

UNIX Shell Programming

Featuring the KornShell

Instructor Guide
EY-G994E-LO-0995

September 8, 1995

April, 1995

Digital Equipment Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Digital or an authorized sublicensor.

© Digital Equipment Corporation 1995.
All Rights Reserved.
Printed in U.S.A.

Reproduction or duplication of this courseware in any form, in whole or in part, is prohibited without the prior written permission of Digital Equipment Corporation.

The following are trademarks of Digital Equipment Corporation: Alpha AXP, AXP, DEC, DEC C DECchip, DECnet, DECsystem, DECthreads, DECwrite, Digital, OpenVMS, Digital UNIX, ULTRIX, VAX, VAXcluster, VAXstation, VMS, VMScluster, the AXP logo and the Digital logo.

Open Software Foundation, OSF, and OSF/1 are trademarks of the Open Software Foundation, Inc.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company, Ltd.

All other trademarks and registered trademarks are the property of their respective holders.

This document was prepared using DECwrite version 2.0.

Contents

Preface	vii
Course Outline	ix

Chapter 1: Review

Standard Files	1-2
Redirection of Standard Output/Input	1-3
Redirection of Standard Error	1-4
Pipelines	1-5
Wild Cards	1-6
History of Past Commands	1-7
Editing Past Commands and Filename Completion	1-8
Shell Variables	1-9
Special KornShell Variables	1-10
Exporting Variables into Child Processes	1-11
Foreground and Background Processes	1-12
Input / Output in Background Processes	1-15
Basic <i>vi</i> Editing	1-16

Chapter 2: Day 1

Shell Command Execution	2-3
Executing Shell Scripts as an Argument to <i>ksh</i>	2-7
Executing Shell Scripts by Name	2-9
Positional Parameters (<i>\$1</i> , <i>\$2</i> , ...)	2-11
Default Parameter Value	2-13
Escaping Wild Cards and Variable Substitution	2-15
Passing Arguments to Shell Scripts	2-17
Input to Shell Script (<i>read</i>)	2-19
Testing Arguments	2-21
Integer Comparison Operators	2-23
String Comparison Operators	2-25
File Enquiry Operators	2-27
Logic Operators	2-29
Making Decisions (<i>if</i>)	2-31
Positive Iteration loops (<i>while</i>)	2-33
List Processing (<i>for</i>)	2-35
Debugging Shell Scripts - Execute Trace	2-37
Debugging Shell Scripts - Verbose Trace	2-39

Execute Trace Inside a Shell Script	2-41
Command Substitution	2-43
Shell Script Examples	2-45

Chapter 3: Day 2AM

Executing Commands without Forking (<i>dot, exec</i>)	3-3
Grouping Shell Commands	3-7
Command Execution Dependent on Previous Command (&&,)	3-11
Multi-way Branching (<i>case</i>)	3-13
Negative Iteration Loops (<i>until</i>)	3-15
Iteration Control Statements (<i>break, continue</i>)	3-17
KornShell Aliases	3-19
KornShell Functions	3-21
Debugging Functions	3-23
Exported Functions and Aliases	3-25
ENV Environment Variable	3-27
Difference between <i>\$ myscript</i> and <i>\$ ksh myscript</i>	3-29
Autoloading Functions	3-31
Redirecting Input/Output in Logic Statements	3-33
Redirecting Both Standard Output and Standard Error	3-35
Function and Shell Script Examples	3-37

Chapter 4: Day 2PM

List Processing (<i>select</i>)	4-3
UNIX Signals	4-5
Common Signals	4-7
Handling Signals in a Shell Script	4-11
<i>trap</i> Examples	4-13
Integer Arithmetic in the KornShell	4-17
Integer Arithmetic in the KornShell	4-19
Floating Point Arithmetic	4-21
Variable Arrays	4-23
Assigning Values to an Array	4-25
Loading an Array from a Disk File	4-27
Changing IFS (Internal Field Separator)	4-29
Order of Evaluation (<i>eval</i>)	4-31
Tools for Shell Scripts (<i>sed, od, cut</i>)	4-35
Setting and Changing Positional Parameters	4-39
Function and Shell Script Examples	4-43

Chapter 5: Day 3

Here Documents	5-3
Reading/Writing Files from a Shell Script	5-9
Command Options Processing (<i>getopts</i>)	5-13
Pattern Matching Operators	5-21
Using a Lock File to Synchronize Access	5-25
Testing for Superuser	5-27
Setuid Bit for Shell Scripts	5-29
KornShell Command Line Parsing	5-33
Co-processes	5-35
Using Multiple Co-processes	5-37

Readability and Maintainability	5-41
Performance	5-45
Shell Script Examples	5-47

Chapter 6: Exercises

Day 1 Lab Exercises	6-1
Day 2AM Lab Exercises	6-4
Day 2PM Lab Exercises	6-6
Day 3 Lab Exercises	6-8

Appendix A: Common Symbols

List of Symbols	A-2
-----------------------	-----

Appendix B: Review Tutorial

Introduction	B-3
Redirection of Standard Output/Input	B-7
Redirection of Standard Error	B-9
Pipelines	B-11
Wild Cards	B-13
History of Past Commands	B-15
Editing Past Commands and Filename Completion	B-17
Shell Variables	B-19
Special KornShell Variables	B-21
Exporting Variables into Child Processes	B-23
Foreground and Background Processes	B-25
Input / Output in Background Processes	B-31
Basic <i>vi</i> Editing	B-33

Preface

Files associated with these course materials can be copied from:
TEACH::SYSS\$PUBLIC:[K\$HELL]

ksh.ps	Postscript Batch file for the course workbook
ksh_instr.ps	Instructor Guide (workbook with facing instructor pages)
kshell.tar	tar file containing: shell script examples of this workbook solutions to the lab exercises to be loaded into /class/kshell and /class/kshell/soln

It is expected that the student will have attended a UNIX Utilities & Commands course or have the equivalent knowledge. The first part of Day 1 is a review of the most important Korn shell topics of that course. Text discussion of the Review chapter is provided in the Appendix. It is intended that the Review chapter be discussed briefly (30-45 minutes) to serve as a warmup for those with the appropriate knowledge. Students lacking in the proper prerequisites can read the Appendix later to catch up.

This workbook is structured as a teaching document. The pages are intended to be lectured sequentially; they are not grouped into categories that would facilitate later reference. The lack of logical structure was a deliberate design decision. It was desired to cover topics lightly on Day 1 so as to get the students programming ASAP. The topics covered lightly on Day 1 are discussed again in more depth on successive days.

Many example shell scripts and fragments of shell scripts are used to introduce and to illustrate programming concepts. Many topics, e.g. option processing, are discussed solely using examples.

The course has 4 lecture periods and 4 lab periods :

Day 1, Day 2AM, Day 2PM, Day 3

Solutions for the lab exercises and on-line versions of the lecture examples are provided (see above).

Day 1 contains only one lecture/lab period in order to introduce sufficient programming topics so that meaningful lab exercises can be attempted. Most of the Day 1 topics were also covered at the end of the U&C course. Day 1 is intended to lay a basic shell programming foundation for the heavier topics to come. Day 2 is split into two lecture/lab periods - morning and afternoon. The shell script examples are very important, perhaps more important than the lab exercises, to communicate practical skills. If time is short, one might sacrifice lab time rather than lecture time. Day 3 has only one lecture/lab period. Many students desire to leave early on the last day so all the Day 3 topics were grouped.

The examples and lab solutions were written on Digital UNIX. With the exception of some parts of option processing (leading colon on p. 5-8), no differences were found between the Korn shell on ULTRIX and Digital UNIX. The examples and lab solutions were tested on ULTRIX.

Course Outline

Day 1:

- Review of Basic Concepts
- Shell Command Execution
- Shell Scripts
- Arguments to Scripts
- Conditions and Control Statements I
- Debugging Scripts
- Command Substitution
- Lab Exercises

Day 2, Morning:

- Alternative Ways to Execute Commands
- Control Statements II
- KornShell Aliases
- KornShell Functions
- Invoking and Executing Scripts
- Lab Exercises

Day 2, Afternoon:

- Menu Processing
- Signal Handling
- Arithmetic Operations
- Variable Arrays
- Command Evaluation
- Tools for Shell Scripts
- Setting Positional Parameters
- Lab Exercises

On ULTRIX the Korn shell is contained in /usr/bin/ksh. On Digital UNIX the Korn shell has 2 pathnames: /bin/ksh and /usr/bin/ksh. It is suggested that /usr/bin/ksh be used when referring to the Korn shell file.

Throughout this workbook, bold typeface is used to represent the characters typed by the user. Normal typeface represents the characters printed by the shell and utilities.

UNIX shells

Bourne shell	original shell from 1970s written at Bell Labs by Stephen Bourne no job control
C shell	enhanced shell from 1980s written at Univ of California at Berkeley has job control no command line editing (but vendors added it in their versions, but works differently in different vendor versions) Uses C language like syntax in shell scripts
Korn shell	enhanced shell from 1990s written at AT&T by David Korn upwards compatible with Bourne shell (all Bourne shell syntax and commands work in Korn shell) has job control and cmd line editing

Frequently the Korn shell, while supporting the older Bourne shell syntax, originated a better way. That better way is used in this workbook.

This course deals only with the Korn shell. No attempt is made to discuss the C shell. Some reference is made to the Bourne shell syntax that works in the Korn shell, but no attempt is made to teach Bourne shell programming.

Day 3:

- Here Documents
- File Input and Output Operations
- Command Options Processing
- Pattern Matching Operators
- Co-Processes
- KornShell Command Processing
- Style and Performance
- Lab Exercises

Chapter 1: Review

- Redirection of Standard Files
- Pipelines
- Wildcard Metacharacters
- Command History and Command Line Editing
- Shell Variables
- Foreground and Background Processes
- Basic vi Editing

Standard Files

Standard input:

The file from which shell accepts commands

Standard output:

The file to which shell sends prompt character (\$)

The file to which most utilities write their output
who, date, ls, ...

Standard error (diagnostic output):

The file to which shell writes error messages

Status of the standard files at login:

standard input	keyboard
standard output	terminal
standard error	terminal

All input/output operations are conducted through files

/class/user/myfile

/dev/tty03

/dev/rmt0h

Redirection of Standard Output/Input

To redirect output to a new file:

```
$ who
user tty02 Dec 5 11:30
kjd tty04 Dec 5 12:53
sam tty00 Dec 5 13:03

$ who > whofile

$ cat whofile
user tty02 Dec 5 11:30
kjd tty04 Dec 5 12:53
sam tty00 Dec 5 13:03
```

To redirect output to an existing file:

```
$ set -o noclobber
$ date > whofile
whofile: file already exists

$ date >| whofile
$ cat whofile
Thu Feb 1 13:38:36 EST 1990

$ set +o noclobber          # clear noclobber
$ date > whofile
$ cat whofile
Thu Feb 1 14:22:17 EST 1990
```

To append redirected output to an existing output file:

```
$ ls >> file
```

To redirect input:

```
$ mail john < message
```

Redirection of Standard Error

To redirect standard error:

```
$ chmod 000 file1
$ cat file1
file1: Permission denied
```

```
$ cat file1 2> problem
$ cat problem
file1: Permission denied
```

To redirect standard error and standard output:

```
$ shell_command > myout 2> myerr
$ shell_command > myout 2>&1
```

To eliminate any output:

```
$ shell_command > /dev/null 2>&1
```

/dev/null is a special file.

It is the UNIX data sink (bit bucket).

When read, it yields EOF.

When written, it provides an infinite sink.

Pipelines

```
$ who
sam  tty10 Jun 30 10:54
mary tty03 Jun 30 08:13
mgr  tty12 Jun 28 17:34

$ who | sort
mary tty03 Jun 30 08:13
mgr  tty12 Jun 28 17:34
sam  tty10 Jun 30 10:54

$ who | wc -l
3

$ who | sort > people

$ cat people
mary tty03 Jun 30 08:13
mgr  tty12 Jun 28 17:34
sam  tty10 Jun 30 10:54
```

To list all files in `/dev` which are block structured devices:

```
$ ls -l /dev | grep '^b'
brw----- 1 root system 8, 3072 Apr 13 10:19 rz3a
brw----- 1 root system 8, 3073 Apr 13 10:19 rz3b
brw----- 1 root system 8, 3074 Apr 13 10:19 rz3c
brw----- 1 root system 8, 3075 Apr 13 10:19 rz3d
```

To list on the printer all files in `/dev` which are block structured devices:

```
$ ls -l /dev | grep '^b' | lpr
```

Wild Cards

*	Matches any string of characters, including a null string
?	Matches exactly ONE of any character
[chars]	Matches exactly ONE of any character among those included between the brackets

```
$ ls -a
.  .login  get.c  put.c  x  x2  xfour  z4
.. a      msg.c  put.o  x1  x3  y4sale  zed
```

```
$ ls x?
x1      x2      x3
```

```
$ ls x*
x      x1      x2      x3      xfour
```

```
$ ls x??
x?? not found
```

```
$ ls [xyz]*
x      x2      xfour  z4
x1     x3      y4sale zed
```

```
$ ls [x-z]4*
y4sale z4
```

```
$ ls [!a-y]*
z4      zed
```

~	Matches user's home directory
~sam	Matches the home directory for user sam

```
$ ls -l ~/srcfile
-rw-r--r-- 1 mary      362 Nov 3 12:13 srcfile

$ cp myfile ~
```

History of Past Commands

```
$ history
9 cat letter.mail
10 date
11 rm mmdc.c
12 ls -l /class/user/save
13 chmod a-x mbox
14 history
```

```
$ r 10
date
Fri Feb 2 12:18:17 EST 1990
```

```
$ r -4
ls -l /class/user/save
-rwxr-xr-- 1 user          580 Jul 5 11:38 save
```

```
$ r cat
cat letter.mail
Don't forget lunch on Friday.
Susan
```

```
$ r ch
chmod a-x mbox
```

Include command number in prompt string:

```
$ PS1='!$ '
18$
```

Editing Past Commands and Filename Completion

```
$ set -o vi
$ <esc>k          #retrieve latest command
$ set -o vi      #cursor at beginning of line
k                #retrieve next latest command
$ chmod a-x mbox #cursor at beginning of line
^C              #terminate history scan
```

vi commands:

h j k l	Move cursor (left, down, up, right)
x	Delete character
istring<esc>	Insert <i>string</i>
Rstring<esc>	Overwrite with <i>string</i>

Filename Completion:

```
$ ls
comm edit files make machine shell

$ lpr ed<esc>\
lpr edit

$ chmod 755 m<esc>\
chmod 755 ma
chmod 755 mak<esc>\
chmod 755 make

$ grep sam /etc/pas<esc>\
grep sam /etc/passwd
sam:A34fjU76Fcv5g:203:0:Samuel Adams:/usr/users/sam:/bin/ksh
```

Shell Variables

```
$ VAR1=Brown  
$ VAR2=stone  
$ CONCAT=$VAR1$VAR2
```

```
$ print "$VAR1"  
Brown
```

```
$ print "$VAR2"  
stone
```

```
$ print "$CONCAT"  
Brownstone
```

KornShell variables are stored as ASCII strings.

Variable names **MUST** begin with a letter or underscore, but may contain digits, letters or underscores.

