Introduction to UNIX® System Problem-solving

- What do people mean when they say that the UNIX® system has
  - flexibility?
  - power?
- Tools? Why aren't other operating system commands thought of as tools?
- UNIX system provides user with straightforward, high-level facilities for writing new commands
  - commands derived from existing system commands: *tools*
  - tools derived from other tools
- Result
  - "intuitive" control over underlying processes
  - proliferation of user tools
  - "Each UNIX system installation reflects interests of its users."

High-level Interface / Low-level Control

Under the UNIX system user-level commands (processes) are recombined with

files — ability to encapsulate programs and to buffer inputs and outputs

pipes — output of $command_1$ made input to $command_2$

```
$ sort file | nl
```

(handwritten: ↑ line number filter)

redirection operators — alternative I / O streams can be specified or merged

```
$ sort file | nl > file.out 2>&1
```

(handwritten: command 2> file — all std error output redirect to file)

control flow operators — processes communicate via statement return value (true / false)

```
$ while
>     command1 succeeds
> do
>     command2
> . . .
>     commandk
> done
$
```

variables — processes communicate via string-valued data

file: `test.prog`

```
$ HERE=`pwd`
$ cd ${HOME}; who | grep ${LOGNAME} > hour_in
$ cd ${HERE}
```

High-level Interface / Low-level Control

- UNIX system command set provides user-level control over underlying processes
- Simple operators, syntax, and input/output facilities extend control
- Commands (processes) recombined with

files
    ability to encapsulate programs and to buffer inputs and outputs

pipesoutput of preceding command can be
    transformed to input of following command

redirection operators
    ability to specify alternative sources of input and output (existing input and output streams can be merged)

control flow operators
    processes communicate via statement return value (true / false)

variables
    processes communicate via string-valued data

Summary of Benefits

Learning the command-level facilities supported under the UNIX system accomplishes several objectives simultaneously. First, you learn the basic utilities that enable you to get on with your work:

- text editing
- electronic mail
- the file system
- other commands for routine work

Second, you become acquainted with tools supporting software development, such as tools for

- configuration management
- source code control
- function library construction
- compiler construction
- other program development tasks

Finally, you enter an environment that is rich in prototyping facilities, particularly for constructing interprocess communication models. This last category is particularly important for programmers who are preparing for telecommunications product development.

4th generation language —

(1) machine language
(2) Assembly
(3) General-purpose language (C, Fortran) →
(4) 4GLs — to enable you to frame your solution in terms of your problems

Summary of Benefits

- UNIX system command interpreter is a fourth-generation language (4GL): *shell language*
  - basic shell primitives are processes
  - language supports high-level concurrent programming
- UNIX system command set includes other 4GLs (*awk, lex, make, sed, yacc*)
- Shell language used as glue for recombining commands and languages
- Result: rich prototyping facilities
  - high-level IPC
    - polling
    - semaphors
    - setting process priority
    - trapping signals
    - more facilities
  - interprocess models constructed quickly and easily

What is an Operating System Anyway?

The view of the machine most users and programmers have is one engendered by software, not hardware. The operating system implements a *virtual machine* that abstracts the raw computing power of the hardware. The UNIX® system itself is called the *kernel*, emphasizing its centrality and isolation from other programs. The kernel, a set of programs standing between the user and the machine, provides the following:

- gives a community of users the illusion that each has his or her own single-user system (time sharing)
- enables users to share data in an orderly and economical fashion (information sharing)
- provides conventions for reading and writing stored data (input/output, or I/O)
- recovers from errors, whether hardware or software errors, and takes measures to protect the integrity of data and the computing environment

What is an Operating System Anyway?

- Software as an abstraction of a machine
- Kernel not necessary, but tames the hardware
- Manages hardware and software resources
  - time-sharing
  - scheduling
  - I/O
  - error-handling

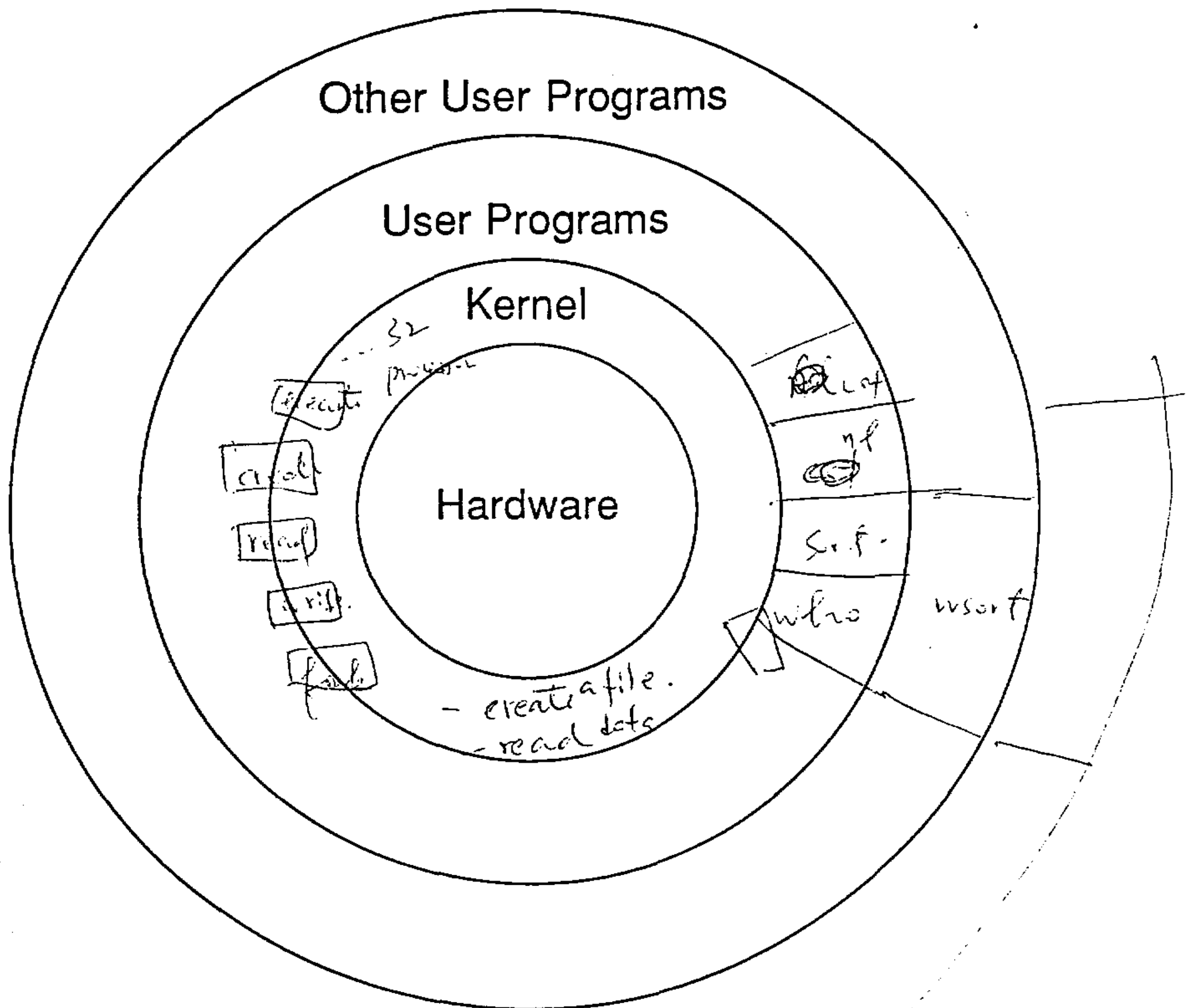
UNIX® System Architecture

The UNIX® system *kernel* abstracts machine services, offering them as a consistent interface having standard syntactical and naming conventions. Consequently, programs written to this kernel interface are easy to move between UNIX® systems. As the illustration on the following page implies, programs executed by the user are actually abstracted from the basic primitives residing in the kernel.

As the extra "onion layers" suggest, that process of abstraction is additive. That is, other commands can be abstracted from user programs, or *command*. And still others can be abstracted from there.

"the Design of the Unix operating system", —

UNIX® System Architecture



cat

read( )
write( )

UNIX® System Architecture (cont.)

The set of system calls constituting the kernel provides the basic building blocks for computation. Interfacing with hardware services, the system calls provide the user with the primitives necessary to support a general computing environment.

That is, the hardware services abstracted by the kernel for the user are not in the form of feature-oriented, monolithic applications. Rather, they are a spare set of primitives providing general computing services. UNIX® System V comprises about 64 system calls; of these, fewer than 32 are used for most activities [Bach 1986].

Used separately and in combination, the system calls are abstracted into higher level functions that are in turn abstracted into more general commands that in turn can be abstracted still further. The limits and direction of this *nested-Russian-doll* model of abstraction and encapsulation is determined by the user. It is a model of computing—that is, problem-solving—which is basic to the UNIX® system environment. Believing that pre-cast solutions frequently constrain as much as they assist, the designers of the UNIX® system developed decomposed "mini-solutions" that can be combined to solve particular problems and recombined to solve variants of those problems. This model of software reuse is provided for both programmers, who combine and recombine modules of source code, and non-programmers, who combine and recombine user commands.

The kernel implements a time-sharing system engendering the illusion that a dedicated processor serves each user. From a hardware perspective users' needs are efficiently satisfied. From a software perspective a community of users is to work in separate, protected environments using private tools, yet share on-line information and general software utilities.

This small system call interface is the set of universal rules from which all UNIX® systems derive. This fact gives rise to the cliche that UNIX® systems encourage portability; a program or library of programs written on one UNIX® system can be easily ported to another UNIX® system, which by definition observes the same set of rules.

UNIX® System Architecture (cont.)

- UNIX® system kernel a minimalist approach
- Single layer of well-chosen primitives exploiting the machine's resources (kernel)
- Highly portable
- Universal rules from which all else derives
- Highly compatible programs all having similar view of the world
  - uniform programs written in general-purpose languages
- The implications of these primitives, of this architecture will be seen at the user program level and throughout the module.
- Implements a general-purpose time-sharing system: multi-tasking, multi-user

Module Objectives

At the conclusion of the module, the student should

- be familiar with the most commonly used UNIX® system commands
- understand the UNIX® system orientation toward problem-solving
- have written several programs demonstrating a knowledge of tools and techniques

Module Objectives

- Ability to use system commands
- Understanding the "UNIX® system approach" to problem-solving
- Working knowledge of several UNIX® system languages:
  - shell
  - `sed`
  - `awk`
  - C
- Ability to prototype IPC applications
- Ability to request OS services from within C programs

What Is the UNIX® System from the User's Perspective?

The ability to combine commands, tools built from commands, and multiple languages into derived solutions requires the following prerequisites:

- an adequate set of primitives
- a consistent and simple set of conventions for combining primitives
- a sharply defined and consistent interface among things to be combined
- facilities designed as the glue to hold everything together

The UNIX® system is notable for these attributes as this unit will show. The flexibility implied here often allows the user to elect multiple solutions to a given problem.

What Is the UNIX® System from the User's Perspective?

- A well-defined orientation for solving problems
- A set of clear rules supplemented by spare, purposeful commands
- Yet multiple (different) solutions for many problems
- Can be used together using a concise syntax well suited to their recombination
- Consistent interface

Summary

- Emphasize design of UNIX® system
- Suggest its influence on an approach to problem-solving
  - decomposable/recomposable primitives
- Briefly describe operating system services

2 UNIX® System Tools

Overview A2-4

Introduction A2-7

Why the Emphasis on Tools? A2-8

cat Command A2-10

cat & sort Tool A2-12

A New Pipe Fitting: the Tee A2-14

A Longer Pipeline: cat, sort, & sed Tool A2-16

More Pipes: cat, sort, sed, & nl Tool A2-18

Little Languages A2-21

Specification-level Languages A2-22

Language for Table Construction A2-24

Language for Formatting Equations A2-26

Language for Field Identification A2-28

Combining Specification-level Languages Input A2-30

Combining Specification-level Languages Output A2-32

Summary A2-34

Overview

The UNIX® system implies a particular way of thinking about problems. A goal of this unit is to identify ''tools-based'' problem-solving.

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What is a tool?
- What is a pipeline?
- Can interpretive languages be recomposed and "piped"?

Why the Emphasis on Tools?

The word *UNIX* has come to imply its companion term *tools*. For the purposes of this module *UNIX® system tool* is defined as follows:

- a user-defined command derived from other commands or languages
- a command suited to a specific set of tasks rather than providing a general service
- a command that can be reused for future (perhaps unrelated) tasks

For the purposes of this module, we will make a distinction between a *command*—a utility offered by the UNIX® system—and a *tool*—a derived command constructed by the user—to clearly distinguish between standard, system primitives and non-standard, derived software.

This distinction is sometimes blurred. A few commands have a sufficiently rich syntax and range of operations—e.g., `sed` or `awk`—allowing the user to build tools from a single command. This prompts users to refer to such commands as tools.

Why the Emphasis on Tools?

- *Tools*:
  - general-purpose programs solving high-level problems
  - derived command made from system primitives (commands)
- Usually suited to immediate task
- Reflects interests of user
  - user able to use and create tools (toolboxes)
- Native commands spare, purposeful: no "special interest" commands for subsets of users
- Range of usages distinguishes command from tool
- Commands allowing fine level of specification are thought of as tools
- Are tools unique to the UNIX® System?
  - how does the UNIX® architecture encourage tool use?

cat Command

The `cat` command—an abbreviation of *concatenate*—displays the entire contents of a file. While the range of `cat`'s behavior includes displaying non-printing characters and tagging the ends of lines with markers, the central function of displaying a file's complete contents does not vary. While the *degree* of that operation is refined through *command options*—hyphen-prefixed characters that modify command behavior—the *kind* of operations remains constant. `cat` is thus primitive to entire display. It is left to other primitives to format a file's contents, to number lines, to segment the contents in user-directed ways, to display only specified ranges of lines, and so forth.

This implies, of course, prescribed techniques for a combined use of primitives.

cat Command

- Display contents of file `roster`

```
$ cat roster
Guy, Warren:Easton:PA:18042:(215) 250-5418
Johnson, Walter:Fayetteville:NC:28301:(919) 486-1522
Qazi, Naseem:Utica:NY:13504:(315) 792-7358
Dann, Wanda:Morrisville:NY:13408:(315) 684-6153
Strauss, Patricia:Providence:RI:02908:(401) 456-8038
Kurzweil, Jack:San Jose:CA:95192:(408) 924-3913
Geotsi, Georgette:Bridgeport:CT:06601:(203) 576-4737
Swanson, William:San Jose:CA:95192:(408) 924-3869
Potter, John:Lock Haven:PA:17745:(717) 893-2346
Ker, Jun-Ing:Ruston:LA:71272:(318) 257-2963
Menon, Unny:San Luis Obispo:CA:93407:(805) 756-2341
Johnson, Jeffrey:Havre:MT:59501:(406) 265-4128
Manchester, Mark:Morrisville:NY:13408:(315) 684-6155
Griscom, Priscilla:Groton:CT:06340:(203) 449-6772
Rider, Michael:Ada:OH:45810:(419) 772-2386
Kiesler, Thomas:Kansas City:MO:64110:(816) 235-1494
Richardson, Joseph:Lake Charles:LA:70609:(318) 475-5873
Levinson, Deborah:Tampa:FL:33620:(813) 974-4815
Hassan, Yassin:College Station:TX:77843:(409) 845-7090
Liu, Gonhsin:Bridgeport:CT:06601:(203) 576-4761
Heep, Uriah:Wapping:UK:SE4:(carrier pigeon)
$
```

cat & sort Tool

The `sort` command is a primitive for sorting lines in a file. Lines are sorted according to a range of criteria—e.g., alphabetical, reverse alphabetical, and so on—and with respect to specified key-words in each line (the first word in each line, by default). The central task of sorting, however, does not vary. A natural companion behavior would be to search, but the user must select a different utility for searching: `grep`. Segmenting the sorted material, editing it, or formatting it similarly is left to other primitives.

The single-mindedness of each primitive allows for its ability to be recombined. Recombination, indeed, would not be possible if each command's operations were not restricted and discrete.

The convention combining `cat` and `sort` on the following page is an operator called a *pipe* (|); the output of `cat` is said to be piped as input into `sort`. The new operation is a superset of the two primitives: it is now possible to sort lines containing non-printing characters as appropriate options are added to `cat`:

```
$ cat -vet roster | sort
```

The pipe expresses the metaphor of plumbing. The output of one command is channeled, or piped, as input into the next command down the pipeline.

cat & sort Tool

- Display the file contents in alphabetical order

```
$ cat roster | sort
Dann, Wanda:Morrisville:NY:13408:(315) 684-6153
Geotsi, Georgette:Bridgeport:CT:06601:(203) 576-4737
Griscom, Priscilla:Groton:CT:06340:(203) 449-6772
Guy, Warren:Easton:PA:18042:(315) 250-5418
```

A New Pipe Fitting: the Tee

The plumbing metaphor applies beyond simply piping one command into another. As with plumbing, the user may attach additional pipes, linking the output of *command*$_1$ as input into *command*$_2$, direct the resulting output of the combined *command*$_1$ and *command*$_2$ into *command*$_3$, and so on:

```
$ command1 file | command2 | command3
```

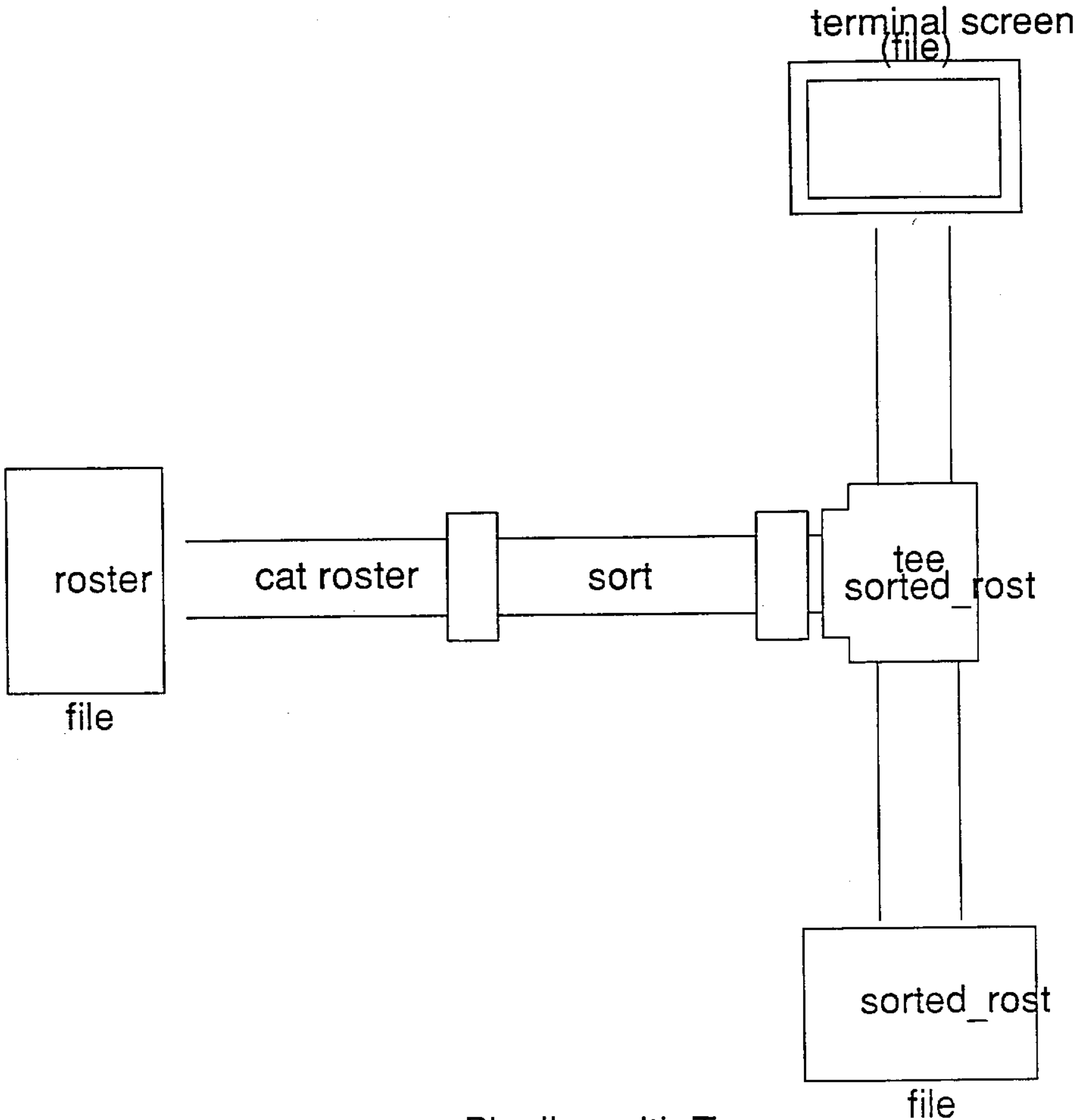
In addition, the resulting output can be directed to a *sink* (e.g., a file), and a *tee* pipe fitting can be attached to direct the stream of output in two directions at once (i.e., into a file and on your terminal).

A pipeline containing the `tee` illustrated on the following page is as follows:

```
$ cat roster | sort | tee sorted_rost
```

The output of the `cat` command is piped into the `sort` command, which in turn is piped into the tee. Finally, the `tee` command directs the output of the command both to a named file (`sorted_rost`) as well as to the terminal.

A New Pipe Fitting: the Tee



Pipeline with Tee

A Longer Pipeline: cat, sort, & sed Tool

In the example shown on the following page, another pipe is attached to add editing operations. The `sed` command is the UNIX® system *stream editor* transforming data while it is in the pipeline as opposed to when it is stored on disk. ("The Shell Programming Language," presented later, will discuss the ability to edit in a pipeline files that are stored on disk.)

The `sed` section of the pipeline shown on the following page truncates all characters beyond and including the first colon. Note that `sed` edits a file that has already been sorted, and that `sort` orders lines only after `cat` has displayed them.

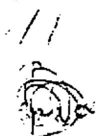
A Longer Pipeline: cat, sort, & sed Tool

- Display only the sorted list of names

```
$ cat roster | sort | sed 's/:.*//'
Dann, Wanda
Geotsi, Georgette
Griscom, Priscilla
Guy, Warren
Hassan, Yassin
Heep, Uriah
Johnson, Jeffrey
Johnson, Walter
Ker, Jun-Ing
Kiesler, Thomas
Kurzweil, Jack
Levinson, Deborah
Liu, Gonhsin
Manchester, Mark
Menon, Unny
Potter, John
Qazi, Naseem
Richardson, Joseph
Rider, Michael
Strauss, Patricia
Swanson, William
$
```

substitute pattern 1 with pattern

//



More Pipes: cat, sort, sed, & nl Tool

Still another pipe is attached for numbering lines of the resulting output from `cat`, `sort`, and `sed`. The `nl` command numbers the resulting output.

More Pipes: cat, sort, sed, & nl Tool

- Number the sorted list of names

```
$ cat roster | sort | sed 's/:.*//' | nl
     1 Dann, Wanda
     2 Geotsi, Georgette
     3 Griscom, Priscilla
     4 Guy, Warren
     5 Hassan, Yassin
     6 Heep, Uriah
     7 Johnson, Jeffrey
     8 Johnson, Walter
     9 Ker, Jun-Ing
    10 Kiesler, Thomas
    11 Kurzweil, Jack
    12 Levinson, Deborah
    13 Liu, Gonhsin
    14 Manchester, Mark
    15 Menon, Unny
    16 Potter, John
    17 Qazi, Naseem
    18 Richardson, Joseph
    19 Rider, Michael
    20 Strauss, Patricia
    21 Swanson, William
$
```

Introduction

This introductory unit provides a general description of the design of the UNIX® operating system. It lists the basic services provided by the operating system, but, more important, it also describes the design of the programming environment and suggests how that design constitutes an approach to problem-solving.

Objectives

- Provide general description of role of UNIX® operating system
  - describe operating system services
- Provide general overview of design of UNIX® system
  - suggest problem-solving model supported by design
- Relate UNIX system concepts and services to general problem of interprocess communication (IPC)

Introduction to UNIX® System Problem-solving

The UNIX® system implies an orientation toward problem-solving as much as it does an interface between a machine and a user. Unfortunately, interested newcomers to the UNIX® system are often left with abstractions such as *flexible* and *powerful* to explain, or fail to explain, the system. The primary aim of this module is to identify the concrete ideas that characterize the system as well as the programming environment those ideas define.

The UNIX® system environment is also said to reflect the interests of those who use it rather than the interests of software application developers or the designs of operating system developers. The ease with which users and programmers are able to make tools suited to their own work, no doubt, has encouraged this idea. Consequently, the twin concepts of the *tool* and the *little language* are central to the introductory part of the module.

Little Languages

Specification-level Languages

The term *specification-level language* refers to a number of high-level languages found in the UNIX® system environment suited to specific activities. That is, the model of well-defined primitives discussed above also applies to languages. Rather than supporting one or two large, general-purpose languages for all programming tasks, the UNIX® system environment supports several small languages each providing a "mini-solution" to its appropriate task.

