional arguments are generated. If the file being transmitted is a directory, then the first argument will be a **−d** so that **put7** can prepare to decode it. The next argument will be a **−mMODE**, where **MODE** is in decimal, and is the mode of the input file on the master machine. The output file's mode will be set to match.

Take7 does not have the **−d** or **−m** arguments since the mode is being sent the other way. This information is sent as part of the initial handshake with **take** and is discussed more thoroughly in the section on OPERATION.

3. Special Files

A special file **/etc/takelist** exists on the slave machine; used in conjunction with the **−i** option. It is used to **relocate** pathnames between the master and slave machines by concatenating the pathname passed from the special file with a directory **prefix**. The following describes the format expected in the special file.

The file consists of three parts: user identification, macro definitions, and access descriptions. The user identification section is a single line that specifies the login name of the 'owner' of take7 and put7. For more information on how this is used, see the section on Protection.

The macro definition section follows immediately. It consists of lines of the form **name=string**. The **name** is any string of upper and lower case letters and numbers and defines the macro name. The **string** is any alphanumeric string. In section three of the file, all occurrences of a dollar sign followed by that macro name will be replaced with that string.

Each line in the third section of the file contains a series of fields separated by colons. The first field is a series of **system identification** names (login names and system ids) separated by 'or' bars (or symbol). Each additional field contains one or more directory prefixes also separated by 'or' bars.

If the **−i** option is used, **take7** or **put7** find the appropriate line of the special file and search each directory for the filename specified.

Both **take7** and **put7** find the appropriate starting line in the same way: if a **system identification** does not appear concatenated after the **−i** flag, then the user name (as determined by **getpwnam**(3S)) is used instead. This name is looked up in the special file and that line is the starting place for the search.

Note that each colon-separated field of the line contains a list of directory names (separated by 'or' bars). In the case of a **take7**, each colon separated field is used left to right until the file specified is found in one or more of the directories listed in that field. If the file exists in more than one of the directories listed, then **take7** exits with an appropriate error message. If exactly one occurrence is found, that file is transmitted to the master machine and any remaining fields are unchecked. If the field being searched contains a single string (i.e., no 'or' bars) and starts with the 'at' sign, then the string is considered to be an **id** and the search is continued at the line for that **id**. In this case the rest of the current line is ignored by **take7**.

In the case of **put7**, no duplication or re-reference (via usage of the 'at' sign) is allowed. The file must occur in exactly one of the directories mentioned on the line. The 'at' sign 'goto' is simply ignored as is the distinction between colon-separated fields and their 'or' bar delimited subfields.

For example, consider the special file sample below:

```
usr68
M=/micro
L=/languages
M3=/usr/m3
ma|macha:$M/ma|/generic:@type1:$M/ma.local
mb|machb:$M/mb|/generic|$L/fortran:@type2
type1:$M/reloc.10000|$M/68mmu
type2:$M/reloc.20000|$M3/xmmu1.20000
```

A command of the form:

```
take -ima /bin/echo /tmp/echo
```

executed on the master machine would invoke:

```
take7 -ima /bin/echo
```

Take7 would look first in '/micro/ma/bin/echo' and '/generic/bin/echo'. If the file is found in both places, an error would result. If it is found in one or the other, then that copy of 'echo' would be transmitted. Otherwise the line for 'type1' would be examined and the directories '/micro/reloc.10000' and '/micro/68mmu' would be searched for 'bin/echo'. Again, if it exists in both places, an error is generated and **take7** terminates; if either one or the other exists, it is transmitted. If neither exists, since there are no more directories or gotos, **take7** would terminate with an error message.

A command of the form:

```
put -ima echo /bin/echo
```

would invoke

```
put7 −mMODE −ima /bin/echo
```

on the slave machine. **Put7** would look in '/micro/ma', '/generic', and '/micro/ma.local' for copies of 'bin/echo'. If it were found in exactly one of these three directories, that file would be replaced. Otherwise the appropriate error message would be generated.

In addition to the differences in the way **take7** and **put7** use the special **/etc/takelist** file, there is another distinction between the two programs. **Take7** has a special user name, specified by the first line of the **/etc/takelist** file. This is so that **take7** can be made set-uid (see **chmod**(1)).

To selectively control access to files through **take7**, all the files should be accessible only through that user id. When someone executes **take7**, he must supply an **−i** option. If no id follows the **−i** option, the user's login name will be used. **Take7** will look up this id and use it to find the appropriate line of the special control file. If this line contains the user's login name, the user is allowed full access to all directories specified on the line. Otherwise, **take7** does a **setuid**(2) to the invoker's real user id. Thus the superuser and the controlling user may reference any line in the special control file. Other users can access only lines that contain their login name.

4. Trouble Shooting

The following sections cover the details of operation for the four programs. These sections should be helpful to users in determining what happened when something goes wrong, as well as an aid to those maintaining the programs. Here is a list of things to check/do in determining why **take/put** doesn't work.

1. Use the **cu**(1C) program (on the master machine) to check out the port. Make sure you are logged in, that the **take7** or **put7** program exists by executing it without arguments. The response should be

   ```
   **bad usage
   ```

   in both cases. If lots of characters are lost try dropping the baud rate (at both ends).

2. If directories or remote commands (i.e., the **−c** option) are failing it may be because **csh**(1) or **sh**(1) are missing from **/bin** or not executable. **Tar**(1) must exist on **both** machines for directory transfers to work.

5. Operation

After determining what files are to be used in the transfer, the master and slave programs must condition the port and synchronize with each other. Echoing is disabled on both sides and the erase and kill characters in effect are checked for conflict with the character set used for transfer.

Each block of data is broken into four 125 byte chunks. To each is added a character count (to determine the short last block) and a checksum byte. The chunks are converted into base 64 (three bytes of data get converted into four characters) and the constructed line is terminated by a line feed. The base 64 character set (in ascending order) is **a-z A-Z 0-9< >**.

Each time a chunk is sent or received from the master machine, a '.' is printed on the standard error if the chunk was error free. Otherwise an 'R' is printed and the chunk is retransmitted. Whenever a retransmission is performed, the program doing the transmitting sleeps one second for each time it has had to retransmit the current chunk. Thus the **take7** program sleeps one second the first time it has to retransmit a chunk, two seconds if it has to retransmit that same block a second time, etc. This sleeping continues until 10 retrys have failed; at this point the transfer fails. The **put** program operates similarly.

In the event that a transfer in progress fails and a simple file (i.e., not a directory) was being created on the master or slave machine, the file is removed. The one exception is a 'put −s' since, in order to replace a file via the **−s** option of **put7**, the file must have existed in the first place. Therefore, if removed because of a transfer error, all future references through the **−s** remapping option would fail.

When the **−c** option is used (i.e., a command is executed on the slave machine with its input or output from the master), the returned status of the **take** or **put** command is the returned status of the remote command if the input or output transfer didn't fail. An error message is printed if the transfer fails. Note that messages printed on the standard error by the remote command are printed on the standard error of **take** or **put**.

6. Directory Transfers

When the file to be transferred is a directory, **tar**(1) is used to convert the tree being sent into a flat file. In all cases the tar command is invoked from the directory to be transferred. Thus if **take** is used to get a directory, that directory is created first, if necessary. Similarly, if **put** is used to send a directory, the directory will be created if necessary.

7. Two Way Operation

In order for transfers to be initiated from either of two machines, two ports must be dedicated. In addition, all four programs, **take**, **put**, **take7**, and **put7**, must exist on both machines. In order to avoid having to always specify the port via the **−p** option, **/dev/tty0** can be the name of the master side of both ports. Remember that the master does not have a login process running, (i.e., its line of /etc/ttys begins with a zero), and that the slave end of each port can be called anything.

Chapter 13: UNIX I/O SYSTEM

CONTENTS

Chapter 13
UNIX I/O SYSTEM

This chapter is an overview of the UNIX I/O system. It guides writers of device driver routines, and therefore focuses on the environment and nature of device drivers, rather than the implementation of that part of the file system dealing with ordinary files. We assume that the reader has a good knowledge of the overall structure of the file system.

This chapter was updated and revised in 1984 by UniSoft Systems to reflect additions to the UniPlus^{+} kernel for System V.

1. Device Classes

There are two classes of device: *block* and *character*. The block interface is for devices, like disks and tapes, which can work with addressable 512-byte blocks. Ordinary magnetic tape only fits in this category because it can read any block using forward and backward spacing. Block devices can potentially contain a mounted file system. The interface to block devices is highly structured; the drivers for these devices share a great many routines as well as a pool of buffers.

Character-type devices have a much more straightforward interface, although the driver itself must do more work.

Both types of devices are named by a *major* and a *minor* device number. These numbers are generally stored as an integer. The minor device number is in the low-order 8 bits and the major device number is in the next-higher 8 bits. The *major* and *minor* macros access these numbers. The major device number selects which driver deals with the device; the minor device number is not used by the rest of system but is passed to the driver at appropriate times. Typically, the minor number selects a subdevice attached to a given controller, or one of several similar hardware interfaces.

The major device numbers for block and character devices are used as indices in separate tables; they both start at 0 and therefore overlap.

2. Overview of I/O

The *open* and *creat* system calls set up entries in three separate system tables. The first is the *u_ofile* table, stored in the system's per-process data area, *u*. This table is indexed by the file descriptors returned by the *open* or *creat*, and is accessed during a *read*, *write*, or other operation on the open file. Each entry is a pointer to the corresponding entry in the *file* table, which is a per-system data base. There is one entry in the *file* table for each *open* or *creat*. This table is per-system because the same instance of an open file must be shared among the several processes which can result from **forks** after the file is opened. A *file* table entry contains flags indicating whether the file was open for reading or writing, and a count which is used to determine when all processes using the entry have terminated or closed the file (so the entry can be abandoned). There is also a 32-bit file offset which indicates where in the file the next read or write takes place. Finally, there is a pointer to the entry for the file in the *inode* table, which contains a copy of the file's inode.