Special KornShell Variables

HOME	Home directory: default argument for cd
PS1	Primary prompt string: default = "\$ "
PS2	Secondary prompt string: default = "> "
HISTSIZE	Size of the history list: default = 128
HISTFILE	File in which history is saved: default = \$HOME/.sh_history
PWD	Present working directory set by cd command
OLDPWD	Previous working directory set by cd command
IFS	Internal field separator: default = whitespace
MAIL	Users mail file: if set, user is informed by shell of arrival of mail every \$MAILCHECK seconds
TERM	Terminal type
PPID	Parent process ID
TMOUT	A number of seconds: if a command is not issued to the shell in this time period, the shell exits
TMOUT=0	Inhibits this automatic logoff feature
RANDOM	A random integer from 1 to 32767
PATH	A list of directories to be searched for a command name

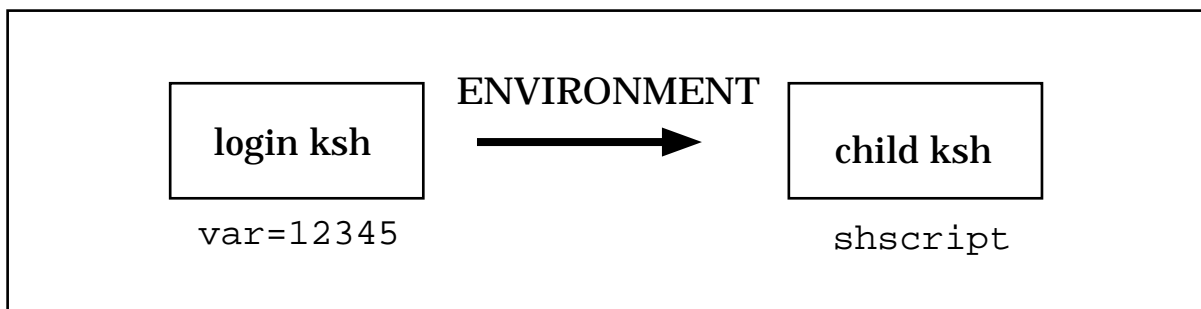
Exporting Variables into Child Processes

```
$ cat shscript
print $var

$ var=12345
$ export var
$ shscript
12345
$

$ var=12345
$ shscript
$
```

Export the value of the variable into all subsequently executed commands or shells.



```
$ var=abcde
$ shscript
abcde
```

If an exported variable is modified, the new value is exported.

```
$ export var=12345  #define & export
$ export            #list environment
ENV=~/.environ
HOME=/usr/users/sam
LOGNAME=sam
MAIL=/usr/mail/spool/sam
PATH=/usr/users/sam/bin:/bin:/usr/bin/:/etc:.
TERM=vt100
TZ=EST05EDT
var=12345
```

Foreground and Background Processes

```
$ ps x
```

```
PID STAT TT TIME CMD
2680 S    01 0:00 - (ksh)
3204 R    01 0:00 ps x
```

```
$ grep '^B' file > report &
[1] 4750
```

```
$ ps x
```

```
PID STAT TT TIME CMD
2680 S    01 0:00 - (ksh)
4751 R    01 0:05 ps x
4750 R    01 0:01 grep ^B
```

```
$ jobs
```

```
[1] + Running grep ^B file>report
```

```
[1] Done grep ^B file > report
```

```
$ grep '^B' file | lpr &
```

```
[1] 4791
```

```
$ ps x
```

```
PID STAT TT TIME CMD
2680 S    01 0:00 - (ksh)
4792 R    01 0:07 ps x
4790 R    01 0:07 grep ^B
4791 R    01 0:01 lpr
```

```
$ jobs
```

```
[1] + Running grep ^B file | lpr
```

To obtain a long listing of process status:

```
$ grep '^B' file > report &
[1] 1706
```

```
$ ps lx
```

```
F      UID PID   PPID  PRI  NI   STAT  TT  TIME  COMMAND
10808001 10 1704     1   15   0    S    01  0:00  - (ksh)
10008101 10 1706   1704   46   0    R    01  0:00  grep '^B'
10008101 10 1708   1704   43   0    R    01  0:00  ps lx
```

```
$ jobs -l
```

```
[1] + 1706 Running      grep ^B file > report
```


To move a foreground job to the background:

```
$ sort abc.dat | lpr
^Z
Stopped

$ jobs
[1] + Stopped sort abc.dat | lpr

$ bg %1
[1] sort abc.dat | lpr &

$ grep '^B' file > report &
[2] 264

$ jobs
[2] + Running grep ^B file > report
[1] - Running sort abc.dat | lpr
```

+ identifies the current job. This job will be the default job for commands like **bg**, etc.

- identifies the previous job. This job may have been the current job at one time, and will be the current job when the current job finishes.

To move a background job into the foreground:

```
$ fg %2
lpr report
$
```

To wait for a background process to complete:

```
$ wait %1
ls
$ a.tmp cat.tmp file klm.tmp rpt.c xyzv.tmp
```

To terminate a background job:

```
$ kill %1
```

To stop a background job:

```
$ stop %2
```

Job references for the **fg**, **bg**, **wait**, and **kill** commands:

%n	Job number
%str	Job whose command begins with <i>str</i>
%?str	Job whose command contains <i>str</i>
%+	Current job
%%	
%-	Previous job

Input / Output in Background Processes

Standard Input:

```
$ ed bigfile
120000
1,$s/this/that/
^Z
Stopped

$ bg
[1] ed bigfile &
$
...some foreground commands...
```

```
[1] + Stopped (tty input) ed bigfile

$ fg
ed bigfile
w
120000
q
$
```

Standard Output:

```
$ ls -l /class/user/save &
[1] 372
$ -rwxr-xr-- 1 user          580 Jul 5 11:38 save

$ stty tostop

$ ls -l /class/user/save &
[1] 373
$ <return>
[1] + Stopped (tty output) ls -l /class/user/save

$ fg
ls -l /class/user/save
-rwxr-xr-- 1 user          580 Jul 5 11:38 save
$
```

Basic *vi* Editing

`$ vi textfile`

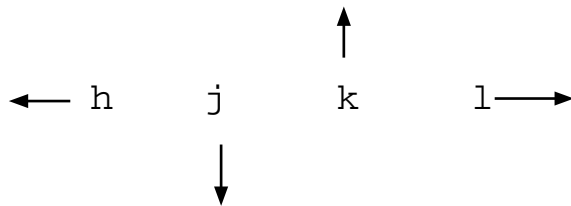
The UNIX operating system developed from the Computing Science Research Group at Bell Laboratories in Jersey. It is said that the creators of the UNIX system has this objective in mind:

to create a computing environment
where they themselves
could comfortably and effectively pursue their own work
programming research.

Until 1980, the UNIX system was mostly confined to
an environment consisting of university computer science
departments and research and development organizations.

~
~
~
~

"textfile" 13 lines, 504 characters



x

dd

iNew <ESC>

aNew <ESC>

J

:r newfile

:wq

:q!

Chapter 2: Day 1

2-2

A process is the kernel environment in which a user program executes.

The kernel schedules process to execute, not programs.

Each program runs in a process; the process is a program wrapper.

Process is to program as nest is to egg.

fork() creates a new process that runs the same program as the parent. (new nest with same color egg)

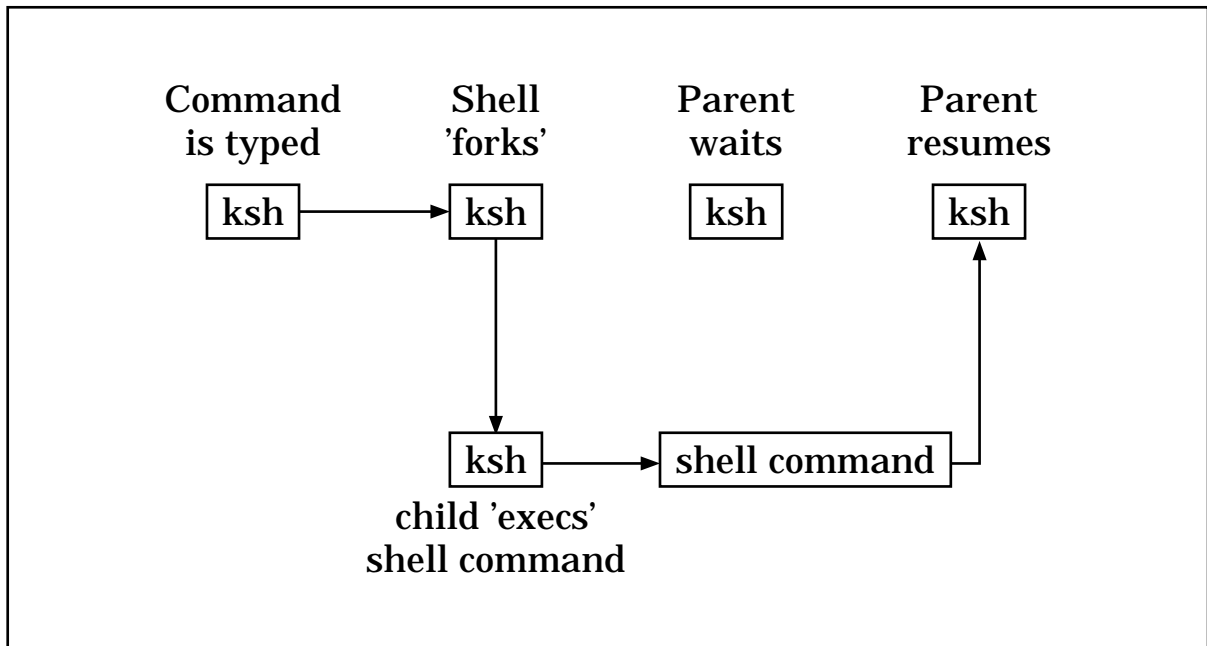
exec() replaces the program in the process with a new program. (replace the egg in the nest)

For man page help on built-in commands, e.g. fg:

```
$ man ksh
```

and search for fg with /fg and n.

Shell Command Execution



Foreground Job

Shell waits for child process to terminate.

Background Job

Shell does not wait for child process to terminate.

Type-ahead Buffer

Characters typed while the shell waits for the foreground job to terminate are buffered in a kernel buffer.

When foreground job completes, the shell reads the type-ahead buffer, echoes the characters to the terminal and executes the command.

Shell Built-Ins

Commands that are executed by the ksh process, e.g. **cd**, **fg**.

A child process and an external program are NOT needed.

Documented under **ksh(1)**.

VERY IMPORTANT PAGE

The ksh that first executes in child process performs redirection and expands wild cards before exec'ing new program.

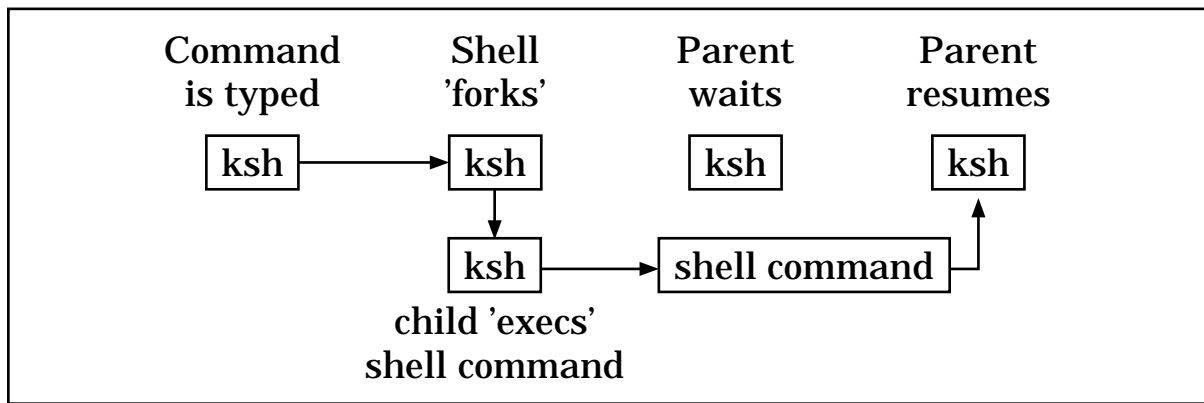
Understanding this fact is necessary to understand the reason for quotes and backslashes in shell commands.

A well behaved UNIX utility (filter) will:

1. read from stdin and write to stdout (can be in pipe)
2. use input file given on cmd line. If not provided, use stdin.

If ksh is unable to expand a wildcard, the wildcard is left in the command line.

If csh is unable to expand a wildcard, an error results and the utility is never exec'ed.



The shell program in the child process performs the following prior to *exec()*:

1. Redirection of standard files
2. Expansion of metacharacters

`$ ls x* > abc` becomes
`ls x1 xabc xyz` to process with standard output = `./abc`

Input file for many shell commands:

1. file name specified on command line, e.g.
`$ sort abc.dat`
2. standard input if no filename on command line, e.g.
`$ who | sort`

`$ sort < abc.dat`

Command passed to sort program: **sort**
 Standard input for sort process: **abc.dat**
 Sort uses standard input (**abc.dat**) as input file.

`$ sort abc.dat`

Command passed to sort program: **sort abc.dat**
 Standard input for sort process: **terminal**
 Sort uses filename on cmd line (**abc.dat**) as input file.

2-4

'print' writes its arguments to standard output.

When an 'ls' command is typed,
 'ls' is the command name and
 the 'ls' program is exec'ed.

When executing a shell script as an argument to ksh,
 'ksh' is the command name and
 the ksh program is exec'ed.

In box:

 cd executed by child ksh (built-in command).

With the 'csh' command at the bottom,
 'csh' is the command name and
 the 'csh' program is exec'ed.

Profile scripts execute in login ksh, not child ksh's.

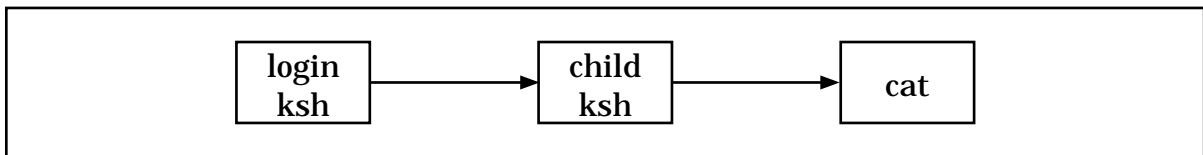
Avoid a discussion of the ENV file at this point.
The topic will be discussed in detail on Day 2.

Executing Shell Scripts as an Argument to *ksh*

```
$ cat phlist
print "name  extension"
print "-----"
cat phone

$ ls -l phlist
-rw-rw-r-- 1 sam  staff          77 Jun 30 15:15 phlist

$ ksh phlist
name  extension
-----
Mary Smith      X4335
Jeff Genua      X116
Albert Julia    X236
Bryan Byrd      X4441
Robert Porter   X117
Tony Lombard    X4211
Thomas Byrd     X379
Arthur Cathey   X394
```



ksh program: 1. runs in child process
 2. reads and executes commands in shell script.

Command name is **ksh**.

User must have read permission to shell script file.

Consider: \$ csh cshell_script

/etc/profile followed by **~/.profile** execute at login in login ksh.

2-5

Ignore at this point the subtle difference between executing a shell script by name and as an argument to ksh. That topic will be discussed on Day 2.

The only difference between executing a shell script by name and as an argument to ksh is the permission needed to the shell script file.

The C shell script at the bottom will be executed by a child ksh if the first line of the C shell script is not

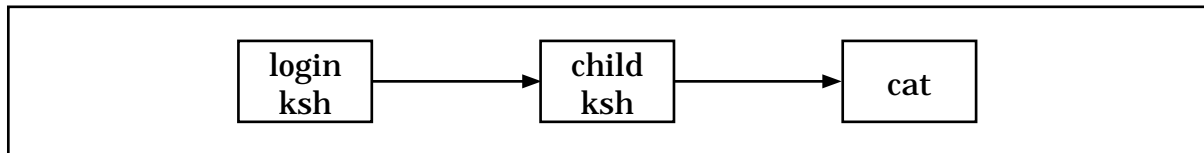
```
#!/bin/csh
```

PRUDENT PRACTICE: The first line of each Korn shell script should be:

```
#!/usr/bin/ksh
```

Executing Shell Scripts by Name

```
$ chmod 775 phlist
$ ls -l phlist
-rwxrwxr-x 1 sam  staff      77 Jun 30 15:15 phlist
$ phlist
name  extension
-----
Mary Smith    X4335
Jeff Genua    X116
Albert Julia  X236
Bryan Byrd    X4441
Robert Porter X117
Tony Lombard  X4211
Thomas Byrd   X379
Arthur Cathey X394
```



exec() system call for **phlist** causes **ksh** to run in child process.

ksh program reads and executes commands in shell script.

Command name is **phlist**.

User must have read and execute permission to shell script file.

Consider: \$ **cshell_script**

If first line in shell script is: **#!/bin/csh**

csh executed in child process by *exec()* system call.

2-6

Suggest that you label \$1, \$2, etc. on the overhead and change labels as 'shift' is executed.

There is NO difference in positional parameters between executing a shell script by name and as an argument to ksh.

The only difference between executing a shell script by name and as an argument to ksh is the permission needed to the shell script file. (Ignore the subtle difference until tomorrow).

Change myscript to
shift 2
and discuss again.

What is value of \$4, \$5, etc?

NULL

A list of symbols is provided at the end of the workbook and in Bolsky p. 172.

Positional Parameters (*\$1, \$2, ...*)

\$	shell_script	arg1	arg2	...	arg9	arg10	arg11	...
	↑	↑	↑		↑	↑	↑	
	\$0	\$1	\$2		\$9	\${10}	\${11}	

\$# Number of positional parameters

\$* All positional parameters

```
$ cat myscript
print $# parameters - $* - in $0
print $2
shift
print $2
```

```
$ myscript abc xyz 1234
3 parameters - abc xyz 1234 - in myscript
xyz
1234
```

```
$ ksh myscript abc xyz 1234
3 parameters - abc xyz 1234 - in myscript
xyz
1234
```

shift [n] Shift the parameter list *n* positions to the right.
n is positive integer (default = 1).

2-7

In example #1 neither \$1 or \$2 is set.
In example #2 \$1 is set and \$2 is not set.
In example #3 both \$1 or \$2 are set.

There other forms that are not discussed in this course.

`${var:=value}` If var is not set, set it to value.

```
$ unset first
$ x=${first:=xyz}
$ echo $x $first
xyz xyz
```

`${var:+value}` If var is set, substitute value, otherwise
substitute nothing.
Positional parameters may not be assigned
in this way.

```
$ long=yes
$ ls ${long:+-l}
```

`${var:?string}`

If parameter is set, use its value, otherwise
print string and exit from shell.
`fname=${2:? "proc requires 2 arguments"}`

These and other forms are listed in Bolsky p. 172.