While the UNIX® system environment supports general-purpose programming languages—C, C++, Fortran, Pascal—it also provides higher-level, interpretive languages that allow the user to specify solutions using better abstracted facilities. High-level languages such as `awk`, `eqn`, `lex`, `pic`, `tbl`, and `yacc` are termed *specification-level languages*.

What is a well-abstracted or specification-level language? It is one that allows you to frame a solution *in terms of the corresponding problem*. That is, a problem should imply the notation that will solve it. It follows that developing a clear understanding of a problem is a significant part of learning its solution language. Specifying formatted tables implies a tabular notation. Formatting equations implies a language suited to equation description. Recognizing lexical fields implies statements well suited to field recognition.

By contrast, general-purpose languages, such as Pascal or C, force the programmer to use a reserved name list and syntax that may not bear resemblance to the problem being solved, sometimes leading to less than purposeful or concise code. By providing mini-solution languages, the UNIX® system environment encourages clearer and more "intuitive" solutions. These smaller solutions can be combined to form larger, more complex solutions.

Specification-level Languages

- *Specification-level languages*
  - high-level
  - well-abstracted
  - allows you to frame a solution in terms of the corresponding problem
- Important feature of the UNIX® system environment
- Known as *little languages*
- Some examples
  - `tbl`
  - `eqn`
  - `awk`

Language for Table Construction

The language `tbl` allows you to specify an input that is compatible with the problem you are trying to solve. The problem is to display data in a tabular format. `tbl`'s input is data entered as tab separated fields. Positioning and type of box is specified in line 2; adjustment of columns is specified in lines 3-4; the specification c results in data that is centered in each column; the

Language for Table Construction

tbl Input:

file: tbl_in

```
.TS
center, box;
c c
l l.
State Ⓣ Capitol
=
Connecticut Ⓣ Hartford
Maine Ⓣ Augusta
Massachusetts Ⓣ Boston
New Hampshire Ⓣ Concord
Rhode Island Ⓣ Providence
Vermont Ⓣ Montpelier
.TE
```

tbl Output:

State	Capitol
Connecticut	Hartford
Maine	Augusta
Massachusetts	Boston
New Hampshire	Concord
Rhode Island	Providence
Vermont	Montpelier

Language for Formatting Equations

The actual problem domain of formatting equations is digital typography. Few scientists want to solve the problem in such terms, however. The language `eqn` enables the user to think in terms of the problem; generally speaking, you use `eqn` to specify equations in the terms in which mathematicians describe them.

Language for Formatting Equations

eqn Input:

file: eqn_in

```
.EQ
a + b over c = pi
.EN
```

eqn Output:

$$a+\frac{b}{c}=\pi$$

eqn

Language for Field Identification

The awk language provides a wide range of features. An important facility is field identification using an intuitive notation. In line 1 of the input shown on the following page, the field separator (FS) is specified to be a colon (:). The first and fifth fields are printed (line 2) using a numeric specification ($1 and $5). Literal strings are denoted by enclosing them in double quotes.

Language for Field Identification

awk Input:

file: awk_in.k

```
BEGIN   { FS = ":"; }
        { print $1 " --> " $5; }
```

awk Output:

```
Guy, Warren --> (215) 250-5418
Johnson, Walter --> (919) 486-1522
Qazi, Naseem --> (315) 792-7358
Dann, Wanda --> (315) 684-6153
Strauss, Patricia --> (401) 456-8038
Kurzweil, Jack --> (408) 924-3913
Geotsi, Georgette --> (203) 576-4737
Swanson, William --> (408) 924-3869
Potter, John --> (717) 893-2346
Ker, Jun-Ing --> (318) 257-2963
Menon, Unny --> (805) 756-2341
Johnson, Jeffrey --> (406) 265-4128
Manchester, Mark --> (315) 684-6155
Griscom, Priscilla --> (203) 449-6772
Rider, Michael --> (419) 772-2386
Kiesler, Thomas --> (816) 235-1494
Richardson, Joseph --> (318) 475-5873
Levinson, Deborah --> (813) 974-4815
Hassan, Yassin --> (409) 845-7090
Liu, Gonhsin --> (203) 576-4761
Heep, Uriah --> (carrier pigeon)
```

$awk -f

Combining Specification-level Languages
Input

A source of confusion derives from the fact that languages and the user commands that interpret them frequently have the same name. Thus, the awk command is the interpreter for the awk language. This fact explains how languages can be combined. Like other commands, a language interpreter accepts input from the output of other commands; conversely, the interpretive language can generate output to be read by other commands.

The following page demonstrates the interfacing of four languages and three UNIX® system utilities. The sed language prepares the contents of the file roster to be processed by tbl. That is, tbl expects its fields to be separated by tabs. The awk language is used to print fields selectively. Note that awk absorbs its fields delimiters (:) during processing. tbl interprets its notation and emits a different notation, which nroff will understand. The col command filters the formatted table, so it can be printed on the terminal screen or paper. The pg command allows a sequence of separate screenfuls to be viewed at the terminal.

The UNIX® system utility sort, as we saw earlier, is used alphabetically to sort the roster of names.

Combining Specification-level Languages Input

file: `roster_tbl`

```
$ sed '
      s/:/ Ⓣ &/g' roster |

sort |

awk '
        BEGIN {
        FS = ":"
        print ".TS"
        print "expand, allbox;"
        print "c c c c"
        print "l l c c."
        print "Name Ⓣ Town Ⓣ St Ⓣ Phone"
        }

        { print $1, $2, $3, $5; }

        END { print ".TE"; }' |

tbl |

nroff | col | pg
```

Combining Specification-level Languages (cont.)
Output

The output shown on the following page begs the question, ''Couldn't I have specified this on my What-You-See-Is-What-You-Get (WYSIWYG) formatter?'' The answer, of course, is, ''Yes.'' The algorithmic solution using the little languages has solved the problem *generically*, though. That is, all such problems can now be solved with a general solution. The file `roster` can be replaced with any file having the same need to convert any four fields into a table.

Combining Specification-level Languages (cont.) Output

Name	Town	St	Phone
Dann, Wanda	Morrisville	NY	(315) 684-6153
Geotsi, Georgette	Bridgeport	CT	(203) 576-4737
Griscom, Priscilla	Groton	CT	(203) 449-6772
Guy, Warren	Easton	PA	(215) 250-5418
Hassan, Yassin	College Station	TX	(409) 845-7090
Heep, Uriah	Wapping	UK	(carrier pigeon)
Johnson, Jeffrey	Havre	MT	(406) 265-4128
Johnson, Walter	Fayetteville	NC	(919) 486-1522
Ker, Jun-Ing	Ruston	LA	(318) 257-2963
Kiesler, Thomas	Kansas City	MO	(816) 235-1494
Kurzweil, Jack	San Jose	CA	(408) 924-3913
Levinson, Deborah	Tampa	FL	(813) 974-4815
Liu, Gonhsin	Bridgeport	CT	(203) 576-4761
Manchester, Mark	Morrisville	NY	(315) 684-6155
Menon, Unny	San Luis Obispo	CA	(805) 756-2341
Potter, John	Lock Haven	PA	(717) 893-2346
Qazi, Naseem	Utica	NY	(315) 792-7358
Richardson, Joseph	Lake Charles	LA	(318) 475-5873
Rider, Michael	Ada	OH	(419) 772-2386
Strauss, Patricia	Providence	RI	(401) 456-8038
Swanson, William	San Jose	CA	(408) 924-3869

Summary

The "Summary" focuses primarily on pipelines. Note that the pipe operator (|) transforms output from the command (or pipeline) preceding it into input for the command (or pipeline) following it.

Summary

`cat` *file*

Concatenates contents of *file* (output written to terminal screen by default)

`cat` *file* `| sort`

Concatenates contents of *file* and then sorts

`cat` *file* `| sort | sed 's/:.*//'`

Concatenates contents of *file*, sorts, and then edits each line, deleting all characters from the first colon to the end of the line

`cat` *file* `| sort | sed 's/:.*//' | nl`

Concatenates contents of *file*, sorts, edits each line, deleting all characters from the first colon to the end of the line, and then numbers each line

Introduction to Files

Overview

Encapsulation and composability have been identified as important issues in the context of the UNIX® system. This unit will focus on important facilities for encapsulating commands (and commands of commands) and for organizing your environment.

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What is the basic facility for storage under the UNIX® system?
- What is the overall design for data storage?
- How are commands encapsulated for reuse?

Files and Directories

Files

When information is not being processed, it is usually stored on disk and associated with a user-defined name of fourteen or fewer characters. This named place on disk is called a *file* and is the primary building block for all stored information in the UNIX® system environment. From the user's perspective, these files are much like ordinary office files. Each file has a name, contents, location, and some descriptive information such as who owns it, who can read or write in it, and how big it is [Kernighan 1984]. Files can contain letters, memos, phone numbers, executable programs, source code for compiled programs, and even data used by programs, machine language, and other non-textual material.

Files are organized so that your information is safely stored away from other users on the system,

Files

- A named location on disk
- Contains ASCII characters, data, binary sequences
- Can be read, written, executed
- No distinction among file "types"
  - text files, data files, object files all share same characteristics

Encapsulating a Tool

We have seen that new tools are made by recombining primitives. The file provides the container for encapsulating derived operations; it is the command you type to execute the new tool. Typing the pipeline into a file, the user is able to reinvoke the tool by simply typing the file name.

Without cluttering the public interface of the operating system, the user's environment can, nonetheless, be extended with these encapsulated tools. Alternatively, tools proven to have a general value can be stored in a public place where a community of users can access them, and perhaps use them as building blocks for still other tools.

The ability to build private environments that are rich in user-defined tools helps to explain another UNIX® system cliche: the UNIX® system environment reflects the interests of the people who use it.

Encapsulating a Tool

- User-defined tool set enhancing the operating system
- Accessible to user alone or community of users
- Overhead on the "programmer" negligible
- Result: an environment that reflects the interests of its users

Directories

The user's files, and therefore work environment, are organized in tree-like groups called *directory structures*. A *directory* is a file that holds other files—both regular files and other directories. The contents of a directory, then, are the names of the files it contains.

Directories

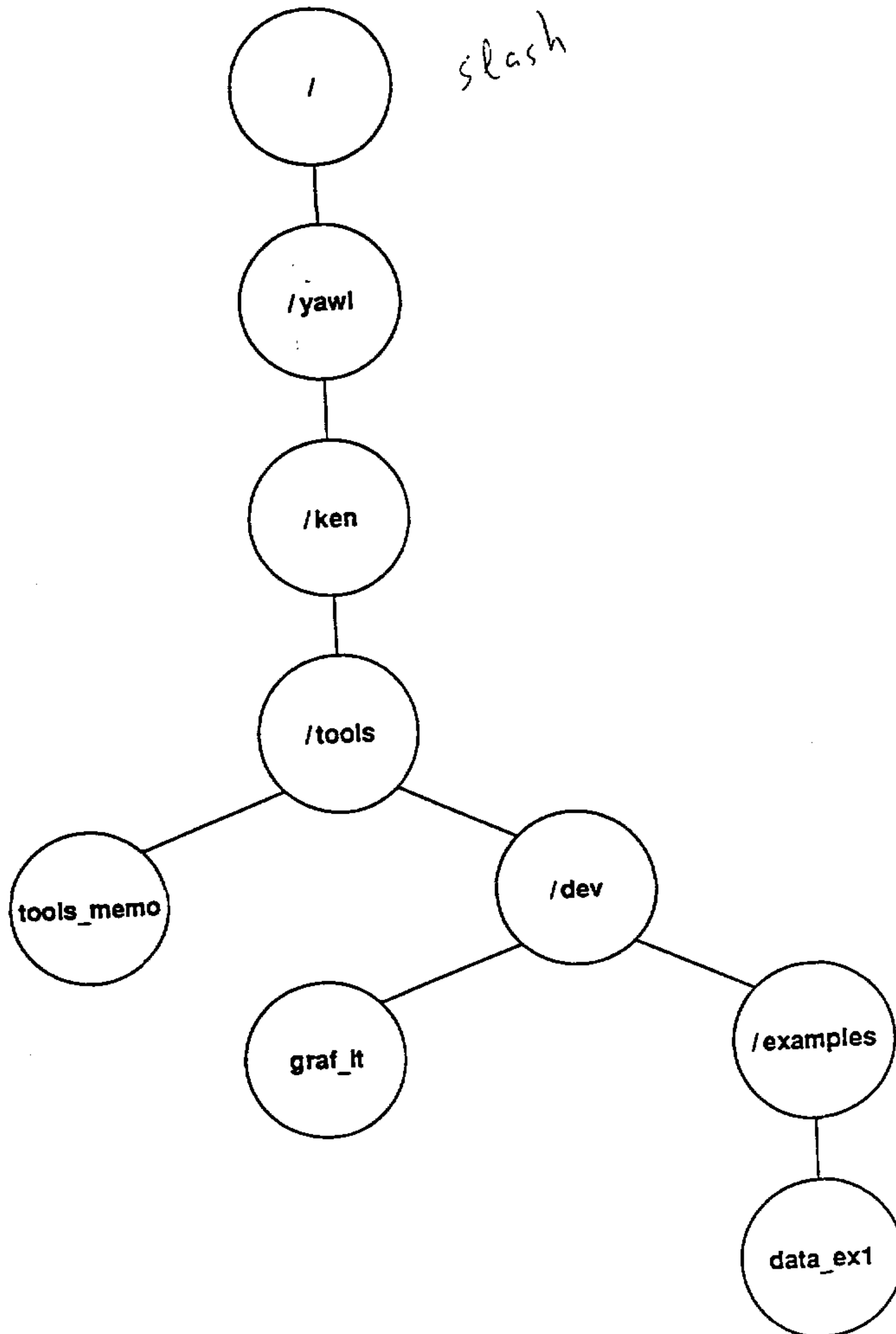
- Files whose contents are other file names
- Inverted tree
- Hierarchical, subordinating structure
  - no theoretical limit on depth

An Inverted Tree

A directory containing directories that in turn contain directories implies a model of subordinate hierarchy. This model is expressed in the metaphor of an inverted tree. A single root occupies the topmost position (known as the *root node*). From this root grow several trunks (known as *files systems* because all branches below are said to be on that trunk or file system). Below these trunks grow branches, which in turn can support other branch growth, and so forth.

The directory structure on the following page illustrates part of the `yawl` file system. Under `yawl` we see a user's login directory, or *home directory*, `ken`. All structure below this level reflects the growth of the user's work environment. The user's home directory contains one file, the directory `tools`. This directory contains a regular file, `tools_memo`, documenting the project, and a directory `dev`, where development occurs. The directory `dev` contains a regular file `graf_it`, a tool for formatting data in scatterplots, as well as a directory containing data to be formatted—in this case a single file name `data_ex1`.

An Inverted Tree



Manipulating Text in Files

The text file is the predominant file, and the environment supports a rich set of tools for manipulating ASCII text. These include executable programs (input to interpretive languages) and source code for executable programs (text that is converted into machine language), instructions for formatting documents and formatted documents, information that can be generated by tools and information that can be supplied as input to tools.

While the structure of files is general and flexible (an arbitrarily long stream of bytes), each command expects its own set of conventions. For example, some commands or languages, such as the typesetting language `troff`, expect an input file to be constituted of lines (records); that is, each input line of the file ends with the *newline character*, octal 012. Other commands such as `awk`, a language for scanning ASCII text, match individual strings of characters, known as *fields*. Thus, it is useful to discuss files in terms of records (input lines) and fields (subdivisions of records).

Manipulating Text in Files

- What is text?
  - C program source code
  - `awk` specifications
  - data (e.g., `roster` file)
  - documents
  - any sequence of ASCII characters
- Files can be thought of in terms of lines (records) and line segments (fields)
- Records can be reordered or selectively grouped according to keywords
- Fields can also be selectively grouped forming columns of file data

Text File Commands

Given the flexibility of files and the uniformity of their data (ASCII), it is not surprising that the UNIX® system environment supports a large set of commands for manipulating text in files. These commands are distinct from little languages:

- generally used at the command line, and not encapsulated as small programs in a text file
- do not imply an accompanying notation, but usually used with one or two command options and one or more target files

Because these are command-line-oriented and because their behavior is specified by command options rather than by a programmable instruction set, options will be listed for each command.

Text File Commands

- Commands for record manipulation:
  - `echo`: writes string (argument) as record on `stdout`
  - `grep`: groups records according to common lexical strings
  - `sort`: reorders records according to alphabetical or numerical sequences
  - `nl`: numbers records
  - `wc`: counts records, (words and characters)
  - `diff`: compares contents of two files (record-oriented)
  - `join`: merges records
- Commands for field manipulation:
  - `awk`: manipulates fields according to position in record
  - `cut`: selects fields according to position in record
  - `paste`: appends fields selected with `cut`

Editing a Text File

Editors supported on the UNIX® system (or in any environment) abound. This module will instruct you in three of them:

- `ed`
- `vi`
- `sed`

`ed` and `sed` are important from a tools point of view. `vi` is a commonly used editor for manual, interactive text modification.

Editing a Text File

- Commands for editing text:

Commands

General Form of Commands

As we have shown, the command is the user's basic building block. The interchangeability implied by this metaphor depends upon two things:

- that commands are able to accept input from and direct output to other commands
- that commands have a consistent form for usage

The pipelines demonstrated above are instances of the former. The general form of UNIX® system commands is shown on the following pages.

Although the operations commands provide range from simple concatenation (`cat`) to programming languages (`awk` and `troff`, among others), most commands conform to the following general form:

```
$ command [options] [command arguments]
```

where *command* is a standard UNIX® system user command, *options* are a list of characters, each preceded by a hyphen, for refining command behavior, and *command arguments* are inputs appropriate to the command. For example, the arguments to the command `mailx` are user login names, the arguments to the command `echo` are literal strings, and the arguments to the command `test` are other commands, values to be tested, or strings to be compared. The usual case, however, is that *command arguments* are files.

By default, commands read input from the *standard input*(a stream of bytes) that may come from the output of other commands or from the user's keyboard. In addition, commands write output to the *standard output*(a stream of bytes) that flows to the terminal screen (a file) by default, may flow as input into other commands, or may be redirected into a named disk file.

The square brackets used in the preceding general form indicate that the enclosed elements are optional. Some commands, such as `login`, which allows you to identify yourself to the system, accept no options. Other commands, such as `tty`, which tells you the name of your terminal, accept no arguments. Still other commands, such as `line`, which reads a line of input and prints it as output, accept neither options nor arguments.

Nonetheless, most commands, whether they are standard utilities or user-defined tools, all follow the same syntax standard to allow for predictable usage and simple rules for recombination.

General Form of Commands

- General form:

[illegible] [options] [command arguments]

Command Input / Output

As we have mentioned, the UNIX® system file structure is flexible—a file is an arbitrarily long stream of bytes. This is an important underlying assumption of pipelines. Processing data in-stream allows for an easy flow of information from one command into another.

The UNIX® system sets aside three special files for managing input, output, and user-level diagnostics: *standard input*, *standard output*, and *standard error*. Standard input is the file from which a command expects to read the information it will process. It is possible to redirect input using the redirection operator <:

```
$ sort < temp
```

where `temp` is a file to be sorted. But by default, `sort` reads from the standard input file. This is an essential property of most commands: if no files are specified, the standard input is processed [Kernighan 1984]. This allows the user to simply type at commands:

```
$ sort
ghi
abd
def
^D
abc
def
ghi
$
```

The standard output is the file to which most commands write by default. If the standard output is not redirected as input to another command or redirected into a named file, the terminal screen is the destination for this stream by default. The standard input is redirected into a named file using the redirection operator >:

```
$ sort < temp > sorted_temp
```

Command Input / Output

- Most commands read input and generate output
- Most commands can access
  - a standard source for input (*stdin*)
  - a standard destination for output (*stdout*)
- User's terminal keyboard is default standard input
- User's terminal screen is default standard output

  - other commands
  - other files

Redirection

Given that most commands use a default source of input (standard input associated with the terminal keyboard) and a default source of output (standard output associated with the terminal screen), alternative flow of input and output is known as *redirection*. This conveys the appropriate con- [illegible] for I/O

You have already seen most of the operators for redirection. The following is a complete list:

<	redirects input such that a command reads from a file
<<	redirects input such that a command reads from input that is explicitly specified between delimiters (special case of input covered later)
>	redirects output such that a command writes to a file (overwrites current file contents)
>>	redirects output such that a command writes to a file (appends to current file contents)
|	transforms a command's output into input for a succeeding command

Redirection

- Most commands already associated with standard input and output
- Subsequent arrangements for I/O called *redirection*

<	Redirects input such that command reads from file
<<	Redirects input such that command reads from explicit input specified between delimiters (special case of input covered later)
>	Redirects output such that command writes to file (overwrites current file contents)
>>	Redirects output such that command writes to file (appends to current file contents)
|	Transforms commands output into input for succeeding command

echo

The command `echo` writes its string argument as a record (by default to the terminal screen). We show the string argument enclosed in single quotes in the general form. This ensures that all characters are interpreted in their literal form.

echo

- General form:

```
$ echo 'string'
```

- Reads *string* and writes it as a record on output
- Single quotes ensure *string* read as literal characters
- Example: a simple greeting

```
$ echo 'Hello, world!'
Hello, world!
$
```

grep

The term *grep* derives from a command commonly specified in the line editor ed:

```
g/RE/p
```

This command globally (g) scans the file for a string (*RE*) and prints (p) the lines in which the string appears. A *global scan* in this context refers to a search on every line.

Because this ed command is so commonly used, it has been abstracted into a separate command. You do not need to edit a file to use it; you simply invoke the command grep from the command line specifying a string and a file.

grep

- General form:

```
$ grep string files
```

- Select all records from *files* containing *string*
- Example: display only the people from Pennsylvania

```
$ grep PA roster
Guy, Warren:Easton:PA:18042:(215) 250-5418
Potter, John:Lock Haven:PA:17745:(717) 893-2346
$
```

Some grep Options

Option	Effect
`-i`	`grep` ignores distinction between upper- and lowercase characters
`-v`	`grep` selects all records *not* containing *string* (inverted search)
`-l`	`grep` prints only names of files containing *string*
`-n`	`grep` prints records containing *string* each preceded by line number in file

sort

The command `sort` provides a general-purpose sorting routine. It can be used with a variety of options, which specify the sense of the sort. For example, the option `-n` anticipates that its data are numbers; the option specifies that lines be sorted according to arithmetic values.

The example on the following page specifies a *sort by dictionary order* (`-d`); only letters, digits, and blanks (horizontal tabs and horizontal spaces) are significant in comparisons. The values will be compared with respect to their sequence in the ASCII character set. The option `-f` specifies a

sort

- General form:

```
$ sort [ options ] files
```

- Reorder the records contained in *files*
- Example: alphabetize the people from Pennsylvania:

```
$ grep PA roster | sort -df
Guy, Warren:Easton:PA:18042:(215) 250-5418
Potter, John:Lock Haven:PA:17745:(717) 893-2346
$
```

Some sort Options

Option	Effect
[illegible]	sorts by "dictionary" order

nl

- General form:

```
$ nl files
```

- Output each input record preceded by the record number
- Example: number that list

```
$ grep PA roster | sort -df | nl
     1 Guy, Warren:Easton:PA:18042:(215) 250-5418
     2 Potter, John:Lock Haven:PA:17745:(717) 893-2346
$
```

wc

- General form:

```
$ wc [ files ]
```

- Count a file's
  - records
  - words
  - characters
- Command option `-l` causes `wc` to print *lines*
- Example: how many people in this class are from Pennsylvania?

```
$ grep PA roster | wc -l
      2
$
```

wc Options

Option	Effect
`-l`	`wc` prints line count only
`-w`	`wc` prints word count only
`-c`	`wc` prints character count only

diff

The `diff` command analyzes two files displaying differences in the second file with respect to the first. It implements a record-by-record comparison, printing only the two lines containing the difference from file to file. In the case of two identical files `diff` displays no output.

Note that `diff` is able to identify the *absence* of records as well as the presence of variant records.

In the example shown on the following page, the operator >> redirects the output of `echo` to the file `nroster`. Unlike the redirection operator >, the operator >> does *not* overwrite the contents of `nroster` but rather appends to its existing contents. Had `nroster` not existed, the operator >> would have caused it to be created.

diff

- General form:

```
$ diff file1 file2
```

- Display differences in $file_2$ with respect to $file_1$

roster?

```
$ grep CA roster > nroster
$ echo 'As, Also K.:Brrr:AK::(carrier penguin)' >>nroster
$
```

diff Output

The output of `diff` has the general form

```
n1 modification n2
< text1
> text2
```

where $text_1$ represents text from $file_1$ and $text_2$ represents text from $file_2$. n_1 indicates the line or range of lines from $file_1$; n_2 indicates the line or range of lines from $file_2$. *modification* indicates the modifications that will make files identical. Modifications are as follows:

c	change line(s) in $file_1$ to match counterpart line in $file_2$
d	delete line(s) from $file_1$ to match counterpart line in $file_2$
a	add line in $file_1$ to match counterpart line(s) in $file_2$

Lines from $file_1$ are preceded by the < symbol (left-hand side). Lines from $file_2$ are preceded by the > symbol (right-hand side).

