Chapter 1: Introduction

This chapter provides an introduction to the manual itself as well as to UNIX and its implementations at Fermilab. In particular, it covers:

- the purpose and intended audience of *UNIX at Fermilab*
- where to send comments and questions about the manual
- where to obtain additional copies of the manual
- the typeface conventions and symbols used throughout the manual
- a summary of the contents of all the chapters and appendices in the manual
- a very brief history of the UNIX operating system and a discussion of its advantages
- an introduction to the *Fermi UNIX Environment*, which is installed on most UNIX systems at Fermilab
- a listing of the supported UNIX operating systems
- an introduction to FNALU, the central Fermilab UNIX cluster
- information on locating on-line Fermilab documents
- a listing of UNIX references

1.1 The *UNIX at Fermilab* Manual

This manual covers basic UNIX concepts and operations, and thus is especially geared to users who are new to UNIX. It also documents the UNIX environment and file systems commonly used at Fermilab, and thus is a handy reference for all users of Fermilab UNIX systems.

> There are many different UNIX systems installed at Fermilab, with differing levels of conformance to Fermilab standards. Therefore, some of the information contained in this manual may not be valid for your system. You will very likely need to obtain supplementary documentation and information specific to your system.
>
> In particular, CDF and D0 have created documentation for their systems and made it available on the World Wide Web (this utility is described in section 3.2.1). You can access this information at the following locations:
>
> **CDF**
>
> **`http://www-cdf.fnal.gov/offline/cdfsga/cdfsga.html`**
>
> **D0**
>
> **`http://d0wop.fnal.gov/d0unix/d0unix.html`**

The purpose of this manual is to help you get started as quickly and painlessly as possible in the UNIX environment here at Fermilab, as well as to help you use UNIX easily and efficiently, whatever your particular needs. We discuss the most commonly used UNIX concepts, commands and tools, and provide enough information to allow you to execute commands and perform tasks typically required by the Fermilab user community. We refer you to commercially available sources of information for details on commands and features that will allow you to exploit the more sophisticated features of the UNIX operating system not covered in this document. We also describe the Fermi UNIX Environment, which includes Fermilab-specific programs, libraries, tools, and usages, and occasionally refer you to other Fermilab documents.

We try to describe here a rather “standard UNIX” and recommend that you try not to use the “value-added” features that the individual vendors provide which may cause you trouble in conversion to another platform. Commands and features for both families of UNIX command interpreters (discussed in Chapter 4), C shell and Bourne shell, are included when they differ. Since some of the systems at Fermilab will have the AFS file system installed, we address some issues concerning AFS in Chapter 7.

It is *not* the purpose of this manual to teach you UNIX or to give you a complete description of UNIX. For that purpose we recommend that you obtain and read one of many good books on UNIX. See section 1.6 for some suggestions. This manual is also not intended as a system administration reference.

For those readers who wish to convert from VMS to UNIX quickly without learning about UNIX ahead of time, see Appendix A for the bare minimum needed to accomplish this task!

> *UNIX at Fermilab* is bound to contain some errors, however we endeavor to minimize the error count! We encourage all the readers of this manual to report back to us:
>
> - errors or inconsistencies that we have overlooked
> - any parts of the manual that are confusing or unhelpful -- please offer *constructive* suggestions!
> - other topics to include (keeping in mind the purpose of the manual)
> - tricks that other Fermilab users might find helpful
>
> Send your comments via email to *cdlibrary@fnal.gov.*

Copies of *UNIX at Fermilab*, document number GU0001, can be obtained from the following sources:

WWW[1]	`http://www.fnal.gov/docs/UNIX/unix_at_fermilab` You can reach this from the Fermilab Computing Division home page, under *UNIX Resources* or via the document database (see section 1.6.1). Both PostScript and HTML versions are available.
Paper Copies	Wilson Hall, 8th floor, NE (outside the former Computing Division library)
AFS	PostScript files are available under `/afs/fnal/files/docs/UNIX/unix_at_fermilab/ps/rev1997/`

1. See section 3.2.1 for information on the World Wide Web (WWW).

1.1.1 Conventions Used in this Manual

The following notational conventions are used in this manual:

bold	Used for shells (e.g., **csh**) and product names.
italic	Used to emphasize a word or concept in the text; many terms found in the index are italicized in the text. Also used to indicate variables and reference book titles.
`typewriter`	Used for filenames, path names, contents of files, output of commands.
sans-serif	Used to indicate keys (on keyboard) and “buttons” on graphical applications.
`typewriter-bold`	In text, used to indicate commands and prompts. In command formats, indicates what the user types “as is”.
bold-italic	In command formats, indicates variables for which the user must make context-specific substitutions.
<Ctrl-***char***>	Indicates a control character. To enter a control character, hold down the control key (labeled Ctrl, probably) while pressing the key specified by ***char***.
[ ]	In command formats, indicates **optional** command arguments and options.
|	When shown in between brackets (**`[x|y|z]`**) in a command line, separates a series of options from which one must be chosen. In UNIX command formats, used to “pipe” output of preceding command to the following one (see section 5.4.3).
`'` ... `'`	Single vertical quotes indicate apostrophes in commands.
`` ` `` ... `` ` ``	Single backquotes in UNIX commands have a special meaning (use output of string in place of string itself). Don’t confuse them with apostrophes.
`...`	Means that a repetition of the preceding parameter or argument is allowed.
`%`	Prompt for C shell family commands (`%` is also used throughout this document when a command works for both shell families).
`$`	Prompt for Bourne shell family commands; also standard UNIX prefix for environment variables (e.g., *$VAR* means “the value to which *VAR* is set”).
{ }	In environment variables, paths, files and other text strings, indicates strings for which the user must make context-specific substitutions. For example, *${PRODUCT}_DIR* is *$WWW_DIR* for the product **www**, and `{username}@{node}` refers to a string like `joe@fsui02`.

All command examples are followed by an implicit carriage return key.

The following symbols are used throughout *UNIX at Fermilab* to draw your attention to specific items in the text:

A UNIX “bomb”; this refers to something important you need to know about UNIX in order to avoid a pitfall.

A VMS “bomb”; this is used to point out common pitfalls of VMS users.

This symbol indicates information for AFS systems (see Chapter 7).

This symbol is intended to draw your attention to a useful hint.

This is a reminder.

We refer you to other documentation sources for further information where you see this symbol.

1.1.2 Summary of Chapters and Appendices

The organization of this manual reflects our compromise between providing a tutorial and a reference manual. You may find it useful to skip around a little rather than reading straight through. We recommend that you get a UNIX account and begin to work with the system as you read.

Chapters:

2: Getting Started on a UNIX System

This chapter describes the login and logout procedures, and discusses some basic UNIX features including the prompt, special keys, and special characters. We present a brief discussion of the file systems you are likely to encounter at Fermilab, and how to change your password according to your file system and installation.

3: Information Resources

This chapter introduces you to the information facilities available from UNIX. Standard UNIX on-line help is available via the man pages. We discuss the World Wide Web and newsgroups, which are very rich sources of information on a virtually unlimited set of topics. A few utilities that allow you to get information about vendor products, other users, and the Fermilab computing systems are also covered. Finally, we include instructions for communicating with the Fermilab Helpdesk.

4: Shells

This chapter discusses the concept of a UNIX shell, and how to manipulate shells. It includes information on the available and recommended shells and their features. The concept of a shell as an interpretive programming language is introduced.

5: Important UNIX Concepts

This chapter introduces you to the UNIX command structure, and to many important commands and concepts. The features introduced in this chapter constitute the core of the UNIX operating system, and many of these tools are quite powerful and flexible. Some of the features are shell-specific, and we provide the distinctions where necessary.

6: The UNIX File System

The UNIX file system has a hierarchical or tree-like structure with the directory called *root* (`/`) as its source. The system is essentially composed of *files* and *directories*. In this chapter we describe techniques for manipulating files and directories, and commands designed to provide information about them.

7: The AFS File System

Fermilab is using the AFS (Andrew File System) as a distributed file service model, and it is installed on several machines on site in a production environment, including the FNALU cluster. This chapter discusses the basic concepts of AFS and provides information on the commands used to manage your files and directories in the AFS environment.

8: Printing

This chapter covers the standard FUE print utility **flpr**, as well as filter programs and techniques available for formatting the output prior to printing. **flpr** is the Fermi implementation of the standard UNIX **lpr** utility. Most software applications supplied by the Computing Division use **flpr** as a default.

9: Working Environment

This chapter describes the methods used to set up your working environment in UNIX. Some of these are standard UNIX (e.g., shell and environment variables), and some are provided and/or customized by FUE (e.g., the login scripts).

10: Accessing Software Products

In this chapter you will learn how to get information about the software products that are provided and supported by the Computing Division. We describe how to access products already installed on your system, and how to obtain a product from the Fermilab `KITS` area and install it on your system.

☞ A notice regarding upcoming changes is included.

11: Editors

Several text editors are available at Fermilab. In this chapter we present our view of the advantages and disadvantages of the available editors, and we provide some basic information on the setup and use of each one. You will learn how to invoke each editor, and how to create, edit, and save a file in each one using a small subset of commands and features. We include only minimal usage information for the VMS-style editors.

12: UNIX Mail Systems

This chapter describes how mail forwarding is managed at Fermilab and discusses the UNIX mail handlers that are currently available and supported.

☞ A notice regarding upcoming changes is included.

13: Connecting to Remote Systems

Several utilities are available to enable you to transfer files between systems, to log into other systems on which you have an account, and to execute commands remotely. These features are described in this chapter.

14: Batch Processing Environment

In this chapter we provide introductory information on **LSF** (Load Sharing Facility), the standard batch processing system at Fermilab, and on **fbatch**, the locally-written interface to **LSF**. We also list the related software components that can be used with **LSF/fbatch**.

15: Tape Handling

In this chapter we discuss the principal tape handling software and facilities available at Fermilab. Start-up information for running and monitoring **OCS** tape mounts is provided, and the **OCS** X interface is introduced. Several tape I/O packages are briefly described.

16: Software Development

This chapter gives an introduction to UNIX software development tools in common use at Fermilab, providing information on:

- Supported languages
- Compiling and linking in C, C++ and FORTRAN
- Debugging

We do not include a discussion of general programming here, but rather, aspects of software development particular to UNIX.

17: The make Utility

The UNIX **make** utility is a tool for organizing and facilitating the update of executables or other files which are built from one or more constituent files. Although **make** can be used in a wide variety of applications, in this chapter we concentrate on its use in the area of software development. We describe how to define relationships between source, object, library and executable files for use by **make**, and how to invoke **make** in its simplest and slightly more complex forms.

18: Code Management

This chapter introduces the recommended code management solution for UNIX, **CVS** (Concurrent Versions System). We also introduce an alternative that is currently being used by a couple of Fermilab experiments, **UCM** (UNIX Code Management). Both packages use **RCS** (Revision Control System) as the underlying protocol. We provide basic information only, and refer you to the complete manuals for these utilities for detailed information.

Appendices:

A: VMS Migration for the Impatient

So, you've decided you're ready to convert (or you've run up against a deadline!), but you don't know the first thing about UNIX. Here's enough information to get you moved over. You can use the rest of the manual to learn about UNIX afterwards.

B: UNIX Product Support (UPS) Overview

In this appendix we discuss the Fermilab product support structure, **UPS**. We recommend that you read and understand this material before performing any of the tasks described in Chapter 10.

☞ A notice regarding upcoming changes is included.

C: Fermi Login Files

This appendix contains file listings of the FUE-customized default login files (the "Fermi Files") used to set up your UNIX environment. If you are on a FUE-compliant system, you are supplied with a copy of each file in your home directory (except for the `fermi.*` and `setup.*` files which are executed directly from `/usr/local/etc`). The files are reprinted here in their entirety except for the copyright disclaimers.

D: awk's Programming Model

This appendix is adapted from a section of the same name in the book *sed & awk*, published by O'Reilly & Associates. It describes the generic structure and organization of an **awk** program.

E: VMS to UNIX Command Reference

This appendix is intended as a convenient UNIX command reference for the migrating VMS user. The UNIX equivalents of commonly used VMS commands are listed in tabular format. No information on syntax, options, or arguments is presented here, however some of the listed commands are described elsewhere in *UNIX at Fermilab*.

F: mh and exmh Customization

This appendix contains information for further customizing the **mh** and/or **exmh** mail readers. Given the multitude of customizable features in these mail readers, we cannot provide you with a comprehensive treatment of the subject here. The information presented in Chapter 12 and this appendix should be sufficient for most users.

G: mh Command Reference

This appendix provides an alphabetical reference to the subset of **mh** commands discussed in Chapter 12.

H: Mail Conversion from VMS

This appendix is intended to guide you through a mail conversion process from VMS to the UNIX **MH** (Message Handling) system. Two options for conversion are presented.

I: Programming Examples

This appendix contains examples of programs that illustrate information presented in Chapters 16 and 17.

1.2 About UNIX

1.2.1 A Brief History

The UNIX operating system was developed at AT&T Bell Laboratories in Murray Hill, New Jersey in the late 1960's. Universities and colleges have played a major role in popularizing UNIX. The Computer Science Department at the University of California at Berkeley made so many popular changes to it that one of the two most popular versions in use today is named the Berkeley Software Distribution (BSD) of the UNIX system. The other major version is AT&T's UNIX System V. Many implementations incorporate features of both systems.

1.2.2 A Word About Features and Components

Because UNIX was originally designed by programmers to support their own projects, one of its strongest points is that it provides an excellent software development environment. UNIX has a large set of powerful utility programs and tools that allow users to easily build systems and applications. It also has several command interpreters, called *shells* that can also be used as high-level programming languages. Keep an open mind to the powerful features of UNIX that may be quite different from systems you are familiar with. Some of the important concepts are introduced in Section 5.4 (e.g., pipes and filters, and device-independence).

Bear in mind that there are many UNIX utilities not described in this guide.

The UNIX operating system has four basic components:

- The *kernel* constitutes the nucleus of the operating system and coordinates the internals such as allocating system resources.
- The *file system*, which is part of the kernel, controls the storage and access of data. It is similar to the VMS file system in that the structure is hierarchical.
- *Commands* are programs that you request the computer to execute.
- Programs (commands) called *shells* serve as command interpreters. They act as links between you and the computer, interpreting and executing commands. They are also high-level interpretive programming languages.

Although there are many implementations of UNIX, there are two families of shells. The Bourne shell family and the Berkeley/C shell family. The shell families and the individual shells are discussed in more detail in Chapter 4.

1.2.3 The Advantages of UNIX

New Technologies

UNIX allows Fermilab to achieve better alignment with the mainstream of computing. The "developmental momentum" is with UNIX and personal computers, and it is primarily for these environments that the new technologies, software products, and software methods are emerging. Other HEP laboratories have also recognized these trends and are moving in this direction.

Multi-Platform Support

There are many UNIX platforms, and while there are certainly variations among them, the differences tend to be small. This allows the Computing Division to more effectively support platforms supplied by many vendors, thus eliminating the link to a single vendor whose pricing policies and technology directions are outside of our control.

Availability of Popular Applications

The general administrative and much of the technical computing load has migrated to MACs and PCs due to the availability of commercial applications. There is also a growing demand from HEP researchers for these applications. The ability to use personal computers as X-terminals offers a real possibility of providing a single platform that allows access to popular applications for word-processing, preparation of presentations, charting, statistical analysis, etc., as well as access to powerful UNIX systems for data analysis and program development.

Choice of Machine and Platform

You first need to determine if you are going to purchase your own desktop UNIX system, get an account on a workgroup UNIX system, or get an account on the central Fermilab UNIX system, FNALU. At this point, you may or may not have a choice of platform; for instance FNALU is comprised of several different platforms. UNIX exists in different *flavors* on the different vendor platforms, meaning that the UNIX operating system is somewhat customized to each platform. The OS is actually named differently on each platform: AIX on IBM, IRIX on Silicon Graphics, Solaris on Sun, Digital UNIX on Digital[1], and LINUX on PC. The differences between flavors tend to be small.

1. See section 1.4 for a note on the OS name for Sun and Digital systems.

Customizable Environment

The "brave new world" of UNIX allows you to set up your work environment and choose your tools to suit your own individual needs. At Fermilab we have been careful to implement and support what we consider to be the most useful of the wide range of available UNIX utilities appropriate for our environment. Still, you will find a much larger array of options than you may be accustomed to. We provide you with defaults for most options, thereby allowing you to get used to the new system gradually, and build your knowledge and proficiency at your own speed.

☞ In order to take full advantage of the flexibility offered, you need to be aware of the options, and you need to make some choices. Two choices you as an individual user need to make initially are the editor and the mail system you want to use. Of course you can change to other ones later. After that, you have the choice of delving further into the marvels of UNIX, or accepting the provided defaults so you can just get to work. UNIX is in fact usable by ordinary mortals as well as by sophisticated hackers! Throughout this manual we give some guidance for making choices.

The available editors come in three basic types:

- modern point-and-click style editors (for graphics-capable terminals) similar to those available for PC Windows and Macintosh
- UNIX editors that provide powerful features for sophisticated editing needs, but that can also be used for simple editing tasks using a small subset of commands
- VMS-style editors

The types of mail systems offered include:

- basic UNIX mail readers
- more fully functional "traditional" mail systems
- modern point-and-click mail readers

Instead of a single command interpreter and user interface, several shells have been developed for UNIX, a number of which we provide and support. Until you get a good idea of what the advantages are of one shell over another, we recommend that you just use the default shell provided. You can change it later.

The choices that you need to make should not be overwhelming. Most people find the range of choices empowering once they're familiar with the system.

1.3 The Fermi UNIX Environment

The on-site UNIX[1] computer systems at Fermilab are used for a wide range of tasks, including general purpose interactive use, batch jobs, farms[2], and data acquisition systems, among other things. In order to most efficiently accommodate the needs of the Fermilab user community, the Computing Division provides a single, operating system-independent UNIX environment known as the *Fermi UNIX Environment*, or *FUE*.

The Fermi UNIX Environment (FUE) is a set of products installed on most UNIX systems at Fermilab that defines the computing environment. One of the main goals of FUE is to provide as much as possible the same environment on the different UNIX platforms. A second important goal is to provide a standard product support methodology.

1. UNIX has been customized to several different hardware platforms, and each of these adaptations (SunOS, IRIX, etc.) is considered a separate operating system (also called OS type in this document).
2. Farms are clusters of RISC-based workstations for which parallel processing software has been developed to support activities such as event reconstruction and Monte Carlo.

The methodology and infrastructure for product support and distribution under FUE is provided via a software support toolkit called UNIX Product Support (**UPS**). **UPS** was developed with the goal of providing a uniform and consistent interface for the management, distribution, installation and use of all the supported UNIX software. The **UPS** methodology is fully described in *UNIX Product Methodology at Fermilab*, document number GU0014. Chapter 10 of the present document describes the operations that installers and end users of the supported software need to know how to perform. Appendix B provides an overview of **UPS** to supplement the information in Chapter 10.

FUE consists of three parts:

- A set of login scripts, called the *Fermi files* that provide a common environment. These are described in section 9.4 and listed in Appendix C.
- Infrastructure to allow access to software products under **UPS**.
- General utilities, system administration tools, and software products in **UPS** format.

UNIX systems at the lab or at users' home institutions can be set up to be FUE-compliant. There are different levels of FUE compliance available; these levels are described in GU0014.

1.4 UNIX OS Support at Fermilab

The Fermilab Computing Division supports many different flavors of UNIX operating systems. The Computing Division currently supports the following platform and operating system combinations:

- Silicon Graphics running IRIX
- IBM running AIX
- Digital running Digital UNIX (OSF1)
- Sun Microsystems running Solaris (SunOS)
- Intel PCs running redhat LINUX (this is just becoming supported as this manual goes to print)

There is a command **`uname`** (also **`funame`** at Fermilab) that returns information about the current system. The OS name returned by these commands for Sun is SunOS and for Digital is OSF1. Note that these are different from the names under which the vendors now market these operating systems.

For the currently supported releases of the operating systems, C compilers and FORTRAN compilers, refer to the document DR0010 (see section 1.6.1 for information on how to locate Fermilab documents). This information changes frequently.

1.5 The Central Fermilab UNIX Resource: FNALU

FNALU is a UNIX computing resource which is available to the laboratory at large and managed by the Computing Division. It provides batch and interactive computing on IBM, Silicon Graphics, Sun, and DEC platforms. The primary file system on FNALU is AFS, described in Chapter 7. AFS is designed explicitly to serve as a distributed file system both within a local area network (LAN) and a wide area network (WAN). FUE is installed on FNALU, and it handles the provided software products in a way that appears the same to all users of the cluster, regardless of platform.

1.5.1 Getting an Account on FNALU

Accounts on the FNALU cluster fall into two categories, based on the user's resource requirements. If your resource requirements are small (< 50Mbytes/1% CPU), you can request an account using the *Computer Account Request Form*. To request an account allowing the use of resources beyond this limit, you need to submit a *Project Request Form* on which you indicate your estimated resource requirements. Both forms are accessible from the Computing Division home page in *Forms* under the heading **Services**. Read the *Proper Use of the Computing Facilities Form* (also in *Forms*) before filling out the appropriate form. Send your completed form to *compdiv@fnal.gov*, or deliver it to the Computing Division administration area, on the Wilson Hall 8th floor cross-over.

Refer to the *FNALU User's Guide*, document number GU0008, for a description of the machines that comprise the FNALU cluster and the available software.

1.5.2 Intended Uses of the FNALU Cluster

FNALU is intended to be a general-purpose, multi-platform cluster with a focus on physics and engineering requirements. The intended uses of FNALU can be summarized into a few categories:

- software repository/server and code development platform for work groups in a distributed environment
- platform for proprietary products available for users on a trial basis
- platform for users with low requirements who cannot purchase UNIX system
- distribution platform for Computing Division products and software products used directly by client systems participating in the AFS file system
- mail reader

1.6 Additional Documents and Information

1.6.1 Fermilab Documents

Unless stated otherwise, the Fermilab documents that we reference throughout this manual can be found on the World Wide Web (a.k.a. *WWW* or *the Web*), an on-line information resource to which we provide an introduction in section 3.2.1. A limited selection of documents is also available in printed form on the shelves outside the former Computing Division library, Wilson Hall, 8th floor, NE corner.

As a general rule throughout this manual, we do not provide the URL[1] for each referenced Fermilab document due to the volatility of the addresses, we only state the title and document number. You can find the documents in the Computing Division document database as described below, where they are maintained.

In addition, lots of information has been gathered under the *UNIX Resources* Web page, organized by topic. This page is linked under the heading **Documentation & Software** on the Computing Division home page.

The principal URLs you may need to reference while navigating the Computing Division Web pages are listed in section 1.6.2.

1. *URL* stands for Universal Resource Locator. A URL is a document's address on the World Wide Web; URL is defined in section 3.2.1 under **WWW Basics**.

Computing Division Document Database

The database contains both UNIX-related and non-UNIX documents. Most of the documents are available for viewing and/or printing in either HTML (see section 3.2.1) or PostScript format. A keyword search facility is provided, which currently allows a one-word entry.

Starting from the *Fermilab at Work* Web page, you can reach the Computing Division document database by selecting in turn:

1) *Computing Division* under the heading **Divisions & Sections** (or select *Computing* under **Work Resources**) to reach the Computing Division home page

2) *Documentation* under the heading **Documentation & Software**

From the document database, under the heading **Computing Division Documents**, you can find documents in the categories of General Documentation (document numbers starting with GG), General UNIX (GU), Division Recommendations (DR) and some other designations.

Software Products Information

Also from the document database, under the heading **Product and Application Information**, you can find documentation for many Fermilab software products. The document numbers correspond to the product numbers, and start with PM (Physics and Math Programs), PU (Utility Programs), and other designations.

Alternatively, you can find UNIX product documentation from a variety of sources directly by product name. Starting at the Computing Division home page, first select *UNIX Applications* under **Documentation & Software**, then select *product documentation* under the heading **Additional Resources and Documentation for UAS Products** to reach the *Fermilab Product Documentation Area*.

1.6.2 Useful URLs

Fermilab home page:

`http://www.fnal.gov/`

Fermilab at Work page:

`http://www.fnal.gov/faw/`

Computing Division home page:

`http://www.fnal.gov/cd/`

Computing Division document database:

`http://cddocs.fnal.gov/cfdocs/productsDB/docs.html`

KITS Product Status Report:

`http://www.fnal.gov/docs/products/kits-report.html`

UNIX Resources page:

`http://www.fnal.gov/cd/UNIX/UnixResources.html`

Fermilab Product Documentation Area page:

`http://www.fnal.gov/docs/products/`

1.6.3 Commercially Available Texts

There are several good books on UNIX. The vendors also have UNIX documentation, for example the Silicon Graphics *IRIS-4D User's Guide* and Sun's beginner's guides. Printed reference documentation for the commands is typically just hard copy of the UNIX on-line help, known as *man pages*, described in section 3.1. Some popular books are:

- *UNIX System V: A Practical Guide* by Mark Sobell, published by Benjamin/Cummings
- *UNIX in a Nutshell* by Daniel Gilly et al, published by O'Reilly & Associates, Inc.
- *UNIX for the Impatient* by Paul W. Abrahams and Bruce R. Larson, published by Addison-Wesley
- *UNIX for VMS Users* by Philip Bourne, published by Digital Press (Note that it has a predictable and detectable bias towards Digital's flavor of UNIX!)

The O'Reilly & Associates, Inc. publishers provide specialized texts on many UNIX applications in addition to their general UNIX texts. These are generally regarded as excellent references.

From the *UNIX Resources* Web page, see *Recommended books* under **The UNIX Operating System** for information on what's available in the stockroom. These books can be found in many bookstores, too.

Chapter 2: Getting Started on a UNIX System

This chapter describes the login and logout procedures, and discusses some basic UNIX features including the prompt, special keys, and special characters. We present a brief discussion of the file systems you are likely to encounter at Fermilab, and how to change your password according to your file system and installation.

2.1 Logging In

There are different ways to access a UNIX system:

- Log into the workstation console directly (in other words, it is on or under your own desk). Consult the instructions for your workstation.
- Connect to a UNIX machine over the network; enter[1]:

  `telnet` ***host***

- For X-terminals see section 9.5.

The system will prompt for your *login name* (or *username*) and for your *password*. Below is a sample login session from an X terminal to node FSGI01 of the FNALU system (note that the system prompt is set to the node name under FUE):

```
Connecting to host "fsgi01"......success.


IRIX System V.4 (fsgi01)

login: username
Password:(password is not shown)
IRIX Release 5.2 IP7 fsgi01
Last login: Tue Oct 24 09:27:14 by UNKNOWN@dcdx03.fnal.gov
Terminal Type is xterm


There are no available articles.
<fsgi01>
```

☞ The login name must be all lower case characters, but the password may contain upper and lower case characters. Be sure to enter them in the correct case, because **UNIX is case sensitive**. Note that if you do use upper case to log in, UNIX may assume you have an upper-case-only terminal and you will have very limited capability. If you do so, either log out and log back in again, or enter the command:

---

1. **telnet** is described in section 13.2. Also described there is the utility **rlogin**, which can be used instead of **telnet**.