Default Parameter Value

```
$ cat myscript
print ${1:-abc}
print ${2:-mouse}
```

Provide a default value for a parameter to be used if the parameter is not set.

```
$ myscript
abc
mouse
```

```
$ myscript xyz
xyz
mouse
```

```
$ myscript dog cat
dog
cat
```

IMPORTANT PAGE:

It makes sense out of what is commonly considered the perverse nature of shell commands.

Discuss variable substitution inside ' " and \ first (left column).

Then discuss wild card substitution in right column.

`print *` Shell expands * to give list of all files in the current directory.

Four cases variables inside single quotes
 variables inside double quotes
 wild cards inside single quotes
 wild cards inside double quotes

ONLY one case works: variables inside double quotes

PS1= at bottom is commonly placed in .profile.

We must use single quotes to avoid evaluating PWD when PS1 is defined - to make PS1 value contain \$PWD instead of current working directory name.

DISCUSS difference between: `print abc`
 `print "abc"`
 `print 'abc'`

No difference, BUT if there are variables or wildcards in the string, there is a difference AND

```
print  a      b      -->  a b
print "a      b"    -->  a      b
```

Escaping Wild Cards and Variable Substitution

Variables

```
$ var=abcde
```

```
$ print $var  
abcde
```

```
$ print '$var'  
$var
```

```
$ print "$var"  
abcde
```

```
$ print \$var  
$var
```

Wild Cards

```
$ ls  
edit make nomake shell
```

```
$ print *  
edit make nomake shell
```

```
$ print '*'  
*
```

```
$ print "*"
*
```

```
$ print \*  
*
```

" "

Gather several fields into a single field
Wild card substitution will NOT occur

' '

Like " ", except that both variable and wild card
substitution will NOT occur

\

Escape the following character

```
$ PS1='$PWD - '  
/usr/users/sam -
```

Display the current working directory in the shell prompt.
Single quotes are essential.

2-9

Three ways to get arguments into a shell script.

Environment variables are commonly used for values that do not vary from command to command,
e.g. LANG used to represent the language of the user.

More about the read command is presented on the next page.

print special characters:

<code>\c</code>	no new line after print (prompting output)
<code>\n</code>	new line
<code>\t</code>	tab

Passing Arguments to Shell Scripts

Before script executes - environment variables

```
$ export ANIMAL=dog
```

On the command line - positional parameter

```
$ myscript mouse
```

While script executes - read command

```
read animal
```

```
$ cat myscript
```

```
print "What animal: \c"
```

```
read animal
```

```
print environment animal = $ANIMAL
```

```
print cmd line animal = $1
```

```
print read animal = $animal
```

```
$ export ANIMAL=dog
```

```
$ myscript mouse
```

```
What animal: lion
```

```
environment animal = dog
```

```
cmd line animal = mouse
```

```
read animal = lion
```

2-10

In example #3 `arg2` is NULL string

Any undefined variable has the NULL string as its value.

<code>read var?prompt</code>	provides no way to give one prompt and assign characters to more than one variable.
------------------------------	---

Why use double quotes with 'read' in the example at the bottom?
So that trailing space is included in prompt string.

Input to Shell Script (*read*)

```
read var1 var2 var3 ...
```

One line of characters are read from standard input.
Successive words of the input are assigned to the variables
named on the **read** command line (in order).
Leftover words are assigned to the last variable.

```
$ cat myscript
read arg1 arg2
print arg1 = $arg1
print arg2 = $arg2

$ myscript
sam tony
arg1 = sam
arg2 = tony

$ myscript
mary betty harry
arg1 = mary
arg2 = betty harry

$ myscript
george
arg1 = george
arg2 =
```

```
read var?prompt
```

prompt is written to standard error.
Line from standard input assigned to **var**.

```
$ cat myscript
read animal?"What animal: "
print read animal = $animal
print cmd line animal = ${1:-horse}

$ myscript
What animal: tiger
read animal = tiger
cmd line animal = horse
```

2-11

It is critical that this page be presented correctly to avoid certain confusion over 0 and 1 with respect to true and false.

Avoid the temptation to state that:

0 is true and 1 is false.

The problem here is the fact that 0 represents false in many programming languages. 0=true is counter intuitive for many people.

The Bourne and Korn shells consider 0 to be true.

The C shell consider nonzero to be true.

The Korn and Borsky book gets all wrapped around the axle over this issue. Avoid the morass.

The test program returns a status of 0 or nonzero.

The if statement tests the return status from the program that is its argument.

Future examples will use `[[]]` syntax exclusively.

It is an improvement over test and `[`.

It avoids some weird syntax that is needed with test and `[`.

`[[` is a shell built-in, not a separate utility.

test and `[` are separate utilities in `/bin`.

Another reason to prefer `[[` over test and `[`.

Examples at bottom:

Left Since `XX=17`, `[$XX]` has value of `17` and shell looks for command by that name.

Right 3 arguments to `[[` command are: `$xx -eq 17`]. A trailing `]]` must be seen to terminate the args to `[[`, so shell prompts for the remainder of the command.

Testing Arguments

```
$ XX=17
$ test $XX -eq 17
$ print $?
0

$ XX=17
$ test $XX -eq 23
$ print $?
1
```

The **test** command returns success status if the condition being tested is true, failure status if the condition is false.

\$? - status from most recent command

zero	SUCCESS
nonzero	FAILURE

```
$ XX=17
$ [[ $XX -eq 17 ]]
$ print $?
0

$ XX=17
$ test $XX -eq 17
$ print $?
0

$ XX=17
$ [ $XX -eq 17 ]
$ print $?
0
```

The KornShell provides three techniques for testing arguments:

[[]] command - new with KornShell (preferred)

test command and **[]** command - Bourne shell & KornShell

```
if [[ $XX -eq 17 ]]
then
...
fi
```

Spaces about **[[** and **]]** are required:

```
$ [[ $XX -eq 17 ]]
[[17: not found

$ [[ $XX -eq 17]]
> (PS2 - secondary prompt)
```

2-12

The 'exit' command is a shell built-in. Its argument is exit status.

0	success
nonzero	not success

Shell variables are stored as character strings. To compare integer values, [[command must convert from string to integer.

There is no floating point capability in the Korn shell, however a hack is provided on Day 2.

List of symbols, like \$# and \$?, at the end of the workbook and in Bolsky p. 175.

Integer Comparison Operators

```
$ cat myscript
if [[ $# -eq 0 ]]
then
    print "Usage: $0 filename"
    exit 2
fi
...
$ myscript
Usage: myscript filename
$ print $?
2
```

<code>n1 -eq n2</code>	success if integers <i>n1</i> and <i>n2</i> are equal
<code>n1 -ne n2</code>	success if integers <i>n1</i> and <i>n2</i> are not equal
<code>n1 -gt n2</code>	success if integer <i>n1</i> is greater than integer <i>n2</i>
<code>n1 -ge n2</code>	success if integer <i>n1</i> is greater than or equal to integer <i>n2</i>
<code>n1 -lt n2</code>	success if integer <i>n1</i> is less than integer <i>n2</i>
<code>n1 -le n2</code>	success if integer <i>n1</i> is less than or equal to integer <i>n2</i>

The `[[]]` command will convert strings to integers and then make the comparison based on integer values

2-13

An undefined variable has a NULL string value.

`[[-n $1]]` tells whether \$1 is defined.

0123 is NOT an octal number - radix for numbers is decimal, but more about that tomorrow.

For `s1 < s2` at bottom:

Is `a < B` ?

NO `a=97, B=66`

`$ man ascii` to get list of ascii codes.

String Comparison Operators

<pre>\$ string=123 \$ [[\$string = 0123]] \$ print \$? 1</pre>	<pre>\$ string=123 \$ [[\$string -eq 0123]] \$ print \$? 0</pre>
--	--

-eq	Convert strings to integer and compare: 123 and 0123 are identical.
=	Perform string comparison: 123 and 0123 are different strings.

-z s1	success if length of string <i>s1</i> is zero
-n s1	success if length of string <i>s1</i> is non-zero
s1 = s2	success if strings <i>s1</i> and <i>s2</i> are identical
s1 != s2	success if strings <i>s1</i> and <i>s2</i> are not identical
s1 < s2	success if string <i>s1</i> comes before string <i>s2</i> based on their ASCII values
s1 > s2	success if string <i>s1</i> comes after string <i>s2</i> based on their ASCII values

2-14

```
if [[ -r myfile ]] ; then  
    we have read access to file
```

'if' tests the status return of [[
if 0 'then' is executed

Avoid the temptation to talk about true and false.

-nt and -ot compare file modification times
 UNIX does not keep file creation time.

-ef hard links ONLY - symbolic links are not -ef

File Enquiry Operators

```
$ ls -l myfile
-rw-rw-r-- 1 sam          10604 Dec 18 09:44 myfile
$ [[ -r myfile ]]
$ print $?
0
```

```
if [[ -r myfile ]]
then
...
fi
```

-a file	success if <i>file</i> exists
-r file	success if <i>file</i> exists and is readable
-w file	success if <i>file</i> exists and is writable
-x file	success if <i>file</i> exists and is executable
-f file	success if <i>file</i> exists and is a plain file
-d file	success if <i>file</i> exists and is a directory
-s file	success if <i>file</i> exists and has a size greater than zero

```
file1 -nt file2  success if file1 is newer than file2
file1 -ot file2  success if file1 is older than file2
file1 -ef file2  success if file1 is another name for file2
```

2-15

In 2nd example at top: What is order of logic compares?

&& has highest precedence, so

1. `-z $HOME && -z $TERM`
2. `that result || -z $LOGNAME`

An arithmetic example sometimes clears confusion.

`2+3*4`

Note that parentheses can be used to override precedence, e.g.

`[[-z $HOME && (-z $TERM || -z $LOGNAME)]]`

What is the meaning of `-z` and `-d` ?

From preceding pages

`-z` true if length of string is zero

`-d` true if file exists and is a directory

Answers to questions:

1. 1

2. 1

3. 1

4. 0

Logic Operators

```
$ [[ -d $HOME && -d /usr2/$HOME ]]
```

! Unary negation operator

&& Logical AND operator

|| Logical OR operator

Shown in decreasing order of precedence

```
$ [[ -z $HOME && -z $TERM || -z $LOGNAME ]]
```

Questions:

```
$ A="Oak"; B="Elm"; C=123; D=0123
```

What value is returned by each of the following?

1. [[\$A = \$C]]

2. [[! -n \$D]]

3. [[\$A != \$B && \$C = \$D]]

4. [[\$A = \$B || \$C -eq \$D]]

NOTE the need for exact syntax.

Korn shell is not as forgiving as the C language.

 'then' requires a line by itself

 'elif' requires a 'then', 'else' does not

 'fi' terminates 'if'

'if' and 'case' were provided in the earliest Bourne shell. 'while', 'for', and 'until' were written into later versions of the Bourne shell.

This is the reason that 'if' and 'case' are terminated by fi and esac.

The later logic constructs were written by Republican programmers.

Many shell commands could be in 'then' or 'else' blocks.

Only one command used in each block of the examples on this page.

Avoid the temptation to discuss true and false.

 'if' tests the status return of [l.

 If 0, 'then' is executed.

At bottom:

 'if' tests the status return of 'who | grep'

 If 0, 'then' is executed.

grep -q silent grep - no output lines written to stdout
 the only result is the status return.

grep -s in ULTRIX.

Making Decisions (*if*)

```
if [[ -d $1 ]]
then
    print "$1 is a directory"
else
    print "$1 is not a directory"
fi
```

```
if [[ -f $1 ]]
then
    print "$1 is a plain file"
elif [[ -d $1 ]]
then
    print "$1 is a directory file"
else
    print "$1 is not a plain file nor a directory"
fi
```

```
if who | grep -q 'sam'
then
    print "sam is logged on"
fi
```

2-17

Avoid the temptation to discuss true and false.

Consider the result if the shell script fragment at top is executed as:

```
$ myscript dog cat
```

Note the need for exact syntax -

'do' and 'done' on separate lines.

'true' is a shell built-in, not a separate utility.

How do we get out of an infinite loop?

```
exit  
break (tomorrow)
```

Positive Iteration loops (*while*)

```
while [[ -n $1 ]]
do
    if [[ -r $1 ]]
    then
        cp $1 $NEWDIR
    fi
    shift
done
```

Loop as long as the command following returns success:

Zero	SUCCESS
Nonzero	FAILURE

```
while true
do
    lines executed in an infinite loop
done
```

true always returns a 0 (success) value.

2-18

`${FILE}new` to avoid search for variable `FILEnew`
`$FILE.new` works - `{}` are not needed.

Consider 2nd shell script fragement executed as:

```
$ myscript dog cat
```

`'for'` used for a known number of iterations
`'while'` used for an indefinite number of iterations

Additional logic constructs will be discussed tomorrow.

List Processing (*for*)

```
for FILE in fa fb fc fd
do
    cp ${FILE} ${FILE}new
done
```

```
for FILE
do
    cp ${FILE} ${FILE}new
done
```

```
for FILE
    is equivalent to:
for FILE in $*
```

2-19

PS1 primary shell prompt
PS2 secondary shell prompt
PS3 'select' prompt string (tomorrow)
PS4 execute trace marker
end of list

the execution at the bottom actually occurred.

Cannot explain the line numbers echoed for [[-a

Why use single quotes in the definition of PS4 ?

To avoid evaluation of LINENO when making the definition
of PS4

To place \$LINENO (not its value) into PS4.

PS4="at \ \$LINENO - " works too

PS4="at \$LINENO - " does not work

Debugging Shell Scripts - Execute Trace

Execute Trace - print shell commands as they are executed.

```
$ nl save #cat with line numbers
1 while [[ -a $1 ]]
2 do
3     cp $1 $1.save
4     shift
5 done
```

```
$ ksh -x save abc.dat xyz
+ [[ -a abc.dat ]]
+ cp abc.dat abc.dat.save
+ shift
+ [[ -a xyz ]]
+ cp xyz xyz.save
+ shift
+ [[ -a ]]
```

```
$ print $PS4
+
$ export PS4='at $LINENO - '
```

```
$ ksh -x save abc.dat xyz
at 1 - [[ -a abc.dat ]]
at 3 - cp abc.dat abc.dat.save
at 4 - shift
at 4 - [[ -a xyz ]]
at 3 - cp xyz xyz.save
at 4 - shift
at 4 - [[ -a ]]
```

2-20

Much less useful than execute trace.

Useful for syntax checking.

For example, take 'do' line from 'save' script and run it with the verbose trace.

verbose trace like compiler errors

execute trace like run time errors

Debugging Shell Scripts - Verbose Trace

Verbose Trace - print shell commands as they are read.

```
$ nl save
1      while [[ -a $1 ]]
2      do
3          cp $1 $1.save
4          shift
5      done
```

```
$ ksh -v save abc.dat xyz
while [[ -a $1 ]]
do
    cp $1 $1.save
    shift
done
```

Useful for syntax checking.

2-21

```
set -x          Bourne shell syntax that works for Korn shell, too
set +x
```

Often used to echo commands that are being executed from a shell script, e.g.

```
set -o xtrace
cc -o $1 $1.c
set +o xtrace
```

instead of

```
print "cc -o $1 $1.c"
cc -o $1 $1.c
```

Execute Trace Inside a Shell Script

```
$ cat save
while [[ -a $1 ]]
do
    set -o xtrace          # turn on execute trace
    cp $1 $1.save
    set +o xtrace         # turn off execute trace
    shift
done
```

```
$ save abc.dat xyz
+ cp abc.dat abc.dat.save
+ cp xyz xyz.save
```

VERY IMPORTANT - this technique is widely used in shell scripts.

Suggest that you write the `'ls s*'` command on board and show its output.

Write that output in place of `$(ls s*)` in definition of `'sfiles'`.

`'for'` command in middle of page will look like:

```
for name in sam mary tom betty
```

`'mail'` gets its letter from stdin - stdin redirected to `./mail_file`

grave accent is back apostrophe - upper left of keyboard.

New Korn shell syntax `'$(cmd)'` will be used in this workbook.

If time permits on Day 1, you might start some of the topics from the next chapter (2AM). The discussion of different ways of executing shell commands on pp. 3-2 to 3-6 could easily be conducted on Day 1. It is probably best not to begin the discussion of Aliases and Functions until Day 2.

Command Substitution

```
$(shell_command)
```

Above is replaced by the output produced by *shell_command*.

```
$ sfiles=$(ls s*)
```

```
$ print $sfiles
```

```
s.cnv.c      start      superx
```

```
$ users
```

```
sam  mary  tom  betty
```

```
# shell script fragment
```

```
for name in $(users)
```

```
do
```

```
    mail $name < mail_file
```

```
done
```

Bourne shell command substitution using the grave accent character is supported by KornShell.

```
`shell_command`
```

```
$ sfiles=`ls s*`
```

It is intended that the shell script examples at the end of each day's lecture be discussed in depth. They are an important teaching technique.