In the example shown on the following page, lines 1-3 of `roster` would have to be deleted in order to match line 0 (before first line) of `nroster`. Line 5 of `roster` would have to be deleted in order to match line 1 of `nroster`. Lines 8-12 of `roster` would have to be deleted in order to match line 3 of `nroster`. Finally, line 14 of `roster` would have to be changed to line 5 of `nroster` in order for the files to be identical.

The command option `-e` causes `diff` to generate a set of `ed` (line editor) instructions for modifying $file_1$ to be identical to $file_2$.

cut

The command `cut` might be thought of as a vertically-oriented `cat`. Its options specify the *field delimiter* and the fields themselves. In summary, it is a command for printing individual columns from a file using a numeric specification.

diff Output

- Results of comparison

```
$ diff roster nroster
1,5d0
< Guy, Warren:Easton:PA:18042:(215) 250-5418
< Johnson, Walter:Fayetteville:NC:28301:(919) 486-1522
< Qazi, Naseem:Utica:NY:13504:(315) 792-7358
< Dann, Wanda:Morrisville:NY:13408:(315) 684-6153
< Strauss, Patricia:Providence:RI:02908:(401) 456-8038
7d1
< Geotsi, Georgette:Bridgeport:CT:06601:(203) 576-4737
9,10d2
< Potter, John:Lock Haven:PA:17745:(717) 893-2346
< Ker, Jun-Ing:Ruston:LA:71272:(318) 257-2963
12,21c4
< Johnson, Jeffrey:Havre:MT:59501:(406) 265-4128
< Manchester, Mark:Morrisville:NY:13408:(315) 684-6155
< Griscom, Priscilla:Groton:CT:06340:(203) 449-6772
< Rider, Michael:Ada:OH:45810:(419) 772-2386
< Kiesler, Thomas:Kansas City:MO:64110:(816) 235-1494
< Richardson, Joseph:Lake Charles:LA:70609:(318) 475-5873
< Levinson, Deborah:Tampa:FL:33620:(813) 974-4815
< Hassan, Yassin:College Station:TX:77843:(409) 845-7090
< Liu, Gonhsin:Bridgeport:CT:06601:(203) 576-4761
< Heep, Uriah:Wapping:UK:SE4:(carrier pigeon)
---
> As, Also K.:Brrr:AK::(carrier penguin)
$
```

cut

- General form:

```
$ cut -ddelim -fn files
```

where fields are separated by *delim* and specified by *n*

- Display selected fields from *files*
- Example: place only phone numbers in a separate file
- Specification:

```
$ cut -d: -f5 roster > phones
```

cut (cont.)

The output shown on the following page resulted from specifying the fifth field (`-f5`), which is the phone number column, and by isolating the fields as being colon-separated (`-d:`).

cut (cont.)

- Execution:

```
$ cut -d: -f5 roster > phones
$ cat phones
(215) 250-5418
(919) 486-1522
(315) 792-7358
(315) 684-6153
(401) 456-8038
(408) 924-3913
(203) 576-4737
(408) 924-3869
(717) 893-2346
(318) 257-2963
(805) 756-2341
(406) 265-4128
(315) 684-6155
(203) 449-6772
(419) 772-2386
(816) 235-1494
(318) 475-5873
(813) 974-4815
(409) 845-7090
(203) 576-4761
(carrier pigeon)
$
```

paste

The command `paste` can be used as a companion to `cut`. While `cut` extracts columns from files (i.e., isolates fields within records), the `paste` command merges two files (each treated as a column) into a target file (a table of columns). Because `paste` assumes each input file to be a column, it is usually the case that `paste`'s input files have been created by `cut`.

paste

- General form:

```
$ paste -ddelim file1 file2
```

where the output field separator is *delim*

- Appends records from file_2 onto records from file_1
- Example: combine phone numbers file and cities from the original data file

$ cat f1
:

$ cat f2
:

$ paste -d <tab> f1 f2

$

paste (cont.)

- Results

```
$ cut -d: -f2 roster | paste -d: phones -
(215) 250-5418:Easton
(919) 486-1522:Fayetteville
(315) 792-7358:Utica
(315) 684-6153:Morrisville
(401) 456-8038:Providence
(408) 924-3913:San Jose
(203) 576-4737:Bridgeport
(408) 924-3869:San Jose
(717) 893-2346:Lock Haven
(318) 257-2963:Ruston
(805) 756-2341:San Luis Obispo
(406) 265-4128:Havre
(315) 684-6155:Morrisville
(203) 449-6772:Groton
(419) 772-2386:Ada
(816) 235-1494:Kansas City
(318) 475-5873:Lake Charles
(813) 974-4815:Tampa
(409) 845-7090:College Station
(203) 576-4761:Bridgeport
(carrier pigeon):Wapping
$
```

sed General Form

The command sed is an interpreter for the *stream editor*, a little language for editing text while it is in-stream, as opposed to being in a file on disk. sed enables you to print or transform text based on string-matching. Because it will be described in detail later in the module, sed will be briefly described here.

sed is most frequently used for editing tasks. The example shown on the following page transforms post office abbreviations into states written in long form. The *sed rule* used in the example is of the form

```
s/RE/new/
```

where s is the *sed substitution operator*, *RE* is the string being matched, and *new* is the string that

sed General Form

```
$ sed 'specifications' [ files ]
```

where *specifications* are applied to *files*

sed Usage

```
$ sed 's/AZ/Arizona/
>      s/CA/California/
>      s/FL/Florida/
>      s/GA/Georgia/
>      s/IL/Illinois/
>      s/KY/Kentucky/
>      s/LA/Louisiana/
>      s/MD/Maryland/
>      s/MI/Michigan/
>      s/NY/New York/
>      s/OH/Ohio/
>      s/PA/Pennsylvania/
>      s/SC/South Carolina/
>      s/TX/Texas/
>      s/WV/West Virginia/' roster
Guy, Warren:Easton:Pennsylvania:18042:(215) 250-5418
Johnson, Walter:Fayetteville:NC:28301:(919) 486-1522
Qazi, Naseem:Utica:New York:13504:(315) 792-7358
Dann, Wanda:Morrisville:New York:13408:(315) 684-6153
Strauss, Patricia:Providence:RI:02908:(401) 456-8038
Kurzweil, Jack:San Jose:California:95192:(408) 924-3913
Geotsi, Georgette:Bridgeport:CT:06601:(203) 576-4737
Swanson, William:San Jose:California:95192:(408) 924-3869
Potter, John:Lock Haven:Pennsylvania:17745:(717) 893-2346
Ker, Jun-Ing:Ruston:Louisiana:71272:(318) 257-2963
Menon, Unny:San Luis Obispo:California:93407:(805) 756-2341
Johnson, Jeffrey:Havre:MT:59501:(406) 265-4128
Manchester, Mark:Morrisville:New York:13408:(315) 684-6155
Griscom, Priscilla:Groton:CT:06340:(203) 449-6772
Rider, Michael:Ada:Ohio:45810:(419) 772-2386
Kiesler, Thomas:Kansas City:MO:64110:(816) 235-1494
Richardson, Joseph:Lake Charles:Louisiana:70609:(318) 475-5873
Levinson, Deborah:Tampa:Florida:33620:(813) 974-4815
Hassan, Yassin:College Station:Texas:77843:(409) 845-7090
Liu, Gonhsin:Bridgeport:CT:06601:(203) 576-4761
Heep, Uriah:Wapping:UK:SE4:(carrier pigeon)
$
```

awk General Form

In general, the command awk signals a departure from the command-line-oriented commands shown above. awk, by contrast, is an interpreter for a powerful, specification-level language. Indeed, following the release of the AWK Programming Language, Release 2.0, it might be viewed as being a general-purpose language. awk will be discussed in detail later in "UNIX® Tools for ASCII Programming."

In this overview, awk will be discussed as an enhancement of the field identification and manipulation operations provided by cut and paste. Many useful programs can be written in two or three lines of awk. In that sense, the usage demonstrated in the next few pages continues the theme of command-line-oriented operations.

In the example shown on the following page, the field delimiter is specified on the command-line (-F:), and the first field is printed using the print function. The field is specified using a predefined *field variable*. These are of the form $*number*, where *number* corresponds to a field appearing in a sequence defined by the delimiter.

awk General Form

```
$ awk 'specifications' files
```

where *specifications* are applied to *files*

awk Usage

- Print the first field of every line

```
$ awk -F: '{print $1}' roster
Guy, Warren
Johnson, Walter
Qazi, Naseem
Dann, Wanda
Strauss, Patricia
Kurzweil, Jack
Geotsi, Georgette
Swanson, William
Potter, John
Ker, Jun-Ing
Menon, Unny
Johnson, Jeffrey
Manchester, Mark
Griscom, Priscilla
Rider, Michael
Kiesler, Thomas
Richardson, Joseph
Levinson, Deborah
Hassan, Yassin
Liu, Gonhsin
Heep, Uriah
$
```

awk

In the example shown on the following page, awk is used as an alternative to cut. The same results could have been achieved as follows:

```
$ grep CA roster | cut -d: -f1
```

It is indicative of the richness of the environment that the same problem can be solved in a variety of ways. What is notable about solving the problem with awk is the ability to extend solutions using a language rather than using a simple, purposeful command like cut. For example, a "friendlier" output could have been done as follows:

```
$ grep CA roster |
> sed 's/, /:/' |
> awk -F: '{ print "Name: " $2, $1 }'
Name: Jack Kurzweil
Name: William Swanson
Name: Unny Menon
$
```

awk

- Example: print only names from list of people from California:

```
$ grep CA roster | awk -F: '{print $1}'
Kurzweil, Jack
Swanson, William
Menon, Unny
$
```

awk (cont.)

sed is not intended for field identification. Its job is to match and transform strings. In the example shown on the following page, sed and awk operate in concert. sed substitutes a colon for a comma followed by a horizontal space. awk, then, is able to isolate the last name (given first) as a separate field.

awk (cont.)

- Example: print the names and states of everyone in the roster

```
$ sed 's/, /:/' roster |
> awk -F: '{ print "Name: " $2, $1, " State: " $4 }'
Name: Warren Guy  State: PA
Name: Walter Johnson  State: NC
Name: Naseem Qazi  State: NY
Name: Wanda Dann  State: NY
Name: Patricia Strauss  State: RI
Name: Jack Kurzweil  State: CA
Name: Georgette Geotsi  State: CT
Name: William Swanson  State: CA
Name: John Potter  State: PA
Name: Jun-Ing Ker  State: LA
Name: Unny Menon  State: CA
Name: Jeffrey Johnson  State: MT
Name: Mark Manchester  State: NY
Name: Priscilla Griscom  State: CT
Name: Michael Rider  State: OH
Name: Thomas Kiesler  State: MO
Name: Joseph Richardson  State: LA
Name: Deborah Levinson  State: FL
Name: Yassin Hassan  State: TX
Name: Gonhsin Liu  State: CT
Name: Uriah Heep  State: UK
$
```

tr

The command tr translates a set of patterns (*pattern*$_1$) into an alternative set of patterns (*pattern*$_2$). Unlike sed, tr is capable of accepting *metacharacters*, called *Regular Expressions*, rather than a literal string as the replacement string. As you will see in the unit "Regular Expressions," these metacharacters can be quite rich, expressing all aspects of a string.

In the example shown on the following page, the string that is matched is each lowercase character (one in a twenty-six character set); the replacement string is a corresponding uppercase character (one in a twenty-six character set, as well).

Unmatched characters, such as comma (,) or colon (:) are passed through.

Note that tr does not accept a file argument per se. Input must be explicitly redirected (<) from *file*. tr is not able to read input from multiple files simultaneously.

tr General Form

```
$ tr pattern1 pattern2 < file
```

Translate characters in *pattern*$_1$ to corresponding characters in *pattern*$_2$

tr Usage

Example: translate lowercase to uppercase

```
$ tr '[a-z]' '[A-Z]' < roster
GUY, WARREN:EASTON:PA:18042:(215) 250-5418
JOHNSON, WALTER:FAYETTEVILLE:NC:28301:(919) 486-1522
QAZI, NASEEM:UTICA:NY:13504:(315) 792-7358
DANN, WANDA:MORRISVILLE:NY:13408:(315) 684-6153
STRAUSS, PATRICIA:PROVIDENCE:RI:02908:(401) 456-8038
KURZWEIL, JACK:SAN JOSE:CA:95192:(408) 924-3913
GEOTSI, GEORGETTE:BRIDGEPORT:CT:06601:(203) 576-4737
SWANSON, WILLIAM:SAN JOSE:CA:95192:(408) 924-3869
SWANSON, WILLIAM:LOCK HAVEN:PA:17745:(717) 893-2346
POTTER, JOHN:LOCK HAVEN:       (318) 257-2963
```

Summary

`awk` *'rules'* *files*

Applies *rules* to *files* for field identification, among other tasks

`cut -d`*delim* `-f`*n* *files*

Extracts *n* fields, separated by *delim*, in *files*

`diff` $file_1$ $file_2$

Identifies differences in $file_2$ with respect to $file_1$ (line-by-line granularity)

`echo` *'string'*

Writes *string* on `stdout` as record

`grep` *'RE'* *files*

Identifies all records in *files* containing string *RE*

`nl` *file*

Numbers lines in *file* (blank lines ignored)

Summary (cont.)

`paste` *file$_1$ file$_2$*

Merges columns (*file$_1$ and file$_2$* each treated as a single column) into a single table of columns (output file)

`sed` *'rules' files*

Applies *rules* to *files* for lexical recognition, editor-like transformations, among other tasks

Storage and Processing

Objectives

- Become familiar with file maintenance commands
- Become familiar with directory structures
- Create and traverse directory structures
- Become familiar with UNIX® system processes
- Become familiar with commands for managing processes

Directory and Process Hierarchies

The System Around You: Storage and Processing

Under any UNIX® system all directories are subordinate to a common root directory (/). Multiple directories, called *file systems*, grow from this root: /bin, /dev, /etc, /tmp, /usr, among others, including file systems that contain the home directories of users. These are the

The System Around You: Storage and Processing

- File systems
- Information about files
  - directories
  - files and their permissions
  - types of files
  - location of named files in directory hierarchy
- Processes
  - identification numbers (PIDs)
  - parent identification numbers (PPIDs)
- Control over processes
  - `nice`
  - `wait`
  - `kill`

Where in the UNIX® System Am I?

The UNIX® system supports the idea of a current place in a current file system. By default the user is in his or her home directory upon logging into the system. On the following page the user's home directory is `ken` under the `yawl` file system. After having logged in, user `ken` moved two directory levels down the tree. He is said to be *in* the directory `uf` (UNIX system fundamentals), which is a directory contained in the directory `curric` (curriculum). The command `pwd` (print working directory) displays the *pathname* (the full directory path from the root directory). The slash (/) separating each level (*node*) of the tree is conventional. All non-terminal strings in the pathname are directories.

By convention, the user `ken` owns all files and directories at the tree node `/yawl/ken` and below, and he may create files and directories in this area as desired. In addition, he can decide who will be allowed to read, modify, and execute any of these files.

Where in the UNIX® System Am I?

- The command `pwd` displays your *current directory*

```
$ pwd
/yawl/ken/curric/uf
$
```

- All directories described in relationship to root: /

Absolute and Relative Pathnames

Two conventions govern how files and directories are specified:

- absolute pathnames
- relative pathnames

The *absolute pathname* is the full name of a file or directory from root to the file in question. The *relative pathname*, by contrast, is specified with respect to the current directory (the result given by `pwd`).

For example, on the following page two commands are executed upon the same file. The first is specified with its absolute pathname; the second is specified with its relative pathname.

Two frequently used relative names are

- `.` the current directory
- `..` the parent of the current directory

Absolute and Relative Pathnames

- Two conventions governing file and directory specification
  - *absolute pathname* full name from root to current file
  - *relative pathname* name specified with respect to current directory
- Specification by absolute pathname

```
$ pwd
/course/os/asee/fund/store_proc
$ wc -l /course/os/asee/fund/store_proc/roster
     28 /course/os/asee/fund/store_proc/roster
$
```

- Specification by relative pathname

```
$ wc -l roster
     28 roster
$
```

What's There?

The `ls` command lists files in the current directory. Used with the command option `-l`, the command produces a long listing, displaying the following by column:

Permissions	a listing of which classes of users may read, modify, or execute the corresponding file
Links	number of links to that file (ignore for now)
Owner	who owns the file (can be changed by the owner with the `chown` command)
Group	the user group to which group permissions pertain (can be changed by the owner with the `chgrp` command)
Bytes	the number of characters the file contains
Month	the month in which the file was created or last modified
Day	the day in which the file was created or last modified
Time	the time in which the file was created or last modified
Name	the file name (must be ≤14 characters)

The permissions column is organized into three sets of three, each containing the sequence `rwx` or hyphens indicating the absence of a given character. The `r` character indicates that the file may be read or copied; the `w` character indicates that the file may be written (modified); the `x` character indicates that the file may be executed. Each set of these sequences corresponds to 1) the owner, 2) the group, and 3) everyone else on the system.

Note that the first character in the permissions column is reserved for other designations, including whether the file is a directory (`d`) or a regular file (`-`).

$ - sh
% - csh

What's There?

- Lists files in current directory

```
$ ls -l
total 473
-rwxr-xr-x   1 ken  ustg     145 Jul 22 08:26 ed_script
-rw-r-----   1 ken  ustg   25375 Jul 22 08:24 junko
-rwxr-xr-x   1 ken  ustg     108 Jul 22 08:25 print
-rw-r-----   1 ken  ustg   59280 Jul 22 08:24 unit01
-rw-r-----   1 ken  ustg  152650 Jul 22 08:26 unit01.o
$
```

Some ls Options

Option	Effect
`-l`	lists files together with file information (long listing)
`-t`	lists files in order of time last modified
`-a`	lists all files, including those whose names begin with dot (`.`)
`-C`	lists files in columnar display
`-F`	lists directories followed by `/`; executables followed by `*`

What's Really There?

Structurally all files are identical: a stream of bytes. The contents of each file may vary though. For example, files to be executed without interpretation must contain machine language. Files to be typeset by the formatter command `troff` must contain ASCII characters, and each line must be terminated with the newline character.

While all text files may be modified with the same text editors, the contents of each conform to the conventions established by the commands that interpret them.

The `file` command reads the files given it and makes an educated guess at what the contents mean, based on a knowledge of various UNIX® system conventions. Files are not "typed" with respect to their contents—all files really are the same—so the `file` command's output is not definitive.

The asterisk (`*`) given as an argument to `file` is a metacharacter meaning "all files in the current directory."

What's Really There?

- The command `file` "guesses" what a file contains based on its contents

```
$ file *
ed_script:     commands text
junko:         [nt]roff, tbl, or eqn input text
print:         commands text
unit01:        [nt]roff, tbl, or eqn input text
unit01.o:      ascii text
$
```

- Not 100% correct (like many such "expert" applications)
- Useful to do before editing or `cat`-ing files

Renaming Files

The function of moving a file in the hierarchy and renaming one amount to the same operation. That is, the sense of spatiality engendered by the UNIX® system is an illusion. Moving a file from one directory to another in a different file system is done by renaming it:

```
$ mv /i4/rpm/bin/mylint /i2/spm/bin/bobslint
```

The command that executes this task is therefore called `mv` (move).

Note that this example specifies the file—both by its old and new names—using absolute pathnames. Usually this indicates that the command is being executed in neither `/i4/rpm/bin` nor `/i2/spm/bin`.

Renaming Files

- Moving and renaming files amount to same task

```
$ mv /i4/rpm/bin/mylint /i2/spm/bin/bobslint
```

- Command for renaming files called `mv` (move)
- Permissions of original file same as renamed file (discussed later)
- `mv` used across file systems

Copying Files

Files can be copied using the copy command cp. The semantics of this command are identical to those of mv with one exception: the original file is not destroyed.

Files obviously cannot be copied to the same directory as the original. When moved to another directory, files can be renamed any valid name. In summary, every absolute pathname must be unique.

Copying Files

- The command `cp` copies files
- Copying is an alternative to renaming
- Files cannot be copied to same directory as original
  - each absolute pathname must be unique
- Semantics similar to `mv`
  - file is renamed
  - permissions of original retained by copy (discussed later)
  - can be used across file systems

```
$ cp /i4/rpm/bin/mylint /i2/spm/bin/bobslint
```

Differences with `mv`

- original is not destroyed
- two independent copies of files exist

Creating New Territory

The tree below the home directory can grow in any downward direction specified by the user. The `mkdir` command is used to make new directories.

New directories cannot be named using a name shared by any file in the current directory. (Directories can share the names of files or directories that reside in them.)

Creating New Territory

- The `mkdir` creates new directory under current directory

```
$ mkdir new; ls -l
total 474
-rwxr-xr-x   1 ken  ustg     145 Jul 22 08:26 ed_script
-rw-r-----   1 ken  ustg   25375 Jul 22 08:24 junko
drwxr-x--x   2 ken  ustg      32 Jul 22 08:30 new
-rwxr-xr-x   1 ken  ustg     108 Jul 22 08:25 print
-rw-r-----   1 ken  ustg   59280 Jul 22 08:24 unit01
-rw-r-----   1 ken  ustg  152650 Jul 22 08:26 unit01.o
$
```

- New directory cannot share name of file in same directory
- Directories renamed with `mv`
  - treated like regular file
  - cannot be used for moving directories into new directories

Moving Around

The cd command allows the user to change directories. The command will accept either relative or absolute directory names as an argument. cd will return the user to the home directory if no argument is given.

As the example shown on the following page demonstrates, cd will accept as an argument a *relative pathname* (i.e., a directory specified with respect to the current one). Alternatively, it will accept an *absolute pathname* (i.e., one whose genealogy from root is explicit).

Moving Around

- The `cd` command moves you to a new current directory in which more directories may be created

```
$ cd new
$ pwd
/yawl/ken/curric/new
$
```

- Arguments may be *relative pathname* or *absolute pathname*

Populating New Territory

Files spring into being in a variety of ways. They can be created with text editors. They can also be named as an argument to a command such as `tee`. Finally, they can be created by redirecting output (>) to them.

For example, on the following page, the file `newfile` is created by redirection. This example also demonstrates an idea discussed earlier.

The default input for most UNIX® system commands is the standard input (`stdin`). Here, the `cat` command is not given any file as a command argument. It therefore waits for input to be typed from the keyboard (associated with `stdin` by default). The end-of-input, or end-of-file (EOF), is specified with `^D` (*control-d*, *control* and *d* simultaneously).

Populating New Territory

Files are created in directories by

- Editing files with text editors
- Redirecting output into new or existing files
  - ed
  - edit
  - ex
  - vi
  - emacs

```
$ cat > newfile
text can be entered using a number of commands
^D
$ ls -l
-rw-r--r-- 1 ken  ustg  48 Jun 25 14:36 newfile
$
```

File System Structure

We suggested that the environment reflects the character of the user's work. In particular, users generally construct special-purpose tools from general software utilities. The file system under each user's home directory reflects the project it contains.

The subordinate hierarchy of the UNIX® system directory structure is a ''natural'' structure for the growth of ideas. Primary classes of information are subdivided into ancillary points, which in turn are split into finer distinctions. The creation of new nodes on the inverted directory tree is proportionate to emphasis in the idea system.

Furthermore, the structure allowing for subdivisions of subdivisions permits users to keep their data out of the way of other users. This structure also provides for privacy. Allowing no access to a given directory permits no access to all files and directories in it or below it.

File System Structure

- Hierarchical, subordinate structure of file system a characterizing feature of UNIX® system
  - relationships among user files user-defined
  - working environment can be designed and redesigned to suit needs of present work
- "Natural" paradigm for organizing ideas
  - primary subjects organized into separate, parallel nodes
  - subtopics organized as children of subject nodes
  - downward growth of subdistinctions at user's discretion

Traversing the File Hierarchy

As we have shown, the command pwd prints a listing of the current (working) directory looking upward toward root. By contrast, the `find` command prints a downward traversal of the current directory looking downward. Beginning at the current directory and traversing all directories below it, `find` allows for display, as well as manipulation, of files.

In order to print the traversal, `find` must be used with its option `-print`.

Traversing the File Hierarchy

- By default the command `find` scans all directories and files below current directory

```
$ cd
$ pwd
/yawl/ken/curric
$ find . -print
.
./nohup.out
./unit03
./s.unit05
./makefile
./p.unit01
./s.unit01
./old_roster
./vi
./s.unit02
./rpm.roster
./unit01
./roster
$
```

Traversing Hierarchy and Pattern-matching

Traversal can be made selectively with the `find` option `-name`. In the example shown on the following page, the argument given to `-name` is a pattern that has been generalized with the asterisk (*). As we have said, the asterisk matches every file name (or remainder of filename) in the current directory.

The concept of current directory obviously changes in the context of directory traversal. The asterisk successively matches files in each part of the tree that is traversed. Only the files that match are recognized by `-print`.

Often you use `find` to look for a specific file. It is tedious to read through all lines of output to find a specific file. The command option `-name`, therefore, is used for pattern matching.

In the example shown on the following page, the asterisk is used to match partial file names that might not be foreseeable.