% stty -lcase

When you log in, a series of login scripts is run to define the functionality of your terminal and to set up your environment. These start-up files have been customized under FUE and are called the *Fermi files*. They are listed in Appendix C. The Fermi files provide the common environment described in Chapter 9.

At login, the system will type out:

```
Terminal Type is {termtype}
```

where `{termtype}` represents the best guess depending on exactly how and from where you entered the system. To change this terminal type, see section 9.7.

2.1.1 C Shell Family

The C shell runs two files when you log in, first a file named `.cshrc` and then a file named `.login`. They are both located in your home directory. See section 9.4.1 for details.

2.1.2 Bourne Shell Family

In the Bourne shell family the `.profile` and `.shrc` files, located in your home directory, are run when you log in.[1] Sometimes **ksh** runs `.kshrc` in place of `.shrc`. See section 9.4.2 for details.

2.2 Logging Out

The logout commands are slightly different for the two shell families. For either one, you can create a `.logout` script and use it in place of the standard commands.

If you are on a system running AFS, it is best to issue the command **unlog** before logging out. See section 7.3.4 for an explanation.

2.2.1 C Shell Family

You can logout with either:

% logout

or

% exit

If you have other processes running you will be informed that you have stopped jobs. You can continue to enter **logout** until all the processes are terminated.

The control character **eof**, which is usually set to <Ctrl-d>, is the normal UNIX way of logging off. Since this is easily entered accidentally, the standard Fermi UNIX Environment disables this way of logging out by including the command **set ignoreeof** in the Fermi files.

1. Bourne shell (**sh**) does not run `.shrc`. Sometimes (more common recently) vendors have linked **sh** to **ksh** (see section 9.4.2), effectively replacing **sh** with **ksh**.

2.2.2 Bourne Shell Family

You can logout with:

$ exit

or (on some systems):

$ <Ctrl-d>

The control character **eof**, which is usually set to <Ctrl-d>, is the normal UNIX way of logging off. Since this is easily entered accidentally, the standard Fermi UNIX Environment includes the command **set ignoreeof** in the Fermi files to disable it (*ignoreeof* is not supported on all Bourne shells).

If you have other processes running you will be informed that you have stopped jobs. You can continue to enter **exit** until all the processes are terminated.

2.3 The UNIX Prompt

After you log in successfully, you will get a prompt. The default UNIX prompt usually indicates your default shell (see Chapter 4 for information on shells). Typically, a **%** indicates the C shell (**csh**) and a **$** indicates the Bourne shell (**sh**). The Fermi files set the prompt to the machine name, which is most likely what you'll see.

You can change your default prompt by altering your start-up files.

In the C shell, include in your `.login` file:

set prompt='*newprompt* '

In the Bourne shell, you need to set the value of the keyword shell variable *PS1*. This is just a variable that is declared and initialized by the shell at start-up. Include in your `.profile` file:

export PS1; PS1="*newprompt* "

2.4 Special Keys

Like all systems, UNIX has a number of special keys that perform particular functions. Some important ones are the keys necessary to backspace over a character when entering a command, to delete the whole line being entered, and to interrupt execution. These are user-specifiable, and have different defaults based on the shell, the version of UNIX, and the login files described in section 9.4. If you are using the Fermi files (most UNIX systems at Fermilab have FUE installed and thus do use these files), these special keys will be set as described in the table (notice the footnote labelled *a* that follows the table):

Name	Control Char	Function
erase	DEL or back-space	Erase character. Backspace and erase one character (the key used depends on terminal setting). Sometimes, especially within **tcl/tk** applications, you must use <Ctrl-h>.
werase	<Ctrl-w>	Delete the rightmost word typed in.
kill	<Ctrl-x>	Kill (erase) the line typed in so far.[a]
intr	<Ctrl-c>	Interrupt the program currently running.
rprnt	<Ctrl-r>	Reprint the line typed in so far.
flush	<Ctrl-o>	Stops terminal output until you press a key.
susp	<Ctrl-z>	Suspend the program currently running and put it in the background. **This does not stop the process as it does in VMS!**
stop	<Ctrl-s>	Stop the display. To resume, press the **start** key
start	<Ctrl-q>	Start the display after stop.
eof	<Ctrl-d>	Send the program an end-of-file character.

a. If you prefer to use <Ctrl-u> for this function, uncomment the line `#stty kill '^u'` in your `.login` file (C shell family), or `#stty kill` in your `.profile` file (Bourne shell family).

To display the current settings for your terminal, enter:

```
% stty -a
```

If the keys don't seem to work as described here or you want to change them, refer to section 9.7.

UNIX relies on the hardware tabs of your terminal. If they are not set or if they are set in an unusual way, displays may appear strange on your terminal. You can set the tabs manually on your terminal, or you can use the **tabs** command to set them. The command with no arguments:

```
% tabs
```

will set tabs in the usual UNIX way, 8 spaces apart.

Special Note Regarding Backspace and Delete Keys

In some unusual circumstances of setup and keyboards you may also need to issue this set of commands to get the backspace key to work as expected:

```
% stty erase "^?"

% stty intr "^C"

% stty kill "^X"
```

Sometimes there is trouble with the delete key. Adding the following text to your `.profile` or `.login` will make the key useful, even on X terminals from different places on different platforms:

```
case $TERM in
vt100)
    stty erase \^? ;;
xterm)
    case "`xdpyinfo | grep ‘vendor string’`" in
        *DigitalEquipmentCorp*)         stty erase \^? ;;
        *Network\ Computing\ Devices*)  stty erase \^H ;;
        *Silicon\ Graphics*)            stty erase \^H ;;
        *)                              stty erase \^H ;;
    esac
    ;;
esac
```

Special Note for ksh Users Regarding Arrow Keys

To make the up and down arrow keys work and therefore to enable command line editing and recall in **ksh**, include the following lines in your `.shrc` file (or `.kshrc`):

```
set -o emacs
alias __A='^P'
alias __D='^B'
alias __B='^N'
alias __C='^F'
```

Note that the `A`, `D`, `B`, and `C` are preceded by *two* underscores, and that you need to insert an actual control character, not simply carat-P or carat-B. A control character typically needs to be preceded by a "quoting" character, which differs from editor to editor.

For this (these) editor(s):	... enter this immediately before the control character:
vi	<Ctrl-v>
emacs	<Ctrl-q>
nu/TPU, fermitpu	<Ctrl-v> <Ctrl-v>
EDT+	<Ctrl-[>
NEdit	Use Insert Control Character from the Edit menu.

We believe this prescription works on all UNIX operating systems, regardless of how you're connected (e.g., **telnet**, **xterm**).

2.5 Special Characters (Metacharacters)

Slashes

The backslash (****) character is used on the UNIX command line to mask the special meaning of the character immediately following it (no spaces in-between) so that the command interpreter takes the character literally. It is called a *quoting* character. For example:

`%` ***command* \<CR>**

causes the carriage return to be ignored, allowing you to continue typing your command on the following line.

The forward slash (**/**) character is the symbol for the root directory. In path names it acts as a separator between directories in the hierarchy, and between the last directory and the file, if one is specified. For example:

/rootdir/.../userdir/subdir1/subdir2/filename

Quotes and Parentheses

Different types of quotes have special meanings.

Normal single quotes (apostrophes) around a string (**'string'**) tell the command interpreter to take the string (**string**) literally. Double quotes around a string (**"string"**) also tell the command interpreter to take the string literally, but allow interpretation of variables that follow a **$** character (**$** preceding a variable name outputs the value of the variable; see section 9.1). Section 5.1.2 further explains single and double quotes in command interpretation, and provides an example.

Single backquotes around a command string (**`string`**) tell the interpreter to run the command(s) in the string, and to use the output of the command(s) in place of the string itself. This is useful for combining two commands into one, and for doing iterative tasks within shell scripts (shell scripts are introduced in section 4.4).

A string of commands enclosed in parentheses (e.g., ***(command1;command2)***) is run in a subshell. In section 5.1.1 we discuss the difference between shell commands and non-shell commands. A non-shell command always runs in a separate shell (a subshell); when enclosed in parentheses, the command starts a second subshell.

Command Separators

The semicolon (**;**) character separates successive commands on a single command line. For example,

% ***command1*** **;** ***command2***

executes ***command1***, and when it finishes, ***command2*** gets executed.

The ampersand character (**&**) is similar to **;** but does not wait for ***command1*** to finish.

A double ampersand (**&&**) runs ***command2*** only if ***command1*** was successful.

Piping commands (the pipe symbol **|**) is discussed in section 5.4.3. A pipe tells ***command2*** to use the output of ***command1*** as input.

A double pipe (**||**) runs ***command2*** only if ***command1*** was *unsuccessful*.

Other Special Characters

Special characters are used in file expansion (section 6.2.2). Note that to prevent file expansion, these characters must be prefaced by a backslash (****). Metacharacters are also used in input/output redirection (section 5.4.2), and in regular expressions (section 5.4.5) as wildcards, delimiters, and other special pattern-matching characters. Refer to these sections for specific information.

2.6 File Systems: Standard UNIX and AFS

The standard UNIX file system is a hierarchy of directories descending from what is known as the *root directory*. UNIX allows parts of the directory hierarchy, also called file systems, to reside on separate storage devices or in separate disk partitions. These are accommodated by means of mount points. A *mount point* is a directory in a file system that corresponds to the root directory of some other file system. The primary file system is the one starting at the true root. The standard UNIX file system is described in Chapter 6.

If the UNIX machine that you work on is part of an integrated system of UNIX machines, for example a LAN (local area network), it is likely that a distributed file serving system has been installed on it. A distributed file system provides a common directory structure and thus the same view of the file system to all participating nodes. This overrides the standard UNIX file system.

Fermilab is using the AFS (Andrew File System) as a distributed file service model, and it is installed on several machines at Fermilab in a production environment, including the FNALU cluster. See Chapter 7 for a discussion of AFS.

A special note to CDF and D0 users: Your UNIX systems are *not* configured to use AFS.

2.7 Information Distribution System: NIS

NIS (Network Information System) is a system that distributes information throughout a cluster. You may also know this by the name *yellow pages*.

Note that the word *cluster*, as used in this manual in reference to UNIX, is not the same as a VAX cluster. We define a UNIX cluster as a group of machines that share both a common password file (or user database), and a common file system, especially for login directories. NIS is usually used to provide the common password file, and the common file system is typically NFS or AFS.

NIS is installed on FNALU and many other UNIX clusters at Fermilab. In order to determine if NIS is running on your system, execute the command:

```
% domainname
```

If it returns a value, then NIS is running on your cluster. If no output is returned, then it is not. Many UNIX clusters use NIS to share a common login area across several machines. Note that it is possible for both AFS and NIS to be installed on a single system.

2.8 Changing Your Password

For any password changes, you will be prompted for your old and new passwords. Your password must be at least six characters long and should not be your login name or any simple permutation of it. We recommend that you limit your password to eight characters, especially if it is used as an AFS password (see section 7.3.2). It is advisable to mix letters and digits in your password.

If your system is configured in a way not covered in the following sections, you may need to contact your local system manager to determine which password(s) to change.

Man pages are available for the commands described below: **passwd**, **yppasswd**, and **kpasswd**.

2.8.1 Standard UNIX Password

This section assumes that AFS is not running on your system.

Your UNIX password is used to log you into the system. This password is stored in one of two ways, depending on the configuration of your system.

If your system uses the standard UNIX file system, and NIS is not installed, your UNIX password is part of your local password file. In this case you need to execute the generic UNIX password-changing command to change your password:

```
% passwd
```

The system will prompt you for the necessary information. Your password is only changed for the machine, or node, on which you execute this command.

If NIS is running on your system, your UNIX password is stored in the NIS password file which is shared by all machines on the NIS cluster. In this case, you need to use the command **`yppasswd`** to change your password. The system will prompt you for the necessary information. When you change your password in this way, it changes for all the machines (nodes) that are part of your cluster.

2.8.2 Kerberos (AFS) Password

Read section 7.3.2 to understand how a Kerberos (AFS) password works. You can change your Kerberos password using the command:

```
% kpasswd
```

The system will prompt you for the necessary information.

This command changes your Kerberos password for all systems that run AFS on-site.

Chapter 3: Information Resources

This chapter introduces you to the information facilities available from UNIX. Standard UNIX on-line help is available via the man pages. We discuss the World Wide Web and newsgroups, which are very rich sources of information on a virtually unlimited set of topics. A few utilities that allow you to get information about vendor products, other users, and the Fermilab computing systems are also covered. Finally, we include instructions for communicating with the Fermilab Helpdesk.

3.1 UNIX On-Line Help

3.1.1 man Pages

On-line help for UNIX system commands and utilities is in the form of *man pages* (*man* stands for manual) which consist of an on-line version of the UNIX documentation set (often called the UNIX Programmer's Reference Manual). You access the man pages with the **`man`** command.

☞ Note that the man pages differ in many instances between UNIX platforms.

The man Command

When you need help on a known command, use the general **`man`** command format:

`% man [`*part*`]` *topic*

where ***topic*** is generally a UNIX command. **man** is really the on-line manual which is divided into several parts. ***part*** is a digit between 0 and 9. If you know in advance which part contains the information you want, you can speed the search by specifying it. More often than not you will just enter:

`% man` *topic*

The word *print* in **man** entries usually means *display on the screen*. Don't be confused by this. Several options are available with the **man** utility, described under **`man man`**. The **`man`** command normally displays complete manual pages that you select by name. One-line summaries can be selected by either by keyword (**`-k`** option), or by the name of an associated file (**`-f`**). These options are described in section 3.1.2.

A typical initial **man** screen can be seen by issuing the command:

`% man ls`

where **`ls`** is the UNIX command to list files in a directory.

```
     ls(1)                    User Commands                    ls(1)



     NAME
          ls - list contents of directory

     SYNOPSIS
          ls [ -abcCdfFgilLmnopqrRstux1 ] [ names ]

     AVAILABILITY
          SUNWcsu

     DESCRIPTION
          For each directory argument, ls lists the  contents  of  the
          directory;  for  each file argument, ls repeats its name and
          any other  information  requested.   The  output  is  sorted
          alphabetically  by  default.  When no argument is given, the
          current directory is listed.   When  several  arguments  are
          given,  the  arguments  are  first sorted appropriately, but
     /tmp/mpa002Zf
```

Man pages are typically formatted with the UNIX text processing utility **nroff** (or **groff**). These utilities are covered in most UNIX texts. If you find that the **man** page is unformatted, run **`setup groff`**, and then rerun the **`man`** command.

Note that built-in shell commands are described under the topic corresponding to their shell.[1] (See section 5.1.1 for information on built-in commands.) For example, to get information on the command **`alias`** for your current shell, you would enter **`man`** ***shell*** (e.g., **`man bash`**) and search there for information on **`alias`** using the ***/pattern*** function described below.

There is an alternate, "quick and dirty" method to verify the format of a command and get a listing of its options. It doesn't work with all commands, but is usually worth a try. Simply enter the command with an illegal option (try **/** or **?** or **.**). For example,

`% ls -/`

will produce the output:

```
ls: illegal option -- /
usage: ls -RadCLHxmnlogrtucpFbqisf [files]
```

The man Command for AFS Commands

For AFS commands, it works a little differently. AFS commands are discussed in section 7.4. The **man** page for an AFS command is found by entering:

`% man fs_`***command***

Note the underscore (**_**) between **`fs`** and the rest of the AFS command; the underscore is only used with the **`man`** command.

---

1. Some platforms provide **man** pages for built-in commands, however in general you may find it easier to look in a reference book!

Manipulating man Pages

man displays the information using your *$PAGER* environment variable, which under FUE is set to **less** (see section 5.4.4). Therefore, **man** pages are normally piped to **less.** The command **`man less`** will give you more information about manipulating **man** via the **less** filter. **less** gives you one page at a time and lets you enter commands at the prompt to control what it does after each page. For example, you page forward with the Spacebar and page backward using **`b`**.

Once in the **man** environment, you can search for patterns by entering the ***/pattern*** option at the command line. The first instance of the string ***pattern*** will appear in the top line of the screen. To find additional instances of the pattern in the text, simply enter a slash (**/**).

To exit from **man** enter **`q`** (for quit).

Printing man Pages

To print a **man** page, you can use the pipe feature (**|**) along with recommended print formatting and printing commands. These features are covered in Chapter 8. As a suggestion, pipe the output of the **`man`** command to **`a2ps -m`** (to convert **man** pages to PostScript format) and then pipe that output to the print command **`flpr`**:

% **man** ***command*** **| a2ps -m | flpr [*options*]**

This formats the output nicely in landscape, two pages to a sheet.

3.1.2 Finding the Right Command

If you don't know exactly what command you need, try the **`-k`** option with a keyword.

% **man -k** ***keyword***

This displays the **man** page name, the section number in the UNIX documentation, and a short description for each **man** page whose name line contains ***keyword***. For example to find a search utility, enter:

% **man -k search**

The system will output several records similar to the following:

```
conflict(8) - search for alias/password conflicts /usr/local/lib/mh/conflict [-m
glookbib(1) - search bibliographic databases glookbib [ -v ] [ -istring ] [ -tn
ident(1) - identify files ident [ -q ] [ \&.\|.\|. ] ident searches for all occu
lkbib(1) - search bibliographic databases lkbib [ -v ] [ -ifields ] [ -pfilename
lsearch(1) - See if a list contains a particular element lsearch ?mode? list pat
lsearch(n) - See if a list contains a particular element lsearch ?mode? list pat
zgrep(1) - search possibly compressed files for a regular expression zgrep [ gre
..
..
```

Some UNIX systems have an additional utility, **apropos**, which can be used to locate commands by keyword lookup:

% **apropos** ***keyword***

apropos ***keyword*** is equivalent to:

% **man -k** ***keyword***

The **`-f`** ***filename*** option for **`man`** prints the manual entry summaries which might pertain to the given filename(s). Any leading pathname components are stripped from the filename before the filename is matched against the summaries. Here is an example using the **`-f`** option, followed by the output:

```
% man -f /etc/passwd
```

```
passwd (1)            - change login password and password attributes
passwd (4)            - password file
```

The summaries are gotten from the **whatis** database. You can run **`whatis`** ***command(s)*** to look up a given command and obtain the header line from the manual section. You can then run **`man`** to get more information on the command. If the line starts `name(section)...` you can do **`man`** ***section name*** to get the documentation for it.

3.1.3 Vendor Product Documentation

Most vendor product documentation is now available on-line, viewable with an X-terminal or workstation.

Flavor:	Command:
AIX	**`info`**
IRIX	**`insight`**
Digital UNIX (OSF1)	**`dxbook`**
Solaris (SunOS)	**`answerbook`** (only usable on Sun workstations at Fermilab)

3.2 The Internet

The Internet is a global network of networks that provides access to hundreds of thousands of computers around the world. As the reach of the network has grown, so has the number of services accessible. The main tools that allow the user to navigate through the Internet are:

telnet	to access remote hosts (see section 13.2)
ftp	to retrieve data files (see section 13.1.1)
Mail	to send mail (note that Web browser mail handlers are *not* covered in Chapter 12)
WWW	to browse the World Wide Web (see section 3.2.1)
News	to scan the numerous Usenet news groups (see section 3.2.3)

There are two ways to reference an Internet host: an alphabetic name and a series of numbers. The alphabetic sequence is called the *host name* (e.g., fnsg01.fnal.gov) and the numeric one is called the *IP address* (e.g., 131.225.8.178). At Fermilab all host names end with *fnal.gov*, where this suffix is called the *domain name*. Since hosts may change their IP addresses, it is a good practice to always use the host name.

An excellent introduction to the Internet services is *The Whole INTERNET, User's Guide and Catalog*, published by O'Reilly & Associates.

3.2.1 The World Wide Web (WWW or "the Web")

Our primary way of delivering information to you, especially relatively static information, is via the World Wide Web (also known as WWW or "the Web"). There are a number of reasons why the Web has become the defacto standard for information delivery at Fermilab, within the HEP community, and even in the computer-literate parts of the world of business:

- The Web is very strong at pulling together information from different places, and of different formats and types (native HTML[1], PostScript, newsgroups, and even VAXNEWS), and making it appear as a seamless whole.
- It is relatively easy to make information available on the Web.
- Browsers (the programs you use to view information on the Web) are available for every supported platform at Fermilab, making it an excellent fit to the distributed environment in which we now work.

Accessing the Web

In order to access the Web, enter (or put in your login script):

```
% setup www
```

Web browsers work best on workstations that support graphics, so if possible you should have on your desktop a workstation, X terminal, PC or Macintosh rather than a "dumb" terminal. If you have a graphics terminal, use either **Netscape** or **Mosaic**[2]. To invoke them on an X terminal enter:

```
% netscape [&]
```

or

```
% mosaic [&]
```

The ampersand (**&**) is discussed in section 5.5.2. In order to use an X application (which these products are), you must have defined your *$DISPLAY* variable correctly. See section 9.2.

If you don't have a graphics terminal, then you must use one of the line-mode programs, **lynx** or **www**. **lynx** is generally thought to be the better of the two. To invoke them, enter:

```
% lynx
```

or

```
% www
```

This section is not intended to provide detailed instruction on the use of any particular Web browser. Once you get any of them running (at least the graphical ones) there is more information than you will ever want available under Help. In addition, from the *UNIX Resources* Web page you will find information both on using browsers and on the files that control what filetypes the browser recognizes. These files are called `.mailcap` and `mime.types`; see section 9.6 for information on them.

1. HTML stands for HyperText Markup Language, the standard language for documents accessed on the Web.
2. Development on **Mosaic** ended in 1996. Many newer features of the Web are not available on **Mosaic**, and in some cases will cause it to crash. In particular, **Mosaic** does not support tables. We recommend using **Netscape**.

Web Basics

Web browsers find information based on URLs (Universal Resource Locators) which are like addresses and which take you to the top level of a Web site, often called a *home page*.

A home page usually refers to the first page of a commercial, educational, government or personal Web site. Each home page can have several layers or pages that it links to, thereby creating a whole Web site. But the home page is generally the first place you would look. It is like looking at the cover of a book and its table of contents at the same time.

The native WWW addresses are of the form:

```
http://address/
```

or

```
http://address/something.html
```

The first part is the protocol, `http` in this case. (A protocol is a set of rules computers observe to exchange information. `http` stands for the HyperText Transfer Protocol; think of it as the identifier of a Web page.) Instead of `http` you might also see `ftp`, `file`, `gopher`, `mailto`, and `news`.

Next you'll see a colon and two forward slashes (except for `mailto`, which has a different format).

Next comes the computer's address in the format described in section 3.2. A computer address, or domain name, is used by computers in routing data across the many networks that make up the Net.

Finally, you often see a directory path at that computer plus a file at the end of the path. Web page files usually end in "`html`", for HyperText Markup Language, although you may also see "`htm`", "`shtm`", or "`shtml`". HTML is the simple scripting language that tells browsers how to display the various elements of a Web page such as links, body text, header text, inline graphics, and external files.

Many HTML files contain links to other documents. Sometimes links are text; sometimes they are images. If a link consists of text, it is underlined and may be in color. You can tell your cursor is on a link when a URL appears at the bottom of the screen. If a link consists of an image, you'll see a URL when you move your mouse pointer across it.

Several Web search "engines" can help you find information you are seeking, and they vary in the number of URLs they contain in their database, how deep they go into Web sites indexing information, what they index, and how frequently they "crawl" or "walk" the Web in surveying sites. There's a valuable Swiss site at URL `http://cuiwww.unige.ch/meta-index.html` that collects several WWW search tools.

Subject directories make it increasingly easy to find information about broad subjects. An excellent directory is one called Yahoo at URL `http://www.yahoo.com/`, a well-constructed directory of tens of thousands of Web pages.

Writing your Own Web Pages

If you want to start writing Web pages in HTML format, see *Creating Web Pages*, available under *Web Help* on the Computing Division home page.

AFS

A note for Web page providers on AFS systems: Set the permissions for `system:anyuser` to `rl` on directories containing files that you want to make accessible via a Web browser. See section 7.6.2 for information on AFS directory permissions.

Fermilab Home Page

Your browser is most likely configured to have either the *Fermilab Home Page* or the *Fermilab at Work* page as your default home page. If not, the *Fermilab Home Page* can be simply accessed as: `http://www.fnal.gov/`. You may want to change your default home page to *Fermilab at Work*: `http://www.fnal.gov/faw/`. To do this, set your environment variable *WWW_HOME* as follows (environment variables are discussed in section 9.1.), depending on your shell:

C shell family:

```
% setenv WWW_HOME http://www.fnal.gov/faw/
```

Bourne shell family:

```
$ WWW_HOME="http://www.fnal.gov/faw/"; \ export WWW_HOME
```

From *Fermilab at Work* you can find information on Fermilab's different divisions and experiments, activities, schedules, etc. Navigate to the Computing Division's pages to find information on UNIX.

3.2.2 UNIX Help on WWW

The man pages can sometimes be cryptic, unwieldy, or both. As an alternative, set a bookmark in your Web browser to *UNIXHelp*, accessible under the heading **The UNIX Operating System** on the *UNIX Resources* page. Here you'll find easy-to-follow instructions on the use of many UNIX features, organized into four categories: Tasks, Commands, Concepts, and Utilities.

Also from the *UNIX Resources* page you'll find *The UNIX Reference Desk* under the heading **Other UNIX Resources**. The resources in this document are divided into the following classes: General, Texinfo Pages, Applications, Programming, IBM AIX Systems, HP-UX Systems, Unix for PCs, Sun Systems, X Window System, Networking, Security, Humor.

3.2.3 Newsgroups

Usenet News (or NetNews) is a way of communicating "articles" among people world-wide. In general, information in newsgroups is volatile information, whereas information in Web pages is of longer term. We have a server here at Fermilab which receives articles from elsewhere and posts the articles originating here. Fermilab has its own newsgroups named `fnal.xxx`. CERN's groups are prefixed with `cern` and SLAC's with `slac`. General information and especially important information is posted in `fnal.announce`. NALCAL, seminar announcements, and the like are posted in `fnal.announce.seminars`. Computing Division forms are in `fnal.announce.forms`. UNIX discussion articles are posted in `fnal.comp.unix`. There are many more newsgroups, both Fermilab groups and others, that you might find of interest.

In order to make a newsreader available on UNIX, enter (or put in your login script):

```
% setup news
```

A number of readers will be made available to you. Line-mode browser commands are **`nn`**, **`rn`**, and **`trn`**. X-based browser commands are **`xrn`** and **`knews`**. **`News`** may also be read from most WWW browsers and some mail readers. The readers keep track of the newsgroups that you are interested in ("subscribed to") as well as which articles in each newsgroup you have read; all of the UNIX readers cooperatively maintain this information, so you can use different readers at different times without losing this information.

3.3 The Info Utility

Info is a facility available to the system support people to communicate with you about events regarding the Fermilab computing systems (shutdowns, for example), or other systems-related information that is newly available. To get a list of the **Info** messages, enter the command **`Info`**. To read an item, enter the command with the ***nametag*** of the item found on the left side of the **Info** list:

```
% Info nametag
```

Note the capital I! If the item is more than one page, press the space bar to continue. Press **`q`** to quit.