An entry in the *file* table corresponds to an instance of *open* or *creat*; if the same file is opened several times, it will have several entries in this table. However, there is only one entry in the *inode* table for a file. Also, a file may enter the *inode* table not only because it is open, but also because it is the current directory of some process or because it is a special file containing a currently-mounted file system.

An entry in the *inode* table differs somewhat from the corresponding inode stored on the disk—the modified and accessed times are not stored, and a flag word containing information about the entry is added. This flag word contains a count used to determine when it may be allowed to disappear, and the device and inumber the entry came from. Also, the several block numbers that give addressing information for the file are expanded from the 3-byte, compressed format to full *long* quantities.

During the processing of an *open* or *creat* call for a special file, the system always calls the device's *open* routine to allow for any special processing (rewinding a tape, turning on the data-terminal-ready lead of a modem, etc.) However, the *close* routine is called only when the last process closes a file; that is, when the inode table entry is being deallocated. Thus, it is not feasible for a device to maintain or depend on a count of its users, although it is quite possible to implement an exclusive-use device which cannot be reopened until it has been closed.

When a *read* or *write* takes place, the user's arguments and the *file* table entry are used to set up the variables *u.u_base*, *u.u_count*, and *u.u_offset*. These arguments respectively contain: the (user) address of the I/O target area, the byte-count for the transfer, and the current location in the file. If the file referred to is a character-type special file, the appropriate read or write routine is called. This routine is responsible for transferring data and updating the count and current location appropriately, as discussed below. Otherwise, the current location is used to calculate a logical block number in the file. If the file is an ordinary file, the logical block number must be mapped (possibly using indirect blocks) to a physical block number; a block-type special file need not be mapped. The *bmap* routine performs this mapping. The resulting physical block number is used (as discussed below) to read or write the appropriate device.

3. Character Device Drivers

The *cdevsw* table specifies the interface routines for character devices. Each device provides five routines: open, close, read, write, and special-function (to implement the *ioctl* system call). Any of these may be missing. If a call on the routine should be ignored (e.g., *open* on non-exclusive devices that require no setup), the *cdevsw* entry can be *nulldev*. If a call on a routine should be considered an error (e.g., *write* on read-only devices) use *nodev*. For terminals, the *cdevsw* structure also contains a pointer to the *tty* structure associated with the terminal.

The *open* routine is called each time the file is opened with the full device number as argument. The second argument is a flag which is non-zero only if the device is to be written on.

The *close* routine is called only when the file is closed for the last time. That is, when the last process closes the file. This means that it is not possible for the driver to maintain its own count of its users. The first argument is the device number; the second is a flag which is non-zero if the file was open for writing in the process which closes it.

When *write* is called, it is supplied the device as argument. The per-user variable *u.u_count* has been set to the number of characters indicated by the user; for character devices, this number may be 0 initially. *u.u_base* is the address, supplied by the user, from which to start taking characters. The system may call the routine internally, For this reason, the flag *u.u_segflg* indicates, if *on*, that *u.u_base* refers to the system address space instead of the user's.

The *write* routine copies up to *u.u_count* characters from the user's buffer to the device, decrementing *u.u_count* for each character passed. For most drivers (which work one character at a time) the routine *cpass* () picks up characters from the user's buffer. Successive calls on it return the characters to be written, until *u.u_count* goes to 0 or an error occurs (when it returns −1). *Cpass* updates *u.u_count*.

Write routines which transfer a large number of characters into an internal buffer may also use the routine *iomove* (*buffer*, *offset*, *count*, *flag*). This routine is faster when moving many characters. *Iomove* transfers up to *count* characters into the *buffer* starting *offset* bytes from the start of the buffer; *flag* should be B_WRITE (which is 0) in the write case. Caution: You are responsible for making sure the count is not too large or non-zero. *Iomove* is much slower if *buffer+ offset*, *count*, or *u.u_base* is odd.

The device's *read* routine is called under conditions similar to *write*, except that *u.u_count* is non-zero. The routine *pass(c)* returns characters to the user. It takes care of housekeeping, like *cpass*, and returns −1 when the last character specified by *u.u_count* is returned to the user. Before that, it returns 0. You can also use *iomove* as you do with *write* —the flag should be B_READ but the same cautions apply.

The "special functions" routine is invoked by the ioctl system call:

```
(*p) (dev,cmd,arg,mode)
```

where *p* is a pointer to the address of the device, *dev* is the device number, *cmd* is the user ioctl command argument, *arg* is the user argument, and *mode* is the file table flag word for the opened device

Finally, each device should have appropriate interrupt routines. When an interrupt occurs, it is turned into a C-compatible call to the device's interrupt routine. The interrupt-catching mechanism makes 16 bits of data available to the interrupt handler in *a-dev* (see <*include/sys/reg.h*>). This is conventionally used by drivers dealing with multiple similar devices to encode the minor device number.

Several subroutines are available for character device drivers. For example, most of these handlers need a place to buffer characters in the internal interface between their "top half" (read/write) and "bottom half" (interrupt) routines. For relatively low data-rate devices, the best mechanism is the character queue maintained by the routines *getc* and *putc*. A queue header has the structure:

```
struct clist    {
       int      c_cc;      /* character count */
struct cblock  *c_cf;      /* pointer to first */
struct cblock  *c_cl;      /* pointer to last  */
}
```

Putc places a character on the end of a queue (*c. &queue*) where *c* is the character and *queue* is a clist structure. The routine returns −1 if there is no space to put the character. Otherwise, it returns 0. *Getc* may retrieve the first character on the queue (*&queue*). This returns either the (non-negative) character or −1 (if the queue is empty).

The space for characters in queues is shared among all devices in the system. In the standard system there are only 600 character slots available. Thus, device handlers, especially write routines, must avoid gobbling up excessive numbers of characters.

The other major help available to device handlers is the sleep-wakeup mechanism. The call *sleep*(*event*, *priority*) makes the process wait (allowing other processes to run) until the *event* occurs. When the *event* occurs, the process is marked ready-to-run and the call returns when there is no process with higher *priority*.

The call *wakeup*(*event*) indicates that the *event* has happened, causing processes sleeping on the event to wake up. The *event* is arbitrary—agreed upon by the sleeper and the waker-up. By convention, it is the address of some data area used by the driver. This guarantees that events are unique.

Processes sleeping on an event should not assume that the event has really happened. They should check that the conditions which caused them to sleep are no longer true.

Priorities range from 0 to 127. A larger number indicates less-favored scheduling. There is a distinction between processes sleeping at a priority less than the parameter PZERO, and those sleeping at a priority greater than PZERO. The former cannot be interrupted by signals, although it is conceivable that it may be swapped out. For this reason it is a bad idea to sleep with priority less than PZERO on an event which might never occur. On the other hand, calls to *sleep* with larger priority may never return if the process is terminated by some signal in the meantime. Incidentally, it is a gross error to call *sleep* in a routine called at interrupt time, since the process which is running is almost certainly not the process which should go to sleep. Likewise, none of the variables in the user area "*u*." should be touched, let alone changed, by an interrupt routine.

If a device driver wishes to wait for some event for which it is inconvenient or impossible to supply a *wakeup* (for example, a device going on-line, which does not generally cause an interrupt), the call *sleep*(*&bolt*, *priority*) may be given. *Lbolt* is an external cell whose address is awakened once every second by the clock interrupt routine.

The routines *spl4*(), *spl5*(), *spl6*(), *spl7*() set the processor priority level as indicated to avoid inconvenient interrupts from the device.

Timeout(*func*,*arg*,*interval*) is useful if a device needs to know about real-time intervals. After *interval* sixtieths of a second, *func* is called with *arg* as argument, in the style (**func*)(*arg*). Timeouts provide real-time delays after function characters (like new-line and tab) in typewriter output and terminate an attempt to read the 201 Dataphone (dp) if there is no response within a specified number of seconds. Notice that the number of sixtieths of a second is limited to 2**31−1, since it must appear to be positive, and that only a bounded number of timeouts can be going on at once. Also, the specified *func* is called at clock-interrupt time, so it should conform to the requirements of interrupt routines in general.

4. The Block-Device Interface

Handling block devices is mediated by a collection of routines. These routines manage a set of buffers containing the images of blocks of data on the various devices. These routines assure that several processes accessing the same block of the same device in multiprogrammed fashion maintain a consistent view of the data in the block. A secondary but still important purpose is increasing the efficiency of the system by keeping in-core copies of blocks that are accessed frequently. The main data base for this mechanism is the table of buffers, *buf*. Each buffer header contains

- A pair of pointers (*b_forw*, *b_back*) maintaining a doubly-linked list of the buffers associated with a particular block device.

- A pair of pointers (*av_forw*, *av_back*) maintaining a doubly-linked list of "free" blocks (blocks which can be reallocated for another transaction). Buffers that have I/O in progress or are busy for other purposes do not appear in this list.
- The device and block number to which the buffer refers.
- A pointer to the actual storage associated with the buffer.
- A word count (the number of bytes to be transferred to or from the buffer).
- An error byte and a residual byte count to communicate information from an I/O routine to its caller.
- A flag word with bits indicating the status of the buffer. These flags are discussed below.