Traversing Hierarchy and Pattern-matching

- Command can selectively match tree nodes based on specified patterns
- Command option `-name` for pattern-matching

```
$ find . -name '*unit*' -print
./unit03
./s.unit05
./p.unit01
./s.unit01
./s.unit02
./unit01
$
```

Removing Files and Directories

To this point the only method for file or directory removal you have is the command `mv`, which always conserves one instance of the object. The commands `rmdir` (remove directory) and `rm` (remove) allow for the erasure of files from the disk. Memory is freed at the point of their usage, and the associated information is overwritten. These commands, therefore, must be used with caution.

Removing Files and Directories

- Command for removing files

  `rm` removes files (read-only file will cause `rm` to prompt you)

  `rm -r` removes directories (read-only contents will cause `rm` to prompt you)

  `rm -rf`
  forces unconditional removal of files and directories

- Commands for removing directories only

  `rmdir` directory must be empty (safe version of `rm -r`)

File and Directory Removal: Examples

As the demonstration on the following page shows, `rmdir` can only remove empty directories.

`rm -r` can remove full directories, but any read-only contents will cause the command to prompt you. (`440` is the octal representation of read-only permissions for the owner and group.) A response of `y` (yes) rather than `n` (no) would have enabled the command to complete directory removal.

The command `rm -rf` will cause the unconditional removal of the directory.

File and Directory Removal: Examples

```
$ mkdir junk
mkdir: cannot make directory junk
$ ls -l
total 65
-rw-r-----    1 lally  ustg      953 Oct 17 10:46 Ccomm
-rw-r-----    1 lally  ustg      528 Oct 17 10:46 Icomm
-r--r-----    1 lally  ustg    30374 Oct 17 10:44 comm
-rw-r-----    1 lally  ustg        0 Oct 17 10:58 junk
$ rm junk
$ mkdir junk
$ cd junk
$ ed file
?file
a
this is read-only.
w
19
q
$ chmod u-w file
$ ls -l file
-r--r-----    1 lally  ustg       19 Oct 17 10:58 file
$ cd ..
$ rmdir junk
rmdir: junk not empty
$ rm -r junk
junk/file: 440 mode ? n
rmdir: junk not empty
$ rm -rf junk
$
```

Introduction to UNIX® System Processes

What's Happening?

The ps command reports on process statuses. The report is a graphic representation of a time-sharing system at work. Typically, many processes are active at once. Each is allocated its microseconds on the central processing unit, and must be scheduled on and off the processor. The command ps reports on that scheduling.

What's Happening?

- Command `ps` reports on system processes
- All active system processes listed
- Information available concerning
  - associated command line
  - name of associated user (UID)
  - associated terminal
  - cumulative execution time
  - process identification number (PID)
  - parent's process identification number (PPID)

Graphic Display of Events

The screen shown on the following page demonstrates ps's option –u for listing all processes associated with a specified user.

Note that the output from ps is a snapshot of history. By the time you see it, events have already changed: new processes complete, others are scheduled, or defunct processes finish the task of cleanup associated with their successful or failed processing.

Graphic Display of Events

- The following is user-oriented display:

```
$ ps -u ken
   PID TTY        TIME COMMAND
  4271 tty05      0:01 sh
 20252 xt013      0:02 vi
 21122 xt014      0:13 vi
 21944 xt012      0:02 vi
 11609 tty05      0:00 layers
 11646 xt012      0:00 ksh
 11647 xt015      0:00 ksh
 11664 xt016      0:00 ksh
 11617 xt013      0:00 ksh
 11620 xt014      0:00 ksh
 11621 xt011      0:00 ksh
 11667 xt016      0:00 ksh
 22348            0:00 <defunct>
 22349 xt014      0:00 ksh
 22350 xt014      0:00 ps
 22351 xt014      0:00 sed
$
```

$ ps

PID	TT	STAT	TIME	command
2448	p6	I	0:00	-csh
8868	p6	S	0:00	sh
9281	"	R	0:00	ps

Families of Processes

Subordinate hierarchy is a model that also applies to processes. Processes typically make copies of themselves (children) to implement a division of labor, each child performing a discrete task. For example, when you log in, an interactive process—the command sh—displays a prompt (typically $) and interprets the commands you type. If the command is a file containing machine language, it is simply run as another process—a child of the sh that first read it. If it requires further interpretation (e.g., such as a file containing instructions to sh), the first sh process copies

spawn children, and so on.

Families of Processes

- A full listing (–f) displays the relation of a process with its parent

creator of the login shell

```
$ ps -fu ken
 UID    PID  PPID  C    STIME TTY   TIME COMMAND
 ken   4271     1  0 07:58:36 tty05 0:01 -sh
 ken  20252 11617  0 16:44:58 xt013 0:02 vi xmacs
 ken  21122 11620  2 17:04:35 xt014 0:14 vi shell_lang
 ken  21944 11646  0 17:39:58 xt012 0:02 vi quoting
 ken  11609  4271  0 11:43:33 tty05 0:00 layers -f .setup
 ken  11646 11609  0 11:45:04 xt012 0:00 ksh -i
 ken  11647 11609  0 11:45:14 xt015 0:00 ksh -i
 ken  11664 11609  0 11:46:06 xt016 0:00 ksh -i
 ken  11617 11609  0 11:43:39 xt013 0:00 ksh
 ken  11620 11609  0 11:43:40 xt014 0:00 ksh
 ken  11621 11609  0 11:43:40 xt011 0:00 ksh -i
 ken  11667 11664  0 11:46:20 xt016 0:00 ksh
 ken  22907 21122  0                0:00 <defunct>
 ken  22908 21122  2 18:04:20 xt014 0:00 [ ksh ]
 ken  22909 22908  9 18:04:20 xt014 0:00 ps -fuken
 ken  22910 22908  2 18:04:20 xt014 0:00 [ sed ]
$
```

Some ps Options

Option	Effect
-f	gives full listing
-e	lists everything (all current processes)
-u*login*	lists processes associated with *login*

Killing Processes

The `kill` command terminates processes. Given the subordinate hierarchy in which processes are organized, processes must be killed selectively. That is, killing one process could terminate not only that process, but also its children, its children's children, and so forth.

Consider the example shown on the following page. What will be result of killing process 11609?

A sure kill is issued with the command

```
$ kill -9 PID
```

Killing Processes

- Before killing a process, it is important to know if it has children
  - avoid undesired chain-effect

```
$ ps -fu ken
 UID    PID  PPID C    STIME TTY    TIME COMMAND
 ken   4271     1 0 07:58:36 tty05  0:01 -sh
 ken  20252 11617 0 16:44:58 xt013  0:02 vi xmacs
 ken  21122 11620 2 17:04:35 xt014  0:14 vi shell_lang
 ken  21944 11646 0 17:39:58 xt012  0:02 vi quoting
 ken  11609  4271 0 11:43:33 tty05  0:00 layers -f .setup
 ken  11646 11609 0 11:45:04 xt012  0:00 ksh -i
 ken  11647 11609 0 11:45:14 xt015  0:00 ksh -i
 ken  11664 11609 0 11:46:06 xt016  0:00 ksh -i
 ken  11617 11609 0 11:43:39 xt013  0:00 ksh
 ken  11620 11609 0 11:43:40 xt014  0:00 ksh
 ken  11621 11609 0 11:43:40 xt011  0:00 ksh -i
 ken  11667 11664 0 11:46:20 xt016  0:00 ksh
 ken  22907 21122 0                 0:00 <defunct>
 ken  22908 21122 2 18:04:20 xt014  0:00 [ ksh ]
 ken  22909 22908 9 18:04:20 xt014  0:00 ps -fuken
 ken  22910 22908 2 18:04:20 xt014  0:00 [ sed ]
$ kill 11609
```

Who Else Is on the System?

The who command reports which users are currently logged on to the system. It also identifies which terminal (a file in /dev) has been associated with their login process. Finally, who identifies when each active user logged on.

Who Else Is on the System?

- who command reports users currently logged on

```
$ who
root        console     Jul  6 09:55
rpm         tty12       Jul  6 21:03
fate        xt012       Jul  6 14:13
$
```

User Information

The system maintains information about each user:

- where the home directory resides
- user identification number (UID)
- group identification number (GID)
- login name of each user
- each user's password

User Information

Every user on a UNIX® System has the following:

Login name	up to eight characters, unique name
Password	key to the system
UID	numeric value associated with login name (identifies files and processes associated with user)
GID	numeric value associated with logical group of users
Home directory	user's starting point in the file structure

```
$ echo $LOGNAME
rpm
$ id
uid=4229(rpm) gid=46014(ustg)
$ pwd
/yawl/rpm
$
```

Summary

`*` matches file names (or partial names) in current directory

`cp` *file*$_1$ *file*$_2$
makes copy of *file*$_1$ named *file*$_2$

`ls` lists files in the current directory

`ls -a`
lists all file names (including those that begin with dot (.)

`ls -C`
lists files in columnar display

`ls -F`
lists files indicating function (directory names preceded by /; executable files preceded by *)

`file` *file*
"guesses" at contents of *file*

`find . -print`
traverses directory tree downward from current directory printing all nodes

Summary (cont.)

```
find . -name RE -print
```

[illegible] directory tree downward from current node

```
find . -type d -print
```

traverses directory tree downward from current node
printing all directory nodes

UNIX® System Security

Overview

The term *security* on the UNIX® operating system applies to restricting access to the system and to its resources. Security is applied to access of the system through the use of *login* and *passwords.* Access to files and directories is obtained through the use of *permissions*. Permissions limit access to the file for all users except the *super-user*. The super-user can be excluded from reading a file by encrypting its contents.

Security is the personal responsibility of each user on the system. This unit will discuss the commands used to secure your environment.

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- Why is security particularly important for the UNIX® system?
- How does the design of UNIX® system aid security?
- How do you secure your environment?
  - secure your directories?
  - secure the system?

Privacy in a Multi-user Environment

We have said that the UNIX® system supports a community of users. The environment design itself emphasizes software reuse through tool development. The idea of a community of users makes reuse—standing on the shoulders of others—all the more attractive.

This raises two opposing questions:

- How do you share files with other users?
- How do you prevent other users from sharing your files?

Depending on the context and the files, both questions are important ones.

Because the design of the UNIX system is one of cooperative computing, security is particularly important.

Privacy in a Multi-user Environment

- UNIX® system design encourages software reuse
  - tool-oriented environment
  - community of users
- Sharing tools raises two important questions
  - how do you share files?
  - how do you avoid sharing files?

A Design for Public and Private Computing

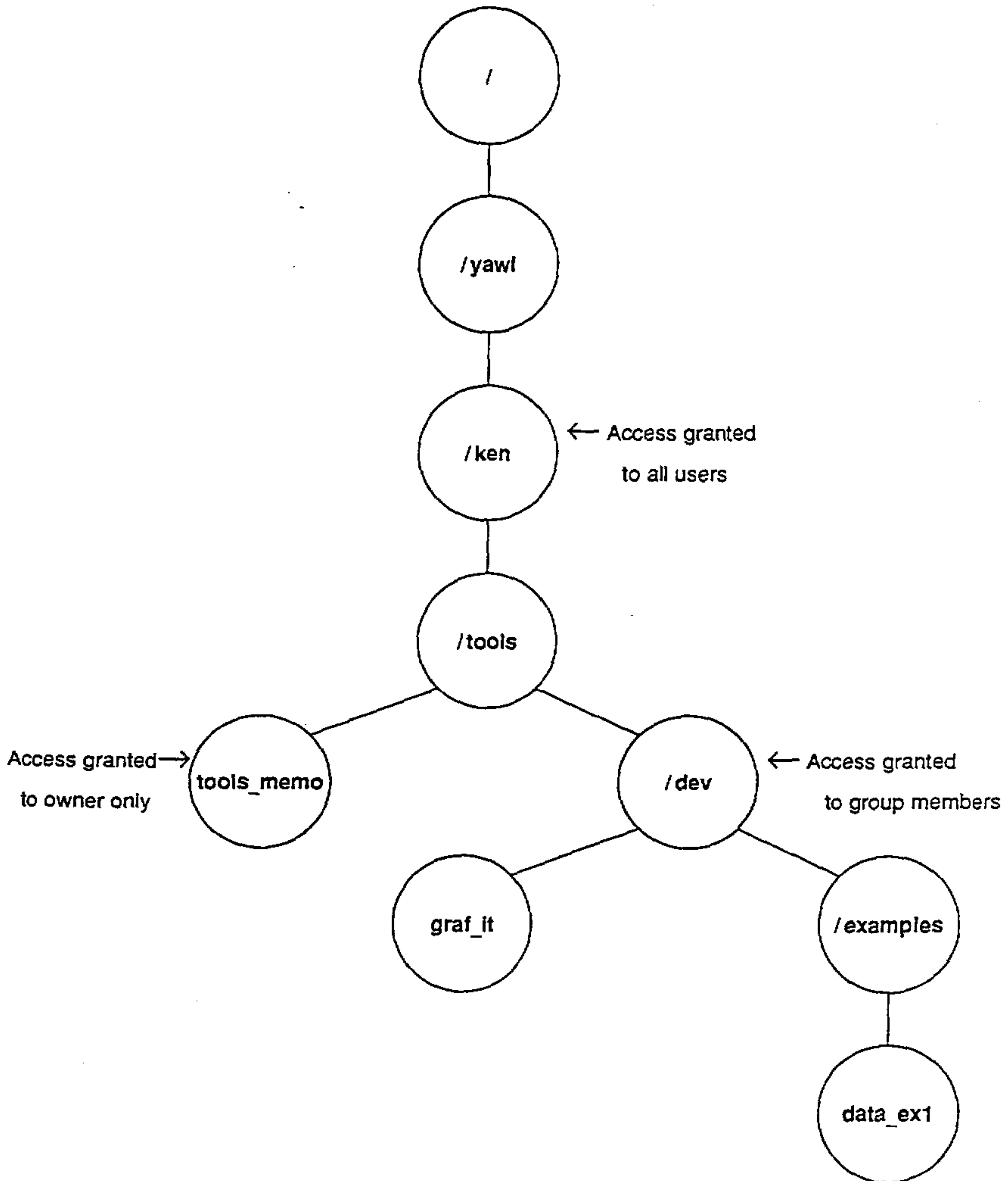
We have discussed the design of the UNIX® system file system as being one of subordinate hierarchy—an inverted tree. Another way of stating this storage paradigm is that every directory is potentially either a gateway or a blockade to all of the directories and correspondent files created beneath it.

As we have said, the UNIX® system directory structure, with its inverted-tree model of division and subdivision, is intuitive to the way you think. It follows that a model for security should also be compatible with the natural order of ideas. Some topics and subtopics are inherently public; some are private.

The UNIX® system facility for restricting *file access*, called *permissions*, is compatible with these constraints.

The illustration shown on the following page demonstrates the point.

A Design for Public and Private Computing



User Classes

Access to files and directories is controlled through the use of permissions. Permissions are applied to three classes of users: owner, group, or other.

The *owner* initially is the person who created the file or directory. The owner is the only person, other than the *super-user*—usually the system administrator—who can change permissions.

The *group* consists of a number of users who have common access to specific files. Groups are

User Classes

- Owner — initially file creator
  - can change permissions
  - can remove file
- Group — related users
  - members of same project
  - members of same academic department
  - employees in same organizational group
- Other — anyone else on system
- Super-user — same status as owner
  - usually system administrator

Permissions

Each class of user is bound by three kinds of permission: *read permissions*, *write permissions*, and *execute permissions*. As the names imply, these permissions permit the corresponding class of user to read or write or execute a file or directory. (Entering a directory is an example of execution.)

It follows that each of the three permissions must apply to each class of users. Consequently, permissions are organized into three groups of three. A typical permissions setting looks like this:

```
-rwxrwxr-x
```

The first group (the owner's designation), the second group (the group's designation), and the third (the designation for others) are all granted read permission (r) and execute permission (x). In addition, the owner and group members are granted write permission (w),

As the example shows, when displayed, permissions are represented as one of three lowercase characters or a hyphen, which indicates that no permissions have been granted with respect to the current column and user class.

Note that the leading hyphen indicates the file is a regular file. A directory is represented with a leading d:

```
drwxrwxr-x
```

Permissions

- For each class of user, three kinds of permission
  - *read permission* — ability to read contents of file or directory
  - *write permission* — ability to modify contents of file or directory
  - *execute permission* — ability to execute file or directory
- Organized into three groups of three
- Typical permissions setting

```
-rwxrwxr-x
```

  - owner — read, write, execute permissions
  - group members — read, write, execute permissions
  - others — read, execute permissions

Displaying File Permissions

The command `ls` lists files in the current directory. Used with the command option `-l`, the command produces a long listing, displaying the following by column:

Permissions	a listing of which classes of users may read, modify, or execute the corresponding file
Links	number of links to that file (ignore for now)
Owner	who owns the file (can be changed by the owner with the `chown` command)
Group	the user group to which group permissions pertain (can be changed by the owner with the `chgrp` command)
Bytes	the number of characters the file contains
Month	the month in which the file was created or last modified
Day	the day in which the file was created or last modified
Time	the time in which the file was created or last modified
Name	the file name (must be ≤14 characters)

The permissions column is organized into three sets of three, each containing the sequence `rwx` or hyphens indicating the absence of a given character. The `r` character indicates that the file may be read or copied; the `w` character indicates that the file may be written (modified); the `x` character indicates that the file may be executed. Each set of these sequences corresponds to 1) the owner, 2) the group, and 3) everyone else on the system.

The permissions associated with a directory are displayed by using the command option `-d`.

Displaying File Permissions

- Displays current directory content

```
$ ls -l
total 473
-rwxr-xr-x   1 toms  ustg       145 Jul 22 08:26 ed_script
-rw-r-----   1 toms  ustg     25375 Jul 22 08:24 junkc
-rwxr-xr-x   1 toms  ustg       108 Jul 22 08:25 print
-rw-r-----   1 toms  ustg     59280 Jul 22 08:24 unit01
-rw-r-----   1 toms  ustg    152650 Jul 22 08:26 unit01.c
$
```

- Displays current directory permissions

```
$ ls -ld .
drwxr-xr-x   4 toms  ustg   256 Jul 22 08:26 .
```

Modifying Access

Changing Permissions

As permissions are displayed as lowercase characters, they can also be specified using lowercase characters. The command chmod changes the *permissions mode* of files. For our current purposes, it has the following general form:

```
$ chmod [ugo][+-][rwx] files
```

where the following apply:

u	user
g	group
o	other
–	delete permissions
+	add permissions
r	read
w	write
x	execute

Changing Permissions

General form

```
$ chmod [ugo][+-][rwx] files
```

where the following apply:

Operator	Meaning
u	user
g	group
o	other
–	delete permissions
+	add permissions
r	read
w	write
x	execute

chmod in Practice

Examples

```
$ ls -l paper
-rw-r--r--   1 toms  ustg     6 Jul 22 08:26 paper
$ chmod u+x paper
$ ls -l paper
-rwxr--r--   2 toms  ustg    32 Jul 22 08:26 paper
$ chmod go-r paper
$ ls -l paper
-rwx------   2 toms  ustg    32 Jul 22 08:26 paper
$
```

Changing File Ownership

The chown command makes it possible to change the ownership on a file. File ownership allows the user to control access to the file through its permissions. Only the owner of the file or directory or the super-user can change the ownership on a file using this command.

If a user copies a file from another person's directory (cp), he or she is automatically given ownership of the file. chown is not needed in this instance. If a file is given away to another users via a cp or mv command, the original owner of the file retains ownership. To change ownership to the new owner of the file this command should be used.

Changing File Ownership

- General form

```
$ chown login file
```

- Example

```
$ pwd
/yawl/toms/curric
$ ls o*
old_roster
$ cp old_roster /yawl/lally
$ cd /yawl/lally
$ ls -l o*
-rw-r--r--   1 toms   ustg   6 Jul 22 08:26 old_roster
$ chown lally old_roster
$ ls -l o*
-rw-r--r--   1 lally   ustg  6 Jul 22 08:26 old_roster
$
```

chgrp

change group

Changing File Group

- General form

```
$ chgrp group file
```

- Example

```
$ cd rje
$ pwd
/yawl/toms/rje
$ ls -l budget
-rw-r--r--   1 toms  finance  350 Jul 22 08:26 budget
$ chgrp common budget
$ ls -l budget
-rw-r--r--   1 toms  common   350 Jul 22 08:26 budget
$
```

Encryption

Encryption of Files

If you are working on a project that requires tight security, changing the security on your files may not be enough. The encryption of files allows a key to be associated with a file that is only unlockable by the owner of the file. The super-user does not have permission to read encrypted files. The key needed to decrypt the file is only known to the owner.

This command is recommended for short term storage of information or information which is sent over the network.

A file can be created encrypted by using the command option `-x` with the `vi` or `ed` editors.

An existing file can be converted to an encrypted file using the `crypt` command. The unencrypted file should be removed to maximize security.

Once the file is encrypted it is unreadable using commands such as `cat` and `pg`. To decrypt the file, use the `crypt` command but redirect the encrypted file as input.

If the key is forgotten or lost, the data remains unreadable and is useless.

Encryption of Files

- General Form

```
crypt < file > encrypted_file
Enter key:
```

Example

```
$ crypt < my_resume > resume
Enter key:
$ rm my_resume
$ cat resume
hdjshkjHJHJHJHJHJjdkalp3hfuhudsiqpfjhjdsa;
hfjdhsahkfdhak;hfueqwuhfehjnmanvklskdjf
hfjdhsafuhahiuhwuehfuhuhfruhuqhuifyhiqh
$ file my_resume
my_resume:     data
$ crypt < resume
Enter key:
   Tom Sing
   10 Milrose Dr.
   Anywhere, NJ 09970

Employment History:
$ vi -x resume
Enter key:
```

Protecting the System

System Security

As a user on the system, you are accountable for any commands executed under your login account. Activities associated with your account are restricted through your *login* and *password*. The *login* and *password* are used as a means of identifying authorized users of the system.

The *login* consists of alphanumeric characters identifying you to other users. It is therefore known by other users and is used by commands such as `mailx` (discussed later).

The *password* provides system security by restricting system access through the login procedure. It consist of six to eight characters and cannot contain spaces. It must have at least two alphabetic and one numeric character.

The `/etc/passwd` file contains information about all the authorized users on the system. The *password* is stored in this file as an encrypted string. Each record in `/etc/passwd` contains colon-separated fields, which contain the following information about each user:

login : *encrypted password* : *UID* : *GID* : *accounting information* : *login directory* : *shell*

If the last field is empty, the default shell, `/bin/sh` is your command interpreter.

Before stepping away from your terminal, it is important to log off the system using `^D` or `exit`. If this procedure is not followed, it may be possible for someone else to use your account.

System Security

- Login — known by others
- Password — private
- User information

```
$ grep -i SING /etc/passwd
toms:ksmJEr890:36262:3333:79554-T.SING(NJ):/yawl/toms:
```

- Use `exit` or `^D` to log-off

Changing the Password

It is important to keep your *password* private and to choose it carefully. It should not be told to others, stored in a function key, or recorded on paper. Don't allow other users to watch you login.

Some rules for choosing a password are as follows:

- may contain six to eight characters
- must contain at least two letters and one numeric or special character (e.g., &).
  - can consist of uppercase and lowercase letters
  - cannot be login or account numbers or the reverse or permutation of them
  - avoid the obvious such as dates, employee data, acronyms, birthdays, proper names

Some systems use *password aging*, which means at regular intervals (for example every four months) the password must be changed. A password should be changed at least four times a year. The `passwd` command is used to alter a password.

This command prompts you for your old password and then for a new password. You are prompted for the new password twice. The passwords entered while using this command are not printed on the screen. Once the password has been changed, you cannot change it again for some period of time, typically a week or two.

Changing the Password

General form

```
$ passwd
```

Example

```
$ passwd
Changing password for toms
Old password:
New Password:
Re-enter new password:
$
```

Summary

`chgrp` *group file*	changes group
`chmod` *mode file*	changes permissions
`chown` *login file*	changes file owner
`crypt <` $files_1$ `>` $file_2$	creates encrypted file
`grep` *login* `/etc/group`	identifies groups
`id`	identifies current user id (UID) and group id (GID)
`ls -l`	lists file permissions
`ls -ld` *file*	list directory permission
`passwd`	changes password
`vi -x` *file*	creates an encrypted file for editing

Introduction to Regular Expressions

Objectives

- Become familiar with a notation for generalizing lexical descriptions
- Be aware of role played by Regular Expression in the UNIX® system environment

A Common Dialect: Regular Expressions

As opposed to a screen editor, which encourages a "topographical" view of a file, the line editor ed encourages the user to view text as classes of information. Such classes might include

- all instances of the literal string *string*
- all instances of words beginning with uppercase letters
- all unsigned integers

and so forth. The notation that makes this possible is known as *Regular Expressions* or *REs*.

REs enable the user to specify minimal units of lexical meaning (*lexemes*) as well as contexts in which those lexemes might be found.