3.4 Other Users: WWW Directories, finger and who

WWW Directories

From the Fermilab at Work page, directories are available to point you to information about Fermilab personnel and the high energy physics community at large. These directories typically contain general information such as email addresses, phone numbers, and office locations.

finger

finger is used to find out about other users. It searches for matching account names and first or last names, if known. Depending on the vendor implementation, it may display the name of the person associated with each account, the login name, the home directory and login shell, the contents of the file `.plan` in the person's home directory, and possibly other information such as waiting mail, and time of last login. If the person is logged in, it also may display information about his or her current sessions.

Note that each vendor has a different implementation of **finger**. In addition, for security reasons many sites disable the output of **finger** over the network. It is therefore unwise to rely on the format, content, or even the availability of **finger** as a tool for finding out about users or their accounts.

The format of the **`finger`** command is:

```
% finger [options] [name...]
```

where ***name*** can be a part of a personal name or a username. If you specify the option **`-m`**, then ***name*** is matched only to account name and not the rest of the personal name.

We encourage you to create a `.plan` file. It is just a text file in which you might include information such as your office location, phone numbers, mail station, home address, schedule, emergency numbers, and so on.

finger can often be used to look up users on a remote machine by specifying the name in the standard internet form *user@host*. This form works on VMS machines with **MultiNet** running, but in this case ***name*** must be the username; otherwise not much useful information is obtained.

As an alternative, point your favorite browser to the location `http://fnal.fnal.gov/finger/`.

who

The command **`who`** lists certain information about the users on your system.

```
% who
```

If used with the **`-q`** option, only the names of the logged in users and the number of users are displayed.

The **`who am i`** form identifies the invoking user. The command format is:

```
% who am i
```

There are a number of options which you can read about in the man pages.

3.5 The Fermilab Helpdesk

The Fermilab Helpdesk (a.k.a. Customer Support) is available to answer questions related to the supported computer systems and software on site. Keep in mind that its first priority is to maintain central systems and networks, and to ensure that Fermilab-supported software is available and usable. Therefore a request which impacts only one individual may not receive immediate attention.

Customer Support is in service Monday through Friday, 9:00 a.m. to 5:00 p.m. You are encouraged to use email for all communications that are not urgent.

Helpdesk Email Address

During business hours:	*helpdesk@fnal.gov*
During off-hours:	*operator@fnal.gov*

Helpdesk Web Page

You can request help and/or keep track of actions taken on your requests from this page. From the Computing Division home page, select *Customer Support* under the heading **Services**.

Helpdesk Phone Number

630-840-2345

During off-hours, you can leave a phone message, or “escape” to Data Center Services (Operations) for requests requiring immediate attention.

Helpdesk Location

FCC 1 West

Chapter 4: Shells

This chapter discusses the concept of a UNIX shell, and how to manipulate shells. It includes information on the available and recommended shells and their features. The concept of a shell as an interpretive programming language is introduced.

4.1 Introduction to Shells

The kernel is the real operating system and is loaded into memory at boot time. Typically the user never interacts directly with the kernel. The utilities are programs stored on disk, and loaded into memory by the kernel when invoked.

A shell is a utility. It is run in user mode, and does not have system privileges. You have a default shell, and you can invoke other shells. Invoking shells is discussed in section 4.1.2.

The shell is the interface between the operating system and the user. It interprets the commands you type and the keys you press in order to direct the operating system to take an action. Shell scripts, which are analogous to DCL command files, allow you to use the shell as an interpretive programming language. They are introduced in section 4.4, but a comprehensive treatment of scripts is beyond the scope of this manual.

There are two families of shells: one based on the Bourne shell (this family also includes the Korn (**ksh**) and Bourne-again (**bash**) shells), and the other based on the Berkeley/C shell. The shells themselves will be discussed and compared in section 4.2.

4.1.1 Determining Your Current Shell

There are several commands available in all the shells that furnish this information. We present four examples below with sample output for **csh**. The first three, **`echo`**, **`env`** and **`finger`**, will show only your login shell. If you have invoked another shell, these commands will not reflect the new shell. **`ps`** lists information about all your active processes.

`%` **`echo $SHELL`**

displays the value of the variable name that follows the **`$`**; example output: `/bin/csh`

`%` **`env or printenv`**

shows all defined environment variables, including SHELL; example output: `SHELL=/bin/csh`

`%` **`finger`** ***your_username***

shows user information and login shell; example output:

```
Login name: username                    In real life: {your name}
Directory: /afs/fnal.gov/files/home/room3/{username} Shell: /bin/csh
```

Finally,

% ps

shows processes, including shell; example output:

```
PID TTY      TIME COMD
6264 pts/11   0:03 csh
```

☞ Note that on some of the more recent OS releases (e.g., AIX+4 and IRIX+6.4, and likely others in the near future) `/bin/sh` is a link (links are described in section 6.3.5) to the korn shell (**ksh**). **ksh** is a superset of **sh**, so this shouldn't present any problems for you. One difference is that your `.shrc` file (see section 9.4.2) gets sourced when you run `/bin/sh` scripts.

4.1.2 Starting a Shell

A shell is started by a login process. A new shell is also started for each invocation of a terminal window or shell script (see section 4.4). Which shell gets invoked is determined by the last field in your entry in the password entry file. In a standard UNIX file system you can display your password entry by the command:

% grep ^*username* /etc/passwd

(**grep** is described in section 6.4.2.; the use of **^** is explained in section 5.4.5.) To display your password entry in an NIS environment, use the command:

% ypmatch *username* passwd

(The NIS command **ypmatch** is not described in this manual.) Sample **ypmatch** output from the FNALU system, for which the default shell has been set to **csh**, looks like:

```
aheavey:!:6302:1525:Name of User:/afs/fnal.gov/files/home/room3/aheavey:/bin/csh
```

When you log in, the login process invokes a shell program (e.g., `/usr/local/bin/tcsh` or `/usr/local/bin/bash`) and transfers control to it. The shell displays a prompt indicating it is ready for your input. The default UNIX prompts are symbols that indicate which shell is invoked (recall from section 2.3 that your prompt is likely to be set differently):

- **%** for the C shell family
- **$** for the Bourne or Korn shells

On FNALU the prompts are set to indicate the host machine, for example **<fsui01>**, or **<fsgi02>**. At any point in your session you can invoke another copy of the same shell or a different shell by typing the shell name at the prompt, for example:

% csh

invokes **csh** (C shell). This new shell, or "subshell", sits on top of your current shell. The execution of the original shell is then suspended (the shell is put to sleep), and the new shell takes control. Upon quitting the new one, the original shell wakes up and resumes control.

The average user at Fermilab does not have the privilege to change the password entry file. Therefore, to change your default shell you will need to ask your system manager.

4.1.3 Exiting a Shell

To exit a shell and return to the calling shell, type **exit** at the prompt. Repeat the **exit** command once for each subshell; when you reach your initial shell, your terminal emulation is closed, and the terminal window disappears. Instead of **exit** you may need to enter <Ctrl-d>.

4.2 Features of Available Shells

This section is excerpted from *Shell Choice, A shell comparison* (dated September 28, 1994) by Arnaud Taddei of CERN. His eleven-page document contains a brief description of the six major shells and provides an excellent comparison of features between the shells. It is available on the Web at `http://consult.cern.ch/writeup/shellchoice`.

Of the six major shells, four are in the Bourne family: **sh**, **ksh**, **bash**, and **zsh**; and two are in the Berkeley/C family: **csh**, **tsch**.

The most up-to-date shells are **tcsh** (Berkeley/C), and **bash** and **zsh** (Bourne). These are also the three shells that are public domain (as opposed to vendor-supported). The public domain shells are the same on all platforms, which is not true of vendor shells. This is desirable when attempting to homogenize user environments. **Note that zsh is not supported at Fermilab.**

Some of the common features of these newer shells are:

- specific startup files
- startup files are the same for any platform
- specific shell variables
- specific built-in commands

The **tcsh** is essentially an enhanced **csh**. Some additional features of the **tcsh** are:

- enhanced completion[1] mechanism (programmable for commands, file names, variable names, user names, etc.)
- multiline editing capabilities (command line editing using **emacs** or **vi**-style key bindings)
- enhanced file expression syntax
- spelling correction (see section 5.3)
- enhanced prompt
- step up/down through history list

The following table should give you an idea of the virtues of each of the shells supported at Fermilab. It is adapted from one in Taddei's document referenced above. More complete feature lists for all the shells can be found there.

++	good
+	existing
-	weak
--	absent

Criteria	**sh**	**ksh**	**bash**	**csh**	**tcsh**
Configurability	-	+	++	+	++
Execution of commands	+	+	+	+	++
Completion	--	+	++	+	++
Line editing	-	+	++	-	++

1. This feature allows you to uniquely specify a file without typing in its whole name.

Criteria	**sh**	**ksh**	**bash**	**csh**	**tcsh**
Name substitution	+	+	++	+	++
History	--	+	++	+	++
Redirections and pipes	+	+	+	+	+
Spelling correction	--	--	--	--	+
Prompt settings	+	+	+	+	++
Job control	--	+	+	+	+
Execution control	+	+	+	+	+
Signal handling	+	+	+	-	-

4.3 Supported/Recommended Shells at Fermilab

On many systems at Fermilab, **tcsh** is used as the default shell. The Computing Division currently supports **csh**, **tcsh**, **sh**, **bash** and **ksh**. **tcsh** or **bash** is recommended for interactive use, and **sh** for scripts. (The C shell family is not recommended for scripts due to inconsistent syntax at different levels of nesting.) **zsh** is not currently supported. From the *UNIX Resources* Web page you can access UNIX support information under **UNIX Products** to find out about any support policy changes.

4.4 Shell Scripts

As mentioned above, a UNIX shell can be used as an interpretive programming language. Besides executing shell commands within the script, you can:

- create and use variables
- process (read) arguments
- test, branch, and loop
- perform I/O

A *shell script* is a file containing a sequence of commands which can be executed by the shell, and flow control commands. The same syntax is used for commands within scripts as for interactive command entry. Section 5.1 explains briefly how the system runs and interprets shell scripts.

Although you can write complex programs using the shell language, you can also create simple shell scripts for running long commands or a series of commands that you use frequently.

In every shell script you write, include the special characters `#!` followed by the pathname of the shell as the first characters in the file. This indicates (a) that this is a script rather than a compiled executable, and (b) which shell to invoke to run the script.[1]

1. On some of the more recent OS releases (e.g., AIX+4 and IRIX+6.4, and likely others in the near future) `/bin/sh` is a link to the korn shell (**ksh**). Therefore on these platforms, the `.shrc` file (see section 9.4.2) gets sourced for any script starting with `#!/bin/sh`.

For example:

```
#!/usr/local/bin/bash
```

at the start of the script invokes **bash** to run it. A `#` found anywhere else in the script is interpreted as the beginning of a comment, and the shell ignores all characters between the `#` symbol and the next newline character.

An introductory reference for script-writing with examples can be found in *UNIXHelp* on the Web. You can get to it via the *UNIX Resources* page under **The UNIX Operating System**.

Note that in order to execute the script, regardless of shell, the script file must have execute permission for the appropriate users (see section 6.6.1 for a discussion of permissions). After you set this permission, the shell will need to rebuild its "hash table" to include the new script. The hash table is a table of executables that the shell recognizes.

To complete these two operations, enter:

% **chmod a+x** ***filename***

% **rehash**[1]

To run a script, the shell must be able to locate it. If its directory is in your path (see section 9.2), you only need to type the script's filename to run it. If not, you can type the the filename preceded by **./** on the command line (the **./** explicitly tells the shell to look for the executable file in the current working directory). Typing the full path of the filename will work too, although it is perhaps the most cumbersome way of telling the shell where the script is. Here we illustrate the three ways to invoke a script:

% ***filename***

% **./*filename***

% ***/full_path/.../filename***

Once the shell locates the script, it interprets and executes the commands in the file one by one.

You may want to maintain a `$HOME/bin` directory for all your programs and shell scripts, and include this directory in your path[2]. The shell uses this variable to locate commands and other executables.

It is important to remember that, like all UNIX commands that are not part of the shell (see section 5.1.1 for an explanation of shell commands), the script file executes in a subshell forked[3] by the parent shell. This subshell retains any environment variables defined in the script as well as any shell variables defined in the file `.cshrc` or `.shrc` (one of these two files may be executed automatically prior to the script, depending on your shell; see section 9.4). At the end of the script, control returns to the parent shell, and any definitions made by the subprocess are *not* passed back to the parent process.

To execute a script for which you do want to pass back changes to the parent shell (for example, setting new shell variables), the syntax for execution differs. For the C shell family, execute the script by typing:

% **source** ***filename***

---

1. The command in **sh** is **hash**; not necessary in other shells.
2. Under FUE, the Fermi files add your `/bin` directory to your PATH.
3. Under UNIX, the term *fork* means create a new process.

For the Bourne shell family, type:

$. ***filename***

The **source** or **.** command executes the script in the context of your current process, so that you can affect this current process, in contrast to normal command execution.

For instance, after you make changes to your `.cshrc` or `.login` file, you can use **source** or **.** to execute it from within the login shell in order to put the changes into effect.

4.5 Other Interpretive Programming Languages

We have mentioned that each UNIX shell can be used as the interpreter for its own programming language. Other interpretive languages supported at Fermilab are **perl** (provided in the FUE **shells** product), and **gawk** (a version of **awk**; see section 5.4.4). These languages are beyond the scope of this manual. The O'Reilly & Associates, Inc. publishers provide excellent reference texts on them.

Chapter 5: Important UNIX Concepts

This chapter introduces you to the UNIX command structure, and to many important commands and concepts. The features introduced in this chapter constitute the core of the UNIX operating system, and many of these tools are quite powerful and flexible. Some of the features are shell-specific, and we provide the distinctions where necessary.

5.1 Processing Environment

5.1.1 Programs, Commands and Processes

A *program* is an executable file. A program is invoked by entering its filename (which is the *command* associated with the executable), often followed by options, arguments, and/or parameters on the command line. The shell allows three types of commands:

- an executable file that contains object code produced by a compilation of source code
- an executable file that contains a sequence of shell command lines (a *shell script*)
- an internal shell command (*built-in* command)

The first two command types may include standard UNIX utilities, commercial products, and user-written programs. All the shells allow both *interactive* command entry in which the commands are typed at the keyboard and executed one by one, and *scripted* entry in which commands are put in a file, called a shell script, and executed sequentially when the script is run. See section 4.4 for a brief discussion of the uses of shell scripts and how to execute them.

Shells execute commands by means of *processes*. A process is an instance of a program in execution. A process can interact with the kernel by invoking a well defined set of *system calls*. The system calls instruct the kernel to perform particular operations for the calling program and they can exchange data between the kernel and the process. For example, a process can use system calls to create new processes and terminate running processes.

When a terminal session begins, the operating system starts a single *parent process*. Creating a new process from an existing process is called *forking*. This new process is called a *child process* or *subprocess*. Each process has a unique process identification number (PID). A subprocess can fork another process and become a parent. A process which is not receiving input from the terminal, either running or stopped, is said to be in the *background* (see section 5.5). The **`ps`** command can be used to print the status of active processes. See the man pages for information about its options.

When you give the shell a command associated with a compiled executable or shell script, the shell creates, or *forks*, a new process called a *subshell*. The new process runs the system call *exec* which invokes yet another program to execute the command in place of the current process (the subshell). Unless the subprocess runs in the background, the parent process remains dormant until its subprocess completes or is stopped. Then control returns to the parent.

To execute most built-in commands, the shell forks a subshell which executes the command directly (no *exec* system call). For the built-in commands **cd**, **set**, **alias** and **source**, the current shell executes the command; no subshell is forked. You can, however, cause the shell to fork a process by enclosing the command in parentheses. The following example illustrates this (use of the semicolon is described in section 2.5; and the commands **cd** (change directory)and **pwd** (print working directory) are described in section 6.5):

`% cd /dir1; pwd`	displays **/dir1** (no subshell is forked)
`% (cd /dir2; pwd)`	due to the parentheses, a subshell is forked, then the commands are issued; displays **/dir2**. Control then returns to the parent process.
`% pwd`	displays **/dir1** since the current process was unaffected by the previous command line.

Most built-in commands exist in all shells, but there may be differences regarding arguments, options, or output format between the shell-specific versions of each command. Some commands for a given shell are not available on all platforms. Refer to a UNIX text for lists of built-in commands.

You do not need to distinguish between built-in and other commands to execute them. However in order to find help in the man pages, you do need to know which is which. Help on shell commands is usually found under the shell name, for example under **man tcsh** or **man bash**. Some platforms provide man pages for built-in commands, however in general you may find it easier to look in a reference book! Help on other commands is found directly under **man** ***command***.

5.1.2 Command Interpretation by the Shell

When the shell receives a command, it interprets it in a series of three (for Bourne shell family) or four (for C shell family) passes. Naturally, if the command is an alias (see section 9.3), it requires an additional pass up front for substitution.

- The first pass for the C shell family looks for the **!** character, and replaces it with the previous command (see section 5.3 for information on command recall).
- The next pass (the first pass for the Bourne shell family) replaces wildcards (used in filename expansion, redirection, and regular expressions; see sections 6.2.2, 5.4.2, and 5.4.5, respectively).
- The next pass looks for the **$** character in order to replace variable names with their values (see section 9.1).
- The final pass splits the command line elements by whitespace to arrive at the final, literal command that the shell must execute.

There are ways to prevent interpretation of special characters in each of these passes. Preceding a character with a backslash (****) works for all special characters; wildcards can be enclosed in single or double quotes; variables can be enclosed in single quotes; and whitespace is ignored if the argument containing the whitespace is enclosed in single or double quotes.

To illustrate the operations that take place in each pass, the following table presents a series of three examples using the **echo** command and the same string, first in single quotes, then double quotes, and finally with no quotes. The **echo** command writes the string to standard output. Assume that the files that match **q*** are `qq` and `qqq`, and the value of the variable *a* is **foo**.

Command -->	`echo 'q* $a x'`	`echo "q* $a x"`	`echo q* $a x`
After first pass[a], only wildcards are interpreted.	`echo 'q* $a x'` (no wildcard expansion due to quotes)	`echo "q* $a x"` (no wildcard expansion due to quotes)	`echo qq qqq $a x` (unquoted wildcard is expanded)
After second pass, unquoted or double quoted variables are replaced by their values.	`echo 'q* $a x'` (no variable replacement due to quotes)	`echo "q* foo x"` (double-quoted variable `$a` replaced by value)	`echo qq qqq foo x` (unquoted variable `$a` replaced by value)

After final pass, command string is broken up according to whitespace. The separate elements are listed vertically.	`echo`
	`'q* $a x'`
	(string treated as one argument due to quotes)
		`echo`
		`q* foo x`
		(string treated as one aregument due to double quotes)
			`echo`
			`qq`
			`qqq`
			`foo`
			`x`
			(no quotes; each argument treated separately)

When you type in the original command, the system returns the string:	`q* $a x`	`q* foo x`	`qq qqq foo x`

a. This would be the *second* pass for C shell family; there were no **!** characters to replace.

5.2 Command Entry

A UNIX command is either a built-in command or the name of an executable file which the operating system will load and execute. When you see the prompt, you can enter a command by typing the command name, any options and arguments, followed by a carriage return.

Recall, the formats displayed in this manual use **`this font style`** to indicate characters to be typed as is, and ***this font style*** to indicate arguments to be substituted. Arguments enclosed in square brackets, **`[...]`**, are optional.

You should be aware that UNIX commands are not noted for their consistency of format. Furthermore, commands, formats, arguments, and options may vary slightly from one UNIX flavor to another. In this manual, we attempt to be as generic as possible, and describe options that are widely available.

UNIX commands are described on-line in the man pages (see section 3.1).

5.2.1 Command Format

The basic format of UNIX commands is:

`%` ***command −option(s) argument(s)*** where:

`%`	is the (default, non-FUE) **csh** prompt.[1]
command	is the UNIX command name of a utility or tool.
option(s)	modifies how the command runs; options are nearly always preceded by a dash and listed one after another. See example below.
argument(s)	specifies data or entities (usually files) on which the command is to operate; arguments are separated by blanks ("white space").

Remember, UNIX is case-sensitive. Therefore UNIX commands must be entered in the correct case. Most of the time commands are entered in lower case.

The components are separated by at least one blank space. If an argument contains a blank, enclose the argument in double quote marks. Normally, options can be grouped; e.g., the **`-lw`** and the **`-l -w`** option specifications are equivalent in the examples below (**`wc`** is a sample command; it lists line, word, and/or character count of one or more files.):

`%` **`wc -lw`** ***file1 file2***

`%` **`wc -l -w`** ***file1 file2***

Some options can have arguments, and there isn't consistency on whether there should be a blank space between the option and its argument. Check the man pages when you're not sure. In the next example which shows the FORTRAN command, ***outputfile*** is the argument of the option **`-o`**:

`%` **`f77 -o`** ***outputfile program*****`.f`**

Looping and conditional commands are also supported. These are more advanced shell commands and are not covered in this manual. Consult a UNIX text for information on these.

5.2.2 Miscellaneous Command Line Features

- To correct typos you can use the erase key (Delete or Backspace) to erase character-by-character, or the Kill key to kill an entire line (see section 2.4).
- More than one command can be entered on a line if the commands are separated by semicolons. The commands will be executed sequentially. See section 2.5 for more information on using multiple commands on one line.
- If you need to continue a command to a new line, you can either keep on typing (without doing a carriage return), or enter a backslash (****) followed directly by a carriage return (no space in-between) and then continue typing on the next line. (Recall the backslash is used to prevent a special character's meaning to be interpreted by the shell. See section 2.5.)
- You can use parentheses to group commands. Since a subshell is created for each group, this can be used to prevent changing the current environment. It can also be used to redirect all output from the commands considered as a group (see section 5.4.2).
- Type ahead works, even if the characters get interspersed with output.

1. **`$`** is the non-FUE default for Bourne shell.

5.3 Command Recall

Command recall is quite different in each shell. One common feature for all shells that support command recall is the **history** mechanism. It maintains a list of commands that have been entered and allows them to be reexecuted. The *history* variable, set to some number at login time in the start-up files, determines the number of commands that are saved in the list. The *savehist* variable specifies how many commands are to be saved for your next session after you log out. The **`history`** command displays the list of saved commands:

```
% history
```

We discuss the following shells separately: **csh**, **tcsh**, and **bash/ksh**. There is no command recall facility for **sh**.

csh

There is no command line editing native to **csh**. Before describing the standard **csh** command recall facility, we should mention a Fermilab product called **cedit** that we recommend for use with **csh** instead. It was designed to mimic VMS line editing, and turns out to provide similar command recall and editing functionality to **tcsh**. To use **cedit**, you need to set it up initially. Enter:

```
% setup cedit
```

To execute it, type:

```
% m
```

followed by <Return>. **`m`** stands for *modify*. Use the up or down arrow keys to scroll to the desired command. The right and left arrow keys and your backspace key allow you to edit the command before reexecuting it. There are several control characters that perform functions within **cedit**. Typing <Ctrl-i> in **cedit** displays the available commands.

Recalling history commands using standard **csh** syntax is fairly easy. Use the commands listed below.

`!!`	Reexecute the previous command
`!`*n*	Reexecute command ***n*** from the history list
`!`*text*	Reexecute the most recent command beginning with ***text***
`!?`*text*`?`	Reexecute the most recent command containing ***text***

For example, to reexecute the 4th command from the history list, enter:

```
% !4
```

and to reexecute the last command starting with **`ls`**:

```
% !ls
```

The dollar sign (**`$`**) can be used to recall the last word of a command. **`!$`** causes substitution of the last word of the last command. For example, you can check the contents of `myfile.f` and then compile it using the following command sequence:

`% less myfile.f`

`% f77 !$`

A couple of nice features you can use with these reexecution commands are *preview* (**p**) and *substitute* (**s**). To substitute a string in the previous command and preview it before execution, use the syntax:

% !:p:s/*oldstring*/*newstring*

To do the same for the ***n***th command in the history list, use:

% !*n*:p:s/*oldstring*/*newstring*

To execute after previewing (and/or substituting), simply type:

`% !!`

tcsh

Recalling commands is easy if you are using **tcsh**. The up/down arrows on the keyboard can be used to recall commands and the left/right arrows can be used to move around within the command to edit it (VMS users will be familiar with this concept).

A command line correction algorithm is available in **tcsh**. To enable it, enter:

`% set correct=all`

This causes all words on the command line to be checked. If any part gets corrected, the system notifies you, and gives you a chance to accept or reject it. For example, say you type in:

`% lz /usr/bin`

The system will return with:

```
CORRECT ls /usr/bin (y|n|e|a|)?
```

Where `y`=yes, `n`=no, `e`=edit, and `a`=abort. You must provide one of these responses.

To turn off command line correction, enter:

`% set correct=none`

ksh

Two styles of command recall are supported; **emacs** and **vi**. The style is determined in one of two ways:

- include the line **set -o *editor*** in either your `.profile` or `.shrc` file, where ***editor*** is either **emacs** or **vi** (this takes precedence if variables below are set differently)
- set either the *EDITOR* or *VISUAL* environment variable to one of these editors

When set to **emacs**, use the usual **emacs** commands to display and modify previous commands, for instance <Ctrl-p> for previous line. When set to **vi**, command recall is initiated by typing the Escape (or <Ctrl-[>) key. Then all the standard **vi** commands can be used. Some of the basic **vi** and **emacs** commands are listed in section 11.3.

bash

Both **csh** and **ksh**-style recall are supported.

5.4 Important Concepts

This section attempts to provide an overview of a few of the important concepts in UNIX which are very different from other systems and may therefore be confusing to the novice user. In order to be able to make effective use of UNIX, these concepts need to be understood.

5.4.1 Path

When you issue a command, the shell program parses the command line and either processes it directly or searches for an executable file with that name in any of the directories specified in your *search path*, which is controlled by the variable *PATH*. If the file is not found in any of the directories in your search path, the shell reports that the command was not found. The file may well be on the disk somewhere, but it is **not in your path**.[1]

FUE attempts to provide an appropriate path, and we recommend that you not change this basic path. However, feel free to *add* directories to it. For the **csh** family, your `.login` file contains a `set path` line for the shell variable *path*.[2] Uncomment this line (remove the `#`) and include additional directories in the shown format:

```
set path=($path /dir1 /dir2... )
```

Or change the environment variable *PATH* (also in `.login`), as follows:

```
setenv PATH "${PATH}:/dir1:/dir2"
```

For the **sh** family, uncomment and add directories to the `PATH` line in your `.profile` file:

```
PATH=$PATH:/dir1:/dir2...
```

See section 9.2 for information on the *PATH* variable.

As an aside, if you add an executable to one of the directories in your search path, it may be necessary for you to either log out and log back in, or to recreate the internal tables used by the shell with the **`rehash`** (**csh**) or **`hash`** (**sh**) command (see section 4.4).

5.4.2 Standard Input and Output Redirection

The shell and many UNIX commands take their input from *standard input* (`stdin`), write output to *standard output* (`stdout`), and write error output to *standard error* (`stderr`). By default, standard input is connected to the terminal keyboard and standard output and error to the terminal screen.[3]

The way of indicating an end-of-file on the default standard input, a terminal, is usually <Ctrl-d>.