The concept pages discussed in the workbook introduce the tools that can be used in shell scripts.

The example scripts show the tools at work.

It is one thing to know how to operate a torque wrench and another to know when and where to use it.

usercheck takes one argument - a number of users

It is important to verify that arguments are present.

'Usage' message shows correct usage.

\$0 is name of shell script.

print writes to stdout

>&2 sends stdout to same as stderr.

'exit' status of nonzero indicates error.

num = number of users currently logged in

PPID defined on p. 1-10

'kill' will be discussed in more detail on Day 2.

For now `kill -9` is a sure kill to process

`#!/usr/bin/ksh`

on first line guarantees that a ksh child will be created, no matter the type of login shell, e.g. C shell.

Shell Script Examples

```
#!/usr/bin/ksh
#usercheck      shell script to determine the number of users
#               logged on.  If that number exceeds the number
#               specified as a command line argument, an appropriate
#               message is written to your terminal and the shell
#               process and its parent process are killed.

if [[ $# -ne 1 ]]
then
    print "Usage: $0 number" >&2
    exit 1
fi

num=$(who | wc -l)

if [[ $num -gt $1 ]] ; then
    print "Too many users logged on"
    kill -9 $PPID
fi
```

2-24

printlater takes one argument - a filename

Second 'if' statement checks if is a directory OR
 if is not readable

Note 2 exit statuses - 1 and 2. Caller can print \$? and see exit point.

let command does arithmetic - more on Day 2

PHILOSOPHY - This writer finds it instructive to use examples prior to the formal presentation of the topic. Others may be troubled by this, but usually students are not ruffled and get to see a tool at work before they see the formal explanation of the tool.

Output of wc -l is line count of \$1.

len = number of 60 line pages in \$1

Note use of wild card in [[\$ans = y*]]
 - y, yes, yup, yeh all are positive.

Why '=' and not '-eq' in \$ans compare?
 string compare, not integer compare

If time permits on Day 1, you might start some of the topics from the next chapter (2AM). The discussion of different ways of executing shell commands on pp. 3-2 to 3-6 could easily be conducted on Day 1. It is probably best not to begin the discussion of Aliases and Functions until Day 2.

```

#!/usr/bin/ksh
#prntlater      shell script which takes one filename as a
#               command line argument and determines how many lines
#               are in that file.  If the file is longer than 2 pages,
#               the shell script asks you if you wish to print it
#               after hours.  If you respond affirmatively, the shell
#               script exits; otherwise the file is printed.  If the
#               command line argument is a directory or is not
#               readable, an appropriate error message is printed.

if [[ $# -ne 1 ]] ; then
    print "Usage: $0 filename" >&2
    exit 1
fi

if [[ -d $1 || ! -r $1 ]]
then
    print "$1 cannot be printed." >&2
    exit 2
fi

let "len=$(wc -l $1)/60+1"
if [[ $len -gt 2 ]]
then
    read ans?"Print after hours?: "
    if [[ $ans = y* ]] ; then
        exit
    fi
fi

lpr $1

```


Chapter 3: Day 2AM

3-2

Top normal execution of shell script
cd of script is done by child ksh (shell built-in)
current directory does not change in login ksh since script
executes in child ksh

\$ ksh show_cron would work the same

Bottom shell script read and executed by login ksh.
current directory changes in login ksh since cd executes
in login ksh (shell built-in).
NOT used for efficiency (to save time to create child process)
UNIX has light weight processes - takes little time to fork().

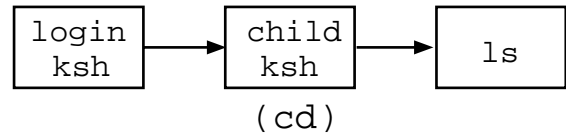
If execute .profile as at bottom, it will execute in child ksh.

\$. .profile should be used

The common use of the dot command is to institute a changed
.profile without logging out and back in again.

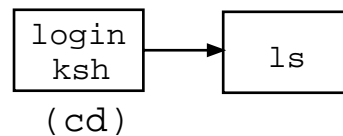
Executing Commands without Forking (*dot*, *exec*)

```
$ cat show_cron
cd /var/spool/cron/crontabs
ls
$ show_cron
adm    cronuucp      milt
root   sys          uucp
$ pwd
/usr/users/sam
```



The . (dot) command executes a shell script in the parent shell:

```
$ . show_cron
adm    cronuucp      milt
root   sys          uucp
$ pwd
/var/spool/cron/crontabs
$ . ls
ksh: syntax error:  ')' unexpected
```



The dot command is used for shell scripts ONLY.

Consider: \$.profile

3-3

exec used for any shell command - program or shell script

dot used only for shell scripts

exec sometimes used as per bottom example -
to specify a final command to execute in a shell script.

exec seldom used.

exec replaces the parent shell with any shell command:

```
$ exec show_cron
```

```
adm    cronuucp    milt
root   sys         uucp
login:
```

```
login
ksh
```

becomes

```
ksh → ls
(cd)
```

```
$ exec ls
```

```
eval  labs  mtpt  opt1  opt2  opt3  opt4
redir redir2
login:
```

```
login
ksh
```

becomes

```
ls
```

```
$ cat myscript
```

```
...
exec echo line 1
echo line 2
...
```

echo writes its arguments to standard output.

echo is a separate program; **print** is a shell built-in.

```
$ myscript
line 1
```

```
login → child
ksh    ksh
```

becomes

```
login → echo
ksh
```

3-4

There is a performance increase when using semi-colon to place several commands on one line.

Performance considerations will be discussed on Day 3.

There is no functional difference between placing commands on several lines and joining them on one line with semi-colons.

Spaces could be used with parentheses, but are not required.

Parentheses are like a shell script created in a shell command.

Following is a command commonly used by system administrators to copy a filesystem from one partition to another.

```
dump -0f - /usr | (cd /mnt ; restore -rf -)
```

dump	0	dump all files
	f -	to stdout (pipe to restore)
()		create a child ksh in which to cd, restore
cd /mnt		set current directory for restore
restore	r	restore all files
	f -	from stdin

If parentheses were left off, then the command

	dump cd	would be executed
followed by	restore	

Grouping Shell Commands

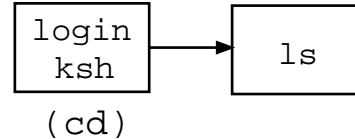
Semi-colon Join several shell commands on the same line.
Each command executes as if typed individually.

```
$ cd /etc ; ls m*
```

```
mail.aliases  miscd  mkfs  
mklost+found  mknod  mkpasswd  
mkproto       motd   mount
```

```
$ pwd
```

```
/etc
```



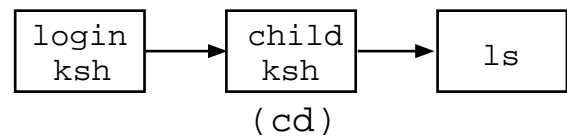
Parentheses Force the group to execute in a child shell.

```
$ (cd /etc ; ls m*)
```

```
mail.aliases  miscd  mkfs  
mklost+found  mknod  mkpasswd  
mkproto       motd   mount
```

```
$ pwd
```

```
/usr/users/sam
```



3-5

Spaces with curlies are essential.

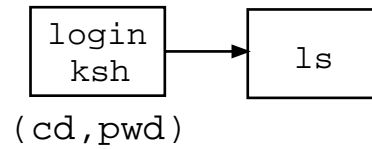
Trailing semi-colon is also essential.

Current directory changes since `cd` executes in login `ksh`.

Curly braces Cause the group to execute in the parent shell, but allow redirection of multiple commands.

```
$ { cd /etc ; pwd ; ls m* ; } > ~/list
```

Note the punctuation.



```
$ cat ~/list
```

```
/etc
mail.aliases  miscd  mkfs
mklost+found  mknod  mkpasswd
mkproto       motd   mount
```

```
$ pwd
```

```
/etc
```

3-6

`grep -q` **silent grep - no output lines written to stdout
the only result is the status return.**
`grep -s` in ULTRIX.

Shorthand way to replace 'if' statements.

```
if who | grep -q sam
then
    print "sam is logged on"
fi
```

Command Execution Dependent on Previous Command (&&, ||)

`cmd1 && cmd2` execute *cmd2* only if *cmd1* succeeds.

```
$ who | grep -q sam && print "sam is logged on"
```

`cmd1 || cmd2` execute *cmd2* only if *cmd1* fails.

```
$ who | grep -q sam || print "sam NOT logged on"
```

3-7

Show results if shell script fragment executes with argument of:

4	is a single digit
44	none of the above
abcd	begins with lower case
lion	is king of the jungle
dog	is name of house pet

Suppose `[a-z]*` is moved to top.

Then lion would print 'begins with lower case'

The order of the patterns is VERY IMPORTANT.

Note the need for exact syntax.

right parenthesis without left parenthesis
'in' on 'case' line
'esac' terminator

We say `dog | cat)`

not `dog)` the way we would in the C Language.
`cat)`

Multi-way Branching (*case*)

```
case $1 in
    [0-9])    print "is a single digit"
              ;;
    dog|cat)  print "is name of house pet"
              ;;
    lion)     print "is king of the jungle"
              ;;
    [a-z]*)   print "begins with lower case"
              ;;
    *)        print "none of the above"
              ;;
esac
```

At most one action is taken.

If no patterns match, no action taken (fall through).

Any shell command(s) can be executed in each action.

Delimiter (; ;) is required to terminate each case.

Patterns may employ standard metacharacters

* ? []

* is the pattern for the default case (matches all).

3-8

Top example:

tom never processed, since after 2 shifts \$1 equalled 'harry'

Bottom example:

Grepping for the word 'Results' in any line of the file.

data.file has such a line.

nov.dat never searched.

Negative Iteration Loops (*until*)

```
until [[ $1 = "harry" ]]
do
    print "Where is harry? - $1"
    shift
done
```

```
$ myscript sam mary harry tom
Where is harry? - sam
Where is harry? - mary
```

Loop as long as the command returns failure status.

```
until grep -q Results $1
do
    print "No Results found in $1"
    shift
done
```

```
$ myscript abc.txt data.file nov.dat
No Results found in abc.txt
```

3-9

Example at bottom:

`break` sends us to the line following `'done'`.

`continue` sends us to the top of the loop.

Note the use of semi-colon to put `'while'` & `'do'` on same line.

How do we get out of an infinite loop?

`exit`

`break`

Note the use of `;` to place 2 parts of a logic statement on the same line, e.g.

```
while true ; do
```

Iteration Control Statements (*break*, *continue*)

`break [n]`

Terminates execution of *n* (default=1) innermost enclosing loops (**for**, **while**, **until**, **select**).

`continue [n]`

Causes execution to resume at the beginning of the *n* (default=1) nearest loop (**for**, **while**, **until**, **select**).

`while true ; do`

`...`

`if [[$num -gt 12]] ; then`

`break # goto command after done`

`... (statements skipped if break)`

`if [[$var -le 0]] ; then`

`continue # goto to top of loop`

`... (statements skipped if continue or break)`

`done`

3-10

Preset alias defined by the ksh program
 user takes no action to get preset aliases

Note the preset aliases on the page - false, history, r, true

Tracked alias no list is available to show the programs that
 will become tracked aliases at first execution.

'hash' is preset alias for 'alias -t'

\$ hash shows list of tracked aliases

\$ hash -r removes all tracked aliases

Problem that can occur:

A program is developed in one directory and then moved to another directory of the PATH variable. Whenever the program is executed, an error results because the shell always looks for the program in the original directory.

Solution: \$ hash -r to remove the tracked alias

Defined aliases commonly defined in .profile

Aliases take no arguments.

KornShell Aliases

```
$ alias delete=rm
$ alias lst='ls -l'
$ alias cprog='cd /class/prog ; pwd'
$ alias psm='ps ax | more'

$ delete myfile
$ lst savefile
-rwxr-xr-x  1 sam  staff   580  Jul 5 11:38  savefile

$ alias lst
lst = ls -l

$ alias
autoload=typeset -fu ← Preset Aliases
cat=/bin/cat
chmod=/bin/chmod ← Tracked Alias
cp=/bin/cp
cprog=cd /class/prog ; pwd ← Defined Alias
delete=rm
false=let 0
functions=typeset -f
history=fc -l
integer=typeset -i
ls=/bin/ls
lst=ls -l
psm=ps ax | more
r=fc -e
true=:
type=whence -v
```

Tracked aliases are defined for common shell commands the first time that shell command is used.

Avoids **\$PATH** search for subsequent executions.

```
$ set -o trackall
```

Every command becomes a tracked alias when first encountered.

There must be a trailing ' ; }' at the end of the function definition when the function is defined on one line.

When the trailing curly brace is on a line by itself (see the following page), there is no need for a trailing semi-colon.

Note the use of the dot command to define the function in the login ksh.

UIL compilation is part of Motif programming.


Functions commonly defined in .profile.

Function is like an alias that takes arguments.

The 'functions' command (to display function definitions) does not want to work properly in a shell script. The function definitions work fine, but the 'functions' command does not work properly.

KornShell Functions

```
$ cat ui.fn
#Invoke the UIL compiler
function ui { uil -o $1.uid $1.uil ; }
```

Note the punctuation 

```
$ . ui.fn
```

Define the function in the login ksh.
Functions commonly defined in **.profile**.

```
$ functions ui
function ui
{
uil $1.uil -o $1.uid ; }
```

```
$ ui abc
```

Execute a UIL compile for **abc.uil** producing **abc.uid**.

Function arguments: **\$1, \$2, ...**
 \$#
 \$*

A function is like a shell script that is built into the ksh process.

3-12

A fancier definition for the same function is:

```
function full {  
    print $([[ $1 != /* ]] && print $PWD/) $1  
}
```

Note the use of the dot command to define the function in the login ksh.

Setting PS4 to include \$LINENO works just as in debugging shell scripts.

However, since all commands execute at line 1, this has limited utility.

```
set -o xtrace  
set +o xtrace
```

work just as in shell scripts.

Debugging Functions

```
$ cat full.fn
#full- function to print the full pathname for the
#      file given as an argument
function full {
    if [[ $1 != /* ]] ; then
        PathName=$PWD/$1
    else
        PathName=$1
    fi
    print $PathName
}
```

```
$ . full.fn                # define the function
$ typeset -ft full         # trace the function
```

```
$ full abc
+ [[ abc != /* ]]
+ PathName=/usr/users/sam/abc
+ print /usr/users/sam/abc
/usr/users/sam/abc
```

An alias can be exported after its definition OR with its definition.
Functions can only be exported after their definition.

This page is the first of 3 pages that explain the subtle difference between executing a shell script by name and as an argument to ksh. This topic is always very difficult to communicate. Even after understanding the difference, some people say, "So what?"

On this first page, we are making the point that aliases/functions are exported ONLY if shell scripts are executed by name.

Variables are always exported, even if the child process runs a program.

For example, 'vi' runs in a child process and looks at the value of the exported variable TERM to know what type of terminal it runs on.

Exported Functions and Aliases

```
$ alias -x delete          # OR alias -x delete=rm
$ typeset -fx ui
```

Mark the alias or function to be exported.

```
$ export LANG=french
```

Mark the variable to be exported.

Exported variables are defined in all future child processes.
Child process can run a program or a shell script.

Exported aliases and functions are only defined:
In future child processes that run a shell script
AND
Only when that shell script is run by name.

```
$ cat myscript
alias delete          # show value of exported alias
```

```
$ myscript          # run by name
delete=rm
```

```
$ ksh myscript      # run as arg to ksh
delete alias not found
```

On this second page of the trio of pages that explain the subtle difference between executing a shell script by name and as an argument to ksh, we are making the point that there is a second shell script that executes at login - the ENV file.

Also we make the point that the ENV file executes first in the process created for a shell script that executes as an argument to ksh.

Most students will see what is coming on the third page.

At this point it might be appropriate to review the 2 points presented on the last 2 pages.

1. Aliases/functions are exported only to shell scripts that execute by name.
2. The ENV file executes in the process created for a shell script that executes as an argument to ksh.

Maybe we can get 2 different sets of aliases defined for a shell script?

***ENV* Environment Variable**

In **~/profile** OR **/etc/profile**:

```
export ENV=~/kshrc
```

At ksh login, the following scripts will execute:

```
/etc/profile  
~/profile  
~/kshrc
```

When executing a shell script as an argument to **ksh**,

~/kshrc executes in the child ksh prior to shell script.

Aliases/functions must be exported in `.kshrc` to be defined in shell scripts that execute as an argument to `ksh`.

It seems that the exporting should not be necessary, since `.kshrc` executes in the process.

The bottom paragraph is a cruel joke to those who have struggled to understand the last 3 pages.

Not all Korn shells exhibit this feature, but Digital UNIX and ULTRIX do.