The interface with the rest of the system is primarily made up to seven routines. Both *bread* and *getblk* return a pointer to a buffer header for the block when given a device and a block number. The difference is that *bread* returns a buffer containing the current data for the block, while *getblk* returns a buffer containing the data in the block only if it is already in core (this is indicated by the *B_DONE* bit; see below). In either case, the buffer (and the corresponding device block) is "busy." Other processes referring to it have to wait until it becomes free. For example, *getblk* can be used when a block is about to be totally rewritten—no other process can refer to the block until the new data is placed in it.

The *breada* routine implements read-ahead. It is logically similar to *bread*, but takes an additional argument—the block number of a block (on the same device) to read asynchronously after the specifically requested block is available.

The *brelse* routine makes the buffer available to other processes when given a pointer to a buffer. It is called, for example, after data is extracted following a *bread*. There are three subtly different write routines, all of which take a buffer pointer as argument, and all of which logically release the buffer for use by others and place it on the free list.

- *Bwrite* puts the buffer on the appropriate device queue, waits for the write, and sets the user's error flag, if required.
- *Bawrite* places the buffer on the device's queue, but does not wait for completion. For this reason, errors are not reflected directly to the user.
- *Bdwrite* does not start any I/O operation at all, but marks the buffer so that, if it is grabbed from the free list to contain data from some other block, the data in it will first be written out.

Use *bwrite* when you want to be sure that I/O takes place correctly, and that errors are reflected to the proper user— for example, when updating inodes. Use *bawrite* when you want more overlap (because no wait is required for I/O to finish) but when you are reasonably certain that the write is required. Use *bdwrite* when you are not sure that the write is needed at the moment. For example, *bdwrite* is called when the last byte of a *write* system call falls short of the end of a block, on the assumption that another *write* will be given soon which will re-use the same block. On the other hand, as the end of a block is passed, *bawrite* is called, since the block will probably not be accessed again soon and you want to start the writing process soon.

The routines *getblk* and *bread* dedicate the given block exclusively to the caller's use and make others wait. On the other hand, *brelse*, *bwrite*, *bawrite*, or *bdwrite* must eventually be called to free the block for use by others.

Each buffer header contains a flag word indicating the status of the buffer. Since they provide one important channel for information between the drivers and the block I/O system, it is important to understand these flags. The following names are manifest constants which select the associated flag bits.

B_READ	This bit is set when the buffer is handed to the device strategy routine (see below). It indicates a read operation. The symbol *B_WRITE* is defined as 0 and does not define a flag. It is a mnemonic convenience for callers of routines, like *swap*, which have a separate argument indicating read or write.
B_DONE	This bit is set to 0 when a block is handed to the device strategy routine and is turned on when the operation completes, whether normally or as the result of an error. It is also used as part of the return argument of *getblk*—if it is 1, it indicates that the returned buffer actually contains the data in the requested block.
B_ERROR	This bit may be set to 1 when *B_DONE* is set to indicate that an I/O or other error occurred. If it is set, the *b_error* byte of the buffer header may contain an error code. If *b_error* is 0, the error code is not specified. Actually, no driver at present sets *b_error*.
B_BUSY	This bit indicates that the buffer header is dedicated to someone's exclusive use. However, the buffer remains attached to the list of blocks associated with its device. When *getblk* (or *bread*, which calls it) searches the buffer list for a given device and finds the requested block with this bit on, it sleeps until the bit clears
B_PHYS	This bit is set for raw I/O transactions.
B_WANTED	This flag is used in conjunction with the *B_BUSY* bit. Before sleeping (described above), *getblk*

	sets this flag. Conversely, when the block is freed and the busy bit goes down (in *brelse*) a *wakeup* is given for the block header whenever *B_WANTED* is on. This avoids having to call *wakeup* every time a buffer is freed on the chance that someone might want it.
B_AGE	This bit may be set on buffers just before releasing them. If it is on, the buffer is placed at the head of the free list, rather than at the tail. It is a performance heuristic used when the caller decides that the same block will not soon be used again.
B_ASYNC	This bit is set by *bawrite*. It indicates to the appropriate device driver that the buffer should be released when the write is finished (usually at interrupt time). The difference between *bwrite* and *bawrite* is that *bwrite* starts I/O, waits until it is done, and frees the buffer. *Bawrite* sets this bit and starts I/O. The bit indicates that *brelse* should be called for the buffer on completion.
B_DELWRI	This bit is set by *bdwrite* before releasing the buffer. When *getblk* (while searching for a free block) discovers the bit is 1 in a buffer it would otherwise grab, it writes block out before reusing it.
B_STALE	This flag invalidates the association between the buffer and the device/block number. It is set when an error occurs or when the buffer is associated with a block on a file system that is unmounted.

5. Block Device Drivers

The *bdevsw* table contains the names of the interface routines and a table for each block device.

As with character devices, block device drivers may supply an *open* and a *close* routine, called respectively on each open and on the final close of the device. Instead of separate read and write routines, each block device driver has a *strategy* routine which is called with a pointer to a buffer header as argument. The buffer header contains a read/write flag, the core address, the block number, a byte count, and the major and minor device numbers. The strategy routine carries out the operation requested by the information in the buffer header. When the transaction is complete, the *B_DONE* (and possibly the *B_ERROR*) bits are set. If the *B_ASYNC* bit is set, *brelse* should be called; otherwise, *wakeup* is called. When the device is capable (under error-free operation) of transferring fewer words than requested, the device's word-count register should be placed in the residual count slot of the buffer header. Otherwise, the residual count should be set to 0. This is for the benefit of the magtape driver—it tells the user the actual length of the record.

Although the most usual argument of the strategy routines is a genuine buffer header allocated as discussed above, all that is actually required is that the argument be a pointer to a place containing the appropriate information. For example, the *swap* routine, which manages movement of core images to and from the swapping device, uses the strategy routine for this device. Care has to be taken that no extraneous bits get turned on in the flag word.

The device's table specified by *bdevsw* has a byte containing an active flag and an error count, a pair of links constituting the head of the chain of buffers for the device (*b_forw*, *b_back*), and a first and last pointer for a device queue. All of these are used solely by the device driver itself, except for the buffer-chain pointers. Typically, the flag encodes the state of the device, and is used at a minimum to indicate that the device is currently engaged in transferring information and no new command should be issued. The error count is useful for counting retries when errors occur. The device queue remembers stacked requests. In the simplest case, it may be maintained as a first-in first-out list. Since buffers which have been handed over to the strategy routines are never on the list of free buffers, the pointers in the buffer which maintain the free list (*av_forw*, *av_back*) are also used to contain the pointers which maintain the device queues.

A few routines are useful to block device drivers. *Iodone(bp)* arranges that the buffer to which *bp* points be released or awakened when the strategy module has finished with the buffer (either normally or after an error). (If after an error, the *B_ERROR* bit has presumably been set.)

The routine *geterror(bp)* can examine the error bit in a buffer header and reflect any error indication found there to the user. It may be called only in the non-interrupt part of a driver when I/O has completed (i.e., *B_DONE* has been set).

6. Raw Block-Device I/O

Block device drivers may be used to transfer information directly between the user's core image and the device without using buffers and in blocks as large as the caller requests. This involves setting up a character-type special file corresponding to the raw device and providing *read* and *write* routines. These routines set up what is usually a private, non-shared buffer header with the appropriate information and call the device's strategy routine. Separate *open* and *close* routines may be provided but this is usually unnecessary. A special-function routine might come in handy, especially for magtape.

A great deal of work has to be done to generate the "appropriate information" to put in the argument buffer for the strategy module. The worst part is mapping relocated user addresses to physical addresses. Most of this work is done by *physio*(*strat*, *bp*, *dev*, *rw*) whose arguments are: the name of the strategy routine *strat*; the buffer pointer *bp*; the device number *dev*; and a read-write flag *rw*, whose value is either *B_READ* or *B_WRITE*. *Physio* makes sure that the user's base address and count are even (because most devices work in words) and that the core area affected is contiguous in physical space. It delays until the buffer is not busy, and makes it busy while the operation is in progress, and it sets up user error return information.

Chapter 14: UNIX IMPLEMENTATION

CONTENTS

Chapter 14
UNIX IMPLEMENTATION

This chapter describes the implementation of the resident UNIX kernel. The first section is a brief introduction. The second section describes how the UNIX system views processes, users, and programs. The third section describes the I/O system. The last section describes the UNIX file system.

1. Introduction

The UNIX kernel consists of 20,000 lines of C code and 500 lines of assembly code. The assembly code can be further broken down into 200 lines included for efficiency (they could have been written in C) and 300 lines performing hardware functions not possible in C.

This code represents 5 to 10 percent of what has been called "the UNIX operating system." The kernel is the only UNIX code that cannot be changed by a user. For this reason, the kernel should make as few real decisions as possible. The user doesn't need a million options to do the same thing. Rather, there should be one way to do a thing, but that way should be the least-common divisor of all the options that might have been provided.

2. Process Control

In the UNIX system, a user executes programs in an environment called a user process. When a system function is required, the user process calls the system as a subroutine. At some point in this call, there is a distinct switch of environments. After this, the process is said to be a system process. In the normal definition of processes, the user and system processes are different phases of the same process (they never execute simultaneously). For protection, each system process has its own stack.

The user process may execute from a read-only text segment, shared by all processes executing the same code. There is no *functional* benefit from shared-text segments. An *efficiency* benefit is that there is no need to swap read-only segments out because the original copy on secondary memory is still current. This is a great benefit to interactive programs that tend to be swapped while waiting for terminal input. Furthermore, if two processes are executing simultaneously from the same copy of a read-only segment, only one copy needs to reside in primary memory. This is a secondary effect, because simultaneous execution of a program is not common. It is ironic that this effect, which reduces the use of primary memory, only comes into play when there is an overabundance of primary memory—that is, when there is enough memory to keep waiting processes loaded.