Redirection of I/O, for example to a file, is accomplished by specifying the destination on the command line using a *redirection metacharacter* followed by the desired destination.

---

1. This concept will be familiar to users of MS-DOS.
2. Shell versus environment variables are discussed in section 9.1.
3. VMS users would know these as the logical devices SYS$INPUT, SYS$OUTPUT, and SYS$ERROR.

C Shell Family

Some of the forms of redirection for the C shell family are:

Character	Action
>	Redirect standard output
>&	Redirect standard output and standard error
<	Redirect standard input
>!	Redirect standard output; overwrite file if it exists
>&!	Redirect standard output and standard error; overwrite file if it exists
|	Redirect standard output to another command (pipe)
>>	Append standard output
>>&	Append standard output and standard error

The form of a command with standard input and output redirection is:

% ***command*** **–[*options*] [*arguments*] < *input file* > *output file***

If you are using **csh** and do not have the *noclobber* variable set (see section 9.2), using **>** and **>&** to redirect output will overwrite any existing file of that name. Setting *noclobber* prevents this. Using **>!** and **>&!** always forces the file to be overwritten. Use **>>** and **>>&** to append output to existing files.

Redirection may fail under some circumstances: 1) if you have the variable *noclobber* set and you attempt to redirect output to an existing file without forcing an overwrite, 2) if you redirect output to a file you don't have write access to, and 3) if you redirect output to a directory.

Examples:

% **who > names**	Redirect standard output to a file named `names`
% **(pwd; ls -l) > out**	Redirect output of both commands to a file named `out`
% **pwd; ls -l > out**	Redirect output of **ls** command only to a file named `out`

Input redirection can be useful, for example, if you have written a FORTRAN program which expects input from the terminal but you want it to read from a file. In the following example, `myprog`, which was written to read standard input and write standard output, is redirected to read `myin` and write `myout`:

```
% myprog < myin > myout
```

You can suppress redirected output and/or errors by sending it to the *null device*, **/dev/null**. The example shows redirection of both output and errors:

```
% who >& /dev/null
```

To redirect standard error and output to different files, you can use grouping:

```
% (cat myfile > myout) >& myerror
```

Bourne Shell Family

The Bourne shell uses a different format for redirection which includes numbers. The numbers refer to the file descriptor numbers (0 standard input, 1 standard output, 2 standard error). For example, **`2>`** redirects file descriptor 2, or standard error. **`&n`** is the syntax for redirecting to a specific open file. For example **`2>&1`** redirects 2 (standard error) to 1 (standard output); if 1 has been redirected to a file, 2 goes there too. Other file descriptor numbers are assigned sequentially to other open files, or can be explicitly referenced in the shell scripts. Some of the forms of redirection for the Bourne shell family are:

<table>
<tr><th>Character</th><th>Action</th></tr>
<tr><td>></td><td>Redirect standard output</td></tr>
<tr><td>2></td><td>Redirect standard error</td></tr>
<tr><td>2>&1</td><td>Redirect standard error to standard output</td></tr>
<tr><td><</td><td>Redirect standard input</td></tr>
<tr><td>|</td><td>Pipe standard output to another command</td></tr>
<tr><td>>></td><td>Append to standard output</td></tr>
<tr><td>2>&1|</td><td>Pipe standard output and standard error to another command</td></tr>
</table>

Note that **<** and **>** assume standard input and output, respectively, as the default, so the numbers 0 and 1 can be left off. The form of a command with standard input and output redirection is:

$ *command* –[*options*] [*arguments*] < *input file* > *output file*

Redirection may fail under some circumstances: 1) if you have the variable *noclobber* set and you attempt to redirect output to an existing file without forcing an overwrite, 2) if you redirect output to a file you don't have write access to, and 3) if you redirect output to a directory.

Examples:

`$ who > names`	Direct standard output to a file named `names`
`$ (pwd; ls -l) > out`	Direct output of both commands to a file named `out`
`$ pwd; ls -l > out`	Direct output of **`ls`** command only to a file named `out`

Input redirection can be useful if you have written a FORTRAN program which expects input from the terminal and you want to provide it from a file. In the following example, `myprog`, which was written to read standard input and write standard output, is redirected to read `myin` and write `myout`.

```
$ myprog < myin > myout
```

You can suppress redirected output and/or error by sending it to the *null device*, **`/dev/null`**. The example shows redirection of standard error only:

```
$ who 2> /dev/null
```

To redirect standard error and output to different files (note that grouping is not necessary in Bourne shell):

```
$ cat myfile > myout 2> myerror
```

5.4.3 Pipes

UNIX uses the concept of a *pipe* to connect the standard output of one program directly into the standard input of another program. This is specified by separating the two commands with the pipe operator, the vertical bar (**|**). The general format is:

`%` ***command1*** **|** ***command2*** **|** **. . .**

where, of course, each command can have options and arguments. To implement pipes of commands, the shell forks off multiple processes. For example if you run the command:

```
% history | more
```

the shell forks twice; the grandchild runs **`history`**, the child runs **`more`** (after hooking up the right file descriptors to the right pipe ends), and the parent shell waits for the process to finish. The **`history`** command, a built-in, is implemented in the grandchild shell process directly, while the **`more`** command requires an *exec* system call.

The **`tee`** command can be used to send output to a file as well as to another command.

```
% who | tee whoout | sort
```

This creates a file named `whoout` which contains the original **`who`** output. It also sorts the **`who`** output and sends it to standard output, the terminal screen. The following example sends the (unsorted) **`who`** output to the file and the screen:

```
% who | tee whoout
```

5.4.4 Filters

A *filter* is a command or program which gets its input from standard input, sends its output to standard output, and may be used anywhere in a pipeline. Examples of filters are the UNIX utilities:

- **more** (and **less**)
- **grep**
- **awk**
- **sort**

The combination of UNIX filters **grep**, **awk**, and **sort** and the use of pipes is very powerful.

more and less

The **more** filter allows you to display output on a terminal one screen at a time. You press Spacebar to move to the following screen, and **`q`** to quit.

less is a much more flexible variant of the standard UNIX utility **more** and is provided under FUE[1]. The command **`less`** lists the output (e.g., specified files) on the terminal screen by screen like the command **`more`**, but in addition allows backward movement in the file (press **`b`** to go back one full screen) as well as forward movement. You can also move a set number of lines instead of a whole page. To view a file with the **less** filter, enter:

`% less [`*options*`] [`*filename*`]...`

1. FUE sets your environment variable *PAGER* to the **less** filter.

The options and usage are described in the man pages for **more** and **less**.

After displaying a page of information, **more** and **less** display a colon prompt (**:**) at the bottom of the screen and wait for instructions.

```
     LESS(1)                  UNIX System V                  LESS(1)



     NAME
          less - opposite of more

     SYNOPSIS
          less [-[+]aABcCdeEimMnqQuUsw] [-bN] [-hN] [-xN] [-[z]N]
               [-P[mM=]string] [-[lL]logfile] [+cmd]
               [-ttag] [filename]...

     DESCRIPTION
          Less is a program similar to more (1), but which allows
          backwards movement in the file as well as forward movement.
          Also, less does not have to read the entire input file
          before starting, so with large input files it starts up
          faster than text editors like vi (1).  Less uses termcap (or
          terminfo on some systems), so it can run on a variety of
          terminals.  There is even limited support for hardcopy
 :
```

You can search for patterns in the file by entering ***/pattern*** at the **less** prompt. Continue to search for the same pattern using a slash (**/**). A further advantage is that **less** does not have to read the entire input file before starting, so with large input files it starts up faster than text editors like **vi**.

grep

The **grep** filter searches the contents of one or more files for a pattern and displays only those lines matching that pattern. **grep** is described in Section 6.4.2.

awk

awk is much more than a filter; it is a powerful pattern scanning and processing language. Although you will need to spend a little time learning how to use **awk**, it is very well suited to data-manipulation tasks. It handles internally what you would have to handle laboriously in a language like C or FORTRAN. You can do in a few lines what would take many, many lines of FORTRAN.

awk works best when the data it operates on has some structure, for example a document with heading levels, or a table. In the case of a table, you can tell it the field separator (spaces, colons, commas, tabs) and it can align and interpret the contents of the field according to the way you use it. Or you can reorder the columns, or change rows into columns and vice-versa.

We present here some very basic information to get you acquainted with the concepts of **awk**, but you will need a more in-depth reference in order to use this utility. A widely-available book on **awk** is *The awk Programming Language* by Aho, Kernighan, and Weinberger, Addison-Wesley. Another good reference, from which much of the information in the present section is extracted, is *sed & awk* published by O'Reilly & Associates.

There are several versions of **awk**, and they differ from platform to platform. "Old" **awk** may be **awk** or **oawk**, "new" **awk** may be **nawk**. FUE provides a GNU version of **awk** called **gawk** as part of the **shells** product.

Some of the features of **awk** are:

- Ability to view a text file as made up of records and fields in a textual database
- Use of variables to manipulate the database
- Use of arithmetic and string operators
- Use of common programming constructs such as loops and conditionals
- Ability to generate formatted reports

With **nawk**, additional features make it easier to write larger scripts. Using **nawk** you can:

- Define functions
- Execute UNIX commands from a script
- Process the result of UNIX commands
- Process command-line arguments more gracefully
- Work more easily with multiple input streams

awk executes a set of instructions for each line of input. You can specify instructions on the command line or create a script file. Input is read a line at a time from one or more files or from standard input. The ***instructions*** must be enclosed in single quotes to protect them from the shell. (Instructions always contain curly braces which are interpreted as special characters by the shell.) We refer you to one of the books on **awk** for the available instructions. We'll use the instruction **`print`** in our examples.

For command lines, the syntax is:

`% awk '`*instructions*`'` ***files***

As an example, say that file `test` contains only the line `Hello, world`. The command:

`% awk '{ print }' test`

produces the output:

```
Hello, world
```

Multiple command lines can be entered by separating commands with semicolons or using the multi-line input capability of the Bourne shell. **awk** programs are usually placed in a file where they can be tested and modified. The syntax for invoking **awk** with a script file is:

`% awk -f '`*script*`'` ***files***

where **`-f`** indicates that the filename of a script follows.

awk interprets each line of the input data file(s) as a record, and each word on that line, delimited by blank spaces or tabs, as a field. You can reference these fields, either in patterns or procedures. **`$0`** represents the entire input line; **`$1`**, **`$2`**, ... refer to the position of individual fields on the input line. As an example, say that the file `personnel` contains a list of employees' first names, last names, and addresses. The command:

`% awk '{ print $1}' personnel`

would produce output of the type:

```
John
Alice
Mary
Eric
```

To use the pattern-matching features of **awk**, you need to be familiar with the metacharacters used in regular expressions (see section 5.4.5). A pattern is enclosed between forward slashes (**/**) on the command line or in a script. When **awk** reads an input line, it attempts to match each pattern-matching rule in a script. Only the lines matching the particular pattern are the object of an action. If no action is specified, the line that matches the pattern is printed (executing the **`print`** statement is the default action).

In our `personnel` example, let's assume that Alice is from Illinois (IL) and Eric is from Iowa (IA). To bring up the complete records with the pattern IL or IA, we could issue the command:

```
% awk '/I./{print}' personnel
```

where the metacharacter **.** matches any single character. We could more simply type:

```
% awk '/I./' personnel
```

and get the same result in either case:

```
Alice Jones 834 S. Jefferson St., Batavia, IL 60510
Eric Smith 24 Birch St., Albert City, IA 50510
```

In Appendix D we present **awk**'s programming model, which is beyond the scope of the present section. This model will help you understand the potential that **awk** offers the programmer.

sort

sort sorts the lines of the specified files, typically in alphabetical order. Using the **`-m`** option it can merge sorted input files. Its syntax is:

% sort [*options*] [*field-specifier*] [*filename(s)*]

For example, start with the `personnel` file contents:

```
John Smith 75 South Ave., Denver, CO 80145
Alice Jones 834 S. Jefferson St., Batavia, IL 60510
Mary Fahey 901 California St., San Francisco, CA 94121
Eric Smith 24 Birch St., Albert City, IA 50510
```

Run the command:

```
% sort personnel
```

to reorder the file contents as follows:

```
Alice Jones 834 S. Jefferson St., Batavia, IL 60510
Eric Smith 24 Birch St., Albert City, IA 50510
John Smith 75 South Ave., Denver, CO 80145
Mary Fahey 901 California St., San Francisco, CA 94121
```

sort is very easy to use. Read the man page for **sort** to see what the available options are and how to specify the sort fields. If a field is not specified, the sort key is the entire line. The sorted output goes to standard output by default.

5.4.5 Regular Expressions

A *regular expression* is a string composed of letters, numbers, and special symbols that defines one or more strings. They are used to specify text patterns for searching. This is similar to wildcards on VMS.

A regular expression is said to *match* any string it defines. The major capabilities include:

1) match single characters or strings of characters

2) match any arbitrary character

3) match classes of characters

4) match specified patterns only at the start or end of a line

5) match alternative patterns

Regular expressions are used by **vi**, **grep**, and **awk** (and at least a couple of utilities not covered in this manual, for instance **ed** and **sed**). **grep** in fact stands for **g**lobal **r**egular **e**xpression **p**rinter. For a complete discussion of regular expressions, refer to a UNIX text. To get you started, we include a table of special characters that can be used in expressions.

Note that regular expression special characters are different from those used in filename expansion.

`.`	Matches any single character Example: _.ing_ matches all strings with any character preceding _ing_; sin*g*ing, *p*ing
`*`	Represents 0 or more occurrences of the preceding character Example: _ab*c_ matches *a* followed by 0 or more *b*’s followed by *c*; *ac*, *abc*, *abbbbbc*
`.*`	Matches any string of characters (`.` matches any character, `*` matches any number of occurrences of the preceding regular expression)
`$`	Placed at the end of a regular expression, matches the end of a line Example: _ay$_ matches *ay* at the end of a line; ... tod*ay*
`^`	Placed at the beginning of a regular expression, matches the beginning of a line Example: _^T_ matches a *T* at the beginning of a line; *T*oday ...
`"`	Delimits operator characters to prevent interpretation
`\`	Turns off special meaning of the following single character (`\` is often called a *quote* character)
`[]`	Specifies character classes
`[...]`	Matches any one of the characters enclosed in square brackets Example: _[bB]ill_ matches *bill* or *Bill*

There is an extended set of special characters available for full regular expressions, including for example `?` and `+`. These can be used in **egrep** and **awk**. Refer to a UNIX book for information.

5.5 Job Control

Any command you give to the shell (true for all shells except **sh**) is a *job* and is given a *job number*. A single command is the simplest job. A series of commands separated by semicolons, or commands piped together, create a single job. A script also creates a single job. A job may consist of many processes, because each command is a process.

The job stays with its environment, for example, the current directory. If you subsequently change directories after putting a job in the background and then resume the background job, you will be in the original directory again.

Job control allows you to work on several jobs at once, switching back and forth between them at will, and it allows you to stop, start, and kill them. When you start up a job interactively, it is by default in the *foreground* and attached to your terminal. You can move that job into the *background* so you can start up another job or observe another job that is already running. You can move any background job into the foreground so it is once again attached to your terminal. You can run any number of background jobs at any one time, but there can be only one foreground job. The use of multiple windows on an X terminal makes much of this transparent.

5.5.1 Priority

You can control the priority of a command or shell with the shell command **`nice`**:

`% nice [+`*n* | –*n*`] [`*command*`]`

n is the value by which you want to increase or decrease priority. Values range from 1 to 19, with the default at 10. The higher the **`nice`** value, the lower the priority of a process, and the slower it runs. (You are being *nicer* to other users!) If no number is specified, **nice** sets the priority to 4. If ***command*** is omitted, the priority is set for the current shell. If ***command*** is specified, it is run at the specified (or default) priority in a sub-shell. You can use **nice** to lower the priority of a command or shell that makes large demands on the system but isn't needed right away.

☞ Note that another **`nice`** command exists, `/bin/nice`. It is not a built-in shell command. If you do **`man nice`**, you will get information on this one. In order to get information on the shell command **`nice`**, do **`man csh`** (using **csh** as an example).

5.5.2 Background, Foreground, and Suspended Jobs

You run jobs in the *background* so that you can perform other tasks in the foreground (i.e., interactively). Jobs are always in one of three states: running in the foreground, running in the background, or suspended. Any job intended to run in the background should have its output and error redirected to a file.

There are two ways to put jobs into the background:

Using the & Metacharacter

One way to start a job in the background is to append the ampersand metacharacter (**`&`**) to the end of the command line. In the first example, the standard output is redirected to a file (in this case, the syntax is valid for both shell families):

`%` ***command*** **`>`** ***output-file*** **`&`**

Note that the parentheses are necessary in the next example in order to send both commands to the background:

`% (`*`command1`*`;` *`command2`*`) &`

The shell prints a line indicating the job number and process ID of its top-level commands, and the job is started as a background job.

Using the Suspend Control Character

The other way is to use the suspend control character, called *susp* or *swtch*, (see section 2.4) which is usually assigned to <Ctrl-z>. It stops or *suspends* the foreground (the currently running interactive) job, moving it to the background; it does not kill it.

After stopping a job, you can either resume it with the **`fg`** command or make it run in the background with the **`bg`** command (see below). You may want to stop a job temporarily to do another task and then return to it interactively, or you may want to stop it in order to let it finish as a background job.

When a background job terminates, this is reported just before the next prompt (so the message doesn't interrupt the current foreground job).

A background job will stop if it tries to read from the terminal. If output is not redirected, a background job can either (continue to) send output to the terminal or be stopped if it attempts to write to the terminal. The following command can be used to toggle this behavior:

```
% stty [-]tostop
```

The minus indicates negation, meaning that background jobs will continue to run even if they attempt to write output to the terminal and that the output will appear on the terminal screen. However, programs which attempt to interrogate or change the mode of the terminal will be blocked when they are not in the foreground whether or not *tostop* is set.

Listing Jobs

The **`jobs`** command lists your jobs:

```
% jobs [-l]
```

This command lists the background jobs, their job number, status, and the command being executed. A plus sign in the output means that job is *current* (in control of your terminal), a minus sign means that job is *next*. *Current* and *next* refer to its relation to the foreground (see **`fg`**). The **`-l`** option lists the process ID as well.

Commands Used for Controlling Jobs

There are a number of commands to control jobs: **`fg`**, **`bg`**, **`stop`**, **`kill`**. All of them can take an argument which specifies the particular job, or they can have no argument. The argument can take two basic forms: a simple process ID number (as displayed by **`ps`**) or a form prefixed with a percent sign (`%`). If no argument is given, the current job is acted upon.

The `%` form of the argument can be **`%-`** where **`-`** indicates the previous job, `%`***n*** where ***n*** is the job number as displayed by the **`jobs`** command, `%`***pref*** where ***pref*** is some unique prefix of the command name and arguments of one of the jobs, or `%`**`?`*****str*** where ***str*** is some unique string in one of the jobs.

You can use the **`fg`** command to move a suspended or background job into the foreground:

`% [fg] %[`*job*`]`

The **`fg`** is not mandatory. If the job specification is omitted, the current job will be brought into the foreground, and the next job becomes current.

Examples (note that the first `%` on each line represents the default **csh** prompt):

`% fg %5`	Bring job number 5 into the foreground
`% %1`	Bring job number 1 into the foreground
`% %`	Bring the current job into the foreground

After stopping a foreground job, you can start it running in the background with the **`bg`** command. **`bg`** puts the current or specified jobs into the background, continuing them if they were stopped. In the following commands, ***job*** stands for job number.

`% bg %[`*job*`]`

We described above how to stop (suspend) a foreground job with the suspend control character (<Ctrl-z>). Similarly, you can suspend a background job with the **`stop`** command:

`% stop %`*job*

You can abort a suspended or background job with the **`kill`** command:

`% kill %`*job*

If you attempt to exit a shell (logout) when there are stopped jobs, you will get a warning message. A second **`logout`** will log you out if you choose not to see what jobs are stopped before you exit. In the C shell family, background jobs will continue running after you log out.

5.5.3 Scheduling Jobs: at and cron

UNIX provides two methods for running jobs at some specified time.

If AFS is installed on your system, there are Kerberos authentication problems with running programs that spawn jobs external to your login process group (Kerberos authentication is described in section 7.3). **at** and **cron** fall into this category. You can run the job, but it will not run with authentication, and most likely will not be able to write into `/afs` space. A work-around is available for executing an authenticated **cron** job (send mail to *helpdesk@fnal* requesting the full details of this procedure).

at

The first is the **at** utililty. This allows the user to queue a job for later execution.

The format of the **`at`** command is:

`% at` *time* **`[`***date***`] [+`***increment***`]`**

at reads the commands from standard input. Standard output and standard error output will be mailed to you unless they are redirected.

The shell saves the environment variables and the working directory that are in effect at the time you submit the job and makes them available when the job is executed.

- The ***time*** can include 1, 2, or 4 numbers. One or two digits is assumed to be hours, four digits to be hours and minutes. It can be specified as two numbers separated by a colon (hours:minutes), either in 24-hour format or with *am* or *pm* appended. The names *noon*, *midnight*, *now*, and *next* are recognized.
- The ***date*** is either a month name followed by a day number (and optionally a year number followed by an optional comma) or a day of the week (fully spelled out or abbreviated to three characters). The words *today* and *tomorrow* are known. If no date is given, *today* is assumed if the hour is greater than the current hour and *tomorrow* if it is less.

• The optional ***increment*** is a number suffixed by *minutes*, *hours*, *days*, *weeks*, or *years* in singular or plural form.

Examples:

% at 8

% at 0800

% at 8:00am Jan 24

% at now + 1 minute

at reads from standard input, meaning you type in the commands (there may or may not be a prompt). When you are finished, terminate input with <Ctrl-d> followed by a carriage return.

You can also redirect the input to a file of commands, for example:

% at now + 1 hour < myscript

at runs in the Bourne shell (**sh**) by default. If you need to force it to run in C shell, you can use the trick illustrated in the following interactive example:

% /bin/csh << xxxxx

? at now + 2 minutes

? source .cshrc

? alias > aout

? <Ctrl-d> (followed by carriage return)

The first line causes **csh** (C shell) to read the following lines up to **xxxxx** or to the end-of-file. There is no **xxxxx**, of course, so it reads until you give it the <Ctrl-d>. The third line runs your `.cshrc`. It is an illegal Bourne shell command, therefore you can tell **at** ran in the C shell and that your `.cshrc` file was executed. You will receive a message similar to the following, and the results will be mailed to you (alas, **at** will say it's using `/bin/sh` even if you've "tricked" it):

```
warning: commands will be executed using /bin/sh
job 826157640.a at Wed Mar  6 18:14:00 1996
```

After 2 minutes, `aout` is mailed to you. It contains a list of all the aliases defined in your `.cshrc` file. If you are running a script using **at** then the script will be run under whatever shell you specify in the script.

For example, say you run:

% at now + 2 minutes

? *script*

? <Ctrl-d> (followed by carriage return)

where ***script*** is a file that contains the line `#!/bin/csh` at the beginning. The commands in the script will execute under **csh**.

cron

The second method for running jobs at some specified time is the **`crontab`** command. It is designed for jobs that need to be run on a regular basis, e.g., once a night, or once per week. Note that **cron**, like **at**, uses the Bourne shell so that output redirection must be specified using Bourne shell syntax. Scripts will be run under what ever shell is specified in the script. If no shell is specified then Bourne shell is used.

The format of the command is:

% crontab [*filename*]

% crontab [*options*]

where ***filename*** is the name of a file containing the commands that you want to have executed. If you do not specify a file, then **`crontab`** will read commands from standard input as you type them, ending with <Ctrl-d>, and the commands will be run in Bourne shell. The system utility **cron** reads the crontab file and runs the commands. Standard output and standard error will be mailed to you unless they are redirected.

The command can also take the following options:

-r	remove crontab file
-l	list contents of crontab file

A **cron** file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

- minute (0-59)
- hour (0-23)
- day of the month (1-31)
- month of the year (1-12)
- day of the week (0-6 with 0=Sunday)

If an asterisk appears in a field instead of a number, **cron** interprets that as a wildcard for all possible values. The sixth field of a line in a **cron** file is a string that is executed by the shell at the specified times.

Examples:

The user creates a **cron** file `myfile`, and runs **`crontab`**:

```
#Myfile
# Run script that archives to 8mm tape for backup.
# Monday-Thursday at 2200 backup everything that has been
# changed.  Every Friday at 2200 backup everything.
0 22 * * 1-4 /usr/buckley/daily  1>>/usr/buckley/cron/backup.log 2>&1
0 22 * * 5   /usr/buckley/weekly 1>>/usr/buckley/cron/backup.log 2>&1
```

% crontab myfile

This command will perform an incremental backup at 10pm Monday to Thursday and a full backup at 10pm on Friday.

Chapter 6: The UNIX File System

The UNIX file system has a hierarchical or tree-like structure with the directory called *root* (**/**) as its source. The system is essentially composed of *files* and *directories*. In this chapter we describe techniques for manipulating files and directories, and commands designed to provide information about them.

In Appendix E you can find a table of UNIX equivalents for commonly used VMS commands.

6.1 Directory Structure

The UNIX system automatically puts you at a specific location in the file system when you log in. This is called your *login directory*. Typically, this is the same as your *home directory*. The name of your home directory is usually the same as your login name. Within this directory you can create files and additional directories (sometimes called *subdirectories*) in which to group the files. You can move and delete your own files and directories and control access to them.

The root of the file system is called *root* and is written as a slash (**/**). In other words, to change to the root directory, type:

```
% cd /
```

There is only one directory tree on a system even if several devices are mounted in that tree. (All devices are viewed as files.) The *current directory* or *working directory* is the directory that you are currently working in, which is also the directory that commands refer to by default. Files in your current directory can, therefore, be specified by their filenames only.

6.1.1 Pathnames

Wherever you can use a filename, you can also use a *pathname*, which is how you point to files that are not in your current directory. You can refer to files in other directories using either a *relative path name*, that is a path specified relative to your current directory, or with an *absolute path name*, that is a path specified relative to the root of the file system.

Absolute path names are preceded with **/**, the root directory. If a pathname does not begin with **/** it is assumed to be a relative path name. Relative path names begin with a directory or filename, a **.** (pronounced “dot”) which refers to the current directory, or **..** (pronounced “dot dot”) which refers to the directory immediately above the current directory. The character **/** also separates components of the pathname, which are directory names, except for the last one, which can be either a simple filename or a directory name.

In summary, every file has a pathname, and its absolute pathname is of the form:

```
/rootdir/dir2/... /filename
```

The following is the form of a relative pathname of a file:

`dir_n/dir_n+1/... /filename`

An example of an absolute path name is:

`/usr/smith/project1/afile`

If my current directory is `/usr/smith`, then I can refer to the file `afile` in subdirectory `project1` with a relative pathname like this:

`project1/afile`

Or, if my current directory is `/usr/smith/project1`, I can refer to a file named `fileb` in `/usr/smith/project2` as:

`../project2/fileb`

Note that you cannot necessarily tell if `fileb` is an ordinary file or a directory name. Many commands will accept a directory name, and if it *is* a directory name, the command in which it is used may perform the action on all files in the directory. This behavior can be dangerous!