This fact means that if you have `#!/usr/bin/ksh` as the first line in every shell script, the following is true:

1. No matter which is the login shell, one will always get a `ksh` child for the shell script.
2. No matter how one executes the script, that script will behave as if it was executed as an argument to `ksh`, i.e.
 - a. the `ENV` file will execute prior to the script and
 - b. exported aliases/functions from that `.kshrc` will be defined for the script.

Aliases/functions will NOT be exported from the login shell.

The `'functions'` command (to display function definitions) does not want to work properly in a shell script.

The function definitions work fine, but the `'functions'` command does not work properly.

Either way of executing shell scripts has this problem.

Difference between *\$ myscript* and *\$ ksh myscript*

In `~/.kshrc`

```
alias -x delete=rm
```

```
$ alias delete
```

```
delete=rm                # defined from ~/.kshrc
```

```
$ alias -x delete='rm -i'
```

```
$ cat myscript
```

```
alias delete            # show value of exported alias
```

```
$ myscript
```

```
delete=rm -i
```

The alias was exported from login ksh.
`~/.kshrc` does NOT execute.

```
$ ksh myscript
```

```
delete=rm
```

`~/.kshrc` executes in child ksh before `myscript`.

Dual script environment exists for aliases and functions.

Shell scripts that execute by name get exported aliases and functions defined in login ksh.

Shell scripts that execute as an argument to **ksh** get **exported** aliases and functions defined in the `$ENV` file.

EXCEPTION for Digital UNIX and some others:

A shell script that has as its first line `#!/usr/bin/ksh` behaves when executed by name as if it had been executed as an argument to **ksh**.

FPATH, like other path variables, could be a list of directories.

In ULTRIX, if one types:

```
$ functions full
```

before using the alias 'full', the login ksh exits.

This is an obvious bug.

In Digital UNIX, if one types:

```
$ functions full
```

before using the alias 'full', one is told that full is an undefined function.

Autoloading Functions

An alternative to defining functions in `~/.profile` or the `$ENV` file.
Reduce startup time by delaying function definition until needed.

1. Place each function definition in a separate file.
The name of the function definition file is the name of the function, e.g. function **full** defined in a file named **full**.

2. Copy all function definition files into one directory, e.g. `~/functions`.

3. Define the `$FPATH` variable to point to that directory, e.g. in `~/.profile`:

```
export FPATH=~/functions
```

4. Define the functions to be autoloaded, e.g. in `~/.profile`:

```
autoload full  
autoload ui
```

The first time the function is invoked, the function will be defined from the file in the **functions** directory.

This page illustrates how all the commands in an entire while loop can have their input/output redirected.

It is a common script programming trick which will occur in several examples that follow.

It might be appropriate to review the redirection symbols.

<	stdin
>	stdout
2>	stderr
>>	append to stdout
>	stdout and override noclobber

Redirecting Input/Output in Logic Statements

```
$ cat myscript
while read line
do
    print $line
done < file1 > file2
```

For all commands in while loop:

```
    stdin  = file1
    stdout = file2
```

```
$ cat file1
This is the first line of file1.
This is the second line of file1.
This is the third line of file1.
```

```
$ ls file2
file2 not found
```

```
$ myscript
```

```
$ cat file2
This is the first line of file1.
This is the second line of file1.
This is the third line of file1.
```

3-18

In some Korn shells, if one types

```
$ command > abc 2> abc
```

stdout and stderr are not directed to the same file.

If writes to stderr are performed, a new file named 'abc' is created, replacing the old file named 'abc' created for stdout.

This is not the case with Digital UNIX.

The above in both ULTRIX and Digital UNIX works just the same as the top example in the workbook.

Reading/writing using file descriptors 3-> will be discussed on Day 3.

Redirecting Both Standard Output and Standard Error

```
$ command > abc 2>&1
```

Redirect the standard output for command to file **abc**.
Redirect the standard error to the same file as stdout.

```
stdout = abc
stderr = abc
```

```
$ command 2>&1 > abc
```

Redirect the standard error to the same file as stdout.
Redirect the standard output for command to file **abc**.

```
stderr = terminal
stdout = abc
```

The order in which redirection is specified is significant.

File descriptor 0	standard input
File descriptor 1	standard output
File descriptor 2	standard error

3-19

The next 3 pages define 3 functions intended to be used together.

ma abc - remembers the current directory with a line in a disk file
abc:current_directory

ga abc - find the 'abc' line in the disk file and sets the current
directory to the last field of that line.

la abc - lists the 'abc' line from the disk file.

If no argument is specified,

ma uses the last segment of the current directory pathname
as the token.

ga prints error message.

la lists all lines from disk file.

'basename' returns the last segment of pathname.

Function and Shell Script Examples

```
#magala.fn      shell script to define functions ma, ga and la
#               that will remember the current directory and enable
#               returning to that directory at a later time.
#               A disk file is used to remember the directory.
# ma           Mark the current directory with the "word" given as
#               argument.
#               If no word is given on the command line, use the
#               basename for the current directory as the word.
#               Write a line into a disk file in the home directory.
```

```
function ma {
    DirFile=$HOME/.dir.data
    Usage="Usage: ma [word]"

    case $# in
        0)      print $(basename $PWD):$PWD >> $DirFile
                ;;
        1)      print $1:$PWD >> $DirFile
                ;;
        *)      print "Too many arguments." >&2
                print $Usage >&2
                ;;
    esac
}
```

```
$ cd /etc
$ ma myetc
```

Writes into DirFile myetc:/etc

```
$ cd /usr/sys
$ ma
```

Writes into DirFile sys:/usr/sys

3-20

```
grep "^$1:" $DirFile
```

Writes to stdout the line of the file that begins with (^) the first positional parameter value (\$1) followed by a colon.

There should be only one such line in the file.

This function has problems if there are multiple lines that have the same token. The repair is left as an exercise.

Why are double quotes used?

to cause ksh to substitute the value of the first positional parameter. The double quotes could be removed without harming functionality.

```
awk -F: '{print $2}'
```

Reads lines from stdin and defines a field separator of :

Prints field 2 for that line

There should be only one line of input. See above..

Why are single quotes used?

to prevent ksh from seeing \$2 as the second positional parameter.

d= field 2 of the match line in the file
(the directory full pathname).

```
# ga      Goto the directory associated with the "word" given as
#          argument.
#          Read the directory name from the disk file in the home
#          directory.
```

```
function ga {
    DirFile=$HOME/.dir.data
    Usage="Usage: ga word"

    case $# in
        1) d=$(grep "^$1:" $DirFile | awk -F: '{print $2}')
            # Look for word in DirFile and set d = 2nd field
            if [[ -n $d ]] ; then
                cd $d
                print "New current directory is $(pwd)"
            else
                print "No entry for \"$1\" in $DirFile"
            fi
            ;;
        *) print "Too many arguments." >&2
            print $Usage >&2
            ;;
    esac
}
```

```
$ ga myetc
```

```
New Current directory is /etc
```

```
$ ga sys
```

```
New current directory is /usr/sys
```

3-21

```
grep $1 $DirFile
```

Writes to stdout the line of the file that contains the value of the first positional parameter.

There should be only one such line in the file.

This function has problems if there are multiple lines that have the same token. The repair is left as an exercise.

```
# la      List the words and their values from the disk file in
#         the home directory.
```

```
function la {
    DirFile=$HOME/.dir.data
    Usage="Usage: la word"

    case $# in
        0)      cat $DirFile
                ;;
        1)      e=$(grep $1 $DirFile)
                if [[ -n $e ]] ; then
                    print $e
                else
                    print "No entry for \"$1\" in DirFile"
                fi
                ;;
        *)      print $Usage >&2
                ;;
    esac
}
```

```
$ la sys
sys:/usr/sys
```

```
$ la
myetc:/etc
sys:/usr/sys
```

3-22

The output from: `! sort file1`
is: `/tmp/bang3722 (assuming pid=3722)`

The shell will substitute in place of
`$(! sort file1)`
the output produced, i.e. `/tmp/bang3722`

The original shell command will then be:
`$ sort /tmp/bang3722 /tmp/bang3723`

In 30 seconds the `/tmp/bang` files will be removed.

```
# !      shell script to store the output of a command (given
#        on the command line) and write that filename to
#        standard output.
#        Example:
#           $ diff $(! sort file1) $(! sort file2)
```

```
TmpFile=/tmp/bang$$
```

```
if [[ $# -eq 0 ]] ; then
    print Usage: $(basename $0) command [args] >&2
    print $TmpFile
    exit 1
fi
```

```
# execute the command, placing output into TmpFile
$* > $TmpFile
```

```
# write the name of TmpFile to standard output
print $TmpFile
```

```
# spawn a process to remove the TmpFile in 30 seconds
(sleep 30 ; rm -f $TmpFile)&
```

There is no shell command to show all the links to a file.

This script makes a valiant attempt by looking for all files in the entire directory tree that have the same inode number and the same size.

Each filesystem can have a file with inode number = 234, so the same inode number does not guarantee that the files are the same.

We could improve the accuracy by looking at other file attributes, like modification time or number of links.

INUM= The next to bottom example shows the output of 'ls -i'.
'awk' prints the contents of field 1 = inode number.

SIZE= The bottom example shows the output of 'ls -l' on ULTRIX.
'awk' prints the contents of field 4 = file size.

FILE= 'find' writes the full pathname for all files in the entire directory tree (1st find arg is /) that has the same inode number as the specified file.

ls -l sends a long listing of those files to 'awk'.

awk prints field 8 for the lines for which field 4 matches SIZE.

\$SIZE is not inside single quotes, so ksh expands \$SIZE to its numerical value.

```
'$4 == ' $SIZE ' {print $8}'
```



```

# links shell script to show all the links to a file whose
# name is specified on the cmd line.
# Each file has a unique inode number on its filesystem.
# We search for all file links that have the same inode
# number.
# The file size is used to ensure we have the same file.
# There is a small probability of error (that another
# file on another filesystem will have the same inode
# number and the same size).
#
# The ls -l output differs between BSD and SysV
# BSD -rwxr-xr-x 1 gar 712 Dec 29 10:46 x
# SysV -rwxr-xr-x 1 gar ins 712 Dec 29 10:46 x
case $# in
  1) INUM=$(ls -i $1 | awk '{print $1}')
     SIZE=$(ls -l $1 | awk '{print $4}') # $5(Digital UNIX)
     if [[ -n $INUM ]] ; then
         FILE=$(find / -inum $INUM -print 2> /dev/null)
         ls -l $FILE | awk '$4 == '$SIZE' {print $8}'
     fi
     ;;
  *) print "Usage: $0 filename" >&2
     exit 1
     ;;
esac

```

```

$ links ~sam/abc
/usr/users/sam/abc
/usr/users/sam/xyz

```

```

$ ls -i ~sam/abc
26642 /usr/users/sam/abc

```

```

$ ls -l ~sam/abc
-rw-rw-r-- 2 sam 361 Dec 29 10:24 /usr/users/sam/abc

```


Chapter 4: Day 2PM

4-2

A shell script fragment is shown at the top.

The execution of that fragment is shown in the middle.

For the first response, `i=fries`

Note that `select` is like an infinite loop. `break` or `exit` is needed to get out of the loop.

Very useful when menus will be used to indicate user preference.

The use of menus is a bit old-fashioned. A graphical user interface, e.g. Motif, is more likely to be considered state-of-the-art.

`<RETURN>` will redisplay the menu.

5 (or any number not in the menu) would cause the variable `i` to be assigned the value `""`.

List Processing (*select*)

```
PS3="Please enter the number for a food: "  
select i in candy carrot fries spinach  
do  
    case $i in  
        candy|fries)      print "Poor choice"  
                          ;;  
        carrot|spinach)   print "Good choice"  
                          break  
                          ;;  
        *)                print "invalid number"  
                          ;;  
    esac  
done
```

```
1) candy  
2) carrot  
3) fries  
4) spinach  
Please enter the number for a food: 3  
Poor choice
```

Please enter the number for a food:

Looping continues until **break** (or **exit**) is encountered.

Default value for **\$PS3** is **#?**.

PS1 (\$)	Primary shell prompt
PS2 (>)	Secondary shell prompt
PS3 (#?)	Select prompt
PS4 (+)	Execute trace marker

A list of signals and their names can be obtained from:

```
$ kill -l
```

Avoid getting dragged into a detailed discussion of signals.

It is the intention to explain only enough about signals to have the pages that follow make sense.

The names for signals indicate the use the kernel makes for those signals.

One user process could send SIGINT to another user process.

However the receiver process could not distinguish between a SIGINT sent by the kernel terminal driver and a SIGINT sent by another user process.

If 2 processes wish to communicate using signals, SIGUSR1 and SIGUSR2 are commonly used.

The man pages indicate the default action for each signal.

```
$ man 4 signal          # in Digital UNIX
```

```
$ man sigvec           # in ULTRIX
```

For 90% of the signals, the default action is process termination.

For some signals, the default action is to stop the process; for others, the default action is to ignore the signal.

Programs do not try to protect themselves against signals.

Only another process with the same uid can send me a signal.

Signals are an inter-process communications technique.

When the user types the interrupt character (^C) at the terminal, the intention is to cause the running program to terminate.

A well-behaved program might handle SIGINT by cleaning up temporary files, etc., and then exiting.

User Programming Example:

Process sam1 reads data from a terminal port and writes it into a disk file.

Process sam2 reads the data from the disk file, processes it and writes a report to the magtape.

Process sam1 sends SIGUSR1 to process sam2 when the data has been completely written to disk.

UNIX Signals

Signals are sent to a process to inform it that an event has occurred.

A process can handle or ignore a signal.

The default action for most signals is process termination.

A signal can only be sent to a process with the same uid as the sender (superuser can send a signal to any process).

Assigned signals in Digital UNIX

1) HUP	14) ALRM	27) PROF
2) INT	15) TERM	28) WINCH
3) QUIT	16) URG	29) LOST
4) ILL	17) STOP	30) USR1
5) TRAP	18) TSTP	31) USR2
6) IOT	19) CONT	
7) EMT	20) CHLD	
8) FPE	21) TTIN	
9) KILL	22) TTOU	
10) BUS	23) IO	
11) SEGV	24) XCPU	
12) SYS	25) XFSZ	
13) PIPE	26) VTALRM	

4-4

EXIT is not a real signal. It is a pseudo signal simulated by the Korn shell to allow logout activity.

The C shell does not have a 'nohup' command.
In the C shell all background processes ignore SIGHUP.

When one logs out of ksh with running background jobs, one gets:
There are running jobs.

This message from ksh warns us; it does not prevent logout.
A second logout command (^D or exit) in succession will logout.

If there are stopped processes at logout, one gets:
There are stopped jobs.

This is also a warning; it does not prevent logout.

The default quit signal is ULTRIX is ^|.
^\\ is the default for Digital UNIX.

A programmer can use the core file with a debugger to debug an aborted application.

Common Signals

EXIT or 0 - pseudo signal

Sent when the KornShell in that process exits

Used to perform logout activity, e.g. clear screen

HUP or 1 - hangup

Sent to all processes in the session when the login ksh exits.

Terminates jobs running in background, if that job was not prefaced with the **nohup** command.

```
$ nohup find / -name 'a*' -print > ffile &
```

INT or 2 - interrupt

Sent when the interrupt character is typed (default = ^C).

Normally used to interrupt the foreground process.

QUIT or 3 - quit

Sent when the quit character is typed (default = ^\).

Same as the interrupt signal except that a core dump of the executing program is produced in a file called **./core**.

The interrupt and quit characters are settable using the shell.

```
$ stty intr ^C quit ^V
```

The kill command sends a signal.
The signal terminates the process.
The kill command does not terminate the process.
(I didn't kill him, the bullet did)

Occasionally 'kill' will not cause the process to terminate because the process is handling or ignoring SIGTERM.

'kill -9' will terminate the process.

Sometimes even 'kill -9' will not work because the process is blocked waiting for some kernel resource and cannot be waked to deliver the signal and be terminated.

It is a bit impolite to always 'kill -9'.
It gives the program no chance to cleanup temporary files, etc., before its termination.

Unless you are superuser, you can only send signals to processes with the same uid as yourself (effectively your own processes).

The signals listed on these 2 pages are the ones we desire most often to handle in shell scripts.

There is little to gain by discussing the other signals listed on p. 4-3.

TERM or 15 - software termination

Default signal number sent by the **kill** command

```
$ kill 3126
```

KILL or 9 - uncatchable kill

Sent by the **kill -9** command

Cannot be handled or ignored

```
$ kill -INT 3245
```

Send the interrupt signal to pid = 3245.

Signal might be caught or ignored by pid=3245.

Default action for INT signal is process termination.

Any signal can be sent using the **kill** command.

4-6

POINT TO EMPHASIZE:

The trap command list is NOT executed until the signal is received.

This shell script illustrates the common reason to handle signals:
to cleanup temporary files.

The use of single quotes or double quotes in the trap command has the same rationale as in other shell commands.
This matter was discussed on Day 1 on p. 2-8.