All current read-only text segments in the system are maintained from the *text*table. A text table entry holds the location of the text segment on secondary memory. If the segment is loaded, that table also holds the primary memory location and the number of processes sharing this entry. When this count is reduced to zero, the entry is freed along with any primary and secondary memory holding the segment. When a process first executes a shared-text segment, a text table entry is allocated and the segment is loaded onto secondary memory. If a second process executes a text segment that is already allocated, the entry reference count is simply incremented.

A user process has some strictly private read-write data in its data segment. As far as possible, the system does not use the user's data segment to hold system data. There are no I/O buffers in the user address space.

The user data segment has two growing boundaries. One, increased automatically by the system as a result of memory faults, is used for a stack. The second boundary is only grown (or shrunk) by explicit requests. The contents of newly allocated primary memory are initialized to zero.

Also associated and swapped with a process is a small, fixed-size system data segment. This segment contains all the data about the process that the system needs only when the process is active. Examples of the kind of data contained in the system data segment are: saved central processor registers, open file descriptors, accounting information, scratch data area, and the stack for the system phase of the process. The system data segment is not addressable from the user process and is therefore protected.

Last, there is a process table with one entry per process. This entry contains all the data needed by the system when the process is *not* active. Examples are the process's name, the location of the other segments, and scheduling information. The process table entry is allocated when the process is created, and freed when the process terminates. This process entry is always directly addressable by the kernel.

2.1 Process Creation and Program Execution

Processes are created by the system primitive **fork**. The newly created process (child) is a copy of the original process (parent). There is no detectable sharing of primary memory between the two processes. (Of course, if the parent process is executing from a read-only text segment, the child shares the text segment.) Copies of all writable data segments are made for the child process. Files that were open before the **fork** are shared after the **fork**. The processes are informed of their part in the relationship, allowing them to select their own (usually non-identical) destiny. The parent may **wait** for the termination of any of its children.

A process may **exec** a file. This consists of exchanging the current text and data segments of the process for new text and data segments specified in the file. The old segments are lost. Doing an **exec** does *not* change processes; the process that did the **exec** persists, but after the **exec** it is executing a different program. Files that were open before the **exec** remain open after the **exec**.

If a program (for example, the first pass of a compiler) wishes to overlay itself with another program (for example, the second pass) then it simply **execs** the second program. This is analogous to a "goto." If a program wishes to regain control after **execing** a second program, it should **fork** a child process, have the child **exec** the second program, and have the parent **wait** for the child. This is analogous to a "call." Breaking up the call into a binding followed by a transfer is similar to the subroutine linkage in SL-5.

2.2 Swapping

The major data associated with a process (the user data segment, the system data segment, and the text segment) are swapped to and from secondary memory, as needed. The user data segment and the system data segment are kept in primary memory to reduce swapping latency. (When using low-latency devices—such as bubbles, CCDs, or scatter/gather devices—this decision has to be reconsidered.) Allocation of both primary and secondary memory is performed by the same simple first-fit algorithm. When a process grows, a new piece of primary memory is allocated. The contents of the old memory are copied to the new memory. If necessary, the old memory is freed and the tables are updated. If there is not enough primary memory, secondary memory is allocated instead. The process is swapped out onto the secondary memory, ready to be swapped in with its new size.

One separate process in the kernel, the swapping process, simply swaps the other processes in and out of primary memory. It examines the process table looking for a process that is swapped out and is ready to run. It allocates primary memory for that process and reads its segments into primary memory, where that process competes for the central processor with other loaded processes. If no primary memory is available, the swapping process makes memory available by examining the process table for processes that can be swapped out. It selects a process to swap out, writes it to secondary memory, frees the primary memory, and then goes back to look for a process to swap in.

Thus there are two specific algorithms to the swapping process. Which of the possibly many processes that are swapped out is to be swapped in? This is decided by secondary storage residence time. The one with the longest time out is swapped in first. There is a slight penalty for larger processes. Which of the possibly many processes that are loaded is to be swapped out? Processes that are waiting for slow events (i.e., not currently running or waiting for disk I/O) are picked first, by age in primary memory, again with size penalties. The other processes are examined by the same age algorithm, but are not taken out unless they are at least of some age. This adds hysteresis to the swapping and prevents total thrashing.

These swapping algorithms are the most suspect in the system. With limited primary memory, these algorithms cause total swapping. This is not bad in itself, because the swapping does not impact the execution of the resident processes. However, if the swapping device must also be used for file storage, the swapping traffic severly impacts the file system traffic. It is exactly these small systems that tend to double the use of limited disk resources.

2.3 Synchronization and Scheduling

Process synchronization is accomplished by having processes wait for events. Events are represented by arbitrary integers. By convention, events are chosen to be addresses of tables associated with those events. For example, a process that is waiting for any of its children to terminate will wait for an event that is the address of its own process table entry. When a process terminates, it signals the event represented by its parent's process table entry. Signaling an event on which no process is waiting has no effect. Similarly, signaling an event on which many processes are waiting will wake all of them up. This differs considerably from Dijkstra's P and V synchronization operations, in that no memory is associated with events. Thus, there need be no allocation of events prior to their use. Events exist simply by being used.

On the negative side, because there is no memory associated with events, no notion of "how much" can be signaled via the event mechanism. For example, processes that want memory might wait on an event associated with memory allocation. When any amount of memory becomes available, the event would be signaled. All the competing processes would then wake up to fight over the new memory. (In reality, the swapping process is the only process that waits for primary memory to become available.)

If an event occurs between the time a process decides to wait for that event and the time that process enters the wait state, then the process will wait on an event that has already happened (and may never happen again). This race condition happens because there is no memory associated with the event to indicate that the event has occurred; the only action of an event is to change a set of processes from wait state to run state. This problem is relieved largely by the fact that process switching can only occur in the kernel by explicit calls to the event-wait mechanism. If the event in question is signaled by another process, then there is no problem. But if the event is signaled by a hardware interrupt, then special care must be taken. These synchronization races pose the biggest problem when UNIX is adapted to multiple-processor configurations.

The event-wait code in the kernel is like a co-routine linkage. At any time, all but one of the processes has called event-wait. The remaining process is the one currently executing. When it calls event-wait, a process whose event has been signaled is selected and that process returns from its call to event-wait.

Which of the runable processes is to run next? Each process is associated with a priority. The priority of a system process is assigned by the code issuing the wait on an event. This is roughly equivalent to the response that one would expect on such an event. Disk events have high priority, teletype events are low, and time-of-day events are very low. (From observation, the difference in system process priorities has little or no performance impact.) All user-process priorities are lower than the lowest system priority. User-process priorities are assigned by an algorithm based on the recent ratio of the amount of compute time to real time consumed by the process. A process that used a lot of compute time in the last real-time unit is assigned a low user priority. Because interactive processes are characterized by low ratios of compute to real time, interactive response is maintained without any special arrangements.

The scheduling algorithm simply picks the process with the highest priority, thus picking all system processes first and user processes second. The compute-to-real-time ratio is updated every second. Thus, all other things being equal, looping user processes are scheduled round-robin with a 1-second quantum. A high-priority process waking up will preempt a running, low-priority process. The scheduling algorithm has a desirable negative feedback character. If a process uses its high priority to hog the computer, its priority drops. At the same time, if a low-priority process is ignored for a long time, its priority rises.

3. I/O System

The I/O system is broken into two completely separate systems; the block I/O system and the character I/O system. In retrospect, the names should have been "structured I/O" and "unstructured I/O," respectively. While the term "block I/O" has some meaning, "character I/O" is a complete misnomer.

Devices are characterized by a major device number, a minor device number, and a class (block or character). For each class, there is an array of entry points into the device drivers. The major device number is used to index the array when calling the code for a particular device driver. The minor device number is passed to the device driver as an argument. The minor number has no significance other than that attributed to it by the driver. Usually, the driver uses the minor number to access one of several identical physical devices.

Using the array of entry points (configuration table) as the only connection between the system code and the device drivers is important. Early versions of the system had a much less formal connection with the drivers, making it extremely hard to handcraft differently configured systems. Now it is possible to create new device drivers in an average of a few hours. The configuration table, in most cases, is created automatically by a program that reads the system parts list.

3.1 Block I/O System

The model block I/O device consists of randomly addressed, secondary memory blocks of 512 or 1024 bytes each. The blocks are uniformly addressed 0, 1,... up to the size of the device. The block device driver emulates this model on a physical device.

The block I/O devices are accessed through a layer of buffering software. The system maintains a list of buffers (typically between 10 and 70) each assigned a device name and a device address. This buffer pool constitutes a data cache for the block devices. On a read request, the cache is searched for the desired block. If the block is found, the data are made available to the requester without any physical I/O. If the block is not in the cache, the least recently used block in the cache is renamed, the correct device driver is called to fill up the renamed buffer, and then the data are made available. Write requests are handled in an analogous manner. The correct buffer is found and relabeled, if necessary. The write is performed simply by marking the buffer as "dirty." The physical I/O is then deferred until the buffer is renamed.

The benefits in reduction of physical I/O of this scheme are substantial, especially considering the file system implementation. There are, however, some drawbacks. The asynchronous nature of the algorithm makes error reporting and meaningful user error handling almost impossible. The cavalier approach to I/O error handling in the UNIX system is partly due to the asynchronous nature of the block I/O system. A second problem is in the delayed writes. If the system stops unexpectedly, it is almost certain that there is a lot of logically complete, but physically incomplete, I/O in the buffers. There is a system primitive to flush all outstanding I/O activity from the buffers. Periodic use of this primitive helps, but does not solve, the problem. Finally, the associativity in the buffers can alter the physical I/O sequence from that of the logical I/O sequence. This means that there are times when data structures on disk are inconsistent, even though the software is careful to perform I/O in the correct order. On non-random devices, notably magnetic tape, the inversions of writes can be disastrous. The problem with magnetic tapes is "cured" by allowing only one outstanding write request per drive.