6.1.2 The Home Directory

Your home directory is the top of your personal branch in the file system, and is usually designated by your username, i.e. `/{path}/{username}`.

Tilde (~)

In most UNIX shells other than **sh**, the *tilde* (**~**) stands for the home directory. Used alone, it specifies your home directory. Followed by a different user's login name, it expands into the pathname of the home directory of that user. This is a convenient way to refer to a user's directory, because it is independent of where the system manager may place the directory on the disk.

The use of tilde (**~**) to refer to a home directory is limited. It isn't available in the Bourne shell, and isn't available in FORTRAN.

Of the following three examples, the first refers to the file `def` from your own home directory, the second to the home directory of user *jones*, and the third refers to file `data1` in the subdirectory `project1` of jones' home directory.

`~/def`

`~jones`

`~jones/project1/data1`

To change to jones' home directory you'd enter:

`% cd ~jones`

logdir

FUE provides the command **`logdir`** which returns the full path of the specified user. **`logdir`** by itself returns the path of the invoking user. For example:

`% logdir` ***username***

is equivalent to:

`% echo ~`***username***

To change directories you'd enter (note the use of backquotes to use the output of the enclosed string):

% cd `logdir** ***username*****`

In contrast to the tilde, **logdir** can be used within commands, scripts, FORTRAN and C programs, and other programs in all shells.

$HOME

The environment variable *HOME* is automatically set to the absolute pathname of your home directory. Environment variables are discussed in section 9.1 and *HOME* is described in section 9.2. To see the value of *HOME*, enter:

% echo $HOME

From some other directory, you can change to your home directory or one of its subdirectories using a command like the following:

% cd $HOME[1]

or

% cd $HOME/mysubdir

6.1.3 Command Line Directory Shortcuts

`.`	Current directory
`..`	Parent directory of the current directory ("up" one directory)
~	Your home directory (all shells but **sh**)
$HOME	Environment variable whose value is your home directory
~username	Home directory of another user (all shells but **sh**)
/	Root directory

6.1.4 Directories and Executables

It is appropriate at this point to mention the relationship between directories and commands. A command is simply the name of an executable file, located in some directory. To execute a command, the shell first needs to find the executable file. The shell therefore needs to be given a set of directories to search. This information is provided via the environment variable *PATH* which is a list of search directories. You can display it with the command:

% echo $PATH

PATH is explained more thoroughly in section 9.2. Standard UNIX commands are generally grouped in a few standard directories (e.g., `/usr/bin`), and your default *PATH* contains these. See section 9.2 to learn how to run executables that you create and store in your own directories.

1. **cd** by itself is equivalent to **cd $HOME**.

The utility **which** is useful in cases where a command may be ambiguous, for example due to aliasing (see section 9.3), and you want to know exactly which executable file or files the command runs. **which** lists the files that would be executed if the specified command(s) had been run. The syntax for **which** is:

% which *command* **[***command2* **...]**

Each argument is expanded if it is aliased, and your path is searched for the executable files associated with the commands. See the man page for more information.

6.2 Files

An ordinary file contains ASCII characters or binary data and is considered by the UNIX system to be merely a sequence of bytes. No structure is imposed on the file and no meaning attached to its contents by the system; the meaning depends on the program that reads the file.

A directory file contains an entry for each file in that directory. The directory entry for a particular file contains the file name and *inode number*. The inode number is a volume data structure used by the file system. It has an associated entry in the *inode table* which contains other information about the file such as the owner, file protection, modification date.

A *hidden file* is an ordinary file whose name begins with a period (called "dot"). The `.login`, `.cshrc`, and `.logout` files described in Chapter 9 are hidden files. The reason they are called hidden is that the **ls** (list files) command does not list them by default. Use the **-a** option with **ls** to see them. Hidden files do not appear in filename expansion of *, either. Filename expansion is discussed in section 6.2.2, below.

VMS

UNIX does not support file versions. If you edit a file and save it with the same name, your earlier version is overwritten. Similarly, if you copy or rename (move) a file to a filename that already exists, the original file is overwritten.

6.2.1 Filenames

A full file specification has only two parts, the directory specification and the file name. A filename is composed of from 1 to 14 characters in old UNIX implementations and a much larger number in more recent versions (up to 255, typically). Although you can use any character in a filename except **/**, UNIX assigns special meaning to many characters (metacharacters), so they should be avoided (see section 2.5). It is safe to use the upper- and lowercase letters, numbers, dash (-), underscore (_), period (**.**), and comma (**,**). As mentioned in the previous section, files beginning with a dot (**.**) are hidden files. The "filenames" **.** and **..** (single and double dot) are reserved. The **.** refers to the current directory, and the **..** refers to the current directory's parent directory. No two files in the same directory can have the same name, but files in different directories can have the same name.

VMS users have not had the luxury of using dashes in filenames. Dashes are much more common in UNIX filenames simply because it's easier to type `my-file` than `my_file`.

Filenames are case sensitive. This means `MYFILE` is different from `Myfile` is different from `myfile` is different from `myFile`, etc.

You cannot distinguish a directory file from an ordinary file by its name, although some people make their own convention by beginning directory filenames with a capital letter, or ending them in `.d`.

Filename extensions are not required in UNIX. You can include a period and an extension in a filename to help describe the contents of the file, but it will not have special meaning to UNIX itself. However, programs can make use of extensions, for example the FORTRAN compiler expects certain extensions. Note, you can have more than one period in a filename, for example, `lex.yy.c`.

6.2.2 Filename Expansion and Wildcard Characters

The UNIX shells have a number of special characters which can be used on the command line when specifying filenames and directory names. They allow the shell to expand the argument into a set of filenames. These characters are called *wildcards*. Filename references that contain these characters are called *ambiguous file references*. Filename expansion is also called *globbing*.

The question mark (**?**) causes the shell to generate filenames which match any single character in that position. For example, `out?` matches `out1` but not `out12`.

The asterisk (*****) causes the shell to generate filenames which match any number of characters (including zero characters) in that position. For example, `myfile` matches `myf*`. The * alone means all files (except those that begin with dot (**.**), which is a special case).

A pair of brackets (**[ ]**) surrounding a list of characters causes the shell to match filenames containing the individual characters in that position. The brackets define a *character class* and each definition can only replace a single character in a filename. In other words, it is like a question mark that will only allow certain characters. For example, `memo1` and `memoa` match `memo[14a]`), but `memo3` and `memo1a` do not. A hyphen can be used to define a range of characters, for example `[a-z]` represents all lowercase characters. Thus `memo[a-z]` matches `memoa` but not `memo2` or `memoB`.

Character	Action
?	matches any single character in a filename
*****	matches any string of characters (including the empty string) in a filename
[ ]	matches any single character from the set enclosed in the brackets

Examples:

`% ls out*`	lists all files beginning with `out`
`% ls out?`	lists all files with 4-character names beginning with `out`
`% ls out[ab]*`	lists all files beginning with `out` followed by `a` or `b` (e.g., `outa4`)
`% ls *out*`	lists all files containing `out`

Filename expansion may surprise you with the results. For example, **`ls b*`** would list all files starting with b in the current directory, but it would also list the contents of all **directories** whose names start with `b` because of the way **`ls`** behaves for a directory argument. If you want to be sure of what filename expansion will result in, you can use the **echo** command to check it before executing a command.[1] For example, say you have a few matching files in your directory for the command:

```
% echo *out*
```

You would obtain output something like this:

```
fout fout275 inandout out1 out2 out
```

Filename expansion in **csh** can be turned off by setting the *noglob* variable:

```
% set noglob
```

To turn it back on, type **unset noglob**.

6.3 Manipulating Files

This section describes the basic file manipulation commands:

- listing the contents of a directory
- displaying the contents of a file
- copying and renaming a file
- deleting a file
- changing a file's access permissions

Section 6.5 describes the commands you can use to change and manipulate directories.

6.3.1 List Directory Contents: ls

The **ls** command, which stands for **list**, is used to list the contents of a directory. **ls** has many options, some of which are system-dependent, so only a few of them are described here. For a complete description of the command, refer to the man pages for **ls**.

ls by itself lists the names of the files and subdirectories in the current directory (in multicolumn format on some platforms), sorted alphabetically.

The format is:

% ls [*options***] [***filenames***]**

where some of the options are:

-a	List all entries, including those that begin with **.** (dot).
-l	List in the long format, giving mode, number of links, owner, group, size in bytes, and (by default) time of last modification, by default sorted by filename.
-C	List in columns (default on some platforms)
-F	Put a / after the name of each file that is a directory, an * after the name of each file that is executable, and an @ after the name of each file that is a symbolic link.

1. **echo** is otherwise useful for sending messages to the terminal from a script and sending known data into a pipe.

-R	Recursively list subdirectories encountered.
-t	Sort by time stamp (latest first) instead of by name. The default time stamp is the last modification time (see **-u**).
-u	Use the time of last access for sorting if used with the **-t** option or printed in the date column if used with the **-l** option. **-ult** both sorts by and displays last access date.
-d	If the argument is a directory, list the directory itself, not its contents. Use with **-l** to get the status (e.g., permissions) of a directory.

If the argument is a directory, **ls** displays the contents of the directory. Note that this can happen unintentionally as a result of filename expansion. This behavior can be prevented with the **-d** option. The **-t** option is useful when looking at recent files:

```
% ls -lt
```

will result in the long output sorted by reverse modification date rather than by filename.

The following is a sample output of **ls -l**.

```
total 251
drwxr-xr-x 3 nicholls g020c 512   May 2  08:53 Tools
drwxr-xr-x 2 nicholls g020c 512   May 2  09:01 bin
-rw-r--r-- 1 nicholls g020c 446   May 4  14:09 defaults
-rw-r--r-- 1 nicholls g020c 95418 May 1  17:42 intro.lpr
-rw-r--r-- 1 nicholls g020c 0     May 10 17:51 lsout
-rw-r--r-- 1 nicholls g020c 6683  May 1  16:46 man.lpr
-rw-r--r-- 1 nicholls g020c 12258 May 9  16:16 out
```

The first line indicates the number of blocks used. The rest of the lines report on (sub)directories or files in the directory being reported on. The first column of the output is called the *mode*. The character in this first column indicates the type of file, and for our purposes here, they are:

d	directory
-	ordinary file

The next 9 characters are interpreted as three fields of three characters each, indicating the read (**r**), write (**w**), and execute (**x**) permissions for *owner* (sometimes called *user*), *group*, and *other*, in that order (see section 6.6.1 for a discussion of permissions).

Next is the number of links to the file or directory. This refers to the number of different names established for it. Normally files have 1, and directories have 1 each for the directory itself, its parent directory, and each of its subdirectories. In the sample output above, notice that the directory `Tools` has 3 and `bin` has 2. Evidently, `Tools` has one subdirectory and `bin` has none.

The next fields are the login name of the owner, the group to which the owner belongs, the size of the file in bytes, the date and time the file was last modified, and, finally, the filename (which can be a directory name).

6.3.2 List File Contents: cat, less, more, head, and tail

UNIX has a number of commands that can be used for displaying the contents of a file at the terminal.

cat

cat, which stands for "con**cat**enate and print," is the standard UNIX file display; it simply prints the file to the screen.[1] When piped to **less** (see section 5.4.4 which describes **less** as a filter), **cat** displays the file contents a screen at a time, and some simple commands may be executed at the supplied prompt.

% **cat** ***filename...*** **[| less]**

As its name suggests, **cat** is in fact quite useful for copying and concatenating files. Output is often piped to a file rather than to the screen, using standard output redirection (see section 5.4.2). The following example concatenates the three specified files and copies them sequentially to a single file called `allthreefiles`:

% **cat fileone filetwo filethree > allthreefiles**

less, more

A shortcut for **cat** ***filename*** **|less** is to use **less** as a file browser:

% **less** ***filename***

And wherever you use **less**, you can alternatively use **more**, although it is not as functional as **less**. You cannot move backwards through the file with **more**.

head, tail

head displays the first ***n*** lines of the specified file or files. If more than one file is specified, the filename is displayed before each set of file contents of ***n*** lines. ***n*** defaults to 10 lines.

% **head [−*n*] [*filename...*]**

tail displays the last lines of a file. Its syntax is a bit different:

% **tail [+|−*n* lbc] [*filename...*]**

The option **+*n*** displays the file starting ***n*** lines down from the beginning of the file, **−*n*** displays the last ***n*** lines. **l**, **b**, or **c** requests display of ***n*** lines, blocks, or characters (default is **l** lines). If more than one file is specified, the filename is displayed before each set of file contents. ***n*** defaults to 10 lines.

tail is useful when you want to see how far a process got. To display the last line of a log file, enter:

% **tail −1** ***logfile***

6.3.3 Copy a File: cp

The command **cp** (stands for **c**o**p**y) can be used to make a copy of a file, leaving the original version intact. You can copy a single file to another one (in the same or a different directory), or you can copy one or more files to a different directory, retaining the same filenames.

1. There are better ways to display a file (see **less** and **more**, which follow **cat** in this section).

The syntax for these two situations varies slightly:

% cp [*options*] *file1 targetfile* — the file ***file1*** is copied to ***targetfile***, where ***targetfile*** may include a path

% cp [*options*] *file1* [*file2 ...*] *targetdirectory*

one or more files (***file1 file2*** ...) are copied to ***targetdirectory***

If the target is a file, its contents are overwritten unless **-i** is specified, in which case you are prompted for confirmation.

Some options are:

-i — If the target filename exists, you are prompted for confirmation before overwriting.

-r — Used only with the ***targetdirectory*** form. Recursively copy a directory, its files, and its subdirectories to ***targetdirectory***.

The first example below copies `myfile` to `anotherfile`, both in my current directory, prompting for verification in case `anotherfile` already exists.:

```
% cp -i myfile anotherfile
```

New users may find it useful to define **cpi** as the alias for **cp -i** to use in place of **cp** so that prompting always occurs. Section 9.3 discusses aliases.

The second example copies files `proj1` and `proj2` to another directory named `newproj` which is parallel to the current directory (has same parent directory as current):

```
% cp proj1 proj2 ../newproj
```

The third example copies the file `oldproj/proj1` to my current directory (`.`), which is a parallel directory to `oldproj` (has same parent directory). The file `proj1` keeps its name.

```
% cp ../oldproj/proj1 .
```

6.3.4 Move (Rename) a File: mv

The **mv** command (stands for **m**o**v**e) allows you to rename a file in the same directory or move a file from one directory to another. If you move a file to a different directory, the file can be renamed or it can retain its original name. **mv** can also be used to move and rename directories.

% mv [*options*] *source1* [*source2 ...*] *target*

Depending on whether the ***source(s)*** and ***target*** are files or directories, different actions are taken. These are described in the table below. If ***target*** is a filename, only one ***source*** file may be specified.

Source	Target	Result
file	***name***	Rename file as ***name***
file	existing file	Overwrite existing file with source file
directory	***name***	Rename directory as ***name***

Source	Target	Result
directory	existing directory	Move directory to be a subdirectory of existing directory
one or more files	existing directory	Move files to directory

An important option is:

`-i` If ***target*** exists, the user is prompted for confirmation before overwriting.

6.3.5 Reference a file: ln

The **`ln`** (link) command allows you to create a link in one directory to a file in the same or a different directory, or to a different directory. Via links, a file or directory can appear to exist in multiple places, but only actually exist in one, thus conserving disk space. Links are often used to easily reference files or directories that would otherwise require a long path name.

The syntax for **`ln`** is similar to that for **`cp`** and **`mv`**, and in fact they are all run by the same executable.

The most commonly used options for the **`ln`** command are:

`-i` You are prompted before overwriting an existing filename.

`-s` This makes a *symbolic*, as opposed to an ordinary or *hard*, link. A symbolic link can point to a file that is in a different file system, whereas a hard link cannot.

A symbolic link displays the link and the file to which it is linked when you run **`ls -l`**; this is the only way to know the name that a file is linked to.

Note that when using the AFS file system, hard links can only be made between files that are in the same directory (the same *volume*, see section 7.5), so use the **`-s`** option even if you're in the same directory tree.

The syntax differs slightly for files and directories:

`% ln [`*options*`]` ***/path/to/file_name /path/to/link_name***

Create the link ***link_name*** to reference the file ***file_name***. If ***link_name*** already exists (as a link or as a file), it gets overwritten (unless you use option **`-i`**).

`% ln -s [`*other options*`]` ***/path/to/file_name /path/to/link_name***

Create symbolic link named ***link_name***, that links to ***file_name*** which exists in the same or another directory.

`% ln -s [`*other options*`]` ***/path/to/file_name1*** **`[`*/path/to/file_name2 ...*`]`** ***directory***

Create a symbolic link in ***directory*** to each of the listed files. The files may all exist in different directories since the **`-s`** option is used. The link names will be the same as the filenames they link to. If files of the same name but in different directories are specified, only the first link of that name will be created.

Let's look at an example:

```
% ln -s /e741/run1/e_mu2/mydata r5742
```

If `r5742` is a directory, this creates a link called `mydata` in the directory `r5742` that points to `/e741/run1/e_mu2/mydata`. You can now reference the data file as `mydata` (i.e. the same filename) as if it were in the directory `r5742`.

On the other hand, if `r5742` is not an existing directory then it represents the name of the link being created. In this case, the command establishes a link called `r5742` that points to the file `/e741/run1/e_mu2/mydata`. Running the command **`ls -l`** should display the following output:

```
lrw-r--r--   1 aheavey  g020         46 Aug 29 14:26 r5742 -> /e741/run1/e_mu2/mydata
```

6.3.6 Remove a File: rm

The **`rm`** command (stands for **re**mo**v**e) is used to remove the entries of one of more files.

`% rm [`*options*`]` ***file...***

Some commonly used options are:

`-i`	Confirmation of removal of write-protected file occurs interactively, whether the standard input is a terminal or not. If used with the **`-r`** option, you are prompted about each directory before it is examined.
`-r`	Causes **`rm`** to delete the contents of the specified directory, including all its subdirectories, and the directory itself (**r**ecursive). This option should be used cautiously.

The file list can include ambiguous file references, so **`rm`** should be used cautiously. You can use the **echo** utility with the same ambiguous file reference to see the list generated.

Removal of a file requires write permission to its directory, but neither read nor write permission to the file itself. If the file has no write permission and the standard input is a terminal, the set of permissions is printed and you are prompted for confirmation. If the answer begins with a **y**, the file is deleted. If the standard input is not the terminal, the files are deleted without confirmation.

New users may find it useful to define **`rmi`** as the alias for **`rm -i`** to use in place of **`rm`** so that prompting always occurs. Section 9.3 discusses aliases.

6.3.7 Copy to/Restore from Archive or Tape: tar

The **tar** utility (**t**ape **ar**chive) can be used to create, add to, list and retrieve files from an archive file. Archive files are often stored on tape. The action taken by the **`tar`** command depends on the *key*, which is essentially a function option. The key must be specified on the command line as if it were the first option. It may be followed by function modifiers, and then by options and/or arguments. The keys and function modifiers must be grouped together before any arguments are listed. **tar** does not require, but does allow, a dash (**–**) before the list of keys and function modifiers. The keys and functions are:

`c`	create a new tar file
`r`	append specified files to tar file
`t`	list all files in the tar file, or all files in a specified file list
`u`	append new or changed files to tar file
`x`	unwind entire tar file or extract specified files from tar file and write each file to the directory as specified in the tar file relative to the current directory

The keys, function modifiers and options are discussed in the man pages. Be aware that they vary in some cases between UNIX flavors. The command syntax varies somewhat from key to key, so check the man pages for that information, too.

When creating a tar file, we have a few recommendations for avoiding problems:

First, when possible, create the tar file on a machine of the same flavor as the target platform. Occasionally a tar file doesn't unwind properly on a different platform.

Secondly, choose your working directory carefully. It is often convenient or desirable to be able to specify simple relative path names for the files to include in the tar file. For example:

```
% cd /path/to/dir
% tar cvf /tmp/filename.tar .
```

creates a tar file with all pathnames relative to `/path/to/dir`. In general you should not specify the pathname explicitly on the command line, unless it will be valid on any other system where the tar file may be unwound and used.

Thirdly, be careful choosing your target directory for the new tar file. Make sure that the target directory is outside of the directory tree that you're including in the tar file. Otherwise the tar file tries to include itself, and can grow infinitely large.

Note to ex-VMS users: **tar** does not provide for extracting a named set of files from a backup file and placing them all in the current directory disregarding their directory specifications on the backup file.

6.3.8 Compress or Expand a File: gzip, gunzip

Several utilities are available on UNIX systems for file compression. **compress** and **pack** are native UNIX utilities, and **gzip** is provided by FUE. We recommend you use **gzip** for file compression, and its associated utility **gunzip** for file expansion. **gunzip** recognizes and can expand files compressed with **compress** and **pack** as well as **gzip**.[1]

The file extensions **gunzip** recognizes include `.gz`, `-gz`, `.z`, `-z`, `_z`, and `.Z`. **gunzip** also recognizes the special extensions `.tgz` and `.taz` as shorthands for `.tar.gz` and `.tar.Z`, respectively. When compressing, **gzip** uses the `.tgz` extension if necessary instead of truncating a file with a `.tar` extension.

You will need to reference the man pages for details on syntax, options and usage. In their simplest forms **gzip** and **gunzip** can be used as follows, starting, for example with the original uncompressed file `bigfile`:

```
% gzip bigfile
```

The result is `bigfile.gz`, whose size is reduced with respect to `bigfile` according to Lempel-Ziv coding (LZ77), the same compression scheme used by **compress**. Whenever possible, **gzip** replaces each file by one with the extension `.gz`, while keeping the same ownership modes and access and modification times. **gzip** will only attempt to compress regular files. In particular, it will ignore symbolic links. If the compressed file name is too long for its file system, **gzip** truncates it.

Compressed files can be restored to their original form using **gunzip**, or equivalently by using the **`-d`** option with **gzip**. If the original name saved in the compressed file is not suitable for its file system, a new name is constructed from the original one to make it "legal".

1. On some systems (namely where **gzip** and **gunzip** are not installed in `/usr/local/bin`) you will need to run **`setup gtools`** in order to access them.

To restore `bigfile.gz` to its original name and size, enter:

```
% gunzip bigfile.gz
```

6.4 Information About Files

This section gives a cursory overview of simple uses for two very powerful commands for dealing with files: **`find`** for searching for files and **`grep`** for searching for strings within files. We also describe **`wc`** which displays the size of a file, **`od`** which creates a dump of a file, and **`file`** which can determine file type.

6.4.1 Find a File: find

The **`find`** utility tests each file in the given pathname list to see if it meets the criteria specified by the expression supplied. It does this by recursively descending the directory hierarchy for each path name. The format is:

% **`find`** ***path-name-list expression***

path-name-list can contain file expansion metacharacters. Each element in ***expression*** is a separate boolean criterion. A space separating two criteria is a logical AND operator, a **`-o`** separating the criteria is a logical OR operator. A criterion can be negated by preceding it with an exclamation point (**`!`**). Criteria are evaluated from left to right unless parentheses are used to override this. Special characters must be quoted (use **`\`**) and there must be spaces on each side of the special character pair.

Some of the criteria that can be used within ***expression*** are:

`-name` *filename*	True if ***filename*** matches the name of the file being evaluated. Ambiguous file references can be used if enclosed in quotes.
`-type` *filetype*	True if the type of the file is ***filetype***, where ***filetype*** is either `d` (directory) and `f` (ordinary file).
`-atime` ***n***	True if the file has been accessed in ***n*** days.
`-mtime` ***n***	True if the file has been modified in ***n*** days.
`-newer` *filename*	True if the file has been modified more recently than ***filename*** has.
`-print`	Causes the matching path names to be displayed on the screen.
`-exec` *command* **`\;`**	True if ***command*** returns a zero exit status. ***command*** must be terminated with a quoted semicolon (note the **`\`**). An empty pair of braces ({ }) within the command represents the filename of the file being evaluated.
`-ok` *command* **`\;`**	Same as **`-exec`** except the generated command line is displayed and executed only if the user responds by typing `y`.

In the previous list, ***+n*** means more than ***n***, ***−n*** means less than ***n***, ***n*** means exactly ***n***.

☞ Note that **`find`** doesn't do anything with the found files, it doesn't even display the names, unless instructed to.

Examples:

- Search the current directory and all subdirectories for the file `lostfile`:

  `% find . -name lostfile -print`

- List all files ending in `.html` in your `/wwwork` subdirectory:

  `% find wwwork -name '*.html' -print`

- This command will prompt you if you want to execute **`more`** on each file that begins with the letter `d` in the current directory and all subdirectories (Enter `y` if you want the file displayed.):

  `% find . -name 'd*' -ok more {} \;`

- List all files in the current directory that *don't* begin with `m`:

  `% find . ! -name 'm*' -print`

- Find all files in the current directory and all subdirectories that contain the string *hello*:

  `% find . -exec grep -l "hello" {} \;`

- Remove all files in your directory tree that are named `a.t` or have the extension of `.o` and haven't been accessed in a week:

  `% find ~\(-name a.t -o -name '*.o') -atime +7 -exec rm {} \;`

Note that using the `find` command uses many system resources.

AFS

In particular on AFS systems, you may accidentally end up searching servers all over the world if the top of the search is at the root directory (**/**). Generally you should be careful to only search the part of the UNIX tree that interests you. Here is an example:

- Look for ***filename*** starting at the root directory (**/**), and exclude searches in the `/afs` and `/nfs` branches:

  `% find / \( -name /afs -prune \) -o \( -name /nfs -prune \) -o -name` ***filename*** **`-print`**

6.4.2 Search for a Pattern: grep

The **`grep`** utility searches the contents of one or more files for a pattern.

The format is:

`% grep [`*`options`*`]` ***pattern*** **`[`*`file`*`...]`**

Some of the options are:

`-c`	Display only a count of lines that contain the pattern.
`-i`	Ignore upper/lower case distinctions during comparisons.
`-l`	Display only the name of each file that contains one or more matches.

The pattern can be a simple string or a regular expression (see section 5.4.5). You must quote regular expressions that contain special characters, spaces, or tabs (this can be done by enclosing the entire expression within single quotation marks).

Examples:

- Find all non-hidden files in the current directory containing the string *smith*:

  `% grep -i smith *`

- Search the file `abc` for a string beginning with *f*, followed by 0 or more *r*'s, and ending in *og* (e.g., *frog, fog, frrog*):

  `% grep 'fr*og' abc`

- Search the file `myfile` for a line beginning with a **T**:

  % grep '^T' myfile

  or

  % less myfile | grep '^T'

- Search `/usr/jones/junk` for the characters *file* followed by a number (e.g., *file1, file3*):

  % grep 'file[0-9]' /usr/jones/junk

- Display a line if Smith is logged in:

  % who | grep smith

- Show all processes being run by Smith:

  % ps -ef | grep smith

- Show all environmental variables containing ***string*** in their name or their translation:

  % env | grep *string*

- Show all aliases containing ***string*** in their name or their translation:

  % alias | grep *string*

6.4.3 Count a File: wc

The **wc** command, which stands for **w**ord **c**ount, counts the number of lines, words, and characters there are in the named files, or in the standard input if the argument is absent. If there is more than one file, **wc** totals the count as well.