Resetting a signal restores its default action.

The 'trap' command can take signal names or signal numbers.

/tmp is a directory to which all users have write permission.
It is commonly used when creating temporary files in shell scripts.
The user may not have write permission to the current directory.

In order to generate a process unique name in /tmp, we use:

\$0	name of the shell script file
\$\$	pid of this process

Handling Signals in a Shell Script

```
#shell script to find a user that is logged on
who > /tmp/$0$$
trap "rm /tmp/$0$$ ; exit 3"    HUP INT QUIT TERM
for name ; do
    if grep -q $name /tmp/$0$$
    then
        print "$name is logged on"
    fi
done
rm /tmp/$0$$
```

A trap command list is established when the **trap** command is executed.

The trap command list is NOT executed until the signal is sent.

The **trap** command without arguments displays current trap settings.

Signals can be CAUGHT:

```
trap 'command list' signal [ signal ... ]
```

Signals can be IGNORED:

```
trap "" signal [ signal ... ]
```

Signals can be RESET:

```
trap signal [signal ... ]
```

4-7

Example at top:

Frequently users type not one `^C`, but `^C^C^C`.

This signal handler begins by ignoring other signals of the same type.

Example at bottom:

C shell has a shell script `~/ .logout` that executes at logout.

Korn shell uses a signal handler for the pseudo signal 0 (EXIT).

This handler must be declared for the login ksh in `.profile` or the ENV file.

`/usr/games/fortune` is part of the Games software subset and may or may not be loaded onto the lab systems.

A list of common shell variable names is provided on p. 1-10 and in Bolsky p. 178.

trap Examples

```
trap '
    trap "" HUP INT QUIT
    rm /tmp/$0$$
    lpr errfile
    exit 1
' HUP INT QUIT
```

By ignoring other signals immediately, multiple traps for the same event are prevented.

Commands to execute upon logout:

```
trap '
    clear
    /usr/games/fortune #fortune cookie
    print "\n\n\n"
    print " Good-Bye" " $LOGNAME"
    print "\n\n\t\t\tc"
    date
' EXIT
```

\$ trap

Display current trap settings.

4-8

This script should be run in background and the terminal attribute 'tostop' should be clear. (`stty -tostop`)

```
$ date +%m-%d          will print  
01-06
```

```
$ date '+%m - %d'      will print  
01 - 06
```



```

#showtime      shell script to output the time every 30
#              seconds. If an interrupt key is entered, print
#              a message saying so and return to the time
#              output. If a quit key is entered, print a
#              message saying so and exit.

trap 'print "Interrupt was typed"' INT
trap 'print "Quit was typed" ; exit' QUIT
while true
do
    date +%r
    sleep 30
done

```

date Formats:

%d	Day of month as a decimal number (01-31)
%j	Day of year (001-366)
%M	Minute number (00-59)
%m	Number of month (01-12)
%r	Time in AM/PM notation
%w	Weekday number (0[Sunday]-6)

and more

Shell variables are stored as character strings.

Consider `[[$XX -eq 0]]`

In order to do an arithmetic comparison, `[[` code converts the string value of `XX` to a number.

The `'let'` command takes only one argument.

We must use single or double quotes if spaces exist in its argument.

To the right of the `'='` in `'let'`, all variable names are replaced by their value, .i.e. `'$'` in front of `'x'` is not needed.

In the 3 examples at the bottom:

`[[` no difference from Day 1
`((and let` are equivalent to one another.

`'((` will be mostly used in the examples that follow.

Generally the syntax of `'((` is easier than `'[[` for persons who have programmed, BUT `'[[` can be used everywhere. `'((` need never be used.

C programmers will have no difficulty with the operators listed.
 Others may have difficulty seeing `'=='` as an operator.

`((x==18))` is the same as `[[$x -eq 18]]`
 `-eq` is an arithmetic comparison operator to `[[`.
 `==` is an operator to `((`.

Integer Arithmetic in the KornShell

```
$ x=17
$ x=$x+1
$ print $x
17+1
```

KornShell does arithmetic using a special command:

```
$ x=17
$ let x=$x+1   or   let "x = $x + 1"   or   let x=x+1
$ print $x
18
```

C language operators:

+	Addition	==	Is equal to
-	Subtraction	!=	Is not equal to
*	Multiplication	<	Is less than
/	Division	<=	Is less than or equal
%	Remainder	etc.	

To test arithmetic values:

```
$ (( x == 18 ))
$ print $?
0

$ let x==18
$ print $?
0

$ [[ $x -eq 18 ]]
$ print $?
0
```

4-10

Discuss the action of the shell script at the top first.

We could use: `while [[$n -le 5]]`

Then introduce the modified script in the middle.

The keyword 'integer' simplifies the script and makes it look more like a programming language.

Note punctuation when declaring 2 integers on one line (no comma).

Number bases from 2 to 36 are supported using:
numbers 0-9
letters a-z.

The reason the number bases supported stops at 36 is that there are 26 letters in the alphabet.

To convert from decimal to hexadecimal:

```
$ x=17
$ typeset -i16 y
$ y=x                                # let is not needed
$ print $y
16#11
$ print $y | cut -c4-10
11
```

Integer Arithmetic in the KornShell

```
# shell script to sum the integers from 1 to 5
total=0 ; n=0
while (( n <= 5 )) ; do
    let total=total+n
    let "n=n + 1"
done
print "sum from 1 to 5 is $total"
```

let is optional if variables are known to be integers:

```
integer total=0 n=0
while (( n <= 5 )) ; do
    total=total+n
    n=n+1
done
print "sum from 1 to 5 is $total"
```

let assigns values using any number base from 2 through 36:

\$ integer x=16#9	\$ typeset -i16 x
	\$ x=9
\$ let x=x+1	\$ let x=x+1
\$ print \$x	\$ print \$x
16#a	16#a

4-11

This is NOT a Korn shell capability; it is a HACK.

Students might experiment with 'bc' (basic calculator) by typing:

```
$ bc
```

and then perform some calculations.

'bc' reads stdin and writes stdout.

'scale' is the number of decimal places for the quotient.

In example at bottom: command substitution is used to define 'monthly'.

```
\n      newline needed to tell bc to use a scale of 2
\$$    literal $ followed by $ to get value of 'monthly'
If \ left off, $$ would yield the pid of this process.
```

Floating Point Arithmetic

```
$ print 5.5 \* 2.2 | bc      # * is a metacharacter
12.1
```

+	Addition
-	Subtraction
*	Multiplication
/	Division (default scale=0)
%	Modulo (remainder when divide)

```
$ print 5.5 / 2.2 | bc
2
```

```
$ print "scale=2\n5.5 / 2.2" | bc
2.50
```

```
$ cat shscript
princ=12000.0
int=.095
monthly=$(print "scale=2\n$princ * $int / 12.0" | bc)
print "Monthly interest is \$$monthly"
```

```
$ shscript
Monthly interest is $95.00
```

4-12

Curlyes must be used to get the value of an array element.

Otherwise - `$var[0]` would be `$var` followed by `[0]`.

Other 'typeset' options are given in ksh man page and in Bolsky p. 194, e.g.

-l	force lowercase
-r	mark variable as readonly

`RANDOM` yields a random integer from 1 to 32767.

`RANDOM%52` yields a random integer from 0 to 51.

This shell script fragment could be used to deal cards.

After loading the arrays, deal a card and zero that array element.

Variable Arrays

`var[0], var[1], ...` - The elements of an array

`${var[0]}, ${var[1]}, .` - The values of those elements
(curlies required)

```
$ cat myscript
integer n=0
typeset -u card      #force uppercase
for suit in clubs diamonds hearts spades
do
    for i in ace 2 3 4 5 6 7 8 9 10 jack queen king
    do
        card[n]="$i of $suit"
        n=n+1
    done
done
print ${card[RANDOM%52]}
```

```
$ myscript
JACK OF CLUBS
```

```
$ myscript
8 OF HEARTS
```

4-13

The syntax is:

set -A array_name list_of_values

The first array element has element number 0.
This will be a problem for FORTRAN programmers.

`${#var[*]}` number of elements of array var

`${#abc}` number of characters in the value of variable abc

See the List of Symbols at the end of the workbook and in
Bolsky p. 175.

Assigning Values to an Array

```
$ set -A var x1 x2 x3 x4
```

```
$ print ${var[1]}    #zero-based array  
x2
```

```
$ print ${var[0]}  
x1
```

```
$ ls /etc/m*  
/etc/mail.aliases      /etc/mknod             /etc/motd  
/etc/miscd             /etc/mkpasswd          /etc/mount  
/etc/mkfs              /etc/mkproto           /etc/mountd  
/etc/mklost+found      /etc/mop_mom
```

```
$ set -A file $(ls /etc/m*)
```

```
$ print ${file[2]}  
/etc/mkfs
```

```
$ print ${file[7]}  
/etc/mop_mom
```

4-14

Stdin for top 'while' loop is 'datafile'.

\ in print statement was needed for unknown reasons.
Apparently the shell has a a problem with #\$.

Note the curlies in the print statement `${line[n]}`.

Loading an Array from a Disk File

```
# loadarray      shell script to load a shell variable array
#               from the lines of a disk file (datafile).

integer n=0
while read line[n] ; do
    n=n+1
done < datafile

n=0
while [[ -n ${line[n]} ]] ; do
    print Line \#n - ${line[n]}
    n=n+1
done
```

```
$ cat datafile
datafile - line 1
datafile - line 2
datafile - line 3
```

```
$ loadarray
Line #0 - datafile - line 1
Line #1 - datafile - line 2
Line #2 - datafile - line 3
```

4-15

Another example of redirecting stdin for a 'while' loop.

Instead of separating fields of input with whitespace, we set the separator to be `:`.

Default value for IFS is: space tab newline

```
$ set | grep IFS
IFS= '
```

```
$ set | grep IFS | od -x
0000000 4649 3d53 2027 0a09
0000010
```

49 is I	46 is F	53 is S
3d is =	27 is '	20 is space
09 is tab	0a is newline	

For the example line of `/etc/passwd` in middle:

```
username=shutdown
uid=0
```

Avoid discussing all the fields of `/etc/passwd`, but

passwd = 13 encrypted characters (2nd field)

gid = 1 (4th field)

personal info = Multi-user shutdown (5th field)

home directory = `/usr/users/shutdown` (6th field)

login shell = `/bin/csh` (7th field)

There is only one special uid - zero.

The superuser is any user with uid=0.

There is nothing special about a uid < 100.

Commonly system administrators use number > 100 for the regular users.

There are many user accounts defined by UNIX to establish

ownership for the several system directories, e.g. root, daemon, bin.

When an new version is released, there are some new user accounts defined by UNIX that might conflict with regular user accounts.

Thus a wise system administrator starts regular user accounts at 100.

Changing IFS (Internal Field Separator)

```
# sysuser      shell script to list all users with
#              uid < 100

IFS=:
while read user junk uid junk2 ; do
    if [[ -n $user ]] ; then
        case $user in
            root)      ;;
            daemon)    ;;
            sys)        ;;
            bin)        ;;
            uucp)       ;;
            *)
                if [[ $uid -lt 100 ]] ; then
                    print $user - $uid
                fi
                ;;
        esac
    fi
done < /etc/passwd
```

```
$ grep ^shutdown /etc/passwd
```

```
shutdown:tJYgmuuYmbr5I:0:1:Multi-user shutdown:/usr/users/shutdown:/bin/csh
```

```
$ sysuser
```

```
operator - 0
field - 0
demo - 0
news - 8
sccs - 9
guest - 49
ris - 11
shutdown - 0
smitty - 13
pcguest - 50
nobody - -2
```

On Day 3 a complete discussion of Order of Evaluation will be provided. The next 2 pages serve to show the need for 'eval' without treating the order of evaluation topic completely.

The 2 examples at the top indicate the heart of the topic.

'eval' is needed to cause ksh to make:

Pass 1 substitute variable num
Pass 2 define the variable line2

'myscript' is modified several times to eliminate errors caused by the order in which the ksh evaluates its command lines.

In the example at the bottom:

line 4 fails because ksh made only one pass.

The command echoed by the error message looks valid, but the order of action has caused the problem

line 5 succeeds, but prints the value of 2 variables:

\$line NULL
\$num 1

Both line 4 and line 5 have problems.

Order of Evaluation (*eval*)

```
num=2
line$num="some characters"
```

The intent is: *line2="some characters"*

Command fails because of the order in which the shell evaluates command lines.

```
eval line$num="some characters"
```

Causes the shell to make a second pass over the command.
num is evaluated in the first pass.

line2 is assigned a value in the second pass.

```
$ nl myscript
```

```
1 integer num=1
2 while (( num <= 3 ))
3 do
4     line$num=$num***
5     print $line$num
6     num=num+1
7 done
```

```
$ myscript
```

```
myscript[4]: line1=1***: not found
1                                     from line 5 (num=1, line is null)
myscript[4]: line2=2***: not found
2
myscript[4]: line3=3***: not found
3
```

4-17

In the example at the top, we fix the problem on line 4.

Now two passes are made over line 4:

Pass 1 get value of variable num

Pass 2 define value of variable line1

line 5 still has a problem.

line 5 prints the value of 2 variables:

\$line NULL

\$num 1

In the example at the bottom, we fix the problem on line 5.

Now 2 passes are made over line 5:

Pass 1 get value of variable num

\ avoids getting value of variable line

Pass 2 get value of variable line1

```
$ nl myscript
```

```
1 integer num=1
2 while (( num <= 3 ))
3 do
4     eval line$num=$num***
5     print $line$num
6     num=num+1
7 done
```

```
$ myscript
```

```
1 from line 5 (num=1, line is null)
2
3
```

```
$ cat myscript
```

```
integer num=1
while (( num <= 3 ))
do
    eval line$num=$num***
    eval print \"$line$num
    # do not evaluate $line on pass 1
    num=num+1
done
```

```
$ myscript
```

```
1***
2***
3***
```

4-18

The purpose of this page is to illustrate some tools and techniques used in shell scripts.

We desire to get a list of filesystem mount points.

Avoid a detailed discussion of filesystems.

A filesystem is a disk partition that is mounted into the directory tree. The mount point is the point of attachment into the directory tree.

The 'df' output at the top shows 4 filesystems:

rz2a	/
rz0c	/usr
rz3c	/usr/users
rz1c	/var/adm/ris

This 'df' command executed on ULTRIX. Digital UNIX has a different output format. The appropriate calculations for Digital UNIX are left as an exercise.

The command in the middle is used to determine where in the 'df' output line the mount point directory begins.

The words dumped in hex by 'od' are byte reversed, i.e.

2f is / 64 is d 65 is e 76 is v

The / (2f) for the mount point occurs at the end of the 3rd line.

Its byte offset is 46. It is the 47th character in the line.

The last character on the 3rd line would be character #48.

'sed' is a stream editor, not an interactive editor.

It normally prints the lines it processes as it performs its duties.

-n suppresses those messages.

'\$' is syntax for the last line.

Note the single quotes to avoid confusing ksh with the \$ in the 'sed' command.

In the example at the bottom:

A \ is used to avoid ksh getting confused over the \$ in the 'sed' command. (just to be different)

'cut' is used to print characters 47 through 80. 80 was chosen because it was large enough to be the end of any line.

This result will be used in the shell script example on the next page.

Tools for Shell Scripts (*sed*, *od*, *cut*)

```
$ df
```

Filesystem node	Total kbytes	kbytes used	kbytes free	% used	Mounted on
/dev/rz2a	15343	12725	1084	92%	/
/dev/rz0c	945726	749032	102122	88%	/usr
/dev/rz3c	390670	220824	130779	63%	/usr/users
/dev/rz1c	622094	376154	183731	67%	/var/adm/ris

```
$ df | sed -n '3,$p' | od -x
```

```
0000000 642f 7665 722f 327a 2061 2020 2020 3120
0000020 3335 3334 2020 3120 3732 3632 2020 2020
0000040 3031 3338 2020 2020 3239 2025 2020 0a2f
0000060 642f 7665 722f 307a 2063 2020 2020 3439
0000100 3735 3632 2020 3437 3039 3233 2020 3031
0000120 3132 3232 2020 2020 3838 2025 2020 752f
0000140 7273 2f0a 6564 2f76 7a72 6333 2020 2020
0000160 3320 3039 3736 2030 3220 3032 3238 2034
0000200 3120 3033 3737 2039 2020 3620 2533 2020
0000220 2f20 7375 2f72 7375 7265 0a73 642f 7665
...
0000420 6f62 6b6f 000a 0000425
```

sed **-n** Suppress all normal output except that by **p** cmd
 3,\$p Print lines 3 to end of file

Locate the character position on the line where the mount point directory begins. (16 bytes in each line - 0a is \n).

```
$ df | sed -n 3,\$p | cut -c47-80
```

```
/
/usr
/usr/users
/var/adm/ris
```

4-19

Based on the example of the preceding page:

```
for d in / /usr /usr/users /var/adm/ris
```

Note the use of the wildcard in the string comparison inside [].