3.2 Character I/O System

The character I/O system consists of all devices that do not fall into the block I/O model. This includes the "classical" character devices—such as communication lines, paper tape, and line printers. It also includes magnetic tape and disks when they are not used in a stereotyped way (for example, 80-byte physical records on tape and track-at-a-time disk copies). In short, the character I/O interface means "everything other than block." I/O requests from the user are sent to the device driver essentially unaltered. The implementation of these requests is, of course, up to the device driver. There are guidelines and conventions to help the implementation of certain types of device drivers.

3.2.1 Disk Drivers

Disk drivers are implemented with a queue of transaction records. Each record holds a read/write flag, a primary memory address, a secondary memory address, and a transfer byte count. Swapping is accomplished by passing a record to the swapping device driver. The block I/O interface is implemented by passing such records with requests to fill and empty system buffers. The character I/O interface to the disk drivers create a transaction record that points directly into the user area. The routine that creates this record also ensures that the user is not swapped during this I/O transaction. Thus, by implementing the general disk driver, it is possible to use the disk as a block device, a character device, and a swap device. The only really disk-specific code in normal disk drivers is the pre-sort of transactions to minimize latency for a particular device, and the actual issuing of the I/O request.

3.2.2 Character Lists

Real character-oriented devices may be implemented using the common code to handle character lists. A character list is a queue of characters. One routine puts a character on a queue. Another gets a character from a queue. It is also possible to ask how many characters are currently on a queue. Storage for all queues in the system comes from a single common pool. Putting a character on a queue allocates space from the common pool and links the character onto the data structure defining the queue. Getting a character from a queue returns the corresponding space to the pool.

A typical character-output device (paper tape punch, for example) is implemented by passing characters from the user onto a character queue until some maximum number of characters is on the queue. The I/O is prodded to start as soon as there is anything on the queue and, once started, it is sustained by hardware completion interrupts. Each time there is a completion interrupt, the driver gets the next character from the queue and sends it to the hardware. The number of characters on the queue is checked and, as the count falls through some intermediate level, an event (the queue address) is signaled. The process that is passing characters from the user to the queue can be waiting on the event, and refill the queue to its maximum when the event occurs.

A typical character input device (for example, a paper tape reader) is handled in a very similar manner.

Another class of character devices is terminals. A terminal is represented by three character queues. There are two input queues (raw and canonical) and an output queue. Characters going to the output of a terminal are handled by common code exactly as described above. The main difference is that there is also code to interpret the output stream as ASCII characters and to perform some translations, e.g., escapes for deficient terminals. Another common aspect of terminals is code to insert real-time delay after certain control characters.

Input on terminals is a little different. Characters are collected from the terminal and placed on a raw input queue. Some device-dependent code conversion and escape interpretation is handled here. When a line is complete in the raw queue, an event is signaled. The code catching this signal then copies a line from the raw queue to a canonical queue performing the character erase and line kill editing. User read requests on terminals can be directed at either the raw or canonical queues.

3.2.3 Other Character Devices

Finally, there are devices that fit no general category. These devices are set up as character I/O drivers. An example is a driver that reads and writes unmapped primary memory as an I/O device. Some devices are too fast to be treated a character at a time, but do not fit the disk I/O mold. Example are fast communications lines and fast line printers. These devices either have their own buffers or "borrow" block I/O buffers for a while and then give them back.

4. The File System

In the UNIX system, a file is a (one-dimensional) array of bytes. No other structure of files is implied by the system. Files are attached anywhere (and possibly multiply) onto a hierarchy of directories. Directories are simply files that users cannot write.

The UNIX file system is a disk data structure accessed completely through the block I/O system. As stated before, the canonical view of a "disk" is a randomly addressable array of 512-byte or 1024-byte blocks. A file system breaks the disk into four self-identifying regions. The first block (address 0) is unused by the file system. It is left aside for booting procedures. The second block (address 1) contains the so-called "super-block." This block, among other things, contains the size of the disk and the boundaries of the other regions. Next comes the ilist, a list of file definitions. Each file definition is a 64-byte structure, called an inode. The offset of a particular inode within the ilist is called its inumber. The combination of device name (major and minor numbers) and inumbers uniquely names a particular file. After the ilist, and at the end of the disk, are free storage blocks available for the contents of files.

The free space on a disk is maintained by a linked list of available disk blocks. Every block in this chain contains a disk address of the next block in the chain. The remaining space contains the address of up to 50 disk blocks that are also free. Thus with one I/O operation, the system obtains 50 free blocks and a pointer showing where to find more. The disk allocation algorithms are straightforward. Since all allocation is in fixed-size blocks and there is strict accounting of space, there is no need to compact or garbage collect. However, as disk space becomes dispersed, latency gradually increases. Some installations choose to occasionally compact disk space to reduce latency.

An inode contains 13 disk addresses. The first 10 of these addresses point directly at the first 10 blocks of a file. If a file is larger than 10 blocks, then the eleventh address points at a block that contains the addresses of the next 128 blocks of the file. If the file is still larger than this, then the twelfth block points at up the 128 blocks, each pointing to 128 blocks of the file. Files yet larger use the thirteenth address for a "triple indirect" address. The algorithm ends here with the maximum file size of 1,082,201,087 bytes for a 512-byte file system, or 2,164,402,175 bytes for a 1024-byte file system.

A logical directory hierarchy is added to this flat physical structure simply by adding a new type of file—the directory. A directory is accessed exactly as an ordinary file. It contains 16-byte entries consisting of a 14-byte name and an inumber. The root of the hierarchy is at a known inumber (*viz.*, 2). The file system structure allows an arbitrary, directed graph of directories with regular files linked in at arbitrary places in this graph. In fact, very early UNIX systems used such a structure. Administration of this structure became so chaotic that later systems were restricted to a directory tree. Even now, with regular files linked multiply into arbitrary places in the tree, accounting for space is a problem. It may be necessary to restrict the entire structure to a tree, and allow a new form of linking that is subservient to the tree structure.

The file system allows easy creation, easy removal, easy random accessing, and very easy space allocation. With most physical addresses confined to a small contiguous section of disk, it is also easy to dump, restore, and check the consistency of the file system. Large files suffer from indirect addressing, but the cache prevents most of the implied physical I/O without adding much execution. The space overhead properties of this scheme are quite good. For example, on one particular file system, there are 25,000 files containing 130M bytes of date-file content. The overhead (inode, indirect blocks, and last block breakage) is about 11.5M. The directory structure supporting these files has about 1,500 directories containing 0.6M bytes of directory content and about 0.5M bytes of overhead in accessing the directories. This comes out to less than a 10 percent overhead for actual stored data. Most systems have this much overhead in padded trailing blanks alone.

4.1 File System Implementation

Because the inode defines a file, the implementation of the file system centers around access to the inodes. The system maintains a table of all active inodes. As a new file is accessed, the system locates the corresponding inode, allocates an inode table entry, and reads the inode into primary memory. As in the buffer cache, the table entry is considered to be the current version of the inode. Modifications to the inode are made to the table entry. When the last access to the inode goes away, the table entry is copied back to the secondary store ilist and the table entry is freed.

All I/O operations on files are carried out with the aid of the corresponding inode table entry. Accessing a file is a straightforward implementation of the algorithms mentioned previously. The user is not aware of inodes and inumbers. References to the file system are made in terms of path names of the directory tree. Converting a path name into an inode table entry is also straightforward. Starting at some known inode (the root or the current directory of some process), the next component of the path name is searched by reading the directory. This gives an inumber and an implied device (that of the directory). Thus, the next inode table entry can be accessed. If that was the last component of the path name, then this inode is the result. If not, this inode is the directory needed to look up the next component of the path name, and the algorithm is repeated.

The user process accesses the file system with certain primitives. The most common of these are **open**, **creat**, **read**, **write**, **seek**, and **close**.

In the system data segment associated with a user, there is room for some (usually between 10 and 50) open files. This open file table consists of pointers that can be used to access corresponding inode table entries. Associated with each of these open files is a current I/O pointer. This is a byte offset of the next read/write operation on the file. The system treats each read/write request as random with an implied seek to the I/O pointer. The user usually thinks of the file as sequential with the I/O pointer automatically counting the number of bytes that have been read/written from the file. The user may, of course, perform random I/O by setting the I/O pointer before reads/writes.

With file sharing, it is necessary to allow related processes to share a common I/O pointer and yet have separate I/O pointers for independent processes that access the same file. To fill these two conditions, the I/O pointer cannot reside in the inode table nor can it reside in the list of open files for the process. A new table (the open file table) was invented for the sole purpose of holding the I/O pointer. Processes that share the same open file (the result of **forks**) share a common open file table entry. A separate open of the same file will share the inode table entry, but will have distinct open file table entries.

The main file system primitives are implemented as follows. **open** converts a file system path name into an inode table entry. A pointer to the inode table entry is placed in a newly created open file table entry. A pointer to the file table entry is placed in the system data segment for the process. **creat** first creates a new inode entry, writes the inumber into a directory, and then builds the same structure as for an **open**. **read** and **write** access the inode entry as described above. **seek** manipulates the I/O pointer. No physical seeking is done. **close** frees the structures built by **open** and **creat**. Reference counts are kept on the open file table entries and the inode table entries to free these structures after the last reference goes away. **unlink** decrements the count of the number of directories pointing at the given inode. When the last reference to an inode table entry goes away, if the inode has no directories pointing to it, then the file is removed and the inode is freed. This delayed removal of files prevents problems arising from removing active files. A file may be removed while still open. The resulting unnamed file vanishes when the file is closed. This is a method of obtaining temporary files.