% wc [-lwc] [*names*]

The options **l**, **w**, and **c** may be used in any combination to specify that a subset of **l**ines, **w**ords, and **c**haracters be reported. The default is **-lwc.**

UNIX users frequently count things by piping them into **wc**. For example, to display the number of users logged into the system, you can execute:

% who | wc -l

6.4.4 Dump a File: od

The **od** (**o**ctal **d**ump) command can be used to examine the contents of a file in various formats: octal, decimal, hexadecimal, and ASCII. The default is octal.

The format is:

% od [*options*] [*file*] [*offset*] [|less]

If ***file*** is not included, standard input is assumed. The options are:

-c	Produces a character dump.
-d	Produces a decimal dump.
-o	Produces an octal dump.
-x	Produces a hexadecimal dump.

The **-c** option prints non-printable characters as a printable character preceded by a backslash: **\0** is null, **\b** is backspace, **\f** is form-feed, **\n** is new-line, **\r** is return, and **\t** is tab.

The ***offset*** specifies where in the file the dump is to begin, if different than the beginning of the file. It is of the form **`[+]`*****n*****`[.][b]`**. The **`+`** is only necessary if you have no file specified so that the command interpreter knows this is the offset not the file. Without **`.`** or **`b`**, ***n*** indicates the dump starts at (octal) byte ***n*** of the file. A **`.`** displays ***n*** in decimal, a **`b`** in 512-byte blocks.

We recommend that you always pipe the output of **`od`** to **`less`** (or **`more`**) so that you can manipulate it. Large files can be unwieldy, and you may not be able to stop the output once it's going!

6.4.5 Determine File Type: file

The **`file`** utility can be used to determine the file type of a file according to its contents. It bases its guesses on a list of "magic numbers" recorded in a "magic file", `/etc/magic`. Some of the file types are:

- ASCII text
- C program text
- directory
- executable

`file` determines the filetype by looking at the beginning of the file and comparing it to entries in the magic file. The command format is:

`%` **`file`** ***filename...***

6.5 Manipulating Directories

This section describes the commands you can use to organize and use the UNIX directory structure. It describes how to make and remove directories, and move from one directory to another. Listing the contents of a directory (files and subdirectories) was described in a previous section. Section 6.6.2 explains the meaning of access permissions as applied to directories.

6.5.1 Print Working Directory: pwd

The **`pwd`** command (for **`p`**rint **`w`**orking **`d`**irectory) displays the path name of your working (current) directory. The command format is:

`%` **`pwd`**

6.5.2 List Directory Contents: ls

The **`ls`** command, which stands for **`list`**, is used to list the contents of a directory. **`ls`** has many options, some of which are system-dependent. A few of them are described in section 6.3.1. For a complete description of the command, refer to the man pages for **`ls`**.

6.5.3 Change Directory: cd

When you first log in to the system, you are placed in your home directory, which is then also your current working directory. You can use the **cd** command (for **c**hange **d**irectory) to change your current working directory. The command format is:

% cd [*directory*]

You can specify a complete path or a relative path. You can use `..` (for the parent directory) in your pathname. You must have execute permission (which provides search permission in this case) on a directory to **cd** to it.

If ***directory*** is not specified, you are returned to your home directory.

The following examples illustrate moving to different directories:

- your home directory

  % cd

- a subdirectory called `Tools`

  % cd Tools

- a colleague's home directory (using absolute pathname)

  % cd /usr/jones

- a colleague's subdirectory (using tilde)

  % cd ~jones/ourfiles

- a parallel directory (has same parent directory as current directory)

  % cd ../Tools

6.5.4 Make a Directory: mkdir

The **mkdir** command (for **m**a**k**e **dir**ectory) is used to create a directory. The command format is:

% mkdir *dirname ...*

If a pathname is not specified, the directory is created as a subdirectory of the current working directory. Directory creation requires write access to the parent directory. The owner ID and group ID of the new directory are set to those of the creating process.

Examples:

- create a subdirectory in the current working directory

  % mkdir progs

- create one with a full pathname (the directory `Tools` must already exist)

  % mkdir /usr/nicholls/Tools/Less

6.5.5 Copy a Directory

The most straightforward way of copying a directory and its contents is to pipe the output of the **ls** command (see section 6.3.1) into the file copy facility **cpio** (see the man pages).

First, create the destination directory using **`mkdir`** (see section 6.5.4), if it doesn't already exist. Secondly, from the source directory, run the command (shown with recommended options; see man pages for option information):

`%` **`ls | cpio -dumpV`** *destination_dir*

The ***destination_dir*** must be specified relative to the source directory.

Another way to copy directory hierarchies is to use the **tar** utility, described in section 6.3.7. The following sequence of commands copies a structure from the source directory to the destination directory:

`%` **`cd`** *source_dir*`;` **`tar cf - . | (cd`** *destination_dir*`;` **`tar xfBp -)`**

The "**`-`**" is used for the name of the tar file (argument to the **`f`** option) so that **tar** writes to the standard output or reads from the standard input, as appropriate.

6.5.6 Move (Rename) a Directory: mv or mvdir

See section 6.3.4 for information on **`mv`**. To move a directory (***olddirname***) and its contents to a different position in the directory tree, use the command format:

`%` **`mvdir`** *olddirname newdirpath*

If ***newdirpath*** exists already, then the directory gets moved to ***newdirpath*/*olddirname***. Note that the two arguments cannot be in the same path. For example:

`%` **`mvdir x/y x/z`**

is ok, but

`%` **`mvdir x/y x/y/z`**

is not ok.

6.5.7 Remove a Directory: rmdir

You can remove a directory with the **`rmdir`** command. The directory must contain no files or subdirectories, and you must have write permission to the parent directory.

`%` **`rmdir`** *dirname* **`...`**

You can use an absolute or relative pathname.

You can also use **`rm -r`** as described in Section 6.3.6. **`rm -r`** will delete a directory, all subdirectories, and all files. **This command should be used with extreme caution.**

For example, the following command deletes the directory `temp`, all subdirectories of `temp` and all files contained in those directories, prompting before each removal, and confirming removal of write-protected files (**`-i`**):

`%` **`rm -ir /usr/jones/temp`**

6.6 File and Directory Permissions

6.6.1 File Access Permissions

The UNIX file system allows you to control read, write, and execute access to your files on the basis of user (owner), group, and other (everyone else).[1] In this section we will consider only the standard UNIX file permissions.

Note that in the AFS file system, file permissions are mediated by Access Control Lists (ACLs) that are set on a directory level. The standard UNIX file permissions don't apply in this case except for the owner permissions, which apply to all users. AFS file permissions are treated in section 7.6.

To determine the current permissions, use the long form of the **`ls`** command, **`ls -l`**. Referring to the example below, the nine characters immediately following the first field represent the one-bit flags known as the *mode bits* that control file access. A dash indicates a bit is not set, **`r`** stands for **`r`**ead access, **`w`** for **`w`**rite access, and **`x`** for e**`x`**ecution access. The first set of three characters refer to owner permission, the middle three for group permission, and the last three for all other user classes.

```
    total 251
    drwxr-xr-x 3 nicholls g020c 512   May 2  08:53 Tools
    drwxr-xr-x 2 nicholls g020c 512   May 2  09:01 bin
    -rw-r--r-- 1 nicholls g020c 446   May 4  14:09 defaults
    -rw-r--r-- 1 nicholls g020c 95418 May 1  17:42 intro.lpr
    -rw-r--r-- 1 nicholls g020c 0     May 10 17:51 lsout
    -rw-r--r-- 1 nicholls g020c 6683  May 1  16:46 man.lpr
-rw-r--r-- 1 nicholls g020c 12258 May 9  16:16 out
```

In the example, ignoring the directory files (which have a **`d`** in position 1), the owner has **`rw`** access to the files, whereas group and others have read (**`r`**) access only.

chmod

The **`chmod`** command, which stands for **`ch`**ange **`mod`**e, is used to change access permissions of a file or directory:

`%` **`chmod`** ***mode filename ...***

or

`%` **`chmod`** ***mode directory ...***

In the *absolute form* of the mode where the level of protection is specified in octal format, ***mode*** looks like 741 or 554, for example, where each of the three octal numbers represents the sum of the permissions granted to its class: user, group, and other, in that order. The three types of permission have the values:

read	4 (100 octal)
write	2 (010 octal)
execute	1 (001 octal)

For example, a mode of 741 means owner can read, write, and execute (4+2+1=7); group can read (4+0+0=4); and others can execute the file (0+0+1=1).

1. Note **o** is for **other** and not for **owner** as on VMS.

To give this permission to a file `test`, you would enter:

% chmod 741 test

You can use an alternate form of ***mode*** in the **chmod** command in which ***mode*** is a three-character field specifying an action to be taken. The action is to add or subtract one or more permissions from one or more user classes. It takes the form:

who operator permission(s)

These three positions within the field take the following characters:

who	represents the user class or classes; it takes any combination of **u**, **g**, **o**, and **a** for **u**ser (user is really the owner), **g**roup, **o**ther and **a**ll, respectively, where *all* includes the three individual classes
operator	**+** or **−** for adding or subtracting permissions, or **=** for setting a specific permission and resetting all other permissions for the specified user class(es)
permission(s)	any combination of **r**, **w**, and **x** for **r**ead, **w**rite, and e**x**ecute, indicating the permissions to be permitted, denied, or reset.

Examples of the **chmod** command:

- Remove group execute permission to the file `progs`:

  % chmod g-x progs

- For the files `out` and `out1`, add group read and write, and deny write to other:

  % chmod g+rw,o-w out out1

- Set group read permission and reset all other group permissions to `myfile`:

  % chmod g=r myfile

☞ Note that classes of users or levels of protection not specified in a command are not modified in this form of the command (with the exception that **=** resets other permissions).

umask

With the **umask** command you can specify a mask that the system uses to set access permissions when a file is created. In order to understand **umask** you need to know that access permission at file creation is application-dependent. Each command or application sets a file permission in its *open* command.[1] The system then "subtracts" any user-defined mask, resulting in the final access permission for the file. You can set a umask by this command:

% umask [*ooo*]

where ***ooo*** stands for three octal digits. The user-specified "mask", ***ooo***, has the same positional structure as described above for **chmod**, but specifies permissions that should be **removed** (disallowed).

For example, a mask of 022 removes no permissions from owner, and removes write permission from group and others. Thus a file normally created with 777 would become 755 (this would appear as `rwxr-xr-x` in the format put out by the command **ls -l**). The following command could be put in your `.cshrc` or `.profile`.

% umask 022

The meaning of permissions applied to directories is described in Section 6.6.2.

1. Normally only the loader creates files with execute permission.

6.6.2 Directory Permissions

See section 7.6 for AFS systems.

You can grant or deny permission for directories as well as files, and protection assigned to a directory file takes precedence over the permissions of individual files in the directory.

- Read permission for a directory allows you to read the names of the files contained in that directory with the **`ls`** command, but not to use them.
- Write permission for a directory allows you to create files in that directory *or* to delete any file in the directory, regardless of the file protection on the files themselves. It does not allow you to see the files or use them without **`r`** and **`x`** directory permission. In other words, write permission to a directory allows you to alter the contents of the directory itself, but not to alter, except to remove, files *in* the directory (which is controlled by the file's permissions).
- Execute permission allows you to list the contents of the directory.

File access permissions of directory files are changed with the **`chmod`** command (see section 6.6.1).

6.7 Temporary Directories

By convention, there are directories named `/tmp` (and sometimes `/usr/tmp`) where programs and users can store temporary files. Many programs (e.g., compilers) write temporary files there or in the area specified by the environment variable *TMPDIR*. Since these are public areas, it is necessary to manage this space, which means that you cannot count on files being retained in these directories.

Many systems on site have fairly small `/tmp` areas and therefore you must be careful not to fill up this space. In general, you should only use `/tmp` for very temporary, small files. On many systems files in `/tmp` will disappear after a reboot or after existing for a week. You can set *TMPDIR* to a different location if there is not enough space in these areas.

Contact the administrators of the particular system to find out what the current policy is on the machine.

Chapter 7: The AFS File System

Fermilab is using the AFS (Andrew File System) as a distributed file service model, and it is installed on several machines on site in a production environment, including the FNALU cluster. This chapter discusses the basic concepts of AFS and provides information on the commands used to manage your files and directories in the AFS environment.

On the Web there is a local collection of information regarding AFS. Look under *The UNIX Operating System* on the *UNIX Resources* Web page. The helpdesk has a small quantity of laminated AFS reference cards that list the syntax of the most common AFS commands. The helpdesk is located in FCC 1W. There are also man pages for AFS commands on the systems running as AFS clients.

7.1 Introduction to AFS

AFS is a modern implementation of distributed file serving. The FNALU cluster is running AFS as a distributed file server in a production environment. Several other systems at Fermilab run AFS as well. AFS has several advantages over older file systems such as NFS (see section 2.6), particularly in the areas of:

- consistency of file organization across the distributed network
- authentication of users (security)
- server management
- volume replication and backup
- server function redundancy

All systems participating in the AFS file system have the same view of the file system. All files for a set of machines known as a *cell* exist under `/afs/{cellname}`. The cellname for Fermilab is `fnal.gov`, thus all files that are part of the AFS file system are located under `/afs/fnal.gov`. There is also a symbolic link (see section 6.3.5) to a shorthand version of the cell name, which is simply `/afs/fnal`. Other cells are also accessible. They are listed as directories under `/afs`, for example the CERN cell at `/afs/cern.ch`.

7.2 How to Determine if AFS is Installed on your System

A special note to CDF and D0 users: Your UNIX systems are *not* configured to use AFS.

To find out if AFS is installed, issue the command:

```
% ps -ef | grep afsd
```

If you get output of the form (note the `/usr/vice/etc/afsd`):

```
    root   305     1  0   Sep 30 ?      14:10 /usr/vice/etc/afsd -stat 2800 -dc
ache 2400 -daemons 5 -volumes 128
    root   306     1  0   Sep 30 ?       5:47 /usr/vice/etc/afsd -stat 2800 -dc
ache 2400 -daemons 5 -volumes 128
```

then it is installed and running on your machine. If you do not get output lines like this, AFS is not installed. However it is still possible that you have access to the AFS file system via a translator[1] service. We discuss translator mode in section 7.9. To check, enter the command:

% df |grep afs

If no output is returned, AFS is not running on your machine in any capacity. If output is returned and the line begins with a node name preceding `/afs`, for example:

```
fsut01:/afs                          144000000           0   144000000     0%
```

then AFS is running in translator mode. (If the output line begins with `/afs`, then AFS is actually installed on your machine.)

7.3 Issues Related to Login and File Access

7.3.1 Authentication in AFS

On machines running AFS, as on most systems, providing a username and password is sufficient to identify a user as legitimate, and allow the login to succeed. However to access AFS files, you must provide a password recognized by AFS, called a *Kerberos password*. This *authenticates* you to AFS. Once you are authenticated, AFS issues what is known as a *Kerberos token* to your login process. It is having the token that allows you access to the AFS file system. As long as you remain logged on, the Kerberos token "lives" for a period of time set by the AFS administrator of the system; for example in the Fermilab cell it is set to six days.

The token is passed to all subprocesses of the login process (see section 5.1 for an explanation of subprocesses). All normal UNIX interactive operations are therefore automatically authenticated, and access is granted to files in the AFS tree, provided you have the appropriate permissions (AFS permissions are covered in section 6.6.2). The Fermilab standard batch interface **fbatch** provides for token renewal at job execution time, since you can't control when your batch jobs actually run.

Situations occasionally arise in which you are not automatically authenticated (e.g., some remote login methods) or you lose your token (e.g., you remain logged in for more than six days). When this happens and you need to obtain a new token, issue the command string:

% pagsh

% klog

pagsh starts a special AFS shell under your login process. **klog** prompts you for your AFS password and obtains a Kerberos token associated with this shell, thus granting authentication and access to files.

☞ A few notes:

1) Running **pagsh** first is much more secure than just running **klog**. It ensures that the token is associated with your pagsh process, and thus with all processes you spawn. **klog**

1. In translator mode, a "translator machine" runs AFS and exports the AFS file tree to other systems.

by itself gets a token associated with your UID, which is not always unique. This could potentially allow another user to share the token, which is undesirable.

2) You cannot enter the commands on one line in the format **`pagsh;klog`**. **`pagsh`** starts a new **sh** shell, and **`klog`** needs to be run at the new shell prompt on the next line.

3) **`pagsh`** changes your shell to **sh**, so you will need to run your preferred shell afterwards (e.g., enter **`tcsh`**, **`bash`**, or **`ksh`** on the command line). You may also then want to source your `.login` and `.cshrc` or your `.profile` and `.shrc` scripts to ensure that your FUE environment is back to normal (see section 4.4 regarding *sourcing* a script, and section 9.4 for information on the login files).

There are Kerberos authentication problems with running programs that spawn jobs external to your login process group. **at** and **cron** fall into this category (they are described in section 5.5.3). You can run the job, but it will not run with authentication, and most likely will not be able to write into `/afs` space. A work-around is available for executing an authenticated **cron** job (send mail to *helpdesk@fnal* requesting the full details of this procedure).

Be aware that being logged on as *root* grants you no special permissions in `/afs` file space; there is no such thing as being “authenticated as root”.

7.3.2 Kerberos (AFS) Password

You will have a Kerberos password (sometimes called an AFS password), for any account on a system that uses the AFS file system, for example FNALU. The Kerberos password, which you enter at login time, allows two operations to proceed:

- You can log in to the system.
- Your *Kerberos token* is automatically obtained via the AFS programs **pagsh** and **klog** to allow you access to the file system. The token is attached to your login process.

You can change your Kerberos password using the command:

```
% kpasswd
```

The system will prompt you for the necessary information.

We recommend that you limit your password to eight characters. This enforces consistent behavior in a multi-vendor AFS environment. On some platforms the login program may truncate a long password after eight characters, allowing login to proceed but denying access to the file system.

This command changes your Kerberos password for all systems that run AFS on-site.

7.3.3 Standard UNIX Password on an AFS System

Depending on how the AFS file system was installed, you may or may not have a standard UNIX password in addition to your Kerberos password. In other words, you may *have* a standard UNIX password even if you never need to use it! On FNALU, AFS was installed so that you have only a Kerberos password; no standard UNIX password is defined.

You can find out if you have a standard UNIX password (or an NIS password, see section 2.7) by attempting to change it via the standard UNIX command **`passwd`**. If the command does not succeed (assuming you provide the correct old password if requested), then you do not have a standard UNIX password. Note that the **`passwd`** command returns a different error message on each different UNIX flavor.

If you have both, at login you should provide the Kerberos password so that you obtain your Kerberos token. If instead you provide your standard UNIX password, the system will log you in, but you will not obtain a token and thus will not be able to access AFS files. If the passwords are the same, the Kerberos password automatically takes precedence.

Generally, the Kerberos password is the only password you need. There are exceptions; for example, remote login via a method that doesn't understand Kerberos passwords (e.g., MAC-X to some UNIX platforms). In this case, after logging in using a standard UNIX password, you would need to run the command string **`pagsh`** and **`klog`** as described in section 7.3.1:

7.3.4 Managing your Token

View Active Tokens

To see what tokens you currently hold, you can issue the command:

`%` **`tokens`**

The output should look similar to this:

```
Tokens held by the Cache Manager:

User's (AFS ID 6302) tokens for afs@fnal.gov [Expires Oct 21 10:22]
User aheavey's tokens for krbtgt.FNAL.GOV@fnal.gov [Expires Oct 21 10:22]
   --End of list--
```

If the output showns no tokens (or only the `krbtgt` token, the second one shown above[1]), then you only have access to (usually a very limited number of) files designated as accessible to the special user *system:anyuser* (a pre-defined AFS protection group; see section 7.7). As its name implies, this designation includes anyone who can access the system (e.g., a user with a standard UNIX password but no Kerberos password).

Get Back an Expired Token

If you remain logged on beyond the set token expiration period, you will find that you no longer have access to AFS files. The system will likely return the message `Permission denied` when you attempt a file operation. To get back the token associated with your login process, issue the commands[2]:

`%` **`pagsh`**

`%` **`klog`**

If you are unexpectedly unable to edit your files, try this first! Expired tokens are often the reason for this problem. See the notes in section 7.3.1 regarding these commands.

Destroy a Token

Logging out does not destroy your token; it remains "live" for up to 26 hours afterwards. This is a security risk. Prior to logging out, we advise that you issue the command:

`%` **`unlog`**

---

1. The `krbtgt` token was created on FNALU for an application that was never implemented, and it does not affect AFS access. It will probably be removed in the near future.
2. If your token has expired within the previous two hours, you do not need to run **`pagsh`** before **`klog`**.

to destroy the token. If you create a `.logout` file (see section 9.4), you should include this command in it.

Token Issues for Remote Login

One practical issue raised by the Kerberos environment involves the use of the Berkeley networking programs such as **rlogin**, **rsh**, and **rcp**. **telnet** automatically authenticates the user and avoids the issues discussed here. All these utilities are described in Chapter 13.

Normally systems are *equivalenced* to enable the use of the **rlogin**, **rsh**, and **rcp** protocols. The equivalencing of the machines implies that once you are logged into one system, you may log into the equivalenced machines without providing further proof of identity, such as a password. This doesn't fit in with the Kerberos authentication system.

On FNALU, token-passing is available for **rsh** and **rcp**.[1] If you have authenticated into the `/afs/fnal.gov` cell, you can use **rcp** and **rsh** more or less normally between AFS machines; the token will be passed along with the request, and the network communications will be authenticated automatically.[2]

rlogin is more complicated. When equivalencing between two machines is enabled, the **rlogin** protocol does not ask for a password. **rlogin** works, but the remote user is not authenticated at login time, and cannot access most files.

☞ We recommend that you use **telnet** rather than **rlogin** in order to avoid this problem.

If you need to use **rlogin** for some reason, immediately after login issue the commands:

`% `**`pagsh`**

`% `**`klog`**

as described in section 7.3.1, to obtain authentication.

7.4 AFS File System Commands and man Pages

AFS provides a command (with many options) that allows you to address file system issues such as checking permissions, checking quota, making mount points, finding where a volume is mounted in the file tree, and so on. The AFS file system (**`fs`**) command is entered in the format:

`% `**`fs`** ***main_option –option(s) argument(s)***

Many of the options can be abbreviated, and option flags can often be omitted from the command. Check the man pages, as described below.

To get a list of the main options of the **`fs`** command, enter:

`% `**`fs help`**

Here is an edited output listing showing only a few of the options:

```
fs: Commands are:
listacl        list access control list
listquota      list volume quota
lsmount        list mount point
quota          show volume quota usage
```

1. If you are on a non-FNALU node, check with your system administrator before running these utilities.
2. This is set up with the usual UNIX method of using `/etc/hosts.equiv` or `.rhosts` to equivalence systems.

```
rmmount         remove mount point
setacl          set access control list
setquota        set volume quota
whereis         list file's location
whichcell       list file's cell
```

To get usage information on a particular **fs** option, enter:

% fs ***option*** **-help**

For example:

% fs setacl -help

```
Usage: fs setacl -dir <directory>+ -acl <access list entries>+ [-clear ] [-negat
ive ] [-id ] [-if ] [-help ]
```

Man pages are available on-line on the AFS systems[1], but you'll find that the names of the manual pages are a little less than clear. Since the **fs** command is *one* command with several main options (e.g., **setacl**, **listacl**), to get the man pages you must attach the command and the main option via an underscore (**_**):

% man ***command_option***

where ***command*** is **fs**, for example:

% man fs_setacl

The underscore is only used with the **man** command, not when issuing the command itself.

7.5 AFS Volumes and Quota

UNIX divides disks into partitions. AFS further divides partitions into subsections called volumes. A volume houses a subtree of related files and directories. Normally, volumes are considerably smaller than traditional file systems. For example, each user's home directory would normally be stored in a separate volume. Large sub-directories are further sub-divided. You do not need to know which file server houses any volume. AFS locates volumes automatically.

To examine the quota on a volume within AFS, the **fs listquota** command may be used. You can request information on several directories at a time. For example the following command requests information on the current working directory (**.**) and on another one specified via an environment variable (see section 9.1):

% fs listquota . $UAFWWW

```
Volume Name            Quota    Used    % Used   Partition
room.aheavey          130000  126024       97%<<       75%      <<WARNING
files.reports.UNIX   2000000   77370        4%         63%
```

The output includes the name of the volume containing the specified directory(ies), the quota size, amount used, percent used, and the percent of space used on the partition containing the volume. You might also get a warning! All sizes are in kilobytes.

1. If you have trouble finding the man pages for AFS commands on your node, check your *$MANPATH* variable to see that it includes `/usr/afsws/man`.

7.6 File and Directory Permissions

7.6.1 File Permissions

File permissions work quite differently from those in standard UNIX, which are described in section 6.6.1. In AFS, you can use the **`chmod`** command just as you would in a standard UNIX file system. However, it behaves differently. Although in AFS all the permission bits on a file may be examined or changed, **only the owner bits are actually used in AFS, and they apply to all users of the file (as permitted by users' ACL settings; see below)**. To turn off write access to a particular file by all users, including the owner, you just need to turn off the owner write bit of the file.

7.6.2 Directory Permissions via Access Control Lists (ACLs)

All other AFS permissions are done with Access Control Lists (ACLs) which take effect at the directory level only. Every directory has its own ACL that defines who can access the directory and its files. Each entry in an ACL consists of a username or an AFS protection group paired with a set of permissions (e.g., read, write). An AFS protection group is simply a list of usernames grouped to share a set of permissions in one or more ACLs. If a user is in two or more ACL entries (e.g., is a member of two groups listed in the ACL) with different permissions assigned, the user gets the union of the permissions.

The permissions granted in a directory's ACL represent the *maximum* permissions. If a file in the directory has more restrictive permissions set, the user is limited by the restrictions on the file. If a file has more lenient permissions set, the user is limited by his ACL entry.

ACL rights include:

`l` lookup rights (allows user to issue an **`ls`** command on files in the directory, examine the directory's ACL, and access the directory's subdirectories which are protected by their own ACLs)

`i` insert rights (allows user to create new files or copy files into the directory)

`d` delete rights (allows user to remove files or move them to other directories)

`a` administrator rights (allows user to change the ACL for a directory; note that you always have this right for your home directory even if you accidentally remove this ACL.)

`r` read rights (allows user to look at the directory's contents and to read the data in the files contained in the directory)

`w` write rights (allows user to modify the contents of the files in the directory and to change the UNIX mode bits with the command **`chmod`**)

`k` lock rights (allows user to run programs that need to *flock* files in this directory; see the man pages for **flock**)

Rights may also be referred to by special names that designate commonly-assigned combinations of rights. These are called *combination rights*. The defined combination rights are:

`write` all rights but **`a`** (i.e. **`lidrwk`**)

`read` **`l`** and **`r`** rights only

`all` all rights (i.e. **`lidarwk`**)

`none` no rights; this removes the group's or user's entry from the ACL entirely

Combination rights can be used in commands, as shown in the examples in section 7.6.2.