In the example:

```
$ mtpt ~sam/abc
```

```
MtPoint=/usr
```

```
MtPoint=/usr/users
```

execute in sequence.

When the 'for' loop finishes, MtPoint has the value of '/usr/users'.

```
$ cat mtpt
# mtpt shell script to display to mount point
#     directory for the file whose name is specified
#     on the cmd line.
for d in $(df | sed -n 3,\$p | cut -c47-80 )
do
    if [[ $1 = $d* ]]
    then
        MtPoint=$d
    fi
done
# When finish loop, MtPoint is lowest mount point on
# df output that is a substring of the full pathname
# given as $1
print $MtPoint
```

```
$ mtpt ~sam/abc
/usr/users
```

Positional parameters (\$1, \$2, ...) are set upon entrance to a shell script.

They can be changed using the 'set' command with the following syntax:

```
set positional_parameter_list
```

i.e. `set abc def xyz`

would set `$1=abc`
`$2=def`
`$3=xyz`

In the example at top, there are 5 fields in the output 'who | grep'. They become \$1, \$2, \$3, \$4, \$5.

In the example at the bottom:

When we change the IFS to colon, there are 7 fields in a line of /etc/passwd.

They become \$1 through \$7. The uid is \$3.

The backslash (\) in the print arguments avoids confusing ksh with an opening and no closing single quote.

It makes the single quote a literal, not an opening single quote.

Setting and Changing Positional Parameters

```
$ who | grep '^sam '  
sam      tty03    Dec 30 13:54
```

```
$ cat myscript  
set $(who | grep '^sam '  
print $# positional parameters  
print sam logged onto $2  
print sam logged in on $3 $4
```

```
$ myscript  
5 positional parameters  
sam logged onto tty03  
sam logged in on Dec 30
```

```
$ grep ^sam: /etc/passwd  
sam:tJSPBSw4uVSTs:910:2:S. Adams:/usr/users/sam:/usr/bin/ksh
```

```
$ cat myscript  
IFS=:  
set $(grep ^sam: /etc/passwd)  
print sam\'s uid is $3
```

```
$ myscript  
sam\'s uid is 910
```

In the example at the top, there are 8 fields in the 'ls -l' output.

When those 8 fields are deposited in the 'set' command in the first line of myscript, the command becomes:

```
set -rw-rw-r-- 3 sam 361 ...
```

The 'set' command takes options, e.g. `set -x`

The leading dash in the first set argument leads 'set' to look for -r option.

None exists, so an error results. But that is not what we wanted.

In the example at the bottom, we solve the problem by placing a dash as the first argument to the 'set' command in myscript.

This tells 'set' that it will not receive any options.

```
$ ls -l ~sam/abc
-rw-rw-r-- 3 sam          361 Dec 29 10:24 /usr/users/sam/abc
```

```
$ cat myscript
set $(ls -l ~sam/abc)
print ~sam/abc has $2 links.
```

```
$ myscript
myscript: -rw-rw-r--: bad option(s)
~sam/abc has links.
```

Leading *-r* is interpreted as option.

```
$ cat myscript
set - $(ls -l ~sam/abc)
print ~sam/abc has $2 links.
```

Single dash suppresses interpretation of options.

```
$ myscript
~sam/abc has 3 links.
```

In the next 2 pages, 3 functions are presented that repeat the functionality of ma, ga, and la from Day 2AM.

Instead of using a disk file, these functions m, g, and l will define an exported variable.

This solves the problem 'ga' has if the same token appears twice in the disk file.

The functions perform as follows:

m abc defines an environment variable abc whose value
 is the current directory pathname

g abc changes the current directory to the value of the variable
 abc

l lists all environment variables defined by 'm'.

If no argument is provided,

m uses an environment variable whose name is the basename
 for the current directory

g print an error message.

In examples at bottom,

m myetc executes:
 export myetc=cd /etc

m executes 2 passes over the command
 Pass 1 evaluate the variable PWD
 Pass 2 execute the export command

Function and Shell Script Examples

```
# mgl.fn          shell script to define functions m, g and l
#                  that will remember the current directory and
#                  enable returning to that directory at a later
#                  time
# m              Mark the current directory with the "word" given as
#                  argument.
#                  Define an exported variable whose value is a
#                  "cd" command.
```

```
function m {
    Usage="Usage: m word"
    case $# in
        0)  eval export \"$(basename $PWD)=cd $PWD\"
            ;;
        1)  export "$1=cd $PWD"
            ;;
        *)  print "Too many arguments." >&2
            print $Usage >&2
            ;;
    esac
}
```

```
$ cd /etc
```

```
$ m myetc
```

Defines an environment variable **myetc** whose value is **cd /etc**.

```
$ cd /usr/sys
```

```
$ m
```

Defines an environment variable **sys** whose value is **cd /usr/sys**.

'env' list the values of the exported (environment) variables.

In the first example in the middle,

g myetc greps in the 'env' output for a line that
begins with myetc=

If it finds such a line, it executes 2 passes over eval "\$\$1"

Pass 1 evaluate the variable \$1 (myetc is its value).

The eval "\$\$1" command is now \$myetc

Pass 2 executes the command which is the value of the
variable 'myetc' (cd /etc is its value)

The function l greps in the 'env' output for lines that contain the
characters '=cd '.

'sed' is used to substitute a tab in place of the characters '=cd '.

The -n option should not be used here.

We depend upon the report of the substituted line that 'sed' gives
when the -n option is not provided.

```
# g      Goto the directory associated with the "word" given as
#        argument.
#        Execute the value of "word".
```

```
function g {
    Usage="Usage: g word"
    case $# in
        1)  if env | grep -q "^$1=" ; then
                eval "$1"
                print "New current directory is $(pwd)"
            else
                print $Usage >&2
                print "Invalid word argument - \"$1\"." >&2
                print "Use \"l\" to list marks and dirs." >&2
            fi
            ;;
        *)  print "Too many arguments." >&2
            print $Usage >&2
            ;;
    esac
}
```

```
$ g myetc
New current directory is /etc
```

```
$ g sys
New current directory is /usr/sys
```

```
# l      List the words and their values.
function l {
    env | grep '=cd ' | sed 's/=cd /    /' # tab inside //
}
```

Substitute tab for =cd.

```
$ l
myetc  /etc
sys    /usr/sys
```

Because of the length of the Day 2PM lecture, it might be advisable to delay the discussion of the remaining examples until the morning of Day 3.

The next 2 pages contain a single script that replays a shell command typed on the command line.

It is the most difficult script that will be discussed.

It is suggested that you begin by explaining the purpose of the script and its options. It might be helpful to show some examples and explain their workings, e.g.

```
replay ls -l
replay -s 10 ls -l
replay -d 3 ls -l
replay -t ls -l
```

Continue the explanation by looking at the commands at the bottom of the 2nd page that actually perform the work:

```
eval $*
```

with the options shifted off the cmd line.

Then begin at the beginning of the script and discuss the commands executed when we type:

```
replay -d 3 ls -l
```

Delay=3 and the '-d 3' are shifted off.

```
$# = 2          and $* = 'ls -l'
```

We break on the default case and continue onto the 2nd page.


```

# replay shell script to invoke a command repeatedly. The
#       command is specified as a cmd line argument.
# Options: -d   delay interval (default = 5 sec)
#           -s   size in number of lines
#           (default = 23 or LINES var)
#           -t   show the tail of the display

Delay=5                # delay interval
Tail=""                # show tail of display
Size=${LINES:-23}      # size of display
Usage="Usage: $0 [-d delay] [-s size] [-t] command"

trap 'clear; exit' 1 2 3 15
while [[ $# -gt 0 ]] ; do
    case $1 in
        -d) if [[ -n $2 ]] ; then
                Delay=$2
                shift ; shift
            else
                print "d option requires an argument" >&2
                print $Usage >&2
                exit
            fi
            ;;
        -s) if [[ -n $2 ]] ; then
                Size=$2
                shift ; shift
            else
                print "s option requires an argument" >&2
                print $Usage >&2
                exit
            fi
            ;;
        -t) Tail=True
                shift
                ;;
        -*) print "Unrecognized option \"$1\"" >&2
                exit 1
                ;;
        *) break          # saw first non-option argument
                ;;
    esac
done

```

4-25

`let $Delay` will perform arithmetic on its arguments.

If the 'let' arguments are non-numeric, 'let' will exit with a nonzero status.

We provide no calculations to perform; we only check the status return.

If 'let' returns 0, we do the null statement (mentioned here for the first time).

In the middle of the page, the command is executed once to verify that it is a valid command. Its output is discarded.

The 'q' command to 'sed' takes only 1 argument
- the number of lines to print from the top of the file.

The command given is equivalent to:

```
sed -n 1,${SIZE}p
```

```

# Verify that Delay is numeric
if let $Delay > /dev/null 2>&1 ; then
    :          # null statement
else
    print "d option argument not an integer" >&2
    exit 1
fi

# Ensure a command was supplied
if [[ $# -eq 0 ]] ; then
    print "No command supplied" >&2
    print $Usage >&2
    exit 1
fi

# Ensure that command is valid
if eval $* > /dev/null 2>&1 ; then
    :          # null statement
else
    print "$* is not a valid command" >&2
    exit 1
fi

# Run the command repeatedly
while true ; do
    clear

    if [[ -n $Tail ]] ; then
        eval $* | tail -$Size
    else
        eval $* | sed ${Size}q
    fi

    sleep $Delay
done

```

'pdir' takes as its command line a string of characters to be sought in a PhoneList file.

The PhoneList file must be created by some other shell script or a text editor.

Match = lines of the PhoneList file that contain the command line arguments.

"\$*" in case Betty Smith is the search string on the command line.

awk commands:

-F/ set / to be the field separator

length() special awk function to return the length of a field

printf() special awk function to do a formatted print.
Its first arg is a format string.
The remaining args are fields to be printed in the format specified by the 1st arg.

Consider PhoneList file lines like:

Betty Smith//306-6429

Ray Jones/234-7654//

```

# pdir  shell script to format and display entries from a file
#       containing names and phone numbers.
#
#       File Format: Name/Home Number/Work Number
PhoneList=~/.phonelist
# Verify that the phone file exists
if [[ ! -r $PhoneList ]] ; then
    print "Phone List file not found - $PhoneList" >&2
    exit 1
fi

# Find lines in the file that match my arguments
Match=$(grep "$*" $PhoneList)

# If no match
if [[ -z $Match ]] ; then
    print Unable to find number for $* >&2
    exit 2
fi

print $Match | awk -F/ '{          # / is field separator
    if (length($3) != 0) {
        if (length($2) != 0)
            printf "%-s\t%s(H)\t%s(W)\n", $1, $2, $3
        else
            printf "%-s\t%s(W)\n", $1, $3          #no home number
    } else
        printf "%-s\t%s(H)\n", $1, $2          #no work number
    },'

```

awk program	\$1	Field 1 = Name
	\$2	Field 2 = Home Number
	\$3	Field 3 = Work Number
	%-s	Left justified
	\t	Tab character

Chapter 5: Day 3

At the top part of the page, background examples dealing with stdin inside shell scripts are provided as a lead-in to here documents.

The End-of-File character for here documents can be any string. Its value follows the <<. One must use characters that will not occur in the here document on a line by themselves.

Several good examples illustrating the use of here documents are provided on the following pages.

A review list of redirection characters is provided at the bottom of the page.

Here Documents

Sending mail from a shell script:

```
$ cat myscript
...
mail sam mary                # letter from stdin
...

$ myscript                   # stdin is terminal
                             # letter from terminal

$ myscript < abc             # stdin is ./abc
                             # letter from ./abc
```

```
$ cat myscript
...
mail sam mary < xyz          # letter from ./xyz
...

$ myscript                   # stdin is terminal
                             # letter from ./xyz

$ myscript < abc             # stdin is ./abc
                             # letter from ./xyz
```

Desire the letter to be the following lines of the shell script.

```
$ cat myscript
...
mail sam mary << EOD
    Welcome to the Club
    Meetings held every Tuesday.
EOD
...
```

>	Redirect standard output
<	Redirect standard input
2>	Redirect standard error
>>	Append to standard output
2>>	Append to standard error
<<	Here document

5-3

In the example at the top, the 'ed' commands for the example are:

```
1,$s/tom/elaine/g      substitute elaine for tom in every line from the first to the
                        last.
                        g means make all changes on the line, not just the first
                        occurrence.
w                        write the output file
q                        quit
```

The backslash (\) in the first 'ed' command tells ksh not to look for the value of variable s, but to pass the \$ to 'ed'.

\$2 and \$3 do not get a backslash, because we want ksh to make the substitution.

ed - similar to sed -n (don't show normal output)

Tabs are provided in front of the 'ed' commands to set them off from the shell commands of the script.

In the example at the bottom, there is no need for the dash in <<.

```
In shell script    $1=John Smith
                   $2=123 Elm Street
                   $3=Detroit, MI 48086
```

date formats	%a	day of the week, e.g. Fri
	%r	time of day in AM/PM format

Note the use of double quotes to bind the %a and %r.

```
$ cat eghere
```

```
ed - "$1" <<- EOD
    1,\$s/$2/$3/g
    w
    q
```

```
EOD
```

```
$ eghere userfile tom elaine
```

Edit **userfile** and substitute *elaine* for every occurrence of *tom*.

Line 1 to last line substitute globally on line.

ed - Suppress printing of messages by **ed** commands.

<<- Ignore leading tabs in here document.

```
$ cat formletter
```

```
cat <<- +++
On $(date +"%a, at %r"), I sent a notice to:
$1
at the mailing address of:
$2, $3
+++
```

```
$ formletter "John Smith" "123 Elm Street" \
"Detroit, MI 48086"
```

```
On Fri, at 10:55:40 AM, I sent a notice to:
John Smith
at the mailing address of:
123 Elm Street, Detroit, MI 48086
```

5-4

A here document is used in this example to write a child shell script that is spawned in background.

The child script will kill its parent in 15 seconds.

At the bottom of the parent script, the parent kills the child.

This script is a race to see who can kill whom first.

The name of the child script is /tmp/time_resp3722 assuming pid=3722.

The backslash (\) in its second line is to cause the 2nd line to be

```
kill $PPID
```

rather than

```
kill 234 (login ksh pid = 234).
```

+++ is the here document end-of-file string.

\$! is the pid of the most recent background process. See the List of Symbols at the end of the workbook and in Bolsky p. 175.

```

#time_resp      shell script to prompt the user to input their
#               name and address. The name and address will be queried
#               separately. After both responses, the shell script
#               will output the answers. If the user does not respond
#               within 15 seconds, the command will exit to the
#               calling process, printing an error message.

trap '
    print "Parent is being killed by its child process." ;
    exit
' TERM

cat > /tmp/$0$$ <<- +++
    sleep 15
    kill \$$PPID
+++

ksh /tmp/$0$$ &
read name?"Name?: "
read addr?"Address?: "

kill $!
rm /tmp/$0$$

print "Name:\t$name"
print "Address:\t$addr"

```

5-5

The default value of the file descriptor for the 'read' command is 0 (stdin).

The command used to close a file is cryptic; one does it that way and it works.

It does not follow from previous functionality.

The actual number of file descriptors is configurable by the system administrator.

Reading/Writing Files from a Shell Script

```
# readfile      Open a file for reading using a file
#               descriptor other than 0,1,2.
# open file1 for input using file descriptor 3
exec 3< file1

# read from file descriptor 3
while read -u3 line
do
    print "file1 contents: $line"
done

# close file descriptor 3
exec 3<&-
```

```
$ readfile
file1 contents: This is the first line of file1.
file1 contents: This is the second line of file1.
file1 contents: This is the third line of file1.
```

exec n<	Open file for read using file descriptor <i>n</i> .
exec n>	Open file file for write using file descriptor <i>n</i> .

ULTRIX	64 file descriptors 0-63
Digital UNIX	4096 file descriptors 0-4095

File descriptor 0	standard input
File descriptor 1	standard output
File descriptor 2	standard error

5-6

This example is straightforward after the previous page.

'print' writes to stdout.

We redirect stdout to the same as file descriptor 3, so print writes to file3.

In order to read from one file and write to another:

```
exec 3< infile
exec 4> outfile
read -u3 line
print $line >&4
```

> redirects sdtout
if noclobber is set (set -o noclobber) and the file exists,
an error is generated.

>| redirect stdout and ignore noclobber.

exec 3>> open file for append


```
# writefile    Open a file for writing using a file
#              descriptor other than 0,1,2.
# open file3 for output using file descriptor 3
exec 3> file3

print "First line of file3"          >&3
print "Second line of file3"         >&3
print "Third line of file3"          >&3

# close file descriptor 3
exec 3>&-
```

```
$ writefile
$ cat file3
First line of file3
Second line of file3
Third line of file3
```

```
$ set -o noclobber
$ writefile
writefile[4]: file3: file already exists
```

```
exec 3>| file3
```

Open file for write and override noclobber.