There is a type of unnamed FIFO file called a **pipe**. Implementation of **pipes** consists of implied **seeks** before each **read** or **write** to implement first-in first-out. There are also checks and synchronization to prevent the writer from grossly outproducing the reader and to prevent the reader from overtaking the writer.

4.2 Mounted File Systems

The file system of a UNIX system starts with some designated block device formatted as described above to contain a hierarchy. The root of this structure is the root of the UNIX file system. A second formatted block device may be mounted at any leaf of the current hierarchy. This logically extends the current hierarchy. The implementation of mounting is trivial. A mount table is maintained containing pairs of designated leaf inodes and block devices. When converting a path name into an inode, a check is made to see if the new inode is a designated leaf. If it is, the inode of the root of the block device replaces it.

Allocation of space for a file is taken from the free pool on the device on which the file lives. Thus a file system consisting of many mounted devices does not have a common pool of free secondary storage space. This separation of space on different devices is necessary to allow easy unmounting of a device.

4.3 Other System Functions

There are some other things that the system does for the user—a little accounting, a little tracing/debugging, and a little access protection. Most of these things are not very well developed because our use of the system in computing science research does not need them. There are some features that are missed in some applications (for example, better inter-process communication).

The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be. Because of this outlook, many features found in most other operating systems are missing from the UNIX kernel. For example, the UNIX kernel does not support file access methods, file disposition, file formats, file maximum size, spooling, command language, logical records, physical records, assignment of logical file names, logical file names, more than one character set, an operator's console, an operator, log-in, or log-out. Many of these things are symptoms rather than features. Many of these things are implemented in user software using the kernel as a tool. A good example of this is the command language. Each user may have his own command language. Maintaining such code is as easy as maintaining user code. The idea of implementing "system" code with general user primitives comes directly from MULTICS.

Chapter 15: ERROR MESSAGES

CONTENTS

Chapter 15
ERROR MESSAGES

This manual describes some UniPlus$^+$ System V Release 2.0 error messages and appropriate actions and references for each.

The messages appear in alphabetical order with one entry per page. The table of contents lists each error message. The format of each error message entry is described below.

Error Message

DESCRIPTION	This section describes the error message. It may also refer you to further information.
ACTION	This section describes probable causes for each error message, and corrective action to resolve the problem.
REFERENCES	This section contains the name of the UniPlus$^+$ system source code module(s) producing the error message.

Can't allocate message buffer

DESCRIPTION	At system initialization time, it was found that too much memory was being allocated for messages. Messages are currently unusable.
ACTION	Check the MSGSEG and MSGSSZ entries in the system description file. MSGSEG is the number of segments to allocate and MSGSSZ is the size each segment should be. The product of the two numbers (MSGSEG and MSGSSZ) is the amount of memory to allocate for messages. When this amount exceeds the amount of memory currently available in the machine, the above message appears during system booting. The system size must be reduced by either modifying the above entries or by other means. Then a new system must be generated and booted. If the above solutions are unacceptable, more memory must be added to the machine.
REFERENCES	os/msg.c

cksum: out of data

DESCRIPTION	The actual length of the mbuf chain passed to in_cksum was shorter than it should have been.
REFERENCES	in_cksum.c

cmemalloc: improper allocation countscat=# size=#

(scatter loaded kernels only)

DESCRIPTION	After attempting to allocate consecutive pages for a raw I/O request in a scatter loaded kernel, the validity check to see if the correct number was allocated failed. Processing continues as if the attempt was successful. This should not happen
ACTION	Requires no action on its own. Depends on any further errors that may result.
REFERENCES	malloc.c

cxrfree error cx_daddr = 0

(kernels with multiple contexts and shared page registers only (STANFORD style))

DESCRIPTION	In releasing the page registers for a process, resources were expected but not found. This is probably a software problem and should be reported.
ACTION	Depends on subsequent error messages.
REFERENCES	context.c

DANGER: mfree map overflow #, lost # items at

DESCRIPTION	One of the tables, mapped through the system's *malloc* (see *malloc*(3C) in the *User Manual, Sections 2 - 6*) mechanism, has overflowed. The first number indicates the address of the table. By searching for this address in the system namelist, the name of the malfunctioning table can be discovered. The *crash* command (see *crash*(1M) in the *Administrator Reference Manual*) may also be used to find the name of the table.
ACTION	Increase the number of entries currently allocated for the malfunctioning table in the system description file. Generate a new system. Boot the system.
REFERENCES	os/malloc.c

DANGER: out of swap space needed # blocks

DESCRIPTION	The system ran out of swap space in trying to swap and there were no sticky images around to free. This process will wait (possibly indefinitely) for swap space.
ACTION	Grow the available swap space.
REFERENCES	text.c

/dev/swap doesn't match swapdev

changing to *x*, *y*

DESCRIPTION	The system was configured so that **/dev/swap** did not match swapdev as defined by conf.c.
ACTION	The file **/dev/swap** is automatically converted to agree with swapdev and has a major device number *x* and a minor device number *y*. If this is not the desired swap area, reset the swap parameters in the system description file used by *config*. See *config*(1M) in the *Administrator Manual*.
REFERENCES	main.c

Device error on ***device-type*** **drive #, [ctl #,] [slice #]**

DESCRIPTION	This message indicates that a hardware error has occurred on a block type device. The error messages from device drivers vary and are hardware specific depending on the type of controller used. This message is followed by: bn = # er = #,# This is the block number in error, followed by the contents of two of the device registers.
ACTION	See your hardware reference manual.

Double Panic: <message>

DESCRIPTION	The system got a second panic when trying to clean up after a first panic. It has stopped to prevent further damage. This can indicate a severe software or hardware problem.
ACTION	Reboot system and run *fsck*.
REFERENCES	prf.c

dpfree failed

(68451 memory management kernels only)

DESCRIPTION	The memory management unit indicated a failure in trying to clear a freed descriptor. This usually indicates a hardware problem. The system will continue, possibly causing problems when the descriptor is used later.
ACTION	None necessary.
REFERENCES	dp.c

duplicate IP address!! sent from ethernet address: %x %x %x %x %x %x

DESCRIPTION	An ARP packet was received from a machine on the network with the same Internet address as the local machine.
REFERENCES	if_ether.c

exec error: u_error # u_dent.d_name <name>

DESCRIPTION	An unexpected error occurred in attempting to execute a program. *name* specifies the name of the program, *u_error* specifies the error. The program is probably corrupted.
ACTION	Check the program. The image may be corrupted, or there may be a hard disk error in the image.
REFERENCES	sys1.c

File table overflow

DESCRIPTION	The system file table has overflowed and a new reference (see *open*(2), *access*(2), *dup*(2) in the *User Reference Manual, Sections 2 - 6*) to a file failed. This means that the system was not created with a large enough file table to support the maximum number of open files on the system.
ACTION	Increase the number of entries currently allocated for the open-file table (files) in the system description file. The number of inode table entries may also have to be increased. Generate a new system. Boot the new system.
REFERENCES	os/fio.c

forward: src %x dst %x ttl %x

DESCRIPTION	A packet is being forwarded: the source and destination addresses and the time to live are given.
REFERENCES	ip_input.c

iaddress > 2^24

DESCRIPTION	When updating a file's inode on the file system, a block number in the inode was found to be greater than permissible.
ACTION	This message indicates a software and/or hardware error. It is usually caused by a corrupted file system. To check the state of any file system suspected to be corrupted, unmount the file system and check it with the *fsck* command (see *fsck*(1M) in the *Administrator Reference Manual*). If the suspect file system is the root system, the system will have to go single user to check it. The error can also be generated by new device drivers which have not been completely debugged. Also suspect are any standard UNIX system device drivers that have been modified without authorization. Check any drivers that meet the above criteria. If none of the above apply, the error can also be caused by a disk drive and/or controller. This could be a memory problem also. Contact your support organization.
REFERENCES	os/iget.c

Inode table overflow

DESCRIPTION	The system inode table has overflowed and a new file could not be accessed (see *open*(2), *create*(2), *access*(2), and *stat*(2) in the *User Reference Manual, Sections 2 - 6*). The inode table overflow message means that the system was not created with a large enough inode table to support the maximum number of open files on the system.
ACTION	Increase the number of entries currently allocated for the inode table (inodes) in the system description file. Generate a new system. Boot the system.
REFERENCES	os/iget.c

insufficient swap space for available memory
Largest runnable process is

DESCRIPTION	There is insufficient swap space available for the amount of memory found on the system. The size of the largest process has been limited to prevent swapping problems.
ACTION	Grow the available swap space.
REFERENCES	smachdep.c

lo%d: can't handle af%d

DESCRIPTION	A request was made to the loopback driver's output routine with a destination address who's address family was not AF_INET.
REFERENCES	if_loop.c

m_expand returning 0

DESCRIPTION	A request to allocate memory to use for mbufs was denied.
REFERENCES	uipc_mbuf.c

memalloc error: tried to allocate # units

cmemalloc error: tried to allocate # units
(scatter loaded kernels only)

DESCRIPTION	The scatter loaded memory allocation routine was called with an invalid size. The request is ignored. This is a kernel problem.
ACTION	Depends on subsequent problems that may arise.
REFERENCES	malloc.c

memfree: illegal index # (0x#)

(scatter loaded kernels only)

DESCRIPTION	Memfree was called with an out-of-range scatter index. Processing continues as if the attempt was successful. This probably indicates a kernel problem.
ACTION	Requires no action on its own. Depends on any further errors that may result.
REFERENCES	malloc.c

<name> overflow - internal table data discarded.