☞ A couple of notes:

- In general, users are granted all permissions on their home directories.
- When a child directory is created, it inherits the parent directory's ACL. The child directory's ACL can then be changed. Users of the child directory must have at least lookup rights (**`l`**) on the parent directory.

Examining a Directory's ACL

You can examine a directory's ACL rights with the command:

`%` **`fs listacl`** ***/path/to/directory***

This returns a listing of all the users/groups that have any permissions on the directory, and what the permissions are. The directory path can be absolute (starting from root) or relative to the current working directory. For example, if you run the command:

`%` **`fs listacl /afs/fnal.gov/files/wwwdocs/cd/webwg/tools`**

The system returns information in the format:

```
Access list for /afs/fnal.gov/files/wwwdocs/cd/webwg/tools is
Normal rights:
  lauram:www_cd_webwg_tools rlidwk
  nicholls:www_cd rlidwk
  hanson:newsmachine rlidwka
  nicholls:wwwdocs rlidwka
  system:administrators rlidwka
  system:anyuser rl
```

The group `lauram:www_cd_webwg_tools` has read, list, insert, delete, write, and lock permissions in this directory (all but administer permissions), i.e. the group has *write* rights. Any member of that group has these permissions in this directory.

Adding/Changing/Deleting a Directory's ACL

You can modify a directory's ACL for a particular AFS group or for an individual using the **`fs setacl`** command. The **`fs setacl`** command only changes the ACL for a single directory, not for a directory tree. The command syntax is:

`%` **`fs setacl -dir`** ***/path/to/directory*** **`-acl`** ***group permission(s)***

where ***group*** is either a group or an individual username. When it is a group, it must be entered in the format `{group_owner}:{group_name}`.

☞ We recommend that you generally define ACL entries for groups rather than individuals; it is much easier to maintain. When you need to add or remove permissions for an individual, it is easier to add/remove the user from one or more groups than to track down every directory for which the user appears in the ACL.

The directory path in the command can be absolute (starting from root) or relative to the current working directory. Any pre-existing permissions for the group or individual are invalidated; the specified permissions collectively become the new set of permissions. The permissions apply to all members of the specified group.

For example, to modify the ACL for the current directory (**`.`**) to include only read and lookup rights for any user (including unauthenticated users), enter:

`%` **`fs setacl -dir . -acl system:anyuser rl`** or, using combination rights syntax:

% fs setacl -dir . -acl system:anyuser read

See the combination rights (e.g., **read**) in section 7.6.2. The group `system:anyuser` is described in section 7.7.

A note for Web page providers: set the permissions for `system:anyuser` to `rl` on directories containing files that you want to make accessible via a Web browser.

To remove all permissions in an ACL for a particular group (or individual), issue the **fs setacl** command with no permissions, e.g.,

% fs setacl -dir *****/path/to/directory******* -acl *******group******* ""**

or, using combination rights syntax:

% fs setacl -dir *****/path/to/directory******* -acl *******group******* none**

See **man fs_listacl** and **man fs_setacl** for further information. (Note the underscores in the **man** command.)

7.7 AFS Protection Groups

An AFS protection group is a list of usernames grouped to share a set of permissions on one or more directories. Any user can include any existing protection group in any ACL within your AFS cell. A protection group is designated in the format:[1]

`{group_owner}:{group_name}`

AFS provides three predefined protection groups:

`system:anyuser` This is similar to world permissions in UNIX. Any AFS user (anywhere in the world, and not necessarily authenticated) can access files or directories, according to the permissions granted (e.g., read, write).

`system:authuser` This is a more restrictive version of `system:anyuser`. Only users who have authenticated within the local cell (`/afs/fnal` at Fermilab) may access files, according to the permissions granted (e.g., read, write).

`system:administrators`

This group includes only the few people in the `/afs/fnal` cell authorized to administer AFS.

As determined by your project's `/afs` area manager(s), you may need to manage, and possibly create, protection groups.

Groups can be owned by other groups or by individual userids. Group members often are not allowed to add or remove other members of the group. If a group is owned by a group, then all the members of the *owner group* can by default add or remove other members from the *owned group*. This can avoid problems when key individuals are unavailable. Having one group consisting of a few key individuals, and using this group as the owner for all your other groups is a nice, neat way to organize your groups. Find out from your `/afs` area manager how group ownership and permissions are assigned within your project or on your system.

1. You may encounter groups that do not have an owner prefix; these are special groups created by the system administrators.

AFS provides the **pts** command (protection server) for group-related tasks. Like the **fs** command, **pts** has several main options. Also like the **fs** command, you need to use an underscore between **pts** and the main option to access the man page. Issue the command **pts help** to list the main options (this list has been abbreviated to contain only the options we discuss in this section):

```
pts: Commands are:
adduser         add a user to a group
chown           change ownership of a group
creategroup     create a new group
delete          delete a user or group from database
examine         examine an entry
listowned       list groups owned by an entry or zero id gets orphaned groups
membership      list membership of a user or group
removeuser      remove a user from a group
setfields       set fields for an entry
```

7.7.1 Permissions for Performing Group-Related Tasks

Group characteristics (e.g., membership, ownership) can only be seen and/or modified according to the permissions set on the group. We refer you to the man page for **pts_setfields** (notice the underscore) for the full story, and present only a brief explanation here.

Every group has a set of five access flags, which represent permissions for performing sensitive tasks regarding (1) status, (2) ownership, (3) membership, (4) adding members, and (5) removing members. There is a **pts** main option associated with each of these tasks:

status (s)	**pts examine**
owned (o)	**pts listowned**
membership (m)	**pts membership**
add (a)	**pts adduser**
remove (r)	**pts removeuser**

Each flag has one of three possible values: its first letter in lowercase, its first letter in uppercase, or a hyphen. The value determines which users can issue the corresponding command option for the group as follows:

lowercase letter (`s`, `o`, etc.)	all members of the group
uppercase (`S`, `O`, etc.)	all users (i.e. `system:anyuser`)
hyphen (`-`)	group owner and members of `system:administrators` only

As an example, we'll issue a **pts examine** command and examine its output:

% pts examine lisa:uss-group

```
Name: lisa:uss-group, id: -316, owner: lisa, creator: hanson,
membership: 14, flags: S-M--, group quota: 0.
```

The permissions information is contained in the `flags` entry. The flags `S-M--` are the default flags when a group is created (all users can check status and membership information, only group owner and administrators can verify ownership and add/remove group members).

If you can't successfully issue one of the **pts** command options, check the access flags! Of course if you can't issue **pts examine** to check the flags, then you don't have status permissions for the group.

7.7.2 Listing Information about Groups

List Members of a Group

To list the members in a group, use the command:

% pts membership ***group***

For example:

% pts membership lauram:www_cd_webwg_tools

returns the output:

```
Members of lauram:www_cd_webwg_tools (id: -454) are:
   nicholls
   hathaway
   stolz
   george
   lauram
   dwalsh
   nelly
```

List Groups in which an Individual is a Member

To list the groups to which an individual belongs, again use **pts membership**, but with the user's id as the argument:

% pts membership ***username***

For example:

% pts membership aheavey

```
Groups aheavey (id: 6302) is a member of:
  nicholls:www_reports
```

List Groups Owned by Group or Individual

To show what groups a particular group or user owns, issue the command:

% pts listowned ***group***

where ***group*** is actually either a group or an individual username. If you try to list groups owned by someone other than yourself, you may find that you do not have permission to do so.

Here are a couple of examples. To check groups owned by the *group* `nicholls:wwwdocs`, issue the command:

% pts listowned nicholls:wwwdocs

Output is returned in the format:

```
Groups owned by nicholls:wwwdocs (id: -306) are:
  nicholls:www_cd_support
  nicholls:www_cd_mgmt
  nicholls:www_faw_events
  nicholls:www_orgs_folkclub
  nicholls:www_directorate
  nicholls:www_cd_ups
  nicholls:www_cd_webwg
```

To check groups owned by the *individual user* `lauram`, issue the command:

% pts listowned lauram

Output is returned in the format:

```
Groups owned by lauram (id: 1866) are:
  lauram:wwwmachine
  lauram:expwwwmachine
  lauram:expwwwadm
```

Show Group Ownership

To find a group's owner, use the command:

% pts examine -name ***group***

This is helpful to determine if a group is owned by an individual or a group. For example, to find the owner of the group `nicholls:www_reports`, run the command:

% pts examine nicholls:www_reports

```
Name: nicholls:www_reports, id: -378, owner: nicholls:wwwdocs, creator: hanson,
  membership: 5, flags: S-M--, group quota: 0.
```

Its output in the entry `owner` indicates that it is owned by a group (`nicholls:wwwdocs`), not by the individual `nicholls`.

7.7.3 Modifying Group Characteristics

Change the Owner of a Group

Note: It is best to change the owner of the group *before* you run **fs setacl** to add directory permissions for the owned group.

You can change ownership of a group using the command:

% pts chown -name ***owned_group*** **-owner** ***owner_group***

Let's take for example the group `owner1:groupname1`, where `owner1` is an individual. We want to change its ownership to a group. The group we want to own it is designated `owner2:groupname2`. We issue the command:

% pts chown -name owner1:groupname1 -owner owner2:groupname2

The owned group is now designated `owner2:groupname1`. Notice that it takes its owner designation from the owner group, and maintains its former group name. Here's a more real-life example for clarity:

```
% pts chown -name lauram:www_cd_webwg_tools -owner \
  nicholls:wwwdocs
```

The old `lauram:www_cd_webwg_tools` is now designated `nicholls:www_cd_webwg_tools`.

You can change a group's ownership to itself (and set the group's access flags appropriately if needed) to allow all members of the group to add/remove other members and perform other administrative tasks. To change the group's ownership to itself, issue the **pts chown** command with the same group as both arguments:

% pts chown -name nicholls:wwwdocs -owner nicholls:wwwdocs

The group designation `{group_owner}:{group_name}` does not change. If you need to reset the group's access flags, see **`man pts_setfields`**.

☞ Note that there is a potentially confusing consequence of the way the group names change. All groups *look like* they're owned by individuals. You can always issue the command:

`%` **`pts examine -name`** ***group***

to determine if the owner is an individual or a group, as shown under **Show Group Ownership** in section 7.7.2.

Add a Member

To add a member, use the command:

`%` **`pts adduser -user`** ***username*** **`-group`** ***group***

For example:

`%` **`pts adduser -user nelly -group lauram:www_cd_webwg_tools`**

The new member (`nelly`) must have an account on the system/cluster that mounts the AFS files he or she needs to access.

Remove a Member

To remove a member from a group, use the command:

`%` **`pts removeuser -user`** ***username*** **`-group`** ***group***

Create a Group

☞ Check with your `/afs` area manager before creating new groups. As groups proliferate, system management can become more difficult.

To create a new AFS protection group, use the command:

`%` **`pts creategroup -name`** ***group***

or, leaving off the **`-name`** option flag for simplicity:

`%` **`pts creategroup`** ***group***

Always enter a group in the format `{group_owner}:{group_name}`; don't enter only the `{group_owner}` portion. By default, the group owner is yourself.[1]

As an example, user *lauram* could run the command:

`%` **`pts creategroup lauram:www_cd_webwg`**

Remove a Group

To remove a group, use the command:

`%` **`pts delete -nameorid`** ***group***

1. There is an option (**`-owner`**) to set the owner to another individual or a group, but we recommend that you just use **`chown`** afterwards as described in section 7.7.3.

For example:

```
% pts delete -nameorid lauram:www_cd_webwg_tools
```

7.8 Implications of ACLs and Examples

The implementation of security in our Fermilab AFS cell is based on the notion that sharing information is more important than trying to protect it. Therefore, in most cases, the default has been to set ACLs to have the least security that is still reasonable. As currently implemented, all user home directories come with their ACL set so that `system:anyuser` has **`rl`** (read and lookup) permissions. A `Mail` subdirectory (used by the **MH** mail readers) is provided with more secure permissions.

The practical implication of this is that *anyone* on the internet running an AFS client can read your files, unless you change the ACL. Home directories are *writable* only by their owners (that is, the owner has **`rldiwka`** permission), but the world can read them. This is probably fine in many cases, but you should be aware of it and protect your files as you see fit, according to the guidelines presented below.

7.8.1 Protecting your Subdirectories

You can protect any single directory by changing its ACL to turn off permission for `system:anyuser` as well as for other users or groups that should be denied permissions. For example, if you use the mail reader **pine**, you may want to protect the message subdirectory `mail`. To make it completely inaccessible by `system:anyuser`, you'd issue this command:

```
% fs setacl $HOME/mail system:anyuser none
```

On the other hand, if you need to allow others to write into any of your directories, the default permission is too constraining. Say you are in a collaborative effort with user *mrchips*. You could allow him full permission in your `shared` directory by issuing the command:

```
% fs setacl $HOME/shared mrchips all
```

Recall from section 7.6.2 that if a user is in two or more groups that have different permissions on a directory, the user gets the union of the permissions.

Also, recall from section 7.6.2 that the **`fs setacl`** command only changes the permission for a single directory. If you have a directory hierarchy on which you want to change permissions, you'll have to use a UNIX command that traverses down the tree and changes all the directories as it goes. The **`find`** command can be used (see section 6.4.1), but it must be used judiciously in the AFS environment! **This is not recommended for inexperienced UNIX users (see section 7.11.2).** As an extension of the above example, say you had a directory hierarchy under `shared` to which you wanted to allow *mrchips* full access. The **`find`** command could be used instead of **`fs setacl`**, as follows:

```
% find $HOME/shared -type d -print -exec fs setacl -dir {} -acl\
  mrchips all \;
```

This would traverse down from the `shared` directory, changing the ACL for each of the directories it finds. The **`-print`** argument causes the system to print out all the directories the command encounters, allowing you to monitor the progress.

7.8.2 Protecting your Home Directory

We strongly recommend that you make your home directory world readable, and simply keep your private files in protected subdirectories. That said, ...

We do *not* recommend that you set the ACL on your home directory such that `system:anyuser` has no permissions (i.e. combination rights **`none`**) in order to keep your top level directory private. There are at least a couple of undesirable consequences:

- If you ever managed to log in unauthenticated, you wouldn't be able to enter your home directory. In fact you might not be able to log in at all, depending on the platform.
- The dot files in your home directory (e.g., `.login`, `.profile`) would be unreadable by unauthenticated system processes, which could cause them to break. For example, if `system:anyuser` does not have at least **`rl`** permissions on your home directory, the **sendmail** program will not be able to read your `.forward` file, and your mail forwarding will break.

If for some reason you really want to protect your home directory, you can do so to the extent that only **`l`** (lookup) permission is granted. However, you must make sure that any files that *must* be world readable, such as your `.forward` file, remain accessible. **Be aware that it is not always obvious which files must remain world readable in order to preserve the behavior of your environment.** You can protect your home directory as follows (Proceed with caution!):

1) Every AFS home directory is created with a subdirectory called `public`. Move the files that must remain world readable into this directory.

2) For each file moved into `public`, create a symbolic link in your home directory to the file in the `public` directory. Use the same filename.

3) *After* all the necessary files are moved and linked, *then* shut off all permissions except **`l`** (lookup) on your home directory.

Note that you *must* leave the **`l`** permission turned on or programs won't be able to find the file in `public` via the symbolic link.

Here is a sample session, assuming the only file that must remain world readable is `.forward` (there would actually be many files). It would be run from the user's home directory:

```
% mv .forward public/.forward

% ln -s public/.forward .forward

% fs setacl . system:anyuser l
```

7.9 AFS in Translator Mode

If your machine is accessing AFS via a translator node, you do not get authenticated when you log in, and in fact you can't run the **pagsh** or **klog** programs discussed in section 7.3.1. You cannot access your AFS login area. You only have access to directories for which an ACL is defined for `system:anyuser`. You have access to files in those directories according to the ACL entry for `system:anyuser` and the file owner bits, as usual.

At Fermilab most of the **UPS** products are set with *read* permissions (**`rl`**) for `system:anyuser`, thus allowing access to products maintained in the `/afs` products area from a machine running in translator mode. This is not true for products that are site-licensed (e.g., **edt**), which are made accessible only to users on site.

7.10 File Locking in AFS

The file locking mechanism in AFS does not really follow POSIX semantics. There are a few issues to mention:

- Files may only be locked as a whole; regions of a file may not be locked.
- File locking only works properly and reliably from a single system. If a file is locked from one client and an attempt is made to access the file from another client, the error `EWOULDBLOCK` is returned.
- There is no deadlock prevention in AFS, so deadlock situations can occur with file locking.
- Any program that attempts to use byte-range file locking in AFS will get a message from the cache manager warning that other users may be accessing the same file. Usually these messages can be safely ignored.

Generally we don't recommend including applications that depend on file locking in the AFS file space. Contact the helpdesk for more information or for resolution of a problem.

7.11 Frequently Asked Questions

7.11.1 Lost Access to Files

Why can't I access files I'm supposed to be able to edit?

First see what permissions you have by checking:

- which groups have `rlwidk` permissions on the directory
- that you are in at least one of these groups
- that the owner of the file has the standard **`rw`** UNIX permissions (see section 6.6.1)

Remember that authentication lasts only a set period of time (6 days in `/afs/fnal.gov`). If your authentication has expired, you will not have access to your files. You can reauthenticate by running **`pagsh`** and **`klog`** (see notes in section 7.3.1). Also we encourage you to use **telnet** in (which always authenticates to AFS) instead of using **rlogin**. See section 7.3.4 and chapter 13 for information on these utilities.

7.11.2 AFS and the UNIX Command "find"

Why shouldn't I use "find" in AFS space?

You should be *very careful* about using any command that traverses the file system on a machine that has `/afs` mounted. Be aware that a **`find`** on your system starting at root (**`/`**) will traverse the *whole* AFS file tree, including all the other AFS sites mounted on our cell. This is particularly problematic on some workstations, like Solaris 1 Suns, which by default run a nightly **cron** job that traverses the whole file system. Also note that the **`-mount`** and **`-xdev`** options (e.g., **`find / -mount ... -print`**) won't recognize an `/afs` file system boundary; **`find`** can't tell the difference between local files and AFS files. The **`find`** command is discussed in section 6.4.1.

7.11.3 Error Messages

What does it mean if I get an error message like this:

```
afs: Waiting for busy volume 536870945 in cell fnal.gov
```

This is an error message from AFS that indicates that you are trying to access a volume that is busy. There may be a number of perfectly normal reasons for this. It probably means either that your volume is in the process of being cloned for a nightly backup, or that one of the system administrators is in the process of moving your volume to a different disk because the one you are on is filling up. Normally the process that generated the error will just hang harmlessly for a few minutes until whatever locked the volume is finished. If this goes on for more than 20 minutes or so, contact the helpdesk and inquire about what is going on.

7.11.4 Retrieving Old Files

What if I need to retrieve yesterday's copy of a file?

Daily backups of the entire Fermilab cell are available from `/afs/fnal.gov/files/backup/`. For example if your home area is `/afs/fnal.gov/files/home/room3/joe`, you should be able to find yesterday's files in `/afs/fnal.gov/files/backup/home/room3/joe`. The backups are done at 12:45 a.m. seven days a week.

Not all of the volumes are readily available (as of November 1997), but work on that is proceeding. In the interim, if you cannot locate your files, contact the helpdesk to request that your backup volume be mounted so that you can access it.

7.11.5 Link Failure

Why did my link fail?

Hard links can be used only within an AFS volume, not across volumes. Generally, you should use symbolic links. Links are discussed in section 6.3.5.

Chapter 8: Printing

This chapter covers the standard FUE print utility **flpr**, as well as filter programs and techniques available for formatting the output prior to printing. **flpr** is the Fermi implementation of the standard UNIX **lpr** utility. Most software applications supplied by the Computing Division use **flpr** as a default.

Note that many X-windows applications allow you to specify in a pop-up window the print command you want to use.

8.1 The FUE Print Command: flpr

The Fermilab print utility **flpr** (pronounced "flipper") implements the **lpd** printing protocol, and can be used to print UNIX files to any print server using this protocol, including VMS print queues and most UNIX systems. Related commands are **`flpq`** (to check the print queue) and **`flpk`** (to kill a submitted print job; not supported on many hosts). The format of each command is:

`% `**`flpr [`***`options`*****`] [`*****`file`*****`] ...`**

`% `**`flpq [`***`options`*****`] [`*****`users|jobs`*****`] ...`**

`% `**`flpk [`***`options`*****`] [`*****`users|jobs`*****`] ...`**

The options are described in the man pages. All three programs will display a list of their options when given an argument of **`-\?`**. Which options actually function depends on the functions supported by the **lpd** host.

The following command produces the version of **flpr** in use, when and where it was created, the **flpr** defaults in effect, and the command format:

`% `**`flpr -\? -v`**

To print a file, generally you will just need to enter the following:

`% `**`flpr -q`** ***queue file***

This assumes that your implementation of **flpr** uses FNPRT.FNAL.GOV as the default host computer (most do), and that your printer is known to FNPRT (most are). If the **`-q`** option is not specified, the queue is taken from (in order of precedence):

1) environment variable *FLPQ*

2) user control file (`.flprrc`)

3) system control file(s)

4) data compiled into **flpr**

A full list of queues supported by FNPRT is available by running the command **obtain printer** to create the file `flp.printers` in your directory[1].

If your printer host is not known to **flpr**, you can use the option **-h** to specify it on the command line, set your own default printer host with the environment variable *FLPHOST*, or put an appropriate entry in your `.flprrc` user control file.

You can set environment variables such as *FLPQUE* and *FLPHOST* with the **setenv** command for the C shell family (see section 9.1):

% **setenv FLPQUE** *queue*

% **setenv FLPHOST** *host*

and for the Bourne shell family:

$ FLPQUE=*queue***; export FLPQUE**

$ FLPHOST=*host***; export FLPHOST**

A useful feature of **flpr** is the capacity to use *nicknames.* A nickname specifies the printer host computer and the queue as a pair. If you often use a printer unknown to FNPRT, this feature may be handy for you. You would then enter a print command in this format:

% **flpr -P** *nickname file*

The `flp.printers` file mentioned above defines many nicknames, most or all of which are associated with the host FNPRT, and is therefore more useful as a queue reference for the **-q** option. An example nickname is `wh10w_lw`, which is defined as:

`printer wh10w_lw fnprt.fnal.gov wh10w_lw`

Why are printer nicknames useful? If you're using a printer not recognized by the **flpr** defaults, you'll need to specify both host and queue unless you use a nickname. Also, we recommend that you use nicknames rather than queue names in shell scripts. If a printer fails or is removed, then its associated nickname can be redefined, and the scripts don't need to be changed. If a queue name is specified in a shell script, then this capacity does not exist. You can define a nickname yourself, or if a widely-used printer is down, it may be appropriate for your system administrator to establish a substitute printer by changing the nickname in `flp.printers`.

A printer nickname does not have to be the same as the printer queue name. It may be a short, easily remembered name, such as *ps* for a postscript printer or *lp* for a text printer. Such personal printer nicknames can be established in your personal **flpr** control file, `$HOME/.flprrc`.

As an example `.flprrc`, assume you use the FNPRT queues `b0trwqms_hp` for text printing, and `b0trwqms_ps` for Adobe PostPscript printing. You want to use the nicknames *lp* for text printing and *ps* for PostScript. Further, you would like the print jobname to contain identifiers for your login name and the file being printed. You could build a `.flprrc` as follows[2]:

```
host fnprt.fnal.gov
queue b0trwqms_ps
identifier %l$%b
printer ps fnprt.fnal.gov b0trwqms_ps
printer lp fnprt.fnal.gov b0trwqms_hp
```

1. This file actually specifies printer-queue combinations. These names are the same as the queues when FNPRT is the host, the case for most if not all of the entries.
2. The *identifier* in the file is equivalent to the **-I** option described in the **flpr** man page.

If this file exists, it will be used automatically. **flpr** also accesses any system-wide defaults files, such as `flp.defaults` and `flp.printers` stored in a system directory. The `flp.defaults` file contains default values for the host, username, printer queue, protocol temporary filename format and one or more associations between a nickname and a host and printer queue pair.

Another option for the **`flpr`** command that may occasionally be useful is **`-l`** ***login-name***. This specifies the remote login name to be used in case you want another user name associated with the print job. The default name is the same as that on the current system.

From the *UNIX Resources* page on the Web, see *Printing from UNIX at Fermilab* under **Our Local Environment** for information on the Fermilab print server, and how to list and check print queues.

8.2 Pre-Printing Options

a2ps and **psnup**, described below, are now in the **UPS** product **psutils**. You may need to run **`setup psutils`** before using them. They can be used from the UNIX command line as well as from many applications. **pr** is also available to format the pages before printing with **flpr**. **pr** is explained in the man pages.

a2ps and **psnup** are described in detail in the man pages and in the Fermilab DCD Release Note 41.0 *Users Guide to UNIX Printing Utilities*.

8.2.1 Convert ASCII to PostScript: a2ps

The **a2ps** utility converts ASCII to two-column PostScript by default, and encloses the text in boxes with headers indicating date, time, filename and page number. You also get a line saying "Printed by ***username*** from ***node***". You can override the defaults; see the man pages.

a2ps is useful if the printer can only accept PostScript. You can use it to produce different output formats (e.g., "2-up", where two pages of your file are shrunk to fit on a single sheet). It provides options to add items like line numbers, user, file, and system information to the output. The command syntax is:

`% a2ps [`*global options*`]` *files* `[`*positional options*`]` *files*

where ***global options*** apply to all files being printed, and ***positional options*** are applied only to the files found in the remainder of the command line (with the exception of **`-H`** as noted in the man page which only applies to the *next* file).

Here is an example where we pipe the output of **`a2ps`** to **`flpr`** (**`-p`** indicates portrait mode):

`% a2ps -p` *file1* `| flpr -q wh10w_lw`

8.2.2 Print Multiple Pages per Sheet: psnup

The **psnup** utility takes a PostScript file and prints it "***n***-up". This refers to how many pages get printed on a single sheet. Portrait and landscape mode alternate as ***n*** changes. Occasionally this procedure doesn't work due to problems with PostScript variance.[1] The syntax of the command is:

1. **psnup** inserts PostScript code in front of the PostScript it is given. Because PostScript is a programming language, this doesn't always work. The file may be printed in its original format, it may never be printed, or it may print fine!

% psnup [-p*n*] [-r] [-R] [-s*n*] *files*

where:

-p*n*	the number of pages of PostScript (or *spots*) that should be printed per sheet of paper. ***n*** is constrained to be 2, 4, 8, 16, 32, or 64.
-r	sets the first spot in lower right and progresses horizontally to the upper left. This is handy for when the pages have already been reversed by another program, and you are printing on a printer that reverses pages.
-R	sets the first spot in the upper left hand corner and progresses horizontally to the lower right. This is for non-reversing printers.
-s*n*	***n*** is the number of the spot you want the first page of output to be placed. *The first spot on the page is 0, not 1.*

8.2.3 Set Duplex Mode

Duplex mode refers to printing on both sides of the paper. This is often desirable for larger documents, although not all printers support this feature.