Roughly speaking, REs constitute a dialect shared by commands throughout the UNIX® system environment. Learned in one place—usually through ed—the notation can be carried over to other commands providing general purpose lexical recognition. Some commands accepting REs notation are as follows:

- `ed`
- `edit`
- `vi`
- `ex`
- `grep`
- `egrep`
- `sed`
- `awk`
- `find`

Note that all UNIX® system commands do not recognize the full set of REs. Some commands, such as `grep`, in fact recognize a relatively small subset of REs.

Because Regular Expressions will be covered in detail in "UNIX® System Tools for ASCII Programming," the treatment given them in this unit is an introductory overview only.

A Common Dialect: Regular Expressions

- Class of metacharacters extending lexical recognition
- Limited context sensitivity (beginning and end of record)
- Regular expressions (with minor variations) recognized by many commands such as
  - `ed`
  - `edit`
  - `vi`
  - `ex`
  - `grep`
  - `egrep`
  - `sed`
  - `awk`
  - `find`

Regular Expressions in ed

The set of REs supported in ed is large. The *character class operators* ([ ]) are used to generalize individual character matches. The *wildcard operator* (.) will match any single character except newline, making it a rich operator. REs are specifically for lexical, not syntactical, recognition. Two context operators are supported, nonetheless. The *beginning-of-line operator* (^) is used to anchor characters at the beginning of a line. The *end-of-line operator* ($) is used to anchor characters at the end of a line. The *replacement operators*, which are peculiar to ed and to the stream editor sed, are used to buffer a string that can later be expanded using an evaluation operator.

Regular Expressions in ed

- Character classes: [ ]
- Complement character classes: [^]
- Wildcard: . (matches any character except newline)
- Repetition character: * (zero or more instances of preceding character)
- Escape character: \ (causes character to be interpreted literally)
- Context sensitivity:
  - beginning of record: ^
  - end of record: $
- Insertion of left-side pattern on right side: &
- Replacement operators: \(\)
  - evaluation form

\1	first isolated string
\2	second isolated string
k	*k*th isolated string

Character Classes

The following demonstration of Regular Expressions (*REs*) will be clearer using a sorted roster. We will, therefore, use a roster that's been sorted contained in the file `sort_roster`:

```
$ sort roster | tee sort_roster
$
```

In the example shown on the following page, all last names beginning with characters in the range from uppercase A-F are printed. As we shall describe, the circumflex (^) preceding the square brackets *anchors* the characters to the beginning of the line.

Character Classes

- Bracketed notation indicates a class of characters
  - hyphenated letters or numbers indicate a range of characters
- Example: all last names beginning with character in class `A` through `F`

```
$ ed sort_roster
g/^[A-F]/p
Dann, Wanda:Morrisville:NY:13408:(315) 684-6153
```

Complement Character Classes

It is sometimes convenient to specify a character in terms of what it is not. This is done with the *complement character class.*

The complement character class is specified with the character ^ appearing in the initial position within square brackets. If the ^ appears anywhere else in the set, it is interpreted as literal.

Note that four characters are special inside a character class:

^	complement class (when following left square bracket)
-	(when not following left square bracket) range of characters
\	quoting character
]	end of character class

In the example shown on the following page, you can see that the circumflex is an overloaded character. Its context decides its meaning. When it appears outside of square brackets, it signifies the beginning of the line. When it appears inside of square brackets, it signifies that all characters in the set specify a complement character class.

The specification `/^[^C-F]/` indicates all lines beginning (^) with a character not (`[^ ]`) in the range A-F.

Complement Character Classes

- Bracketed notation beginning with a caret (^) indicates the ASCII complement to that notation:
- Example:

```
g/^[^A-F]/p
Geotsi, Georgette:Bridgeport:CT:06601:(203) 576-4737
Griscom, Priscilla:Groton:CT:06340:(203) 449-6772
Guy, Warren:Easton:PA:18042:(215) 250-5418
Hassan, Yassin:College Station:TX:77843:(409) 845-7090
Heep, Uriah:Wapping:UK:SE4:(carrier pigeon)
Johnson, Jeffrey:Havre:MT:59501:(406) 265-4128
Johnson, Walter:Fayetteville:NC:28301:(919) 486-1522
Ker, Jun-Ing:Ruston:LA:71272:(318) 257-2963
Kiesler, Thomas:Kansas City:MO:64110:(816) 235-1494
Kurzweil, Jack:San Jose:CA:95192:(408) 924-3913
Levinson, Deborah:Tampa:FL:33620:(813) 974-4815
Liu, Gonhsin:Bridgeport:CT:06601:(203) 576-4761
Manchester, Mark:Morrisville:NY:13408:(315) 684-6155
Menon, Unny:San Luis Obispo:CA:93407:(805) 756-2341
Potter, John:Lock Haven:PA:17745:(717) 893-2346
Qazi, Naseem:Utica:NY:13504:(315) 792-7358
Richardson, Joseph:Lake Charles:LA:70609:(318) 475-5873
Rider, Michael:Ada:OH:45810:(419) 772-2386
Strauss, Patricia:Providence:RI:02908:(401) 456-8038
Swanson, William:San Jose:CA:95192:(408) 924-3869
```

Wildcard

- Wildcard matches every character except newline
- Specified as dot (.)
  - enforces convention of line

Repetition of a Character

The only repetition character supported by ed is the asterisk (*). It specifies zero or more instances of the character immediately preceding it. It must be preceded by a character (even a space) in order to function as a repetition character.

Note that specification, ''zero or more instances of a character'' is richer than even the wildcard.

Repetition of a Character

- Dot-star combination (`.*`) matches zero or more instances of any character except newline
- Dot-star-character combination (`.*`*c*) includes *last* instance of character *c*
  - an inclusive match
- Ampersand (&) inserts the matched string into the replacement string

Repetition of a Character (cont.)

```
g/([0-9][0-9][0-9]).*$/s//>&</p
Dann, Wanda:Morrisville:NY:13408:>(315) 684-6153<
Geotsi, Georgette:Bridgeport:CT:06601:>(203) 576-4737<
Griscom, Priscilla:Groton:CT:06340:>(203) 449-6772<
Guy, Warren:Easton:PA:18042:>(215) 250-5418<
...Station:TX:77843:>(409) 845-7090<
```

Repetition of the Absence of a Character

- Complement-character-class-star combination (`[^c]*`) matches every character except *first* occurence of complement set
  - an exclusive match

Repetition of the Absence of a Character: Example

```
g/^[^:]*/s//{&}/p
{Dann, Wanda}:Morrisville:NY:13408:(315) 684-6153
{Geotsi, Georgette}:Bridgeport:CT:06601:(203) 576-4737
{Griscom, Priscilla}:Groton:CT:06340:(203) 449-6772
{Guy, Warren}:Easton:PA:18042:(215) 250-5418
{Hassan, Yassin}:College Station:TX:77843:(409) 845-7090
{Heep, Uriah}:Wapping:UK:SE4:(carrier pigeon)
{Johnson, Jeffrey}:Havre:MT:59501:(406) 265-4128
{Johnson, Walter}:Fayetteville:NC:28301:(919) 486-1522
{Ker, Jun-Ing}:Ruston:LA:71272:(318) 257-2963
{Kiesler, Thomas}:Kansas City:MO:64110:(816) 235-1494
{Kurzweil, Jack}:San Jose:CA:95192:(408) 924-3913
{Levinson, Deborah}:Tampa:FL:33620:(813) 974-4815
{Liu, Gonhsin}:Bridgeport:CT:06601:(203) 576-4761
{Manchester, Mark}:Morrisville:NY:13408:(315) 684-6155
{Menon, Unny}:San Luis Obispo:CA:93407:(805) 756-2341
{Potter, John}:Lock Haven:PA:17745:(717) 893-2346
{Qazi, Naseem}:Utica:NY:13504:(315) 792-7358
{Richardson, Joseph}:Lake Charles:LA:70609:(318) 475-5873
{Rider, Michael}:Ada:OH:45810:(419) 772-2386
{Strauss, Patricia}:Providence:RI:02908:(401) 456-8038
{Swanson, William}:San Jose:CA:95192:(408) 924-3869
```

Replacement Operator

The replacement operators allow you to isolate a string with the notation \(\) and reinsert the string with the notation *n*, where *n* corresponds numerically to a string isolated in a sequence. \1 reinserts the first string isolated; \2 reinserts the second string isolated; and so forth.

In the example shown on the following page, previous operations are reversed. In line 5 the last name and first name are isolated with the replacement operators \(\). The reversal occurs when \2 is given first, and \1 is given second.

Note that the comma is deleted (line 6) as a ''cleanup'' measure.

Replacement Operator

```
1,$s/{ //
1,$s/ }//
1,$s/<$//
1,$s/>//
1,$s/\([^,]*\)\(, [^:]*\)/\2 \1/
1,$s/^, //
1,$p
Wanda Dann:Morrisville:NY:13408:(315) 684-61
Georgette Geotsi:Bridgeport:CT:06601:(203) 576-47
Priscilla Griscom:Groton:CT:06340:(203) 449-67
Warren Guy:Easton:PA:18042:(215) 250-54
Yassin Hassan:College Station:TX:77843:(409) 845-70
Uriah Heep:Wapping:UK:SE4:(carrier pigeo
Jeffrey Johnson:Havre:MT:59501:(406) 265-41
Walter Johnson:Fayetteville:NC:28301:(919) 486-15
Jun-Ing Ker:Ruston:LA:71272:(318) 257-29
Thomas Kiesler:Kansas City:MO:64110:(816) 235-14
Jack Kurzweil:San Jose:CA:95192:(408) 924-39
Deborah Levinson:Tampa:FL:33620:(813) 974-48
Gonhsin Liu:Bridgeport:CT:06601:(203) 576-47
Mark Manchester:Morrisville:NY:13408:(315) 684-61
Unny Menon:San Luis Obispo:CA:93407:(805) 756-23
John Potter:Lock Haven:PA:17745:(717) 893-23
Naseem Qazi:Utica:NY:13504:(315) 792-73
Joseph Richardson:Lake Charles:LA:70609:(318) 475-58
Michael Rider:Ada:OH:45810:(419) 772-23
Patricia Strauss:Providence:RI:02908:(401) 456-80
William Swanson:San Jose:CA:95192:(408) 924-38
```

Summary

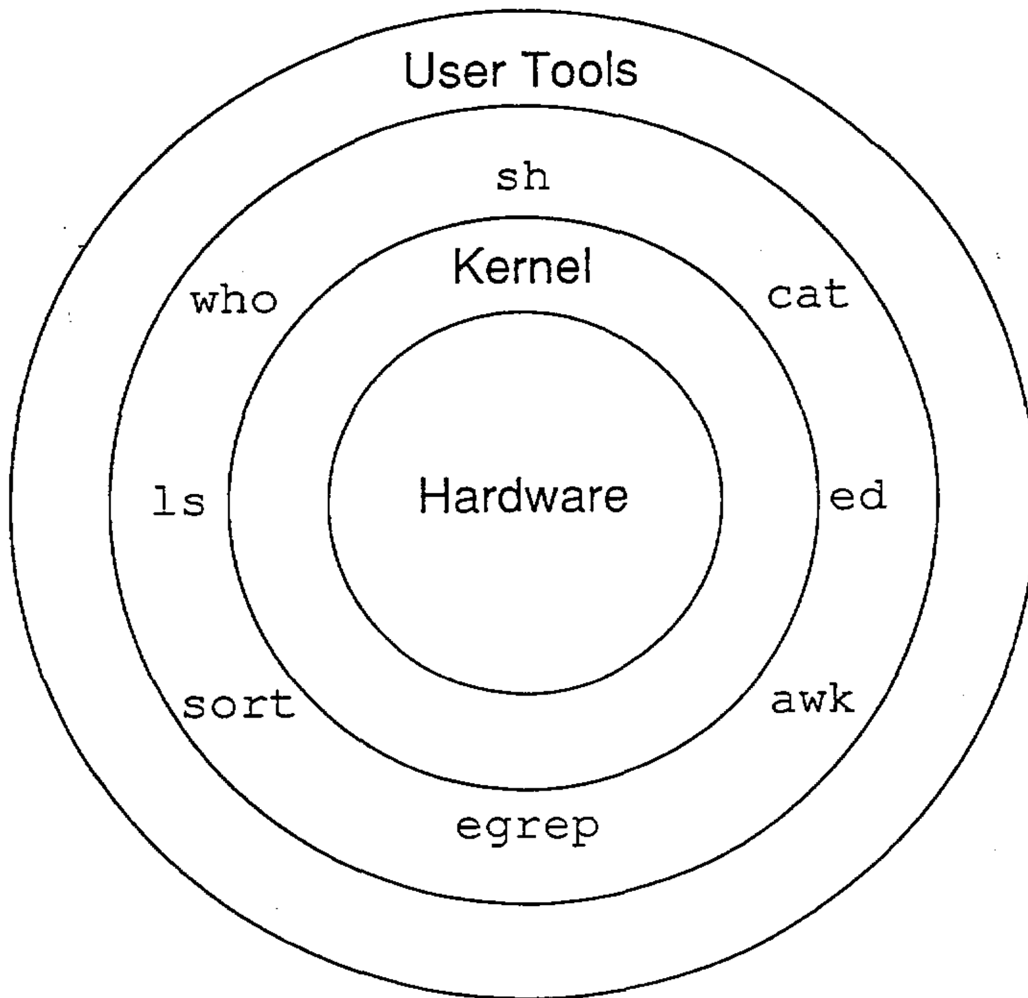
Operator	Function
.	Match of any operator except newline
*	Repetition operator (zero or more)
[ ]	Character class
[^]	Complement character class
^	Context operator (beginning of line)
$	Context operator (end of line)
\(\)	Replacement operators
\	Quoting character

Where Are We?

The environment that the user enters upon logging onto the UNIX® system is loosely known as *the shell*. The term was probably coined to suggest a layer around the operating system constituting the user interface. In fact, the shell is a user command situated at the same level as other user commands. Its interactive nature and role as a command interpreter engenders the illusion of a "shell layer."

← csh
% sh
$ ps

Where Are We?



What Do We Have?

In the first part of this module, we described a small set of concepts whose central connecting threads are abstraction—the ability to generalize a set of primitives into derived tools—and encapsulation—the ability to capture what was generalized into a named object that can be reinvoked. We discussed conventions generally called redirection—the pipe symbol (|), the output symbol (>), the input symbol (<)—to demonstrate how simple commands and even languages can be combined into more complex solutions.

But what is the ''glue'' that binds all of this together into meaningful units of computation?

What Do We Have?

- The *illusion* of another system layer
- Rich set of primitives (commands) designed to build tools
- Ability to connect commands in interesting ways
  - pipes
  - redirection
- Ability to create new commands (reflecting user's ideas)
- Ability to execute commands concurrently (back-ground jobs)
- Ability to amplify the power of commands using other commands
- Endless ability to encapsulate functionality in a com-mand, such that it may in turn be encapsulated in another command
- What holds all of this together?

The Shell: a Language of Commands

Aside from its role as a command interpreter, the `sh` command is also a programming language. The UNIX® system commands constitute most of the shell language's reserved name list. Command pipelines are examples of the shell language's simple constructions. The redirection symbols are also important syntactical elements of the language.

As these high-level constructions suggest, the shell is a *specification-level* language. Given a set of user applications or programming utilities, such as the sorting utility `sort`, the user naturally runs into cases where he or she wants the output of one application to flow as input into another. The shell language provides the appropriate notation to implement that flow.

In addition, the user is able to expand the shell reserved name list by writing tools for the local environment.

The Shell: a Language of Commands

- A command just like every other command
- Interprets other commands
- Primitives of the language are commands
- Since user can create new commands, the language is "user-extensible"

Attributes of the Shell Language

Although the `sh` command is the UNIX® system command interpreter, it is also a programming language. Its reserved name list is rich and well-defined: the UNIX system command set. Variables are supported and require no declaration. Expressions for sophisticated lexical matching (Regular Expressions) are also supported.

The shell language provides the flow-control statements normally found in general-purpose programming languages:

- `if-then-else` statement
- `while` loop statement
- `until` loop statement
- `for` loop statement
- `case` statement

(A fuller discussion of flow-control statements will appear later.) Shell also supports *parameter passing*. That is, it allows users to pass arguments (actual parameters) from the command line into shell programs (formal parameters).

Attributes of the Shell Language

- Very high-level, interpretive, concurrent programming language
- Commands as reserved words
- Named variables
- Metacharacters provide
  - filename expansion (wildcards)
  - I/O rerouting
  - quoting: control over metacharacters
- Rich set of flow control statements
  - `if-then-else` statement
  - `while` loop
  - `until` loop
  - `for` loop
  - `case` statement
- Parameter passing
- Shell language statements can be entered directly from keyboard
- Statements can be put in a file, creating a new command

How High-level?

Like any interpretive language, the shell language cannot perform as quickly as a compiled language. It does not require, however, the time-consuming (and error-prone) ''housekeeping'' of compiled languages. The C program on the following page, for example, required the user to declare and initialize variables, to open files before reading, to test for boundary conditions in the opened file, and to close the files at the conclusion of processing.

From the perspective of language design, as well, the C language is a stark contrast to shell. Unlike the specification-level model, C requires a notation appropriate for lower-level operations and is not intuitive to the tasks the program will perform, such as displaying a list of the current users on the system.

How High-level?

- How many users currently on the system?
- C specification

file: `users.c`

```
#ident "@(#)icis_shell/intro ⓘ 1.4
#include <stdio.h>

main()
{
    char buf[BUFSIZ + 1];
    FILE *fp1, *fp2;

    if ((fp1 = popen("who"   ,  "r")) == (FILE *) NULL ||
        (fp2 = popen("wc -l", "w")) == (FILE *) NULL)
        exit(1);

    while (fgets(buf, BUFSIZ, fp1) != (char *) NULL) {
        fputs(buf, fp2);
    }

    fclose(fp1);
    fclose(fp2);
}
```

functions

- Execution

```
$ cc -o users users.c
$ users
3
$
```

Compare this with D1-17

How High-level? (cont.)

- How many users currently on the system?
- Shell specification

```
$ who | wc -l
3
$
```

How High-level? (cont.)

By contrast, shell requires only two primitives, piping the output of the who command as input to the wc command. The execution of this program depends only on specification.

A Language of Commands

The Anatomy of a Command

Under the UNIX® system, both compiled binary executables and executable files containing shell statements are regarded as commands. Most commands accept arguments, including command options and files containing data.

original c program

shell scripts

The Anatomy of a Command

- Simple general form

```
$ command argument_list
```

- *command*
  - name of a compiled program in a binary file (`a.out`)
  - name of an ASCII file containing shell language statements
- *argument_list*
  - optional space-separated list of words
  - interpretation of arguments depends on *command*

How Commands Interpret Arguments

The interpretation of command arguments depends on the command. While hyphen-prefixed arguments uniformly are interpreted as *command options*—predefined strings that refine command behavior—all other arguments are treated differently from command to command.

For example, the `cat` command expects at least one of its arguments to be the name of a file. While `cat` accepts options (e.g., `-v` for verbose display, including non-printing characters), any arguments not prefixed by a hyphen must be one or more file names.

By contrast, the command `mailx`, aside from command options, accepts only the names of users. Any other string will be interpreted as an undeliverable address.

The `echo` command interprets its arguments as literal strings, displaying whatever it reads. It is important to be aware that `sh` interprets `echo`'s arguments first. Thus while a string such as `r*` is literal to `echo`, it is not a literal string to `sh`; it is an expression that expands to all file names in the current directory beginning with `r`.

How Commands Interpret Arguments

- Argument interpreted as a file name

```
$ cat rpm
```

name of file

- Argument interpreted as a user name

```
$ mailx rpm
```

name of user

- Argument interpreted as a literal string

```
$ echo rpm
```

literal strings

Running a Command

The sequence with which sh reads a line of input is important. Unexpected results are sometimes attributed to the executed command when they should have been pinned to the behavior of sh.

sh reads from left to right. If it finds that the first string is a file containing machine language, the file is executed directly without further interpretation. If it contains ASCII characters, the currently running sh process makes a copy of itself to interpret the file. The parameters to the file (command) are passed into the program contained there, expanded, and the first sh process suspends execution until the file has been interpreted by the new sh process. The exit status of the executed file is recorded, so conditional statements have a basis for decision. At the end of execution, the sh copy exits (the child process dies), and the sh process that had begun parsing

Running a Command

- Sequence of events
  - line of input is read by the shell
  - shell searches for *command*
  - if *command* is a binary executable, it is run directly
  - if *command* is an ASCII file, another copy of the shell reads the file for commands
  - *argument_list* is passed to the command
  - shell suspends execution until *command* terminates (*foreground* processing)
  - shell examines exit status of *command*
    - 0 value: *true*
    - 1 - 255: *false*
    - truth values used by flow-control statements

Consecutive Commands

The semicolon operator (`;`) is a statement terminator. Thus, the following executions of the `who` command and `date` command:

```
$ who
root        console      Jul 23 12:39
rpm         xt002        Jul 23 13:32
lally       tty05        Jul 23 11:43
jeromes     xt021        Jul 23 10:38
$ date
Sat Jul 23 13:44:49 EDT 1988
$
```

and

```
$ who; date
root        console      Jul 23 12:39
rpm         xt002        Jul 23 13:32
lally       tty05        Jul 23 11:43
jeromes     xt021        Jul 23 10:38
Sat Jul 23 13:45:19 EDT 1988
$
```

are almost identical. The `sh` command in the second case omits to print a prompt following each execution. The semicolon operator causes these commands to be executed consecutively.

Consecutive Commands

- Sequential processing general form

$command_1$; $command_2$; . . . ; $command_n$

- Commands are executed left to right
- $command_n$ must complete before $command_{n+1}$ will execute

Concurrent Commands

By contrast, the pipe causes all operands (commands) to be executed concurrently:

```
$ ed shell_lang
64346
! ps -fu tq | sed 's/^/Ⓣ/'
     UID   PID  PPID  C    STIME TTY      TIME COMMAND
      tq  4271     1  0 07:58:36 tty05    0:01 -sh
```

Concurrent Commands

- Pipeline general form

$command_1$ | $command_2$ | . . . | $command_n$

- $command_1$ through $command_n$ are executed concurrently
- Standard output of $command_1$ is routed by the shell into the standard input of $command_2$
- More generally
  - standard output of $command_n$ is routed by the shell into the standard input of $command_{n+1}$
- Example

```
$ ps -fu rpm | nl
   1    UID    PID  PPID  C    STIME TTY     TIME COMMAND
   2    rpm 25247    441  2 10:47:39 ttyih   0:01 -ksh
   3    rpm 25316 25247  5 10:48:36 ttyih   0:00 ps -fu rpm
   4    rpm 25317 25247  1 10:48:36 ttyih   0:00 nl
$
```

Summary

- The shell programming language
  - very high-level
  - interpretive
  - concurrent

Shell Statement Encapsulation

Overview

You have seen that statements in the shell language can be executed consecutively or con-

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- How can shell statements be encapsulated?
- What are shell programs?
- How does encapsulation promote reuse?

Encapsulation

Shell language statements can be captured in unnamed objects (statement groups and statement bodies) or in named objects (functions and files). We will examine group and file encapsulation here. Statement bodies will be seen in the units on flow control. Functions are beyond the scope of this module.

Encapsulation

- Statements that perform a task can be encapsulated in
  - unnamed objects
    - statement groups
    - statement bodies
  - named objects
    - functions
    - files
- Encapsulations form building blocks
- Can be combined to solve larger and larger problems

Groups of Statements

The pipe operator (|) has a higher precedence than the terminator operator (;), as the example on the following page demonstrates. The standard output of the `date` command is directed to the terminal screen. Then the `who` command is executed, and its standard output is routed by the shell into the standard input of the `wc` command.

If you wish the standard outputs of `date` and `who` to be redirected together, then they can be grouped with parentheses. The grouping causes the statements within to act as a unit. Within the group, `date` is executed and its output is sent down the pipe. Then `who` is executed and its output flows down the same pipe. From outside, the group appears to be a single command.

Groups of Statements

- Pipe (|) has higher precedence than terminator (;)
- Example

```
$ date; who | wc
Sat Jul 23 13:50:06 EDT 1988
      6      30     222
$
```

- Consecutive commands can be grouped with parentheses
- Standard output of group can be redirected
- Example

```
$ ( date; who ) | wc
      7      36     251
$
```

Shell Program Files

Commands such as `cat` and `sort` exist as binary executable files and are invoked by name. Shell statements as well can be encapsulated in files, invoked by file name, and used in combination with other commands. For example, the standard input and standard output of shell programs can be redirected. A shell program also can be used as a primitive in other shell programs. Thus shell programs can be the building blocks with which complex problems are solved.

Shell Program Files

- ASCII file containing shell statements
- Usage identical to a binary executable such as `/bin/cat`
- Can be used as primitives within other shell programs
- Ability to create new commands around existing tools

Creating a Shell Program

A shell program can be created simply by placing shell statements in a file and then giving that file permissions allowing execution. The shell program can be executed by invoking the file as though it were a binary executable.

Creating a Shell Program

- Place the shell statements in a file
- Change the file permissions to allow execution
- Use the file name as the command name

Creating a Shell Program: Example

The file howmany contains a shell program that will output the number of users currently logged into the system. The file can be treated as a command once permission for execution has been given.