The define constant <name> is too small.
(68451 memory management kernels only)

DESCRIPTION	Memory was released when there was insufficient space in the memory map. Memory will be lost. System will continue to run without that memory with possible impact on performance or capability to run processes.
ACTION	Grow the named map.
REFERENCES	cmalloc.c

No swap space for exec args

DESCRIPTION	During an exec, the system uses swap space to pass arguments and environment from the calling program to the called program. There was no space available and the exec was aborted.
ACTION	Grow the swap space.
REFERENCES	sys1.c

Not enough swap space to fork

DESCRIPTION	The system makes a preliminary check when forking to make sure that there is enough swap space for this program. If there is not enough, it will print this warning message and return failure on the fork.
ACTION	Grow the available swap space.
REFERENCES	sys1.c

out of mmu registers

(68451 memory management kernels only)

DESCRIPTION	The system has run out of descriptors in trying to map in a process. This probably indicates a software problem. The process will be run anyway and will probably abort.
ACTION	None necessary.
REFERENCES	uregdp.c

panic: ** ABORTING ******

DESCRIPTION	Printed after a system stack dump. It will be preceded by some other kernel error message.
ACTION	See other message.
REFERENCES	smachdep.c

panic: accept

DESCRIPTION	The queue of incoming connections to a socket was nonempty but an attempt to remove the first connection on the queue failed.
REFERENCES	netipc.c

panic: bflush: bad free list

DESCRIPTION	In attempting to do delayed writes for a device (sync or umount), the buffer queue was invalid. This should not happen.
ACTION	Reboot the system.
REFERENCES	bio.c

panic: devtab

DESCRIPTION	The system got an error in attempting to hash the device and block number to look for a buffer. This should never happen.
ACTION	Reboot system.
REFERENCES	bio.c

panic: dpfrelse

(68451 memory management kernels only)

DESCRIPTION	The system has run out of descriptors in trying to map in a process. Probably indicates a software problem.
ACTION	Reboot the system.
REFERENCES	dp.c

panic: findmajor

DESCRIPTION	The getmajor routine was not able to find the major device number. This should never happen. It can occur if a device strategy routine is being used that is not in the block device table.
ACTION	Reboot the system.
REFERENCES	bio.c

panic: icmp_error

DESCRIPTION	An unknown ICMP message type was given to icmp_error.
REFERENCES	ip_icmp.c

panic: iinit

DESCRIPTION	System could not read the superblock or root inode of the root file system. Usually a hardware problem or a bad device driver.
ACTION	Correct problem and reboot. Possibly remap bad block.
REFERENCES	main.c

panic: interrupt stack overflow

DESCRIPTION	The system has run out of supervisor stack. Probably some routine is using excessive local variables. If it persists, it may be necessary to grow the Udot area.
ACTION	Reboot the system.
REFERENCES	clock.c

panic: IO err in swap

DESCRIPTION	An unrecoverable error has occurred during a system swap operation. The processor has halted.
ACTION	This message indicates an error on the disk drive and/or controller problem. The following steps should be taken. Generate a system dump. Change the location of the swap device to a different section on the current disk or replace the disk with another. If the problem is alleviated, the error was caused by a bad spot on the disk. If the problem still exists, suspect disk drive and/or controller problems. Contact your support organization. Meanwhile, attempt to boot from a different disk drive.
REFERENCES	io/bio.c

panic: ip_init

DESCRIPTION	No protocol table entry exists for protocol family PF_INET, protocol IPPROTO_RAW.
REFERENCES	ip_input.c

panic: kernel memory management error

DESCRIPTION The kernel got a bus or address error. This message is preceded by *kernel address error* or *kernel bus error.*

```
vaddr=# paddr=# ireg=# fcode=#
```

vaddr is virtual fault address
paddr is physical fault address
ireg is instruction register
fcode is the function code

```
pc=# sr=# up->u_procp=# pid=# exec=NNN
#   #   #   #   #   #   #   #
#   #   #   #   #   #   #   #
```

pc is the program counter
sr is the status register
up->u_procp is process pointer
pid is process id of active process
exec is the name of the active process
the rest is the data and the address registers

It can indicate either a software or a hardware problem. The program counter gives a good starting place to look.

ACTION Reboot the system.

REFERENCES trap.c

panic: kernel parity error

DESCRIPTION Memory detected a parity error while the kernel was running. Message preceded by *trap: kernel parity error.*

```
vaddr=# paddr=# ireg=# fcode=#
```

vaddr is virtual fault address
paddr is physical fault address
ireg is instruction register
fcode is the function code

```
pc=# sr=# up->u_procp=# pid=# exec=NNN
#   #   #   #   #   #   #   #
#   #   #   #   #   #   #   #
```

pc is the program counter
sr is the status register
up->u_procp is process pointer
pid is process id of active process
exec is the name of the active process
the rest is the data and the address registers

ACTION Run memory check if one is provided with the system. Reboot the system.

REFERENCES trap.c

panic: lost vmap segment

(68451 memory management kernels only)

DESCRIPTION	In freeing shared data pages, a conflict exists in the memory map. The operation will be skipped, potentially causing later problems.
ACTION	None necessary.
REFERENCES	shm.c

panic: m_copy

DESCRIPTION	In copying one mbuf chain to another, one of the following occurred: • the offset or length was negative • the offset is positive but no more mbufs remain • the length is positive but no more mbufs remain • the offset of an mbuf in the chain is greater than MMAXOFF
REFERENCES	uipc_mbuf.c

panic: No data pages

panic: No text pages
(kernels with multiple contexts and shared page registers only (STANFORD style))

DESCRIPTION	Insufficient page registers were available to set up the process. This should not happen.
ACTION	Reboot the system.
REFERENCES	cxureg.c

panic: no fs

DESCRIPTION	The in-core super-block of a mounted file system cannot be found. This should never happen. The processor has halted.
ACTION	Generate a system dump. Reboot the system. The *crash* command (see *crash*(1M) in the *Administrator Reference Manual*) can be used to gather more information from the system dump about the nature of the problem. Probable causes of the error include both software and hardware problems. Check the configuration information in the system description file to make sure it is correct. Check any device drivers that have not been completely debugged. Check any UNIX system device driver that has been modified without authorization. If none of the above apply, suspect hardware problems with the disk drive and/or controller. Contact your support organization.
REFERENCES	os/alloc.c

panic: no imt

DESCRIPTION	A mount point was not found in the system mount table when traversing a file system boundary. This should never happen. The processor has halted.
ACTION	Generate a system dump. Reboot the system. The *crash* command (see *crash*(1M) in the *Administrator Reference Manual*) can be used to gather more information from the dump about the nature of the problem. Probable causes include both software and hardware problems. Check the configuration information in the system description file, make sure it is correct. Check any new device drivers that have not been completely debugged. Also, check any UNIX system device driver that has been modified without authorization. If none of the above apply, suspect hardware problems with the disk drive and/or controller. Contact your support organization.
REFERENCES	os/iget.c

panic: no procs

DESCRIPTION	A process table entry cannot be found during a fork when it is known that an entry is available. This should never happen. The processor has halted.
ACTION	Generate a system dump. Reboot the system. Contact your support organization.
REFERENCES	slp.c

panic: out of swap space

DESCRIPTION	The system ran out of swap space in trying to swap out a process.
ACTION	Grow the available swap space.
REFERENCES	text.c

panic: raw_usrreq

DESCRIPTION	Raw_usrreq was called with an unknown user request.
REFERENCES	raw_usrreq.c

panic: receive

DESCRIPTION	The count of characters in a socket's receive queue is nonzero but the pointer to the mbuf chain containing the data is NULL.
REFERENCES	socket.c

panic: receive 2

DESCRIPTION	In dealing with a protocol in which addresses are always passed with messages, either an mbuf could not be allocated to hold the remote address or the size of the remote address was zero.
REFERENCES	socket.c

panic: receive 2a

DESCRIPTION	In dealing with a protocol in which addresses are always passed with messages, the message contained the sender's address but no access rights.
REFERENCES	socket.c

panic: receive 3

DESCRIPTION	In dealing with a protocol in which addresses are always passed with messages, the message contained the sender's address and access rights, but no data.
REFERENCES	socket.c

panic: receive 4

DESCRIPTION	In a protocol in which only atomic messages are exchanged, the "end of message" marker is not present but no more data remain.
REFERENCES	socket.c

panic: rtfree

DESCRIPTION	Rtfree was called with a NULL argument.
REFERENCES	route.c

panic: sbappendaddr

DESCRIPTION	The pointer to the first mbuf of data in a message is NULL.
REFERENCES	socket2.c

panic: sbdrop

DESCRIPTION	A request has been made to drop some data from a socket buffer but the pointer to the first mbuf in the data chain is NULL.
REFERENCES	socket2.c

panic: sbflush

DESCRIPTION	Sbflush was called to flush data from a socket but the receive queue was "locked."
REFERENCES	socket2.c

panic: sbflush 2

DESCRIPTION	After flushing all data from a socket buffer, either the character count, the mbuf count, or the pointer to the first mbuf in the data chain is nonzero.
REFERENCES	socket2.c

panic: soaccept: !NOFDREF

DESCRIPTION	A socket on the queue of incoming connections for another socket already has a file table reference.
REFERENCES	socket.c

panic: soclose: NOFDREF

DESCRIPTION	A close was attempted on a socket for which a file table reference should no longer exist.
REFERENCES	socket.c

panic: sofree dq

DESCRIPTION	The list of "accept sockets" for a socket was not NULL but an attempt to remove the first connection on the queue failed.
REFERENCES	socket.c