PostScript Files

To print a PostScript file in duplex mode on a printer that supports this feature, you must prepend some specific text to the file. An easy way to do this is to maintain this text in a separate file, and concatenate it to your PostScript file when you are ready to print it in duplex mode. The two-line text file, which we'll call `duplexps` here, must have the following contents:

```
%!
statusdict begin true setduplexmode end
```

To print your PostScript file in duplex mode, enter the command:

% cat duplexps *postscript_file* | flpr -q*duplex_printer*

Text Files

For text files, you need to prepend a different sequence to the file. The technique we present here works with any PCL-based printer with a duplex device in it (assuming no filters are in the way) . Since text files are easy to edit, just add the following line to the top of your file[1]:

```
<ESC>&l1S<ESC>&a0G
```

The trick is to get the *escape* character in your file! The sequence <Ctrl-[> will work for escape, but it must be preceded by a "quoting" character, which differs from editor to editor.

1. After the first ampersand (&), the characters are: lower case L (l), one (1), uppercase S (S). The character before the final G is a zero (0).

For this (these) editor(s):	... enter:
vi	<Ctrl-v> <Ctrl-[>
emacs	<Ctrl-q> <Ctrl-[>
nu/TPU, fermitpu	<Ctrl-v> <Ctrl-v> <Ctrl-[> <Ctrl-[>
EDT+	<Ctrl-[> <Ctrl-[>
NEdit	Use Insert Control Character from the Edit menu; escape is 27 decimal. The escape character may not echo on your screen depending on your font.

... to insert an escape character. Then print the file in the usual manner to an appropriate printer.

8.3 Other Print Utilities

There are two other traditional printing systems under UNIX. One of these is **lpr**, upon which **flpr** is based. **lpr** requires that each printer be defined in the local `/etc/printcap` file, which must be done by the system administrator. One of **flpr**'s advantages is that no such definition is required since the host name and print queue can be specified on the command line. **lpr** is the default print command used by some applications, however. You can obtain in your current directory the file `printcap` which contains all printcap entries for print queues defined on FNPRT by executing the command:

```
% obtain printcap
```

The other traditional printing system is called **lp**. This has its own set of files which must be set up by system administrators and is recommended for use only in situations where the other options are not available to you.

Chapter 9: Working Environment

This chapter describes the methods used to set up your working environment in UNIX. Some of these are standard UNIX (e.g., shell and environment variables), and some are provided and/or customized by FUE (e.g., the login scripts).

9.1 Shell Variables and Environment Variables

Every UNIX process runs in a specific *environment*. An environment consists of a table of *environment variables*, each with an assigned value. When you log in certain *login files* are executed. They initialize the table holding the environment variables for the process. (Exactly which files run will be made clear later in this chapter.) When this file passes the process to the shell, the table becomes accessible to the shell. When a (parent) process starts up a child process, the child process is given a copy of the parent process' table. Environment variable names are generally given in upper case.

The shell maintains a set of internal variables known as *shell variables*. These variables cause the shell to work in a particular way. Shell variables are local to the shell in which they are defined; they are not available to the parent or child shells. Shell variable names are generally given in lower case in the C shell family and upper case in the Bourne shell family.

9.1.1 C Shell Family

The C shell family explicitly distinguishes between *shell variables* and *environment variables*.

Shell Variables

A shell variable is defined by the **`set`** command and deleted by the **`unset`** command. The main purpose of your `.cshrc` file (discussed later in this chapter) is to define such variables for each process. To define a new variable or change the value of one that is already defined, enter:

`%` **`set`** ***name*=*value***

where ***name*** is the variable name, and ***value*** is a character string that is the value of the variable. If ***value*** is a list of text strings, use parentheses around the list when defining the variable, e.g.,

`%` **`set`** ***name*=(*value1 value2 value3*)**

The **`set`** command issued without arguments will display all your shell variables. You cannot check the value of a particular variable by using **`set`** ***name***, omitting **=*value*** in the command; this will effectively unset the variable.

To delete, or unset, a shell variable, enter:

`%` **`unset`** ***name***

To use a shell variable in a command, preface it with a dollar sign (**$**), for example **$*name***. This tells the command interpreter that you want the variable's value, not its name, to be used. You can also use **${*name*}**, which avoids confusion when concatenated with text.

To see the value of a single variable, use the **echo** command:

% **echo $*name***

If the value is a list, to see the value of the ***n***th string in the list enter:

% **echo $*name*[*n*]**

The square brackets are required, and there is no space between the name and the opening bracket.

To prepend or append a value to an existing shell variable, use the following syntax:

% **set** *name=prepend_value***${*name*}**

or

% **set** *name=***${*name*}***append_value*

☞ Note that when a shell is started up, four important shell variables are automatically initialized to contain the same values as the corresponding environment variables. These are *user*, *term*, *home* and *path*. If any of these are changed, the corresponding environment variables will also be changed.

Environment Variables

Environment variables are set by the **setenv** command, and displayed by the **printenv** or **env** commands, or by the **echo** command as individual shell variables are. Some environment variables are set by default (e.g., *HOME*, *PATH*).

The formats of the commands are (note the difference between **set** and **setenv**):

% **setenv [***NAME value***]**

% **unsetenv** *NAME*

where ***value*** is interpreted as a character string. If the string includes blanks (i.e. if it encompasses multiple values), enclose the string in double quotes ("), e.g.,

% **setenv** *NAME* **"***value1 value2* **..."**

The current environment variable settings can be displayed using the **setenv** command with no arguments.

To use an environment variable in a command, preface it with a dollar sign (**$**), for example **$***NAME*. This tells the command interpreter that you want the variable's value, not its name, to be used. You can also use **${***NAME***}**, which avoids confusion when concatenated with text.

To prepend or append a value to an existing environment variable, use the following syntax:

% **setenv** *NAME* **"***prepend_value***${***NAME***}"**

or

% **setenv** *NAME* **"${***NAME***}***append_value***"**

If the pre- or appended value is the value of a preexisting environment variable, enclose the variable name in braces, too, e.g.,

% **setenv** *NAME* **"${***NAME***}${XYZ_VAR}"**

☞ Appending and prepending is commonly used with the *PATH* variable, and a colon is used as a separator, e.g.,

```
% setenv PATH "${PATH}:${XYZ_DIR}"
```

9.1.2 Bourne Shell Family

The Bourne shell family does not really distinguish between shell and environment variables. When a shell starts up, it reads the information in the table of environment variables, defines itself a shell variable for each one, using the same name (also uppercase by convention), and copies the values. From that point on, the shell only refers to its shell variables. If a change is made to a shell variable, it must be explicitly "exported" to the corresponding environment variable in order for any forked subprocesses to see the change. Recall that shell variables are local to the shell in which they were defined.

Shell variables are defined by assignment statements and are unset by the **unset** command. The format of the assignment statement is:

$ *NAME=value* **[; export** *NAME***]**[1]

where there are no spaces around the equal sign (**=**). The **unset** command format is:

$ unset *NAME*

where ***NAME*** is the variable name, and ***value*** is a character string that is the value of the variable. If the string includes blanks (i.e. if it encompasses multiple values), enclose the string in double quotes, e.g.,

$ *NAME=***"***value1 value2* **..."**

The values of all the current variables may be displayed with the **set** command.

To use a variable in a command, preface it with a dollar sign (**$**). This tells the command interpreter that you want the variable's value, not its name, to be used. For example, to see the value of a single variable, enter:

$ echo $*NAME*

You can also use **${***NAME***}**, which avoids confusion when concatenated with text.

To prepend or append a value to an existing environment variable, use the following syntax:

$** *NAME=prepend_value***$*NAME*

or

$** *NAME=***$*NAMEappend_value*

☞ Appending and prepending is commonly used with the *PATH* variable, and a colon is used as a separator, e.g.,

```
$ PATH=${PATH}:${XYZ_DIR}
```

1. In most cases you will want to include the optional part of the command, so that it reads: *NAME=value***; export** *NAME*

9.2 Some Important Variables

These variables are important for all shells, unless noted otherwise.

DISPLAY

In order to use an X windows application, the environment variable *$DISPLAY* must be set correctly. Normally the Fermi login files set it correctly for you. Its value is of the form ***node:screen.server***. At Fermilab, this will generally look like ***node*`.fnal.gov:0.0`** where *node* is your machine name.

Finding the value of *name* is different on different types of terminals and workstations:

UNIX workstation	On a local window, run **`funame -n`** to get the name.
NCDX terminal	("X-terminal") Use the TCP/IP Name found under Show Version from the startup menu.
Tektronix X terminal	Enter setup by pressing F3 (the setup key). Select setup. Under Configuration Summaries, select TCP/IP. A screen appears with the nodename. This is the name you want.
VMS workstation	If the workstation is connected directly to the monitor, then use the name of the workstation.

HOME

Your home directory is the top of your personal branch in the file system, and is usually designated by your username, i.e. `/{path}/{username}`. The value of the variable *HOME* is the pathname of your home directory. The command **`cd`** without arguments always returns you to *$HOME*. In all shells except **sh**, the tilde (**~**) symbol used in filename expansion, expands to the value of this variable. For example `~/myfile` is equivalent to `$HOME/myfile`. `~{username}` is equivalent to the $HOME directory of user `{username}`.

PATH

The *PATH* variable lists the set of directories that the shell looks in to find the commands that you enter on the command line. (For the C shell family, the shell variable *path* takes its value from *PATH*.) If the path is set incorrectly, some commands may not be found. If you enter a command with a relative or absolute pathname, the shell will only search that pathname for it, and not refer to *PATH*.

If you include the current working directory, "dot" (`.`), in your *PATH*, the shell will always find your current working directory. This allows you to run executable files from your current working directory by typing in only the filename. The Fermi login files include the dot at the end of the path for you.

For the C shell family, see the following line in the `setpath.csh` file (see section C.1.3):

```
set path = ($path .)
```

For the Bourne shell family, see the following line in the `setpath.sh` file (see section C.2.3):

```
PATH = ${PATH}.:
```

See section 9.4 for information on these files. If "dot" is not in your *PATH*, then in order to execute a file, you need to precede the executable filename by **`./`** on the command line. This provides the current directory pathname explicitly.

LINES and COLUMNS

These variables control the number of lines and columns are displayed on your screen. The **csh** family syntax is:

% setenv LINES *n*

% setenv COLUMNS *n*

to set the number of lines or columns to ***n***.

Note for Solaris (SunOS 4.x) and OSF1: Use instead:

% stty –rows *n*

% stty –cols *n*

MANPATH

The *MANPATH* variable lists the set of directories that the shell looks in to find man pages.

SHELL

This variable is set to your default shell. Your default shell is determined by the last field in your password entry (see section 4.1.2).

ignoreeof

This shell variable is in **csh**, **tcsh**, **ksh** and **bash**. **sh** doesn't have it. When the *ignoreeof* variable is set, you cannot exit from the shell using <Ctrl-d>, so you cannot accidentally log out. You must use **exit** or **logout** to leave a shell (see section 2.2).

noclobber

This shell variable is in **csh**, **tcsh**, **ksh** and **bash**. **sh** doesn't have it. With the *noclobber* variable set, you are prevented from accidentally overwriting a file when you redirect output. It also prevents you from creating a file when you attempt to append output to a nonexistent file.

noclobber has no effect on utilities such as **cp** and **mv**. It is only useful for redirection. See sections 5.4.2 and 6.3.

9.3 The Alias Command

The **alias** command allows you to create your own names or abbreviations for commands by performing string substitution on the command line according to your specifications. Aliases are recognized only by the shell that invokes them; spawned processes do not "inherit" them.

Never use the actual command syntax as an alias for itself. If for some reason an error occurs and the login file which defines your aliases doesn't run, UNIX executes the standard version of the command. Normally you'd see an error message in this case, but what if you miss it? This can be disastrous. For example, if you are accustomed to using **rm** (*remove file(s)*, see section 6.3.6) as an alias for **rm –i** (*remove file(s), but prompt for confirmation*), when you run **rm** you will expect a confirmation prompt. If the alias didn't get defined you won't get a prompt, and you may end up removing files you need. That is why we suggest **rmi** as an alias for this command.

9.3.1 C Shell Family

The format of the **`alias`** command is:

`% alias [`*new* `[`*old*`]]`

When you enter ***new*** the shell substitutes ***old***.
The first example causes **`ls -l`** to be executed when the command **`ll`** is entered:

`% alias ll ls -l`

The next example creates the command **`dir`** to list directory files only:

`% alias dir 'ls -l | grep ^d'`

`grep` in this case searches for a **`d`** in the first column of each line.

9.3.2 Bourne Shell Family

Alias

The **`alias`** command is supported by **ksh** and **bash**, but not **sh**. For the entire Bourne shell family you can use *shell functions* instead of aliases; we discuss these below. The format of the **`alias`** command is:

`% alias` ***name*** **= '*alias_contents*'**

The first example causes **`ls -l`** to be executed when the command **`ll`** is entered:

`% alias ll='ls -l'`

The next example creates the command **`dir`** to list directory files only:

`% alias dir='ls -l | grep ^d'`

`grep` in this case searches for a **`d`** in the first column of each line.

Shell Functions

The Bourne and Korn shells support shell functions, which are similar to shell scripts in that they store a series of commands for execution at a later time. Shell functions are more quickly accessed than scripts because they are stored in memory instead of a file, and the shell preprocesses them. They can be used in place of aliases in order to be completely portable between **sh**, **ksh**, and **bash**. For information on the format of shell functions and how to use them, refer to the **sh** man pages or a UNIX text.

9.4 Tailoring Your Environment

This section discusses the FUE-customized login files (also called the *Fermi files*) used to set up your UNIX environment. All the files discussed in this section are printed in Appendix C. You will automatically have your own copy of these files in your home directory[1]. The default files exist in `/usr/local/etc` and you can recopy them to your home directory if you ever need to. Once you understand the functions of the various files, you can tailor them to suit your tastes.

Many of these files include sample code that you may want to activate. A pound sign (#) in the first column indicates a comment line. To activate a command line that's been "commented out", remove the #.

9.4.1 C Shell Family Fermi Files

The C shell executes *hidden* FUE-customized files at various times in your session. They include the files `.cshrc` and `.login`, which you may choose to further modify.

When you log out, the shell looks for a logout script in your home directory called `.logout`. FUE does not provide this file, but you can create it yourself, and it will get run automatically. This file is not required.

As an example, including the **clear** command in your `.logout` file contents clears the screen when you logout.

AFS

If your system is running AFS, we recommend including the **unlog** command in the `.logout` file. This is explained in section 7.3.4.

.cshrc and fermi.cshrc

Upon logging in, the first file to execute is the `.cshrc` located in your home directory. The shell also executes this file each time you invoke a new C shell, for example when you execute a C shell script or otherwise fork a new process.

Your `.cshrc` file first executes `/usr/local/etc/fermi.cshrc`, which:

- sets up the machine id, type, and operating system
- sets up **UPS**
- establishes a reasonable default *path* (and therefore *PATH*) by running `/usr/local/etc/setpath.csh` (see section C.1.3), and *MANPATH*
- sets *fermimail* as the standard mail alias

`fermi.cshrc` also calls `/usr/local/etc/local.cshrc` which may set other environment variables.[2] You do not have a copy of `fermi.cshrc` in your home directory; it is not designed to require individual customization. See section C.1.2 for the file listing.

The file `.cshrc` should contain all your aliases so that child processes have access to them; many suggested aliases are provided for you to activate, and you can define your own. You can also set shell variables (*noclobber* and *ignoreeof* are already set for you) and parameters that are local to a shell.

---

1. Exceptions are the `fermi.*` and `setup.*` files which are called directly from `/usr/local/etc`.
2. This is a file for things that the local system manager wants to add to the login scripts. It may or may not have been created on your system.

☞ Don't set any environment variables here. Any changes to their values will remain after you terminate a forked process, thus changing your standard environment for the duration of your login session. See section C.1.1 for the default file listing.

.login and fermi.login

The `.login` file is executed only at login time. After execution of `.cshrc`, the `.login` file located in your home directory is run. The default `.login` file first executes the file `/usr/local/etc/fermi.login`.

`fermi.login` performs several actions:

- sets **umask** (default file access permissions) so others can read and execute but not modify or delete your files
- determines the terminal type (and makes "best effort" at determining *DISPLAY* variable)
- sets common terminal characteristics
- sets a host of environment variables

You do not have a copy of `fermi.login` in your home directory; it is not designed to require individual customization. See section C.1.6 for the file listing.

Next, the `.login` file sets your prompt, and sets the variables *history* and *savehist*. You can edit your `.login` to modify your path and/or terminal settings, change the default values of environment variables or create your own, and/or include commands that you want to execute once, at the beginning of each session (for instance **`setup`** ***product*** commands). See section C.1.5 for the default file listing.

.logout

The C shell executes the `.logout` file in your home directory (if you have created one) when you log off the system.

Execute files to modify current session

☞ If you modify your `.cshrc` or `.login` files and you want them to take effect in the current session, you must execute them with the **`source`** command:

```
% source .cshrc
```

```
% source .login
```

This is explained in section 4.4.

9.4.2 Bourne Shell Family Fermi Files

The Bourne shell executes *hidden* FUE-customized files at various times in your session. When you log on in the Fermi environment, the `.profile` and `.shrc` files in your home directory are executed for **sh**, **bash**, and **ksh**. Your `.shrc` file is also executed at any time a new **bash** or **ksh** is invoked.[1]

---

1. On some of the more recent OS releases (e.g., AIX+4 and IRIX+6.4, and likely others in the near future) `/bin/sh` is a link (links are described in section 6.3.5) to the korn shell (**ksh**). **ksh** is a superset of **sh**, so this shouldn't present any problems for you. One difference is that your `.shrc` file gets sourced when you run `/bin/sh` scripts.

The name of the file `.shrc` is determined by the *ENV* environment variable which is set to `~/.shrc` in the standard `.profile`, it is not a standard UNIX feature.

.profile and fermi.profile

The `.profile` file first executes `/usr/local/bin/fermi.profile`. This file performs several actions:

- sets **umask** (default file access permissions) so others can read and execute but not modify or delete your files
- sets up the machine id, type, and operating system
- establishes *PATH* (by running `/usr/local/etc/setpath.sh`; see section C.2.3) and *MANPATH*.
- determines the terminal type
- sets a host of environment variables
- sets common terminal characteristics

You do not have a copy of `fermi.profile` in your home directory; it is not designed to require individual customization. See section C.2.2 for the file listing.

The `.profile` file sets your prompt and the variables that govern your history list, your default editor, and your command line editor. You can edit your `.profile` to modify your path and/or terminal settings, change the default values of variables[1] or create your own, and/or include commands that you want to execute once, at the beginning of each session (for instance **`setup`** ***product*** commands). See section C.2.1 for the default file listing.

.shrc and fermi.shrc

The `.shrc` file first executes `/usr/local/etc/fermi.shrc` which sets up **UPS** and performs some machine-dependent functions. See section C.2.5 for the default file listing.

The `.shrc` file should contain all your aliases[2] so that child processes have access to them; many suggested aliases are provided for you to activate, and you can define your own. You can also set variables (*noclobber* and *ignoreeof* are already set for you except in **sh**) and parameters that are local to a shell, and you can activate and define functions. See section C.2.4 for the default file listing.

Execute files to modify current session

If you modify your `.shrc` or `.profile` files and you want them to take effect in the current session, you must execute them with the `.` command:

```
$ . .shrc
```

```
$ . .profile
```

This is explained in section 4.4.

1. Remember for **sh**, there is not really a difference between shell and environment variables; see section 9.1.
2. Aliases are available for **bash** and **ksh**, but not for **sh**; see section 9.3.2.

9.4.3 Storing Customized Code

If you wish to maintain versions of distributed code customized to your own needs, we recommend that you store them in the following directories:

`$HOME/bin`	for machine-neutral code
`$HOME/bin.$ARCH`	for architecture-specific code; `$ARCH` is the value returned by **`funame -s`** (e.g., SunOS, IRIX).

The path names for these directories will be added to your *PATH* when the Fermi files are invoked.

9.5 X Terminal Support

A very helpful collection of information regarding X terminal configuration and use is maintained on the Web. From the Computing Division home page, select *X-terminals* under the heading **Systems and Networking**.

In this section we provide only a brief overview of the configuration and start-up information.

9.5.1 Configuration

If you have a new X terminal, you'll first need to configure it. The Computing Division currently provides boot/configuration/font service for NCD and Tektronix X terminals. There are two types of boot service, basic and complete. Basic service enables you to immediately boot the terminal, manage windows, reach the network, and have a complete set of fonts for most applications. Complete service includes the *Save Configuration* capability. Complete service requires a special request form. Refer to the Web pages for instructions.

Upon reboot, the terminals are by default loaded with a basic server and configuration. The server contains local telnet, setup, and console clients as well as two window managers. The configuration contains font service, access to the FNAL name servers, color tables, and other basic X terminal needs.

The service is provided on FNALU, but in practice, anyone with an NCD or Tektronix X terminal can use the service. In particular:

New users	If you have your terminal and there is no boot software available on your local nodes, you should use the central boot service.
Users in small groups	Groups with a small number of terminals may prefer to use the service instead of getting their own copy of the software.
Special projects	Projects which will be using the terminals for a short amount of time or without a particular host computer should use the service.
Users whose own server is down	For temporary use.

9.5.2 Connecting to Host Computers

For an NCD X terminal:

1) Select *Start Terminal* from pulldown menu on background

2) Type in host name

3) Login to host

For a Tektronix terminal:

1) Select *Telnet* From "Host Connections" menu in *Launcher*, or select or type in host name in *TekHostMenu*

2) In telnet window, type **`open`**

3) Login to host

For either type of X terminal, in order to run an X client program, you need to set your *DISPLAY* environment variable (see section 9.2).

9.6 Multimedia File Support

Applications such as Web browsers and mail handlers need to be able to handle files of many different types. The standard used for identifying multimedia file types is called Multipurpose Internet Mail Extensions (MIME).

When a server sends a document to a client, it usually includes a section that identifies the document's type so that the file can be presented properly. The identifier is called a MIME type, and consists of a general type (e.g., text, image, application, audio, video) and a subtype which specifies the format. These two elements are separated by a slash. Examples of MIME types are:

```
text/plain
text/html
image/jpeg
image/gif
application/postscript
```

A file called `.mailcap`[1] is used to map MIME types to external viewer programs, thus providing a recipe for displaying/playing multimedia files. When you first run **`setup www`** in a FUE environment, a default `.mailcap` file gets created in your `$HOME` directory if it doesn't already exist. If an earlier version of the file is found, the terminal displays a message saying that you can update it from the file in `$WWW_DIR/lib/TypeMap`. For most situations, the `.mailcap` file should be sufficient as provided.

Each entry in the `.mailcap` file consists of two fields separated by a semicolon (`;`). The first field is the MIME type in the format `type/subtype`. You may see asterisks used as wildcards to specify all of the subtypes of a particular type (e.g., `video/*`). The second field specifies the display command. It requires a full shell command, including the pathname for the external viewer and any command line arguments. Some examples of entries from the `TypeMap` file are:

```
image/xwd; display %s
image/x-xwd; display %s
image/x-xwindowdump; display %s
audio/*; sfplay %s
```

1. Its name refers to the fact that it was originally designed for multimedia mail, however its role has since expanded, and will probably continue to do so as more and more programs become multimedia.

```
video/mpeg; mpeg_play %s
video/*; animate %s
application/postscript; ghostview %s
application/x-dvi; xdvi %s
application/pdf; xpdf %s
```

The `%s` is a **printf**-style parameter for the string representing the filename.

Sometimes, if the MIME type is not sent in the file's header, the multimedia application displaying it needs to determine the file's MIME type from its file extension. In this case, the application references a file called `mime.types` which provides the mapping between file extensions and MIME types. This file is usually not required, and in fact a default `mime.types` file is not even provided.

To add support for a new MIME type with an associated file extension, you would need to create a `mime.types` file to provide the file extension mapping, and then edit your `.mailcap` file to include an entry that maps the new MIME type to an external viewer that can display the data.

For example, say you want to add support for MIME type `application` with (fictional) subtype `xyz`. The files come with the extension `xyzz`, viewable via the program **viewxyz**. You would need to create `mime.types` and include the following line in it:

```
application/xyz xyzz
```

Note there is no semicolon (`;`) in a `.mime.types` entry. Then in `.mailcap`, you would need an entry as follows:

```
application/xyz; viewxyz %s
```

More information on the `.mailcap` and `mime.types` files can be found on the Web under the heading **WWW** from the *UNIX Resources* page.

9.7 Terminal Characteristics

You can specify your terminal type to UNIX if the default is not suitable. To do so, enter the command for the C shell family:

`% `**`set term=`***termtype*

or for the Bourne shell family:

`$ TERM=`*termtype***`; export TERM`**

where ***termtype*** is the name of a terminal type supported on the system. vt100, vt220 and xterms are acceptable terminal types. If you always use the same kind of terminal, you may want to put this command in your `.login` or `.profile`. Note that the standard Fermi files attempt to set this variable correctly.

In Section 2.4 we listed some terminal control functions. Recall that you can display the settings with the **`stty`** command:

`% `**`stty -a`**

The format on each machine is different but should indicate approximately the same information. The following is the output from a Silicon Graphics workstation. The settings reflect the FUE defaults.

```
speed 9600 baud; line = 1;
intr = ^C; quit = ^; erase = DEL; kill = ^X; eof = ^D; eol = ^@; swtch = ^Z
lnext = ^V; werase = ^W; rprnt = ^R; flush = ^O; stop = ^S; start = ^Q
-parenb -parodd cs8 -cstopb hupcl cread clocal -loblk -tostop
-ignbrk brkint ignpar -parmrk -inpck istrip -inlcr -igncr icrnl -iuclc
ixon ixany -ixoff
isig icanon -xcase echo echoe echok -echonl -noflsh
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tab3
%
```

In this display the second and third lines display the FUE default control characters. The character **`^`** indicates the control key (e.g., **`^C`** represents <Ctrl-c>). Your reference books will most likely tell you to delete a character with the # key and delete a line with the @ key, **but this is not correct at Fermilab**. Use the character indicated as erase in the **stty** output for single character deletion, and **`kill`** for whole line deletion. The Fermi UNIX Environment defaults for these operations are the delete key and <Ctrl-x>, respectively.

You can display a description of all of the options reported by **stty** with the command:

% **man stty**

If you don't like the FUE defaults, you can also set these functions with the **`stty`** command. The form for setting them is:

% **stty** ***control-char c***

where:

control-char	is one of the functions in the table in section 2.4.
c	is the representation of the key to be used for that function. A control character is specified preceded by a caret: **`^x`** represents <Ctrl-x>.

Example:

% **stty kill '^y'**

There are two special representations: **`^?`** is interpreted as the delete key and **`^-`** is interpreted as undefined. You must include the quotes as shown in the example so that special characters are not interpreted incorrectly. You must be careful not to have two functions represented by the same key.

There are many other options that can be set with **stty**. Others that might be of interest are **`echoe`** which specifies that deleted characters are erased, and **`-tabs`** which specifies that the tab character be translated into the appropriate number of spaces. Refer to the man pages for more information.