Shell programs can be documented by preceding commentary with the # character. If a line or a word begins with a #, then all characters from the # to the end of the line are ignored.

Creating a Shell Program: Example

- Specification

file: howmany

```
# display the number of users
#    logged into the system

who | wc -l
```

- Execution

```
$ chmod +x howmany
$ howmany
     12
$
```

Creating Commands from Commands

The shell program `howmany` can be used as a primitive within other shell programs. The ease with which shell statements are encapsulated and invoked contributes to making the UNIX® environment a very productive one.

Creating Commands from Commands

- Specification

file: report

```
# display the date and the number of users
#    logged into the system

date

echo "\tnumber of users = \c"
howmany
```

- Execution

```
$ chmod +x report
$ report
Wed Aug 30 19:52:26 EDT 1989
        number of users =      12
$
```

Summary

- Encapsulation mechanisms
  - groups
  - files
- Recombination mechanisms
  - pipe operator
  - commands of commands

Flow Control

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- When should the flow of control be altered by the success or failure of a command?
- Does the shell language have conditional or looping constructs?
- What can be tested?

The if-then Statement

- General form:

```
if
   command_list
then
   body
fi
```

- *command_list* executed
- If exit status of *command_list* is true, *body* is executed
- Otherwise, *body* is not executed
- Execution continues at statement following `fi`

The if-then-else Statement

- General form:

```
if
   command_list
then
   body1
else
   body2
fi
```

- If exit status of *command_list* is true, $body_1$ is executed
- If exit status of *command_list* is false, $body_2$ is executed
- *command_list* or either *body* can contain any valid shell statements, including another `if`

if-then-else: Examples

The `egrep` command is successful if and only if at least one input line contains a string that matches the regular expression. In the examples on the following page, the success or failure of the `egrep` command determines which messages are sent to the standard output.

All output redirected to `/dev/null` is automatically discarded. Here the output of `egrep` is discarded because only the success or failure of the command is required to choose which message to send.

The `:` command does nothing. It is used in flow-control statements where a command must appear but none is desired.

if-then-else: Examples

```
if
  egrep TX roster > /dev/null
then
  echo There are Texans in the class
fi
```

```
if
  egrep NJ roster > /dev/null
then
  echo New Jersey is represented
else
  echo No one from the Garden State
fi
```

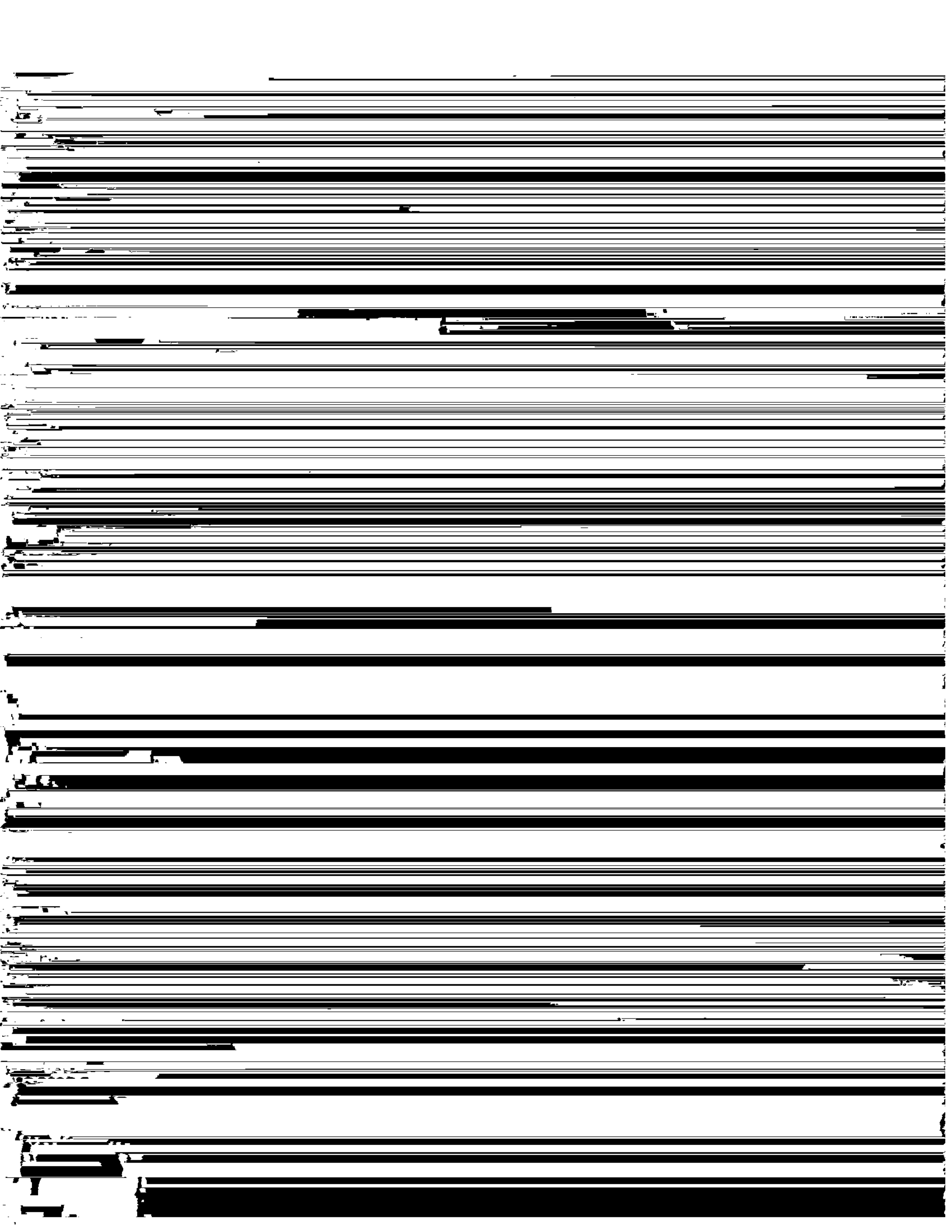
```
if
  egrep CA roster > /dev/null
then
  :    # do nothing
else
  echo No one from California
fi
```

- The : command does nothing

$echo 'hi' > /dev/tty

The test Command

The command `test` expands the usefulness of the `if` statement. Testing more than the exit status of a command, the `test` command evaluates file size, file permissions, and file type (directory or regular file), among other things. It adds boolean (logical) evaluations to the language and also allows compound predicates to be formed, using the `-a` (and) or `-o` (or) operator.



The test Command (cont.)

The table below summarizes the most common tests.

test	returns true if string s1 is
test s1 = s2	identical to string s2
test s1 != s2	not identical to string s2
test s1 -eq s2	numerically equal to string s2
test s1 -ne s2	numerically unequal to string s2
test s1 -lt s2	numerically less than string s2
test s1 -le s2	numerically less than or equal to string s2

The test Command (cont.)

- General forms:
- Compare two strings for equality

- Determine numeric inequality

The test Command: Examples

- Before we edit a file, check to see if it is writable
- If it isn't, check to see if it exists

file: ed_front

```
if
   test -w prog.c
then
   ed prog.c
else
   if
      test -f prog.c
   then
      echo prog.c is not writable
   else
      echo prog.c does not exist!
   fi
fi
```

The test Command: Examples (cont.)

- Before we edit a file, check to see if it is writable, readable, and a regular file
- If it isn't, explain why not

file: ed_front2

```
if
   test -f prog.c -a -r prog.c -a -w prog.c
then
   ed prog.c
else
   if
      test -f prog.c
   then
      echo prog.c is unwritable or unreadable
   else
      echo prog.c is not a regular file
   fi
fi
```

The while Loop

Frequently repetition is an important part of a task. For example, it might be useful to have a program report, every five minutes, the continued presence of a user on the system. The ''machine'' that drives this repeated task until some condition is met is known as a *driver*. The shell language's general driver is called a `while` statement, or `while` loop.

The while Loop

- General form:

```
while
   command_list
do
   body
done
```

- *command_list* is executed
- While exit status of *command_list* is true, execute *body*
- If exit status of *command_list* is false, continue execution at statement following `done`
- *body* may be executed zero times

The while Loop: Examples

- Loop while the file named memo is empty

```
while
  test ! -s memo
do
  echo Memo not started yet
  sleep 60
done
```

- Print message as long as rpm is logged in

```
while
  who | egrep rpm > /dev/null
do
  echo Bob is still at work!
  sleep 300
done
```

The while Loop: Examples (cont.)

The command `true`, like the `:` command, does nothing, but does it successfully. Either command can be used to create an infinite loop.

The command `banner`, like `echo`, prints its arguments to the standard output, but in a more noticeable form.

The while Loop: Examples (cont.)

- Infinite loop

```
while
  true
do
  banner Hello
done
```

/usr/5bin/banner

Summary

- The `if-then-else` statement
  - choosing a course of action
- The `while` statement
  - repeating a course of action
- The `test` command
  - searching for the truth

Shell Variables

Overview

The preceding flow-control statements provide important instruments for commands to communicate with each other. That is, the success or failure (even type of failure) of one command can be communicated to another. The most important facility for communication among commands, tools, and programs, however, is the variable. More than capturing the simple success or failure of a preceding command, the variable can also record the computed results of that command.

This unit describes how shell variables are created and used, and introduces the most useful environment variables.

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What are shell variables?
- How are they created and used?
- What are environment variables?

Shell Variables

Shell language variables are simple and require little of the programmer. They spring into being by simply being named. That is, no declaration or definition is necessary.

All shell variables are string variables (type `character` in Pascal or type `char` in C). It follows that integer arithmetic of the form

```
num op num
```

where `num` is a variable containing an integer value and `op` is an arithmetic operator, is not possible.

Shell Variables

- Facility for communicating among programs
- Named variables
- Not declared
- One type
  - ASCII text
- Recursive definition possible
- Shell predefines large set of variables

```
$ PATH=/bin:/usr/bin:/usr/lbin
$ echo $PATH
/bin:/usr/bin:/usr/lbin
$ PATH=$PATH:$HOME/bin:.
```

Valid Names: Rules

Shell variables may have names of any length and all characters in the name are significant.

Shell variable names may contain any alphabetic or numeric characters or an underscore character (_), although the first character in the name must be either an alphabetic character or the underscore character (no digits). Special characters are not permitted in variable names.

Shell variable names should represent the values they contain. Since there is no length restriction, cryptic shell variable names can be easily avoided.

Valid Names: Rules

- Name can be any length
- All characters significant
- Allowable characters
  - alphabetic
  - numeric
  - underscore
- Must start with alphabetic character or underscore
- Special characters not permitted in variable names

Setting a Variable's Value

A variable can be assigned using three different mechanisms:

1. explicit assignment (=)
2. `read` statement
3. `for` statement

The first two are described on the following page. We will describe the third shortly.

Setting a Variable's Value

- General form

```
name=string
```

- Set the value of *name* to *string*
- *string*: whitespace-terminated character string

- General form

```
read var1 var2 ... varn
```

- Shell reads a line from the standard input
- Input line is broken up into whitespace-separated *words*
  - value of each variable (var_i) is set to the corresponding word in the input line
  - if fewer variables than words, var_n is set to the remainder of the input line

Accessing a Variable's Value

- General form

```
$ name
```

- Shell replaces all occurrences of variables preceded by a dollar sign with the value of the variable
- If *name* is not set, the null string is substituted

Setting/Accessing a Variable's Value

On the following page, the variable `Name` is assigned the string `Bob`. The variable springs into being by being named. No declaration is necessary. The `echo` command then displays the string assigned to `Name`.

However, the `echo` command did not access the value of `Name`. The shell command interpreter, `sh`, replaced `$Name` with the value of the variable. The `echo` command received the value, `Bob`, as an argument to be displayed.

In the second example, `read` statements are used to assign variables.

Setting/Accessing a Variable's Value

```
$ Name=Bob
$ echo $Name
Bob
$
```

```
$ read city state zip
Summit NJ      07901
$ echo city state zip
city state zip
$ echo $city $state $zip
Summit NJ 07901
$ read x y
This is a test
$ echo $x
This
$ echo $y
is a test
$
```

Setting/Accessing a Variable's Value (cont.)

```
$ cmd=date
$ echo $cmd
date
$ $cmd
Sun Jun 28 08:47:58 EDT 1987
$
```

Environment Variables

The following names are used by the shell command interpreter (and some commands such as `vi` and `mailx`) and should be used only for the purposes described.

HOME

HOME is set to the path name of the login directory. The login procedure sets the variable from the information taken from the user's `/etc/passwd` entry.

This variable is useful in making portable shell programs. Instead of specifying an absolute path name for a file, the programmer can specify the file's relative position to the user's login directory.

LOGNAME

LOGNAME is set by the login procedure to the user's login name, and is often used when a shell program needs to send a mail message to the user who is executing the program.

MAIL

If this variable is set, `sh` will look in the designated file to see if any new mail has arrived since the last prompt. If new mail has arrived, then `sh` will print the message ''you have mail'' before prompting.

The value of this variable does not dictate where mail is sent to or read from; it only supplies information to `sh` telling it where to look for new mail.

MAILCHECK

This variable specifies in seconds how often `sh` checks for the arrival of new mail. The default value is 600 (10 minutes).

Environment Variables

- HOME
  - set to login directory
  - useful in creating portable shell programs
- LOGNAME
  - set by login procedure to login name
- MAIL
  - if set, `sh` will print "you have mail" if mail arrived since last prompt
- MAILCHECK
  - how often (in seconds) `sh` checks for arrival of new mail

Environment Variables (cont.)

PATH

This variable is set by the login procedure to a list of directory path names separated by colons. When you request that a command be executed, `sh` searches the ordered list of directories for an executable file whose name is the command. The first such file found is executed. If no such file is found, `sh` tells you so.

PS1

`PS1` (Prompt String 1) contains the primary prompt string, by default the dollar sign followed by a space.

PS2

`PS2` (Prompt String 2) contains the secondary prompt string, by default the character > followed by a space. If `sh` recognizes that you have started a command or statement and have not finished it, then `sh` will prompt for more input with the value of `PS2`.

TERM

`TERM` should be set to your terminal type or model. Binary descriptions of many terminal types can be found under the `/usr/lib/terminfo` directory. Several programs, such as `vi`, use the descriptions to manipulate terminal screens.

Environment Variables (cont.)

- `PATH`
  - set by login procedure to a list of directories
  - used to find commands
- `PS1`
  - primary prompt string
- `PS2`
  - secondary prompt string
  - used for incomplete commands or statements
- `TERM`
  - set to terminal type or model
  - used by commands that manipulate terminal screens

Summary

- Shell variables
  - facility for communicating among programs
  - named
  - not declared
  - one type
- Assignment
  - explicit assignment (=)
  - `read` statement
- Access
  - `$var`
- Predefined set of environment variables

Flow Control with Variables

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What are the primary methods of communication in the shell language?
- How can they be combined?

Combining Flow Control and Variables

Flow control statements and variables are the primary methods of communication in the shell language. Each is a valuable facility in its own right, but more importantly, they can be combined easily to build very powerful tools.

The program on the following page uses the variable `name` to read file names from the standard input. As long as file names are provided, the body of the loop will be executed and the named files manipulated. When there is nothing more to read, the `read` statement will fail and the loop will terminate.

Suppose that when prompted for a file name, the user enters a string of several words. The variable `name` will contain all the words separated by spaces. In the subsequent test, `$name` will be replaced by the multi-word string. The quotes around `$name` ensure that the value of the variable will be passed as a single argument to the `test` command. Without the quotes, the `test` command would receive each word as a separate argument and would be confused.

Likewise, if the user simply presses the return key when prompted, `$name` will be replaced by nothing. The quotes ensure that the `test` command will receive an empty string as an argument. The use of quotes in the shell language will be described in more detail later.

Combining Flow Control and Variables

- Two methods of communication
  - flow control statements
  - variables
- Combination produces powerful tools
- Example

file: ed_1

```
while
   echo Enter filename:
   read name
do
   if
      test -f "$name" -a -r "$name" -a -w "$name"
   then
      ed "$name"
   else
      if
         test -f "$name"
      then
         echo "$name" is unwritable or unreadable
      else
         echo "$name" is not a regular file
      fi
   fi
done
```

The for Loop

The `while` and `read` statements together allow the creation of drivers that operate on arbitrary numbers of objects as they enter a program through standard input. There are times, however, when a driver is needed to operate on a known set of objects. For example, you may wish to display various information for each file in the current directory. For this type of task, the `while` statement may be awkward.

The shell language has a second driver called the `for` statement. The `for` statement is given a list of words and a body of statements. The body is executed for the first word in the list, executed again for the second word in the list, and so on until the list of words is exhausted. The body of statements can refer to the current word in the list by accessing the value of a loop-control variable.

Recall that the `for` statement was listed as the third mechanism for assigning a variable.

The for Loop

- General form

```
for name in word_list
do
  body
done
```

- *name* is the loop-control variable
- For each word in *word_list*
  - set the value of *name* to the current word
  - execute *body*
- Loop terminates when *word_list* is exhausted

The for Loop: Examples

- Edit a set of files

file: `ed_files`

```
for fname in memo junk paper.2
do
   ed $fname
done
```

- Find out a lot of information about a set of files
  - do not run the commands on files that are not readable

file: `ed_files2`

```
for i in roster roster_new roster big_file small_file
do
   if
      test -r $i
5  then
      echo File $i:
      ls -l $i
      file $i
      wc $i
10 else
      echo Cannot open $i
   fi
done
```

The break and continue Statements

During an iteration of a loop, an event may occur that requires termination of the loop, or of just the current iteration. The `break` statement within a loop causes the termination of the entire loop. The `continue` statement within a loop terminates the current iteration and causes the next iteration, if any, to begin.

The break and continue Statements

- Alter execution of `for` and `while` loops

- General form

```
break
```

  - Terminates execution of inner-most enclosing `for` or `while` loop
  - Execution resumes at the statement following the next `done`

- General form

```
continue
```

  - Forces jump to next iteration of inner-most enclosing loop

break and continue: Example

The shell program on the following page edits each editable file in the current directory. After a file is edited, the program continues to the next iteration of the loop. When the program encounters an uneditable file, the user is asked whether or not to stop the program, and the `read` statement places the user's response in a variable. If the variable's value matches a string, the `break` statement terminates the loop. If not, the current iteration of the loop ends and the next iteration, if any, begins.

break and continue: Example

- Query the user and act accordingly

file: ed_2

```
for name in *
do
   if
      test -f $name -a -r $name -a -w $name
   then
      echo Editing file $name
      ed $name
      continue
   fi

   echo Cannot edit $name
   echo Enter q to quit:
   read answer

   if
      test "$answer" = q
   then
      break
   fi
done
```

The case Multi-way Branch

The `if-then` statement allows statement execution to depend on the success of a command or test. The `if-then-else` statement allows a choice between two bodies of statements. A choice from more than two bodies of statements can be achieved by nesting `if-then-else` statements.

```
if
   test "$answer" = yes
then
   echo the answer is yes
else
   if
      test "$answer" = no
   then
      echo the answer is no
   else
      if
         test "$answer" = maybe
      then
         echo the answer is maybe
      fi
   fi
fi
```

However, when the choice among several bodies of statements is based on the value of a variable, the `case` statement is a convenient mechanism.

The case Multi-way Branch

- General form

```
case value in
    pattern1) body1 ;;
    pattern2) body2 ;;
    pattern3|pattern4) body3,4 ;;
. . .
    patternn) bodyn ;;
esac
```

- *value* is compared to each *pattern*
- If $pattern_i$ matches *value*, execute $body_i$
- When `;;` is reached, execution continues at next statement after `esac`

$ ps -futeach76

$ exec ksh

The case Multi-way Branch (cont.)

- Patterns consist of
  - literal strings
  - variable substitutions
  - shell wildcards
- Vertical bar (|) provides pattern alternation
  - *body* is executed if *value* matches any member of list of alternate patterns

The case Multi-way Branch: Example

- Give the user several choices

file: `ed_3`

```
for name in *
do
   if
      test -f $name -a -r $name -a -w $name
   then
      echo Editing file $name
      ed $name
      continue
   fi

   echo Cannot edit $name
   echo Enter c to continue or q to quit
   read answer

   case "$answer" in
      c|C) continue
           ;;
      q|Q) break
           ;;
      *)   echo Do not understand. Goodbye.
           break
           ;;
   esac
done
```

Summary

- Two methods of communication
  - flow control statements
  - variables
- Combination produces powerful tools
- `while` and `read` together
  - processing objects from standard input
- The `for` statement
  - processing objects from a list
- The `case` statement
  - choosing from several courses of action

Unit B6 — Problem-solving in the UNIX® Environment — Shell

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What is a shell process?
- How and when are processes created?
- What tools are available to manage them?

Attributes of a Process

So far you have seen how to create and execute shell programs, how shell programs can invoke other shell programs, and how variables and flow

Attributes of a Process

- Every running executable occupies a process
- Executables are created by C compiler, FORTRAN compiler, etc.
- A process must be created by another process
  - process creation performed by kernel upon `fork()` request by a process (parent)
  - newly-created process (child) initially identical to parent
  - a process can replace the executable it is running with an `exec()` kernel request

Attributes of a Process (cont.)

- Input
  - standard input (`stdin`)
  - arguments
  - files explicitly opened by process
- Output
  - standard output (`stdout`)
  - standard error (`stderr`)
  - files explicitly created by process
  - exit status
- Kernel supports re-routing of input sources

The Services That the UNIX® System Provides

Earlier, we stated that the shell language is a very high-level language. This can be seen by the number of services that are provided, from process creation and execution to the treatment of output devices as files. This high-level interface frees users to do their work without dealing with the underlying system implementation.

The Services That the UNIX® System Provides

- Process creation and execution
- Input/output
  - standard interface to user including backspace, line-kill
- File manipulation
  - allows devices to be manipulated as files, e.g., `/dev/tty`

Foreground Processes

As we said earlier, when you execute a program from the command line, the process executing

Foreground Processes

- A command entered from terminal will cause interactive shell to
  - interpret command line
  - create child process to execute command or interpret shell program
  - wait for child process to complete
  - prompt for next command

The ps Command

The ps command reports the status of your running processes. If executed without any options, it produces four columns of information:

PID	process identification number
TTY	name of the terminal-special file from which the process was invoked
TIME	cumulative execution time for this process
COMMAND	name of the command

If the -f option is used, the ps command prints a full report.

UID	login name of the user executing the process
PID	process identification number
PPID	process identification number of the parent process
C	processor scheduling information
STIME	time the process was started
TTY	name of the terminal-special file from which the process was invoked
TIME	cumulative execution time for this process
COMMAND	full command name with its arguments

The following page illustrates a process tree structure. From inside an editing session, the ps command is executed. It lists four processes: the login shell process, the process executing ed, the shell process interpreting the ps command line, and the process running the ps command itself.

The ps Command

- `ps` command prints status of running processes
- Option `-f` illustrates process tree

```
$ ed processes
11843
!ps
   PID TTY       TIME COMMAND
 24609 ttyif     0:02 sh
 25400 ttyif     0:00 ed
 25411 ttyif     0:00 sh
 25412 ttyif     0:00 ps
!
!ps -f
 UID   PID  PPID  C    STIME TTY       TIME COMMAND
 rpm 25415 25400  0 18:04:07 ttyif     0:00 sh -c ps -f
 rpm 25416 25415 15 18:04:07 ttyif     0:00 ps -f
 rpm 24609     1  0 18:01:00 ttyif     0:02 -sh
 rpm 25400 24609  0 18:03:56 ttyif     0:00 ed processes
!
q
$
```

Background Processes

A process can create a child process and continue to execute without waiting for the child to complete. The parent and child will execute independently, concurrently, and with the same priority. The child is said to be a *background process* (sometimes called an *asynchronous* process).

Background processes ignore interrupts (via the *delete* key) so you can interrupt a foreground process without interrupting your background processes. We will see later how to interrupt them. Background processes do not ignore line disconnects, so they will terminate if you log off the system.

If a background process's standard input is not redirected, it will be redirected by default from the file `/dev/null` causing immediate receipt of an end-of-file. If the standard output is not redirected, it will print on the terminal screen and be interspersed with the output of foreground processes. Since background processes ignore interrupts, pressing the *delete* key will not stop their output.

Invoking A Background Process

- Invoked by ending command line with ampersand (&)
- Several background processes can run at same time
- Process identification number printed
  - needed to interrupt the process

```
$ grep NJ roster > output 2> errors &
24435
$
```

The kill Command

The `kill` command terminates the background process whose PID is given as an argument. By default, `kill` sends the termination signal (Signal 15) to the process. If the process ignores this signal, then `kill` with the `-9` option will kill it. Signal 9 is the *uncatchable* kill signal.

If the PID given to `kill` is `0`, then `kill` will send the termination or kill signal to all child processes of the current interactive shell.

The kill Command

- Used to terminate background process
- Argument to `kill` command is process identification number of process to be terminated
- Option `-9` is uncatchable kill signal
- All child processes can be killed at once

```
$ kill 0
```

Background Processes

- Also called asynchronous processes
- Parent processes do not wait for them
- Run at same priority as foreground processes
- Will ignore interrupts (*delete* key)
- Will not ignore line disconnects or logging off
- Usually have input and output redirected
- If standard input is not redirected, input is supplied from `/dev/null`
- If standard output is not redirected, prints to terminal (cannot be stopped)

Invoking A Background Process

To execute a command in the background, follow it with an ampersand (&).