panic: soisconnected

DESCRIPTION	A socket related to an accept cannot be found on the queue of partial connections of the socket to which it is related.
REFERENCES	socket2.c

panic: sosend

DESCRIPTION	The number of scatter-gather write areas remaining to send is negative although it was never 0.
REFERENCES	socket.c

panic: syscall

DESCRIPTION	A system call came through trap rather than syscall. The interrupt vector table is not set up correctly.
ACTION	Fix the interrupt table and rebuild the kernel.
REFERENCES	trap.c

panic: tcp_output

DESCRIPTION	The TCP template pointer for a TCP control block is NULL.
REFERENCES	tcp_output.c

panic: tcp_output REXMT

DESCRIPTION	The TCP retransmit timer was set when an attempt was made to start the persistence timer.
REFERENCES	tcp_output.c

panic: tcp_pulloutofband

DESCRIPTION	The TCP urgent pointer points within the current segment but beyond the last byte of data in the segment's mbuf chain.
REFERENCES	tcp_input.c

panic: tcp_usrreq

DESCRIPTION	Tcp_usrreq was called with an unknown user request.
REFERENCES	tcp_usrreq.c

panic: Timeout table overflow

DESCRIPTION	The system timeout table, which is used to implement software interrupts, has overflowed while attempting to add another entry. The processor has halted.
ACTION	Reboot the system. If this condition persists, increase the number of entries currently allocated for the call-out table (calls) in the system description file. Generate a new system. Boot the new system.
REFERENCES	os/clock.c

panic: udp_usrreq

DESCRIPTION	Udp_usrreq was called with an unknown user request.
REFERENCES	udp_usrreq.c

phys mmu overflow

shared mem mmu overflow
(kernels with multiple contexts and shared page registers only (STANFORD style))

DESCRIPTION	Insufficient page registers were available to set up the phys or shared memory area of a process. This should not happen.
ACTION	The process will continue to run without the specified area set up. If it uses that area, it will probably core dump.
REFERENCES	cxureg.c

procdup: error: a1 == #

procdup: error: a2 == #
(scatter loaded kernels only)

DESCRIPTION	Procdup was called with more memory allocated than specified in the process size. Processing continues, ignoring extra memory. Probably indicates a kernel problem.
ACTION	Requires no action on its own.
REFERENCES	mmachdep.c

Random interrupt ignored

DESCRIPTION	The system got a spurious or random interrupt. This interrupt is generated by the MC68000 if a device requests an interrupt but does not acknowledge when the 68000 responds. This is usually a hardware problem.
ACTION	No action necessary. The system will continue, ignoring the interrupt. If the interrupt should have been handled, further problems may arise later.
REFERENCES	trap.c

real mem = *value*

avail mem = *value*

DESCRIPTION	Real memory is the number of bytes of physical memory on the CPU. Available memory is the number of bytes of physical memory actually available to the user processes.
ACTION	This message is for information only.
REFERENCES	machdep.c

panic: unexpected kernel trap

DESCRIPTION An unexpected system fault has occurred. This message is preceded by trap type #

```
pc=# sr=# up->u_procp=# pid=# exec=NNN
#   #   #   #   #   #   #   #
#   #   #   #   #   #   #   #
```

pc is the program counter
sr is the status register
up-> u_procp is process pointer
pid is process id of active process
exec is the name of the active process
the rest is the data and the address registers

The message may also be preceded by a memory management dump. The trap type is the trap type as specified in a motorola 68000 reference manual.

ACTION Reboot the system. Probable causes include the following.

- New device drivers that have not been completely debugged.
- Unauthorized modifications to existing system device drivers.
- Running out of system resources.
- Incorrect information in the system description file.
- Lack of information in the system description file.

- Hardware problems.

REFERENCES trap.c

out of text

DESCRIPTION	The shared text table has overflowed and a shared text program was not allowed to execute. Too many programs with the sticky bit on (ISVTX) may be present in the system. This is usually a transient condition that occurs under a heavy load.
ACTION	If the condition persists, increase the number of entries currently allocated for the text table (texts) in the system description file. Generate a new system. Boot the system.
REFERENCES	os/text.c

scatter load map too small by # units

(scatter loaded kernels only)

DESCRIPTION	The scatter load map size is too small for the amount of memory on the system.
ACTION	Grow NSCATLOAD by at least the indicated amount and rebuild the kernel.
REFERENCES	smachdep.c

swap error: swapping beyond process

(scatter loaded kernels only)

DESCRIPTION	During swapping, the size of a process indicated that more segments should have been allocated in memory than actually were allocated. The swap will be aborted. This will probably cause problems when the process tries to run again.
ACTION	Depends on subsequent results.
REFERENCES	bio.c

unsuccessful mmu load

lba=# lam=# pba=# asn=# asm=#
memory management unit dump
(68451 memory management kernels only)

DESCRIPTION	The memory management unit has indicated an error in loading due to a conflict in addresses. Probably indicates a software problem, but can be a bad chip. System will continue with process anyway. *lba* is the logical base address *lam* is the logical address mask *pba* is the physical base address *asn* is the address space number *asm* is the address space mask All of the control registers and each of the descriptors will also be dumped.
ACTION	None necessary.
REFERENCES	uregdp.c

WARNING: swap space running out needed # blocks

DESCRIPTION	The system needed to free a sticky image in order to swap.
ACTION	Grow the available swap space or mark less processes sticky.
REFERENCES	text.c

Chapter 16: VIRTUAL TUNING

CONTENTS

Chapter 16

VIRTUAL TUNING

1. Configuration Tuning

Except for special provisions required by the paging environment noted below, tuning guidelines for the UniPlus$^+$ virtual implementation are identical to those described in the *Administrator Guide*.

1.1 Multiple Swap Areas

Virtual UniPlus$^+$ provides the ability to configure systems with multiple swap areas. The procedure for doing so is described in the manual page for **swap**(1M).

Under most circumstances, there is no need to configure more than a single swap area. However, for the following conditions, multiple swap areas should be configured:

- operating system warning messages suggest that the system is running low on swap space and the current swap area cannot be extended.
- load across disk drives is unbalanced and ordinary file systems cannot be relocated.

As usual, whenever reapportioning access across disk drives by moving logical partitions, verify that load across drives is evenly distributed in the new configuration by comparing the *%busy* files for all drives reported by **sar −d**.

1.2 Paging Parameters

Virtual UniPlus$^+$ provides the ability to control activity of the page daemon by selective setting of parameter values. The values of these parameters are specified in **/etc/master** (see **master**(4) and config(1M)).

1.3 Paging Daemon Parameters

In general, the default settings provide near optimal performance for a variety of workloads. However, circumstances when changes might be made to default setting include:

- most active processes are in the swap queue and only one process is in the run queue
- Central Processor Unit (CPU) idle time occurs for a workload where none occurred with a swapping version of System V Release 2.0 (time spent waiting for memory is attributed as idle time).

The first case results when the paging daemon is too active and the second symptom occurs when the daemon is not active enought. The goal is to establish daemon activity that is acceptable.

- *vhandr*—decrease the value to make the daemon more active: increase to make less active (must be integer > 0 and ≤ 300).
- *vhndfrac*—decrease the value to make the daemon more active; increase to make less active (must be integer > 0 and $>$ *maxmem*/*getpgslow* and $<$ 25 percent of available memory).
- *getpgshi*—increase the value to make the daemon more active; decrease to make less active (must be integer >0 an d $>$ *getpgsio* and $<$ 25 percent of available memory).
- *getpgslow*—increase the value to make the daemon more active; decrease to make less active (must be integer ≥ 0 and $<$ *getpgshi*).

The value of *getpgshi* and *getpgslow* will be dynamically adjusted at system boot time if there are fewer free pages (*maxmem*) available than the initial value of *getpgshi* and *getpgslow*.

The following table provides several parameter configurations that are well suited to different job mixes. As noted above, the default configuration should be satisfactory for most applications. however, sites running only large processes (greater than 400 kilobytes) may wish to experiment with different values.

Suggested Paging Parameter Configurations

Parameter	Default
vhandr	1
vhndfrac	16
getpgshi	100
getpgslow	25

1.4 Additional Parameters

In general, the default settings for the following additional parameters should not be altered.

- *nregion*—set at 2.5 times the number of *procs* and increment if and only if region table overflow message appears.
- *maxpmem*—default is 0 and should not be altered. *Maxpmem* is the maximum memory size in pages, a value of 0 uses all available memory.
- *maxumem*—default is 256, but can be reduced to save memory consumed by page tables at the cost of decreasing user process address space. This is the maximum number of pages which a user process is allowed to use.
- *nsptmap*—set at the number of *procs* and increment if and only if the message "DANGER: mfree map overflow . . ." appears.
- *maxsc*—if system warning message low on swap appears, this value should be set lower or the swap space increased. The value must be an integer much greater than 1 (default of 64)

and less than or equal to 100.

- *maxfc*—The value must be an integer much greater than 1 (default of 100) and less than or equal to 100.
- *ppmem*—The value must be an integer greater than 1 (default 10) and $\leqslant$ *getpgslow*. This values is used to determine when to deny a user process memory resources when memory becomes scarce. The super-user can always get the last memory available.

1.5 Record and File Locking Parameters

There are two parameters which control the resources that are allocated for Record and File Locking.

- *flckfil*—the number of files in the system that can have record locks set at one time; the number cannot exceed *files*.
- *flckrec*—the number of internal records locks that may be present on the system at one time.

Colophon

Composed at UniSoft Systems Inc.
on the UniPlus$^+$ Operating System
Designed by the Documentation Department
Printed in Times Roman on Sequoia Matt