You can have several background processes at the same time. If you wish to interrupt one of them, you will need its process identification number (PID). When a background process is created from the command line, its PID is printed on the terminal screen.

The wait Statement

It may be useful for a shell program to place one or more programs in the background, do some other tasks, and then wait until the background processes are completed. The `wait` statement causes a shell program to suspend execution until all child background processes are completed.

If the `wait` statement is invoked from a command line, then the interactive shell process will suspend its execution until all its child processes are completed. If the `wait` statement is supplied with a PID, the shell process will wait only for that process. A process can wait only for its children. It cannot wait for its parents, siblings, cousins, or grandchildren. If a background pipeline is to be waited for, the `wait` statement should be given the PID of the last command in the pipeline (the sink).

The wait Statement

- Waits for background processes to complete
- Allows concurrent execution of independent tasks

```
# place independent tasks in background

task1 > output1 2> errors1 &
task2 > output2 2> errors2 &
task3 > output3 2> errors3 &

# wait for completion of tasks

wait

# use output of tasks
```

↑ all three tasks

The nohup Command

As mentioned earlier, background processes do not ignore line disconnects. If you log off the system or get disconnected by accident, all of your background processes will be terminated. If you wish a background process to ignore line disconnects, then precede the command line with the word `nohup`.

If standard output and standard error are not redirected, then they are redirected to a file named `nohup.out` in the current directory. If the current directory does not give permission to create the file or if the file exists and write permission is denied, then the file is created in the user's login directory.

The nohup Command

- Background command ignores line disconnects (including logging off) if preceded by `nohup`
- If output not redirected, it is appended to file named `nohup.out`
- Examples
  - redirect standard output and standard error

```
$ nohup sort long.file > out 2> errors &
```

  - redirect only standard output
  - standard error redirected to `nohup.out`

```
$ nohup sort long.file >out &
Sending output to nohup.out
```

Summary

- Processes are running programs
- Processes create processes
  - in the foreground by default
  - in the background by request with `&`
- Tools for process management
  - `ps`
  - `kill`
  - `wait`
  - `nohup`

Shell Programs

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- How can a shell program be invoked?
- How can it report its success or failure?
- How can it receive command line arguments?
- What is the scope of a shell variable?

The Anatomy of a Shell Program

You have seen how the execution of a shell program differs from a binary executable. The latter is executed directly by a child process, while the former serves as input to a child process executing the shell command interpreter, `sh`. However, the user sees no difference. Both are invoked by name, and standard input and output are available to each. As we shall see shortly, the commonality extends to the ability to communicate with a parent process through an argument list and an exit status.

The Anatomy of a Shell Program

- Execution different from binary executable such as `/bin/cat`
- Usage identical
  - standard input and output are available
  - arguments can be passed
  - exit status returned on termination
- Allows the user to create new commands around existing tools

Invoking Shell Programs

Shell program execution involves the creation of a child process to execute sh with the shell program file as input. Therefore, there are two ways to invoke a command implemented in the shell language. You have seen the first: grant the file execute permission and invoke the file by name. The command also can be executed by invoking sh explicitly and giving the shell program file as an argument. In the latter method, the file does not need execute permission.

Invoking Shell Programs

- General form

```
$ file
```

  - *file*: ASCII file with execute permission
  - invoking *file* implicitly executes a new shell process to interpret the contents of *file*

- General form

```
$ sh file
```

  - `sh`: invoke the shell command creating a new process
  - command argument *file* contains the shell program
  - `sh` command opens, reads, and interprets the contents of *file*

does not need
execute permission

Reporting Success or Failure

Earlier you saw how flow control statements used the success or failure of a command, such as `egrep` or `test`, to choose a course of action. An executing command reports its success or failure by returning an exit status to its parent. By convention, a zero value represents success, while a value from 1 to 255 represents failure. The large range permits a program to indicate the type of failure. In a shell program, the special variable `$?` expands to the exit status of the last executed command. The program on the following page outputs a message depending on the outcome of `egrep`, which returns an exit status of 1 if the pattern is not found, or 2 if the file is inaccessible or the syntax is incorrect.

When a shell program terminates, it returns by default the exit status of the last executed command. Therefore, by default the success or failure of a shell program depends on its children.

Reporting Success or Failure

- Each process returns an exit status
  - 0 value: *true*
  - 1 - 255: *false*
- Special variable `$?` expands to exit status of last executed command

file: state

```
egrep NJ roster > /dev/null 2>&1

case $? in
   0)  echo NJ represented
       ;;
   1)  echo NJ not represented
       ;;
   2)  echo roster file inaccessible or syntax error
       ;;
esac
```

- Shell program returns status of last executed command by default

The exit Statement

A shell program can terminate itself with an `exit` statement. An argument is returned as the exit status. If no argument is given, the exit status of the previously executed command is returned.

The program on the following page edits a file if it is readable, writable, and an ordinary file. If so, the exit status of the program will be the exit status of `ed`. Otherwise, an error message is sent to standard output, and the program is terminated with an exit code of 1 or 2. The distinction may be useful to the parent process.

The exit Statement

- Causes immediate termination
- Argument used as exit status
- If no argument, exit status is that of previous command
- Example

file: edfront

```
if
   test -f prog.c -a -r prog.c -a -w prog.c
then
   ed prog.c
else
   if
      test -f prog.c
   then
      echo prog.c is unwritable or unreadable
      exit 1
   else
      echo prog.c is not a regular file
      exit 2
   fi
fi
```

Positional Parameters

Until now we have relied on variables to communicate information into a shell program. The shell language also supports a general facility for parameter passing.

Arguments (actual parameters) given to a shell program filename are passed into placeholders (formal parameters). These formal parameters are known as positional parameters and are numbered 0 through 9. The term positional derives from the fact that the position of each actual parameter passed (first, second, third, and so forth) corresponds to the numbered notation of each formal parameter. For example, in the case

```
$ shellprog arg1
```

`arg1` maps to `$1`.

Positional Parameters

- When a shell program is invoked, user may supply arguments
- The procedure accesses the values of the arguments via *positional parameters*
- Positional parameters
  - special variables
  - value of parameters automatically set by shell on invocation of the procedure

- General form

```
$n
```

- Where *n* is a digit from zero through nine
  - `$1`: value of first argument
  - `$2`: value of second argument
  - `$3`: value of third argument
  - . . .
  - `$0`: name of shell program

Argument List and Counter

The special variable $* expands to the entire list of parameters passed to shell program file. Note that by contrast, shell metacharacter * expands to a list of all file names in the current directory.

The special variable $# expands to the number of parameters in the argument list.

Argument List and Counter

- General form

```
$*
```

- The shell substitutes `$*` with the entire argument list
- Often used as the *word_list* in a `for` loop

- General form

```
$#
```

- The shell substitutes `$#` with the number of arguments in the argument list
- Often used to verify correct invocation of a command

Positional Parameters: Examples

- The user specifies the file name as the first argument

file: `edfront2`

```
if
   test -f "$1" -a -r "$1" -a -w "$1"
then
   ed "$1"
else
   if
      test -f "$1"
   then
      echo $0: "$1" is unwritable or unreadable
   else
      echo $0: "$1" is not a regular file
   fi
fi
```

Positional Parameters: Examples (cont.)

The program at right expects at least one argument. If the user fails to supply one, then the program tells the user how to use the command. The usage message includes the string `<file>`, which is standard notation, meaning that a file name is required. The single quotes around `<file>` prevent characters < and > from being interpreted as redirection operators. The use of quotes in the shell language will be described in more detail later.

Positional Parameters: Examples (cont.)

- The user can supply multiple arguments
- Each argument will be treated as the name of a file to edit
- Specification

file: `edfront3`

```
for name in $*
do
   if
      test -f $name -a -r $name -a -w $name
   then
      echo Editing file $name
      ed $name
      continue
   fi

   echo Cannot edit $name
   echo Enter c to continue or q to quit
   read answer

   case "$answer" in
      c|C) continue
           ;;
      q|Q) break
           ;;
      [illegible] understand. Goodbye.
```

Positional Parameters: Examples (cont.)

- Require at least one argument
- Issue usage message if no arguments

file: edfront4

```
if
   test $# -lt 1
then
   echo Usage: $0 '<file>'
   exit 1
fi

for name in $*
do
   if
      test -f $name -a -r $name -a -w $name
   then
      echo Editing file $name
      ed $name
      continue
   fi

   echo Cannot edit $name
   echo Enter c to continue or q to quit
   read answer

   case "$answer" in
      c|C) continue
           ;;
      q|Q) break
           ;;
      *)   echo Do not understand. Goodbye.
           break
           ;;
   esac
done
```

The shift Statement

Positional parameters allow you to access the first nine arguments by number, and `$*` allows you to access them all at once. A particular argument beyond the ninth can be accessed by "shifting" the argument list to the left. The `shift` statement reassigns the positional parameters so that `$1` will expand to the value of the second argument, `$2` will expand to the value of the third argument, and so on. The list cannot be shifted to the right, so the value of the first argument is no longer accessible. An argument to `shift` allows multiple shifts in one statement.

The shift Statement

- Allows access to individual parameters beyond the ninth
- Values of parameters "shifted" to left
  - `$1` expands to value of second argument
  - `$2` expands to value of third argument
  - `$3` expands to value of fourth argument
  - . . .
  - `$0` unaffected
- `$*` and `$#` will reflect changes
- Cannot shift to the right
  - value of first argument no longer accessible
- Optional argument permits multiple shifts

The shift Statement (cont.)

- Specification

file: shiftpar

```
while
   test $# -gt 0
do
   echo $# arguments: $*
   shift
done
```

- Execution

```
$ shiftpar lally rpm tq
3 arguments: lally rpm tq
2 arguments: rpm tq
1 arguments: tq
$
```

The set Statement

Positional parameters can be assigned values within a program. The `set` statement reassigns all positional parameters. The current values can be reused, but any that are not are lost.

The `set` statement can take options. In particular, the option `--` informs `set` that no subsequent string is to be treated as an option. This is necessary when a positional parameter is to be assigned a value that begins with a dash.

The set Statement

- Assigns list of strings to positional parameters
- General form

```
set -- string-list
```

- All positional parameters reassigned
- Existing positional parameter values can be used
- Unused positional parameter values lost

The set Statement (cont.)

- Specification

file: setpar

```
echo $# arguments: $*
set -- tq lally $*
echo $# arguments: $*
set -- $2 $3 $1
echo $# arguments: $*
```

- Execution

```
$ setpar rpm root
2 arguments: rpm root
4 arguments: tq lally rpm root
3 arguments: lally rpm tq
$
```

Command Options

Most commands accept options that refine their behavior. Options usually can be concatenated. For example, `wc -lc` yields a count of both lines and characters. Some commands have options which take arguments. The command `ed -p` *string file* causes the editor to use *string* as its prompt. A command developed in the shell language can have the same flexibility.

Command Options

- Command options
  - refine behavior
  - are preceded by dashes
  - can be concatenated (`wc -lc`)
  - can take arguments (`cut -d: -f1`)
- Arguments to shell programs can be preceded by dashes
- Program must interpret arguments as options

The getopt Command

The command `getopt` is a useful tool for processing options. Given the positional parameters and a list of valid options, it will print a reformatted list of parameters to standard output. The output can be used as arguments to the `set` command—the significance of the backwards quotes will be described later—which reassigns the positional parameters. The newly assigned parameters can be processed more easily.

The getopt Command

- `getopt` command reformats its arguments
- Combined with `set` to reassign positional parameters
- General form

```
set -- `getopt option-list $*`
```

  - where *option-list* is string of valid option letters
  - option letter followed by colon means option takes argument
- Newly assigned positional parameters easier to process
  - concatenated options and arguments are separated
  - string -- inserted after last option
- Invalid options cause failure

The getopt Command (cont.)

The program on the following page is designed to accept two options: `-q` and `-d`. The latter requires an argument. If `getopt` detects an invalid option or missing argument, `getopt` will fail, and so will `set`.

The getopt Command (cont.)

- Specification

file: options

```
if
   set -- `getopt qd: $*`
then
   :   # options valid, so do nothing
else
   echo Usage: $0 '[-q] [-d <directory>] <file>'
   exit 1
fi

echo $*
echo
```

- Execution

```
$ options -qd$HOME prog.c roster
-q -d /i8/rpm -- prog.c roster

$ options prog.c roster
-- prog.c roster

$ options -e
getopt: illegal option -- e
Usage: options [-q] [-d <directory>] <file>

$ options -d
getopt: option requires an argument -- d
Usage: options [-q] [-d <directory>] <file>

$
```

Processing the Command Line

The program on the following page edits files listed on the command line. The program also accepts two options: `-q` causes the program to terminate immediately if an uneditable file is encountered, and `-d` takes as an argument the directory in which the files reside. If the latter option is not given, the current directory is searched.

The loop processes the options and shifts them aside. It terminates when the argument `--` inserted by `getopt` is encountered.

Processing the Command Line (cont.)

The program makes an additional test to ensure that file names were included on the command line. The program then changes directory to where the files reside.

At its conclusion, the program need not `cd` to the original directory because the program is executed in a child process. The parent process still has the original directory as its current directory.

Processing the Command Line

```
# validate options

if
   set -- `getopt qd: $*`
then
   :   # options valid, so do nothing
else
   echo Usage: $0 '[-q] [-d <directory>] <file>'
   exit 2
fi

quit=no    # terminate if file uneditable?
dir=.      # directory where files reside

# loop until -- encountered

while
   true
do
   case $1 in
      -q) quit=yes
          shift      # shift -q aside
          ;;
      -d) dir=$2
          shift 2    # shift -d and argument aside
          ;;
      --) shift      # shift -- aside
          break
          ;;
   esac
done
```

Processing the Command Line (cont.)

```
# check for file names

if
   test $# -lt 1
then
   echo Usage: $0 '[-q] [-d <directory>] <file>'
   exit 1
fi

if
   test -d $dir
then
   cd $dir          # go where the files are
else
   echo $0: error: $dir is not a directory
   exit 3
fi
```

continuation of the previous program

Processing the Command Line (cont.)

```
for name in $*
do
   if
      test -f $name -a -r $name -a -w $name
   then
      echo Editing file $name
      ed $name
      continue
   fi

   echo Cannot edit $name

   if
      test $quit = yes    # immediate termination?
   then
      exit 4
   fi

   echo Enter c to continue or q to quit
   read answer

   case "$answer" in
      c|C) continue
           ;;
      q|Q) exit 4
           ;;
      *)   echo Do not understand. Goodbye.
           exit 5
           ;;
   esac
done
```

Variable Scope

As mentioned in the previous example, each process has its own notion of the current directory. If a shell program changes directory, the process that invoked the program is unaffected because a child process runs the program. Imagine the inconvenience if commands quietly changed the current directory of your interactive shell process.

Shell variables are also local to a process. Except for the special variable `$?`, a child process can never change the value of its parent's variable. In fact, a child cannot access the value of its parent's variable unless the parent explicitly gave access with the `export` statement. Exported variables are accessible to all descendants of a process.

Variable Scope

- By default, variables local to current program (process)
- Parent can grant evaluation (but not assignment) privileges to child processes
  - no extension of scope (no shared memory)
  - no true global variables (e.g., siblings can't see variables of other siblings)
- Facility for granting evaluation privileges: `export`

Variable Scope (cont.)

- Example

```
$ on_first=whose
$ sh
$ echo $on_first

$ on_second=whats
$ export on_second
$ sh
$ echo $on_second
whats
$ control-d $ echo $on_second
whats
$ control-d $ echo $on_second

$ echo $on_first
whose
$
```

- All environment variables typically exported by login (first parent) process

Summary

- Shell programs can be invoked explicitly or implicitly
- They can communicate with the invoking program
  - `exit`
  - positional parameters
- Positional parameters can be accessed
  - `$n` expands to value of n-th argument
  - `$0`, `$*`, `$#`
- Positional parameters can be manipulated
  - `shift`
  - `set`
- Command line options can be processed
  - `getopt`
- Variables are local in scope but values can be inherited
  - `export`

Quoting

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- How can the shell be made to ignore the special meaning of a character?
- What are the differences between the three quoting mechanisms?

Special Characters

Suppose you wish to search a file for lines containing a semicolon. A likely first attempt would not fare well.

```
$ egrep ; roster
usage: egrep [ -bcilnv ] [ strings ] [ file ] ...
roster:  not found
$
```

The semicolon has a special meaning in the shell language. It separates commands that are to be executed sequentially. You have seen other special characters as well: <, >, |, (,), and &. Characters ?, *, [, and] are used for matching file names. Even spaces, tabs, and newlines have special meaning. The first two separate words, while the last separates command lines.

To pass special characters as arguments to commands, the shell command interpreter must be told to treat the characters literally. The mechanisms for ''turning off'' the special meaning of characters are collectively known as *quoting*. There are three methods of quoting: the backslash, single quotes, and double quotes.

Special Characters

- Several characters have special meaning to the shell
  - invocation: ; | ( ) &
  - redirection: < >
  - regular expressions: ? * [ ]
  - delimiters: *space tab newline*
- To pass as arguments to commands, special characters must be quoted
- Quoted characters treated literally

Backslash

The backslash character turns off the special meaning of the single character that follows it. If the character has no special meaning, then the backslash has no effect. After the backslash is interpreted, it is removed from the command line, that is, the backslash is not passed as part of an argument.

Backslash

- Turns off the special meaning of single character that follows it
- No effect if it precedes a non-special character
- Removed after interpretation

Backslash: Example

- Specification

file: backslash

```
egrep \; roster > /dev/null 2>&1
case $? in        exit status
```

Single Quotes

Single quotes turn off the special meaning of all characters between them, including backslashes. In the example on the following page, the single quotes ensure that the enclosed backslashes are passed literally to the `echo` command.

Single Quotes

- Characters enclosed in single quotes have no special meaning
- Used to protect parts of a command line from the shell
- Specification

file: sing_quote

```
echo This \n line has \n several newlines
echo 'This \n line has \n several newlines'
```

- Execution

```
$ single_quotes
This n line has n several newlines
This
 line has
 several newlines
$
```

this \n line

Single Quotes (cont.)

The `egrep` command permits the searching for any of several patterns. In the example on the following page, the `outline2` command prints lines that contain the pattern `^\.VI` or `^\.SB`. The single quotes turn off the special meaning of the backslashes and the pipe.

Single Quotes (cont.)

- Specification

file: outline2

```
egrep '^\.VI|^\.SB' $*
```

- Execution

```
$ outline2 intro
.SB Introduction
.VI "Where Are We?"
.VI "What Do We Have?"
.VI "The Shell: a Language of Commands"
.VI "Attributes of the Shell Language"
.VI "How High-level?"
.VI "How High-level?"
.SB "A Language of Commands"
.VI "The Anatomy of a Command"
.VI "How Commands Interpret Arguments"
.VI "Running a Command"
.VI "Consecutive Commands"
.VI "Concurrent Commands"
$
```

Double Quotes

Double quotes turn off the special meaning of all characters between them, with a few exceptions. [illegible] expansion is permitted within double quotes, so the dollar sign retains its special mean-

Double Quotes

- Double quotes turn off special meaning of most characters
- Variable expansion permitted so $ remains special
- Command substitution permitted so grave accent (`) remains special (described later)
- Backslash remains special when followed by
  - $ to turn off variable expansion
  - ` to turn off command substitution
  - " to include literal double quote
  - \ to include literal backslash

Double Quotes: Examples

In the example on the following page, `$TERM` is expanded inside double quotes. The third `echo` statement shows that file name generation characters are protected from expansion by the double quotes.

Double Quotes: Examples

- Specification

file: doub_quote

```
echo The value of TERM is $TERM
echo "The value of TERM is $TERM"
echo The value of "*" is *
```

- Execution

```
$ double_quctes
The value of TERM is 630
The value of TERM is 630
The value of * is roster prog.c edfront
$
```

Double Quotes: Examples (cont.)

You have already seen double quotes used to preserve null arguments that resulted from expansion of an unassigned variable.

Double Quotes: Examples (cont.)

- Preserving null arguments

file: `ed_1`

```
while
   echo Enter filename:
   read name
do
   if
      test -f "$name" -a -r "$name" -a -w "$name"
   then
      ed "$name"
   else
      if
         test -f "$name"
      then
         echo "$name" is unwritable or unreadable
      else
         echo "$name" is not a regular file
      fi
   fi
done
```

Special Cases of I/O

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- Can the output of a command be inserted into a command line?
- Can a shell program invoke a command and supply the input as well?

I/O Redirection

You have seen how standard input and output can be redirected to files and to other commands. The standard output of a command can be redirected to a command line as well. That is, the output of one command can be used as arguments for another command. This mechanism is known as *command substitution*. Likewise, the standard input of a command can be redirected so that subsequent lines in the shell program are read. This facility is known as a *here document*.

I/O Redirection

- Redirection to files: < >
- Redirection to other commands: |
- Redirection to other command lines
  - command substitution
  - here document

Command Substitution

Back quotes (grave accents), like double quotes, allow for shell evaluation of the enclosed string. These quotes are not used for protection, however. They enclose commands, allowing the shell to execute those commands and substitute the quoted command string(s) with the resulting output. Another name for the more casual term *back quotes* is *command substitution.*

As the example on the following page shows, the shell scans the line performing command substitution as it expands variables, and interprets other quoting facilities. You have already seen another example:

file: `options`

```
if
   set -- `getopt qd: $*`
then
   :    # options valid, so do nothing
else
   echo Usage: $0 '[-q] [-d <directory>] <file>'
   exit 1
fi

echo $*
echo
```

The `getopt` command reformats its arguments and prints the results to standard output. The back quotes cause the output to become arguments to the `set` command.

here Document

The *here document* is a special form of input redirection. Rather than supplying input from a file or pipe, the input is supplied from within the shell program. In the general form on the following page, the command receives its input from the subsequent lines in the file. The closing *endmarker* marks the end of the input.

The placement of the << and the *endmarker* must be as shown. The closing *endmarker* must appear on a line by itself.

here Document

General form:

```
command <<endmarker
input
input
. . .
endmarker
```

- Used to supply input to command from within a shell program
- Routes standard input of *command* from the lines bounded by *endmarker*
- *endmarker* marks the end of the here document

here Document: Example

The program on the following page reports the date and the number of people on the system, and then sends a mail message to user `rpm`, recording the fact that the program was executed. User `rpm` may want to collect statistics on the usefulness of the program before investing further work in it.

The standard input of the mail message is redirected to the lines that follow. An exclamation point is defined to be the closing *endmarker*. The input lines are scanned by the shell interpreter before they are read by the `mail` command, so variable expansion and command substitution are performed.

here Document: Example

- Specification

file: `report_gen`

```
# display the date and the number of users
#    logged into the system

date
echo "\tnumber of users = `who | wc -l`"

# record fact that program was executed

mail rpm <<!
$0 ran by $LOGNAME on `date`
!
```

- Execution

```
$ report_gen
Tue Sep  5 22:04:49 EDT 1989
   number of users =        8
$ mail
1 messages
From rpm Tue Sep  5 22:04 EDT 1989
report_gen ran by rpm on Tue Sep  5 22:04:50 EDT 1989
?d
?q
$
```

here Document with ed

The edit facility can be used in a shell program by specifying a here document. Since the command is embedded in a program, some extra specifications are helpful.

First, suppress the printing of character counts upon entering the editor and each time you ''write'' the file. These counts are helpful when in interactive mode, but may have little meaning within a shell program.

Lines within a here document are indented over one tab to make the program more readable. However, ed does not recognize editor commands unless they are at the very beginning of the line. By appending << with a dash, the leading tabs are stripped before passing the line on to ed. CAUTION: Leading BLANKS are not stripped!

Lastly, turning on the help facility will cause an error message to be printed should the editor run into some problems. Normally just a question mark (?) is printed. If you specify many editor commands, the printing of the ? will not tell you where the error occurred.

here Document with ed

- Dash option to `ed` suppresses character counts
- Command `H` turns on `ed` help facility
- Appending << with dash in here document strips leading tabs (but not spaces) from input
- Example

file: `here1`

```
ed - document <<-!
 Ⓣ H
 Ⓣ g/bad/s//good/g
 Ⓣ w
 Ⓣ q
!
```

Quoting a here Document

A here document allows variable expansion and command substitution on the input lines, so some editing commands can cause unexpected results. Remember that "$" stands for the last line in a file for ed, but stands for variable substitution in the shell. Substitution can be suppressed by quoting the endmarker (e.g., preceding it with a backslash).

Quoting a here Document

- Variable expansion and command substitution allowed within a here document
- A backslash preceding the endmarker quotes the entire specification
- Example

file: `here2`

```
ed - file <<-\!
	H
	1,$s/bad/good/g
	w
	q
!
```

Summary

- Two special forms of redirection
  - command substitution
  - here documents

Command Substitution

- Command string enclosed by back quotes (grave accents) replaced by output of command
- Specification

file: back_quotes

```
echo The current date is `date`
echo "The command 'who | wc -l' tells us
how many people are on the system: `who | wc -l`"
```

- Execution

```
$ back_quotes
The current date is Thu Jul 6 19:36:39 EDT 1989
The command 'who | wc -l' tells us
how many people are on the system:      16
$
```

Summary

- Some characters have special meaning to shell
- Shell will interpret them literally if they are quoted
- Backslash for a single character
- Single quotes for a string of characters
- Double quotes for a string of characters
  - some characters retain special meaning

here DOCUMENT

The here document is used to supply a command with input from within a shell procedure. Normally, this input comes from the terminal and is ended with a control-d (EOF). The here document allows one of these commands to be fed information within the procedure. An example of a command that expects input from the terminal (or standard input) is the **cat(1)** command. If the **cat** command is invoked without a file name, it expects input from the standard input.

FORMAT

```
command << word<CR>
input<CR>
input<CR>
word<CR>
```

word is used to "frame" the input stream to the command. It is important that the format matches the above format with respect to the placement of the << and the **word**. The closing **word** must appear on a line by itself and must be the first and only word on the line.

A "–" in front of the initial "framing word" allows both input and the closing "framing word" to be preceded by tabs. This is useful for readability within shell scripts.

Note: Spaces are not allowed in front of the final framing word.

The framing word can be almost any string of characters, but is often the characters EOF or even a single !. If the initial "framing word" is preceded by a backslash (\) or surrounded by quotes, any variables in the input are not processed by the shell.

here DOCUMENT

- USED TO SUPPLY INPUT TO COMMAND FROM WITHIN A SHELL PROCEDURE

FORMAT

```
command <<word<CR>
input<CR>
input<CR>
word<CR>
```

word IS USED TO "FRAME" THE INPUT STREAM TO THE COMMAND

- '<<-word' allows input and 'framing word' to be indented with preceeding tabs.
- '<<\word' or '<<"word"' prevents the shell from performing substitutions.

here DOCUMENT — EXAMPLE 1

```
$ cat menu1<CR>
cat <<EOF
                         Inventory Control System

             update - Update Inventory Master Files

             report - Inventory On-Hand Report

            reorder - Inventory Reorder Report
EOF

$ menu1<CR>
                         Inventory Control System

             update - Update Inventory Master Files

             report - Inventory On-Hand Report

            reorder - Inventory Reorder Report
$
```

This program causes the **cat** command to take its input from within the shell program. This result could have been achieved by printing the contents of an existing file, but by using the here document, there needs to be only one file (the program file).

Compare this with echo

here DOCUMENT — EXAMPLE 1

```
$ cat menu<CR>
cat <<EOF
        Inventory Control System

 update - Update Inventory Master Files

 report - Inventory On-Hand Report

reorder - Inventory Reorder Report

EOF

$ menu<CR>
        Inventory Control System

 update - Update Inventory Master Files

 report - Inventory On-Hand Report

reorder - Inventory Reorder Report

$
```

here DOCUMENT — EXAMPLE 2

```
$ cat menu2<CR>
cat <<-EOF
                        Inventory Control System

            update - Update Inventory Master Files

            report - Inventory On-Hand Report

           reorder - Inventory Reorder Report

              list - List Files In ${HOME}

                              EOF

$ menu2<CR>
                        Inventory Control System

            update - Update Inventory Master Files

            report - Inventory On-Hand Report

           reorder - Inventory Reorder Report

              list - List Files In /staff/jhw
$
```

This is an example of a similar program that wants to display the value of the variable ${HOME}. When the shell encounters any variable, it substitutes the assigned value directly into the input stream. Also note that when the framing word is initially preceded with a dash, the framing word may be preceded with tabs at the bottom of the input stream.

here DOCUMENT — EXAMPLE 2

```
$ cat menu<CR>
cat <<-EOF
        Inventory Control System

 update - Update Inventory Master Files

 report - Inventory On-Hand Report

reorder - Inventory Reorder Report

   list - List Files In ${HOME}

                EOF

$ menu<CR>
        Inventory Control System

 update - Update Inventory Master Files

 report - Inventory On-Hand Report

reorder - Inventory Reorder Report

   list - List Files In /staff/jhw

$
```

Unit B10 — Problem-solving in the UNIX® Environment — Shell

Signals and Traps

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What events can cause premature program termination?
- When is a program vulnerable to such events?
- How can a program respond to their occurrence?

Signals

External events can affect running programs. The user might interrupt a program by pressing the *delete* key (for a foreground process), or by executing the `kill` command (for a background process). The user might log off the system causing termination of all background processes not protected with `nohup`. The kernel might interrupt a process because of a program or system error.

When such an event happens, a process receives a signal indicating the type of event. Normally, receipt of a signal will cause immediate termination. However, some signals can be ignored or caught. Catching a signal allows a program to take appropriate action before termination. Consider, for example, a program that creates a temporary file, uses it, and then removes it before terminating. An uncaught signal might terminate the program between the time the file is created and removed. The program can be written to catch specific signals and remove the file before terminating.

Shell programs can ignore or catch signals by using the `trap` statement.

Signals

- External condition can interrupt running program
  - user presses *delete* key
  - user executes `kill` command
  - user logs off or is disconnected
  - kernel encounters error
- Process receives signal
- Receipt of signal normally terminates process
- Program can
  - ignore signal
  - catch signal and take appropriate action

Signals (cont.)

There are many signals that a process can receive. Only signals 0, 1, 2, 3, and 15 should be trapped in a shell program. If signal 0 is trapped, then the trap routine will run when the program completes (no matter how it completes).

Signal 9 is the uncatchable signal sent with the `-9` option of the `kill` command. The signal cannot be trapped.

Signals (cont.)

- Catchable

0	normal termination (end of program or `exit`)
1	hangup (logged off or disconnected)
2	interrupt (*delete* key)
3	quit (similar to interrupt)
15	terminate (`kill` command)

- Uncatchable

9	kill (`kill -9`)

The trap Statement

The `trap` statement defines actions to be taken (commands to be executed) on receipt of a listed signal. When a listed signal is received, the commands are executed, and then the program resumes at the point where it was interrupted. However, an `exit` statement can be included

The commands are enclosed in quotes so they are passed to `trap` as a single argument. Single quotes delay variable expansion and command substitution until a signal is received. (You will see more of this later.) The enclosed commands can be separated by semicolons or placed on separate lines.

The trap Statement (cont.)

If the command list is explicitly null, then the listed signals are ignored by the process.

If no command list is given, then the action taken on receipt of a listed signal is the system default, namely termination. This is sometimes referred to as resetting the trap.

If the `trap` statement is given no arguments, it prints the command list for all trapped signals.

Quoting Traps

- Commands within trap are interpreted twice
  - once when read in
  - another time when executed
- Enclosing commands with single quotes prevents variable expansion and command substitution in the initial interpretation

The trap Statement (cont.)

- If *command_list* explicitly null, then listed signals ignored

```
trap '' 1 2 3 15    # signals ignored
```

- If no *command_list*, default action (termination) on receipt of signal

```
trap 1 2 3 15    # traps reset to system defaults
```

The trap Statement: Examples

In the example on the following page, a temporary file is created in `/tmp`, the system directory where programs are free to create temporary files. The special shell variable `$$` expands to the identification number of the process. Using `$$` as all or part of a temporary file name prevents two processes from trying to create the same file even if they are running the same program.

The trap Statement: Examples (cont.)

The program has been modified to trap any of four signals.

The trap Statement: Examples (cont.)

When a signal is received, the command list is executed. However, the sudden arrival of the same signal or another listed signal would interrupt the command list and cause its execution to recurse. Therefore, the traps should be reset in the command list to ignore additional signals.

The trap Statement: Examples (cont.)

The actions taken on catching a signal are the same for normal termination. Rather than duplicate statements, you can catch normal termination by including signal 0. The trap for normal termination should be reset to the system default so the `exit` statement doesn't spring the trap again.

Quoting Traps

When the shell initially reads a `trap` statement, it evaluates the entire statement and saves it in case the signal is received. If the signal is received, then the commands within the `trap` are reevaluated. A problem occurs when, for example, the command list includes the expansion of a variable assigned during the execution of the program. In that case, the first time the `trap` statement is read by the shell, the expansion takes place, replacing the expression with a null value.

To prevent premature expansion, enclose the command list in single quotes. Then the shell will not perform variable expansion or command substitution until the `trap` is invoked.

If an expression needs to be quoted within the command list, then double quotes or backslashes should be used.

Summary

- Catchable and uncatchable signals
- Setting and unsetting traps
- Catching or ignoring signals
- Nested traps

Shell Environment

Objectives

Upon completion of this unit, you will be able to answer the following questions:

- What does the shell language give us?
- What is its place in the UNIX® environment?
- How can we customize our environment?

What Do We Have Now?

We started with an extensive set of commands that share standard interfaces. The shell programming language has given us the ability to ''glue'' these primitives together and use them as building blocks for other commands, which in turn can be used as building blocks. As we extend our set of commands, we are customizing our programming environment. Therefore, the environment we work in is limited only by our imagination.

What Do We Have Now?

- A rich set of primitives (commands) with standard interfaces
- A high-level, interpretive, concurrent programming language
- Endless ability to encapsulate functionality in a command, such that it may in turn be encapsulated in another command
- A customizable programming environment

Customized Environment

Tools are placed in directories according to their function or usefulness to the community at large. Individuals can choose which collections of tools they want to use by setting their PATH variables.

ls -a to see .profile

Customized Environment

- Tools collected in standard places
  - standard commands in `/bin`, `/usr/bin`, and `/usr/lbin`
  - local tools in a local system directory
  - project tools in a project directory
  - personal commands in `$HOME/bin`
- `PATH` variable defines places to look and is user-definable

Customized Environment (cont.)

- Default environment provided at login
- Login shell process reads and interprets `/etc/profile`
  - shell program provided by system administrator
  - modifies environments for all users
- Login shell then reads and interprets `$HOME/.profile`
  - user-defined shell program
  - further modifies environment for user

Simple .profile

file: profile

```
MAIL=/usr/mail/$LOGNAME; export MAIL
```

The trap Statement

- General form

- If a listed signal is received
  - *command_list* is executed
  - program continues at point where signal was received unless *command_list* includes `exit`
- Example

Dot Statement

- Editing `$HOME/.profile`
  - does not change environment of current login session
  - will define environment of future sessions
- Execution of `$HOME/.profile`
  - occurs in child process
  - does not affect parent (login) shell process
- Dot (`.`) statement causes current shell process to interpret file given as argument

Dot Statement (cont.)

- Example

```
$ echo $TERM
2621
$ .profile
Terminal type? 605
Terminal type is 605
$ echo $TERM
2621
$ . .profile
Terminal type? 605
Terminal type is 605
$ echo $TERM
605
$
```

Lesson 1. Debugging Aids

EXPLICIT INVOCATION OF SHELL PROCEDURE

Shell programs are normally executed implicitly by creating a command file using an editor, adding execute permission using the **chmod** command, then executing the command file by using its name as the command name.

Another method of invocation is available for debugging shell programs — explicit invocation. This involves the execution of the **sh** command and supplying it with the file path name of the shell procedure to be executed. The **sh** command, like many other commands, if not supplied with a file name, takes its input from the standard input. If supplied with a file name, it takes its input from that file instead of the standard input.

FORMAT

```
$ sh file_name<CR>
```

The *file_name* must be a fully qualified path name, either absolute (starting with a slash) or a relative path name (relative to the current directory). The command lines are read from that file and executed in the same manner as if the procedure had been invoked implicitly.

Because a fully qualified path name is supplied, the **PATH** variable is not searched to find the file. The **PATH** variable is, however, searched to find the **sh** command (found in */bin/sh*). Because the file is not actually being executed, but rather read as input, the mode of the file does not have to be executable, but must be readable.

EXPLICIT INVOCATION OF SHELL PROCEDURES

FORMAT

```
$ sh file_name<CR>
```

- EXECUTES **sh** COMMAND WITH *file_name* AS INPUT
- *file_name* MUST BE FULLY QUALIFIED PATH NAME OR RELATIVE PATH NAME OF SHELL PROCEDURE
- IF *file_name* IS OMITTED, WILL READ FROM STANDARD INPUT
- **PATH** IS NOT SEARCHED, WILL OVERRIDE BUILT-IN SHELL STATEMENTS
- FILE MODE DOES NOT NEED TO BE EXECUTABLE, BUT MUST BE READABLE

VERBOSE TRACE

The verbose trace feature of the shell makes it possible to watch the reading of a shell program. The shell, after a command line is read in, prints the command line (as read in) on the standard error output, permitting the execution of the program to be traced.

If the shell procedure is being invoked explicitly , then give the **-v** option to the **sh** command as follows:

```
$ sh -v shell.procedure<CR>
```

This causes the verbose trace to be turned on for the duration of the execution of the shell program. It is also possible to turn on the verbose trace from within a shell procedure and then turn it off at some point. This is done using the **set** statement within the shell procedure as follows:

```
set -v
```

After the shell executes the **set** statement, the shell prints all lines as read until either the end of the program is reached or the statement

```
set +v
```

is encountered. This statement turns off the verbose trace, but shows up in the verbose trace (not executed until after the command line is printed).

The verbose trace option is not inherited by child shells. If the trace is turned on at the login shell level, it echoes only the commands as they are typed in at the terminal, not in any shell programs which that shell process invoked. The verbose trace is a shell option. When new shell processes are started, the options are not passed to the newly created shell. To trace a child shell, the child shell must be invoked explicitly (including the -v option) or must contain a **set -v** statement.

VERBOSE TRACE

FORMAT (ON COMMAND LINE)

```
$ sh -v shell.procedure<CR>
```

FORMAT (IN SHELL PROCEDURE)

```
set -v
```

- PRINTS COMMAND LINE AS READ
- BEFORE EXPANSIONS, SUBSTITUTIONS
- MAY BE TURNED OFF WITHIN A PROGRAM BY

```
set +v
```

- SHOWS COMMENTS
- NOT INHERITED BY CHILD PROCESSES

VERBOSE TRACE — EXAMPLE

The following page contains a sample session showing a traced shell procedure. Notice that the **set** command does not appear in the trace. This is because the trace was not turned on when that command line was read. The option was selected after the command line was executed, at a time which comes after the verbose trace print.

This program causes the command lines to print and then execute. Also note that the variable substitution expression was not expanded. In fact, no feature of the shell (parameter substitution, command substitution, metacharacter expansion, etc.) shows in its interpreted or expanded form. The command line prints exactly as read.

VERBOSE TRACE — EXAMPLE

```
$ cat traced<CR>
set -v
# This is a comment
date
echo ${TZ}
set +v          # Turn off verbose trace
date

$ traced<CR>
# This is a comment
date
Wed Apr 13 15:01:44 EST 1983
echo ${TZ}
EST5EDT
set +v          # Turn off verbose trace
Wed Apr 13 15:01:44 EST 1983

$
```

EXECUTION TRACE

The execution trace facility of the shell permits the execution of a shell program to be watched. It causes the shell to print each command line immediately before execution, after the shell has done all of its processing, including parameter and command substitution and metacharacter expansions. A shell program can use the execution trace option two ways. The first involves the use of an explicitly invoked shell procedure as follows:

```
$ sh -x shell.procedure<CR>
```

If invoked this way, the shell causes all the command lines in the file *shell.procedure* to print just prior to their execution. The second method is to trace only part of a shell procedure. This is done using the **set** statement within the procedure as follows:

```
set -x
```

This causes the execution trace option to be turned on. It remains on until either the program ends or the following command line is encountered:

```
set +x
```

The use of these two command lines within a shell program permits the tracing of only part of the program.

The shell prints the command lines on the standard error output and precedes those lines with a plus symbol (+). Execution trace and verbose trace may be combined by either invoking the command as follows:

```
sh -vx command.file
```

or by using the following command line within the shell program:

```
set -vx
```

Looping constructs and pipelines appear radically different in the two traces. These constructs are read only once, so the verbose trace prints them once only. Since the commands may be executed over and over (as with looping constructs) or the execution of the command may involve the separate execution of several commands (as with pipelines), the execution trace may appear to print the same command line more than once.

EXECUTION TRACE

FORMAT (ON COMMAND LINE)

```
$ sh -x shell.procedure<CR>
```

FORMAT (IN SHELL PROCEDURE)

```
set -x
```

- PRINTS COMMAND LINE AFTER SHELL HAS DONE ALL PROCESSING
- AFTER EXPANSIONS, SUBSTITUTIONS
- TRACED COMMANDS PRECEDED WITH "+ " (EXCEPT VARIABLE ASSIGNMENTS)
- MAY BE TURNED OFF WITHIN A SHELL PROGRAM BY

```
set +x
```

EXECUTION TRACE — EXAMPLE

The following is an example of the same program as before, but this time using the execution trace instead of the verbose trace. Note that the parameter expansion shows up in this trace where it does not in the verbose trace example.

```
$ cat traces<CR>
set -x
# This is a comment
date
f=unit
echo ${f}
set +x        # Turn off execution trace
date

$ traces<CR>
+ date
Thu Oct 20 10:06:39 CDT 1988
f=unit
+ echo unit
unit
+ set +x
Thu Oct 20 10:06:39 CDT 1988

$
```

The command line that turns on the execution trace does not show up in the trace, but the command that terminates the trace does. This is because the command is not executed until after the trace is printed.

EXECUTION TRACE — EXAMPLE

```
$ cat traces<CR>
set -x
# This is a comment
date
f=unit
echo ${f}
set +x              # Turn off execution trace
date

$ traces<CR>
+ date
Thu Oct 20 10:06:39 CDT 1988
f=unit
+ echo unit
unit
+ set +x
Thu Oct 20 10:06:39 CDT 1988

$
```

NONEXECUTABLE SHELL

In conjunction with the **-v** option, the nonexecutable option allows a program to be watched without the commands in that program being executed. The nonexecutable shell is usually used in conjunction with the verbose trace to see how a program would execute without actually executing it.

The nonexecutable shell may be invoked either from the command line or from within the shell program. If it is invoked from the command line, the format is as follows:

```
$ sh -n command.file<CR>
```

command.file is the name of the shell program. If this option is turned on from within a shell program, the following command line needs to appear in the shell program:

```
set -n
```

Although this option may be turned on within a shell program, it may NOT be turned off using a set +n. (The command would NOT be executed.) It is, therefore, advisable not to enable this option at the login shell level. It would disallow the execution of any commands thereafter.

NONEXECUTABLE SHELL

FORMAT (COMMAND LINE)

```
$ sh -n command.file<CR>
```

FORMAT (WITHIN SHELL PROCEDURE)

```
set -n
```

- READS COMMAND LINES BUT DOES NOT EXECUTE THEM
- DOES NOT MAKE VARIABLE SUBSTITUTIONS
- USED IN CONJUNCTION WITH VERBOSE TRACE TO WATCH PROGRAMS HARMLESSLY
- **set +n** NOT AVAILABLE
- SHOULD **NOT** BE GIVEN TO LOGIN SHELL
- WILL REPORT CONTROL CONSTRUCT SYNTAX ERRORS

NONEXECUTABLE SHELL — EXAMPLE

This example turns on the verbose trace and then the nonexecutable option. This combination causes the remainder of the program to print. Although this is available, it is often simpler to use the **cat** command to print the command file.

```
$ cat non.exec<CR>
set -v
set -n
variable=first
echo ${variable}
ls -C

$ non.exec<CR>
set -n
variable=first
echo ${variable}
ls -C

$
```

NONEXECUTABLE SHELL — EXAMPLE

```
$ cat non.exec<CR>
set -v
set -n
variable=first
echo ${variable}
ls -C

$ non.exec<CR>
set -n
variable=first
echo ${variable}
ls -C

$
```

UNSET VARIABLE EXIT

The unset variable exit feature of the shell permits a global check on all variables to be performed. The shell normally permits the use of an unset variable. In that case, the value of that expression is null; but if the unset variable exit option were selected, then an error message would be produced and the program would immediately exit.

The unset variable exit option may be invoked on the command line or from within a shell program. If turned on from the command line, the format is as follows:

```
$ sh -u command.file<CR>
```

This causes the file *command.file* to be interpreted, and if any unset variables are referenced, then the program immediately exits. This option is in effect for the duration of the execution of the program. The format for turning on the unset variable exit from within a shell program is as follows:

```
set -u
```

The unset variable exit remains in effect until either the end of file is reached or until the following command line is encountered:

```
set +u
```

UNSET VARIABLE EXIT

FORMAT (COMMAND LINE)

```
$ sh -u command.file<CR>
```

FORMAT (WITHIN SHELL PROCEDURE)

```
set -u
```

- CAUSES A NON-INTERACTIVE SHELL TO EXIT IMMEDIATELY IF UNSET VARIABLES ARE REFERENCED
- HELPFUL IN DEBUGGING — GLOBAL VARIABLE CHECK
- MAY BE TURNED OFF WITHIN A SHELL PROGRAM BY

```
set +u
```

UNSET VARIABLE EXIT — EXAMPLE

The following is a typical example of how the unset variable exit is used to find possible misspellings in variable names. In the fifth line of the program, the variable named **name** is misspelled **nmae**. When executed using the unset variable exit, the shell recognized this error and printed an error message on the standard error, then exited immediately.

```
$ cat unset.var<CR>
set -v
set -u
name=first
cat ${nmae}.file
date
wc -l ${name}.file

$ unset.var<CR>
set -u
name=first
cat ${nmae}.file
unset.var: nmae: parameter not set

$
```

UNSET VARIABLE EXIT — EXAMPLE

```
$ cat unset.var<CR>
set -v
set -u
name=first
cat ${nmae}.file
date
wc -l ${name}.file

$ unset.var<CR>
set -u
name=first
cat ${nmae}.file
unset.var: nmae: parameter not set

$
```

Automatic Variable Export

When the **-a** option is set, any newly created or modified variables are automatically exported to child processes. Programs that re-exported modified variables will still work under **-a**.

By default, this option is turned off (+a) so that it will not conflict with any programs written prior to System V, Release 2.0. As an added note, values modified by a child will not be carried back to the parent.

AUTOMATIC VARIABLE EXPORT

FORMAT (COMMAND LINE)

$ sh -a command.file<CR>

FORMAT (WITHIN A SHELL PROCEDURE)

set -a

- PERFORMS AUTOMATIC EXPORT OF SUBSEQUENTLY CREATED OR MODIFIED VARIABLES
- STILL *CANNOT* CARRY MODIFIED VALUES FROM CHILD TO PARENT
- OPTION IS NOT INHERITED BY CHILD PROCESSES
- MAY BE TURNED OFF BY USING THE FOLLOWING WITHIN A SHELL PROGRAM:

  set +a

AUTOMATIC VARIABLE EXPORT — EXAMPLE

In the example, the program **parent** establishes a variable and then changes its value. The program **child** just prints the value of the variable. Notice the first time the program is executed the value is not known by **child**.

Note that when the **-a** option is not set, the child is not notified of the value of **var**, but when **set -a** is added to the script, **child** is aware of the value of **var**.

AUTOMATIC VARIABLE EXPORT - EXAMPLE

```
$ cat parent
var=new
child
var=change
child

$ cat child
echo "Value of var in child is ${var}"

$ parent
Value of var in child is        no export yet
Value of var in child is


$ cat parent2  #modified
set -a
var=new
child
var=change
child

$ parent2
Value of var in child is new
Value of var in child is change
```

EXIT IMMEDIATE SHELL

The exit immediate feature of the shell permits a program to exit immediately if any command within the program exits with a nonzero status. The format for invoking the exit immediate shell option from the command line is as follows:

```
$ sh -e command.file<CR>
```

This causes the file *command.file* to be interpreted; and if any of the commands failed, the program immediately ends. If this option is selected from within a shell program, it is set as follows:

```
set -e
```

This turns on the exit immediate option that may be turned off by the command line

```
set +e
```

within the same shell program.

EXIT IMMEDIATE SHELL

FORMAT (COMMAND LINE)

```
$ sh -e command.file<CR>
```

FORMAT (WITHIN A SHELL PROCEDURE)

```
set -e
```

- EXITS IMMEDIATELY IF ANY COMMAND EXITS WITH A NONZERO STATUS
- MAY BE TURNED OFF BY USING THE FOLLOWING WITHIN A SHELL PROGRAM:

```
set +e
```

EXIT IMMEDIATE SHELL — EXAMPLE

The following is an example of the execution of a program that has enabled the exit immediate option. The execution trace is also turned on to illustrate what is happening.

```
$ cat xit.im<CR>
set -ex
date
cat not.there
date

$ xit.im<CR>
+ date
Mon Jun 13 10:03:28 EDT 1983
+ cat not.there
cat: cannot open not.there

$
```

In this example, the file *not.there* was not there, causing the **cat** command to exit with a nonzero exit status. Since the exit immediate option was selected, the program immediately ended at that point.

This is not the preferred method for checking for the proper execution of commands within a

EXIT IMMEDIATE SHELL — EXAMPLE

```
$ cat xit.im<CR>
set -ex
date
cat not.there
date

$ xit.im<CR>
+ date
Mon Jun 13 10:03:28 EDT 1983
+ cat not.there
cat: cannot open not.there

$
```