5-7

Review the technique used in processing command line options in previous scripts, e.g. replay.

```
case $1 in
    -r)      ...
```

The examples will carry the burden of explaining option processing.

3 options: a, b, c

b: b option must supply an argument.

The last 2 lines indicate that the command line to be processed could be specified with the 'getopts' command.

The default command line arguments for 'getopts' is \$*.

The default value list for the 'for' loop is \$*.

Command Options Processing (*getopts*)

```
# opt1          illustrate processing command line
#              options with getopts
while getopts ab:c opt
do
    case $opt in
        a)      print "option a "
                ;;
        b)      print "option b arg is $OPTARG"
                ;;
        c)      print "option c "
                ;;
    esac
done
```

```
$ opt1 -b 33 -c -a
option b arg is 33
option c
option a
```

```
$ opt1 -ab dog
option a
option b arg is dog
```

`getopts` *option_string* *name*

Parses the positional parameters.

Places option letter into variable *name*.

Colon in *option_string* following option indicates an argument must be supplied with that option.

\$OPTARG gets value of the option argument.

`getopts` *option_string* *name* *arg1* *arg2* ...

Parses the arguments following *name*.

5-8

The leading colon to 'getopts' does not work in the ksh of ULTRIX. The documentation for ULTRIX indicates that it should, but 'opt2' will not perform as advertised.

```

# opt2      leading colon in option string allows
#           detection of invalid options and missing
#           arguments.
#           For invalid options, a name value of ?
#           will be returned and OPTARG will be set
#           to the invalid option.
#           For missing arguments, a name value of
#           : will be returned and OPTARG will be
#           set to that option.

```

```

while getopts :ab:c opt
do
    case $opt in
        a)      print "option a "
                ;;
        b)      print "option b arg is $OPTARG"
                ;;
        c)      print "option c "
                ;;
        \?)     print "unknown option $OPTARG "
                ;;
        :)      print "missing argument for $OPTARG"
                ;;
    esac
done

```

```

$ opt2 -a -z -b5
option a
unknown option z
option b arg is 5

```

```

$ opt2 -c -b
option c
missing argument for b

```

```

$ opt2 -b -c
option b arg is -c

```

5-9

We are adding to the behavior of 'getopts'.

This does not replace any functionality from previous pages, i.e.

```
$ opt3 -b 56  
option b arg is 56
```

```

# opt3          If an option is preceded by +, rather
#               than -, a + is prepended to the value of
#               opt.
while getopts ab:c opt
do
    case $opt in
        a)      print "option a "
                ;;
        b)      print "option b arg is $OPTARG"
                ;;
        c)      print "option c "
                ;;
        +b)     print "option +b arg is $OPTARG"
                ;;
        +c)     print "option +c "
                ;;
    esac
done

```

```

$ opt3 +c +b mouse
option +c
option +b arg is mouse

```

```

$ opt3 -a +b 17
option a
option +b arg is 17

```

```

$ opt3 +cb 45
option +c
option +b arg is 45

```

5-10

Index begins with one,
i.e. in first example

Index	1	-a	Same as positional parameter number
	2	dog	
	3	cat	

`shift OPTIND-1` works (without the \$)

It seems like it should not, but it is the form documented in Bolsky.


```

# opt4      After processing options, shift the
#           options off the command line.
#           After processing options, OPTIND is the
#           index of the next command line
#           parameter.
while getopts ab:c opt
do
    case $opt in
        a)    print "option a "
              ;;
        b)    print "option b arg is $OPTARG"
              ;;
        c)    print "option c "
              ;;
    esac
done
print "OPTIND = $OPTIND"
shift $OPTIND-1
print "$# parameters remain - $"

```

```

$ opt4 -a dog cat
option a
OPTIND = 2
2 parameters remain - dog cat

```

```

$ opt4 -cb dog cat
option c
option b arg is dog
OPTIND = 3
1 parameters remain - cat

```

```

$ opt4 -bc dog cat
option b arg is c
OPTIND = 2
2 parameters remain - dog cat

```

5-11

The ksh pattern matching operators can be very obscure if not approached properly.

At the top of the page, metacharacters are reviewed.

In the middle, a question is posed.

How do we match abc34 and abc56?

`abc[35][46]` is no good since it also matches abc36 and abc54.

We have hopefully developed a need for this new class of metacharacters.

Pattern Matching Operators

`abc*`

Matches characters *abc* followed by 0 or more characters.

`abc?`

Matches characters *abc* followed by exactly 1 character.

`abc[13579]`

Matches characters *abc* followed by exactly 1 character which is 1, 3, 5, 7, or 9.

`abc[!246]`

Matches characters *abc* followed by exactly 1 character which is NOT 2, 4, or 6.

Desire to match *abc34* or *abc56*.

Consider: `abc[35][46]`

`abc@(34|56)`

Matches characters *abc* followed by 34 or 56.

5-12

The examples shown on the page provide only 2 patterns: 34, 56.

An indefinite number of patterns can be specified, e.g.

```
abc@( 34 | 56 | def | x | 789 )
```

The example at the bottom matches:

foo.	foo.dat	foo.file
bar.	bar.dat	bar.file
bam.	bam.dat	bam.file

`@(pattern1|pattern2|...)`

Matches exactly one pattern.

`abc@(34|56)`

Matches *abc34* and *abc56*.

`?(pattern1|pattern2|...)`

Matches 0 or 1 pattern.

`abc?(34|56)`

Matches *abc*, *abc34*, and *abc56*.

`*(pattern1|pattern2|...)`

Matches 0 or more patterns.

`abc*(34|56)`

Matches *abc*, *abc34*, *abc56*, *abc3456*, *abc5634*,
abc3434, *abc5656*, ...

`+(pattern1|pattern2|...)`

Matches one or more patterns.

`abc+(34|56)`

Is same as **`abc*(34|56)`** except that *abc* is not matched.

`!(pattern1|pattern2|...)`

Matches all strings except those in patterns.

`abc!(*.dat|*.txt)`

Matches *abc* followed by any characters except *.dat* or *.txt*.

Example: `@(foo|bar|bam).?(dat|file)`

On the next 4 pages are 3 programming gimmicks.

A lock file is used to guard access to /etc/passwd. The superuser edits the passwd file with 'vipw' which creates a lock file to insure that 2 superusers cannot be editing the file simultaneously.

Commonly the contents of a lock file would be the pid of the process holding the lock.

This scheme allows a tiny window in which several users could gain simultaneous access to the data file.

User #1 finds that the lockfile does not exist, but loses cpu control to User #2 between the 'while' and the 'print \$\$'.

User #2 executes the 'while' and finds that the lock file does not exist, so he creates one and accesses the data file.

User #2 blocks while accessing the data file and User #1 continues and creates the lock file again.

Both User #1 and User #2 have access to the data file.

A solution to this problem (if one exists) is left as an exercise.

Using a Lock File to Synchronize Access

The idea is to allow only one process to access a data file at any one time.

1. Create a lock file as we begin to access the data file.
2. Remove the lock file when we have finished accessing the data file.
3. If lock file exists, do not access the data file.

```
while [[ -a /tmp/MyLockFile ]]
do
    sleep 2
done

print $$ > /tmp/MyLockFile      # pid into lock file

# access the data file

rm /tmp/MyLockFile
```

5-14

Normally only the superuser will have write permission to `/etc/passwd`, however there could be exceptions.

The way of detecting superuser shown at the bottom looks at the first character of the uid number from the `id` command. Only the superuser could have such a first character equal to 0.

Note that superuser and root are not synonymous.

Superuser is any process with `uid=0`, no matter the username.

'root' is an account distributed with UNIX that has a `uid=0`.

The uid of 'root' could be changed by editing `/etc/passwd`, thus making 'root' a non-superuser.

Testing for Superuser

Superuser is any process with `uid = 0`.

root is a common superuser account.

The superuser is the system administrator and has no limits to his privilege, e.g. superuser can read/write any file.

Test #1: Only superuser has write permission to `/etc/passwd`:

```
if [[ -w /etc/passwd ]] ; then
    ...some privileged commands...
fi
```

Test #2: Look at the output from the `id` command:

```
$ id
uid=910(sam) gid=2(ins)
```

```
if [[ $(id | cut -c5) -eq 0 ]] ; then
    ...some privileged commands...
fi
```

The setuid/setgid bits will not work for shell scripts.

The next 2 pages describe a gimmick to provide setuid capability through an application program 'sush'.

This first page explains the concept of setuid bits.

The uid of the child process running a program matches that of the program file NOT the user running the program because the setuid permission bit is set for the program file.

The 'passwd' command is such a program.

Mary does not have write permission to /etc/passwd, but is still able to modify the password field of her entry in that file.

The 'chmod' command at top sets both the setuid and setgid bits.

4755 sets only the setuid bit.

2755 sets only the setgid bit.

The third bit is the sticky bit - of no interest to shell programmers and of little interest in these modern days to anyone else.

In the old days, before virtual memory, when a program ran, the entire program was DMA loaded into memory. If the sticky bit was set, the memory was not freed when the program terminated, so the 2nd and subsequent program runs were speeded up since the program was stuck in memory.

'ls -l' uses the 'x' field of the owner permissions to designate both 'x' permission for owner and setuid bit.

s	means	both ('x' and setuid)
S	means	only setuid
x	means	only 'x'

Similar functionality is associated with the 'x' for group and the setgid bit. The setgid bit is less often used than the setuid bit.

One can set the setuid bit for a shell script, but it is ignored.

Who can set the setuid bit for a program?

the owner of the program file -or
the superuser (who can do anything)

Setuid Bit for Shell Scripts

```
$ chmod 6755 prog
```

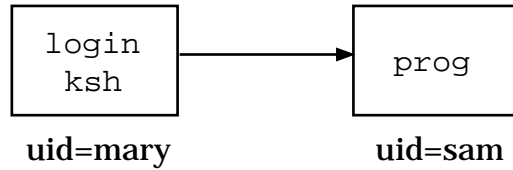
```
$ ls -l prog
```

```
-rwsr-sr-x 1 sam    ins      27916 Jan 4 10:55 prog
```

```
$ whoami
```

```
mary
```

```
$ prog
```



```
$ ls -l /usr/ucb/passwd
```

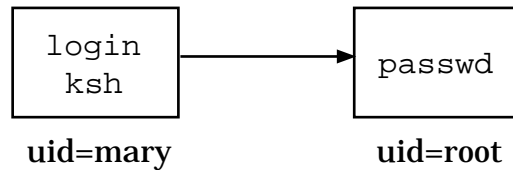
```
-rws--x--x 3 root bin    16384 Aug 8 19:49 /usr/ucb/passwd
```

```
$ passwd
```

```
Old password:
```

```
New password:
```

```
Verify:
```



The *setuid* and *setgid* special permissions have NO effect on the execution of shell scripts.

5-16

Avoid a detailed discussion of the C language program.

It is important that the name of the program 'sush' not be changed to contain less than 3 characters.

We are replacing the program name with 'ksh'; we need at least 3 characters in the program name.

Note the way the effective uid is displayed by 'id'.

The real uid is gar.

The effective uid is root.

That is the one that determines file permission category.

Avoid expanded discussion of real/effective uids.

Note that the 2nd check for superuser on p. 5-14 would fail if called from a setuid script.

```

/* sush.c      program to run a KornShell script
               whose name and arguments are given on
               the cmd line.                                */
main(argc, argv)
    int argc;
    char *argv[];
{
    strcpy(argv[0], "ksh");
    if (execvp("ksh", argv) == -1)
        perror("sush error"), exit(1);
    exit(0);
}

```

The following commands must be typed as superuser:

```

# cc -o sush sush.c
# chmod 4750 sush
# ls -l sush
-rwsr-x--- 1 root    system  27916 Jan 4 10:55 sush

```

The following commands are typed as 'mary':

```

$ cat myscript
id
if [[ -w /etc/passwd ]] ; then
    print "have write access to /etc/passwd"
fi
print $# positional parameters - $*

$ sush myscript abc def
uid=910(gar) gid=2(ins) euid=0(root)
have write access to /etc/passwd
2 positional parameters - abc def

```

5-17

Parameter expansion in #5 include:

- the expansion of variables
- the default parameter value on p.2-7
- other parameter expansions listed in Bolsky p. 171.

In 1st example at bottom,

- USER is substituted at step 5, but
- the tilde expansion at step 3 has already occurred.

In 2nd example at bottom,

- the command substitution is performed at step 4 but
- the tilde expansion at step 3 has already occurred.

Placing 'eval' prior to these commands causes ksh to make a 2nd pass over the command.

Parameter (or command) substitution occurs in the first pass and tilde expansion in the 2nd pass.

The commands work if 'eval' precedes them.

KornShell Command Line Parsing

The KornShell processes the command line in the following sequence:

1. The command is split into tokens and these are organized in three categories:
 - a. Variable assignment words
 - b. Command words
 - c. I/O redirections
2. Command words are tested and alias substitution is performed.
3. Tilde Expansion is performed
4. Command substitution is done
5. Parameter expansion is performed.
6. Results of command substitution are split into words.
7. Pathnames are expanded in command words and I/O redirections.
8. Quote characters are removed.

Command word precedence is as follows:

1. Alias
2. Function
3. Built-in
4. Program

Consider the following:

```
$ cd ~$USER          # '~' expanded before variable
~mary: No such file or directory
$ eval cd ~$USER      # eval forces second pass
$ ls ~$(whoami)       # '~' expanded before cmd subst.
~mary: No such file or directory
$ eval cd ~$(whoami)  # eval forces second pass
```

5-18

The co-process is blocked at 'read file', waiting for text from its stdin.

When 'print -p savefile' is executed by the parent shell,
text is passed to the co-process and
the file 'savefile' becomes the value of 'file'.

The file 'savefile' is copied to the 3 directories in the value list of the
'for' statement.

The co-process prints a final message and terminates.

A 'while' loop could be coded into the co-process to avoid termination
of the co-process after one text string is read and processed.

Co-processes

A co-process is a background job which is started in such a way that you can communicate with it using **read** and **print** commands.

```
$ cat co-script
# co-script
read file

for dir in ~/save /usr/local/save /public/save
do
    cp $file $dir
done

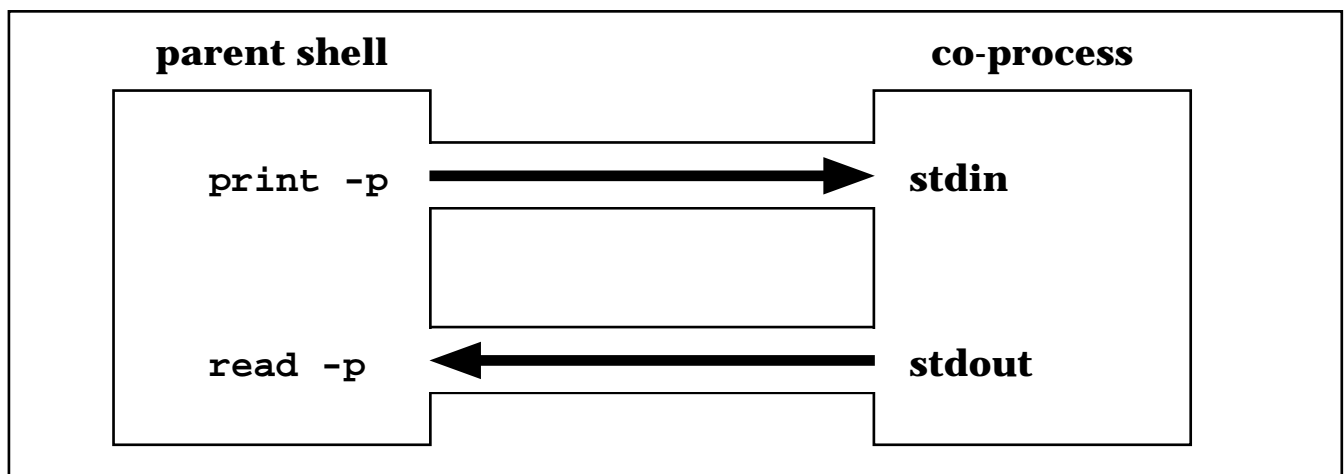
print "$file: Saved successfully"

$ co-script |&
$ print -p savefile
$ read -p
$ print $REPLY
savefile: Saved successfully
```

program | & starts the co-process.

The co-process' stdin and stdout are connected to file descriptor *p* of the parent shell.

The parent shell uses **read -p** and **print -p** commands to exchange information with the co-process.



5-19

file descriptor 3	read from program1
file descriptor 4	write to program1
file descriptor 5	read from program2
file descriptor 6	write to program2

Using Multiple Co-processes

To access more than one co-process, assign another file descriptor to *p*.

```
$ cat program1
# program1
while true ; do
    read text
    print "$text - this is from program1"
done
```

```
$ cat program2
# program2
while true ; do
    read text
    print "$text - this is from program2"
done
```

```
$ program1 |&
$ exec 3<&p
$ exec 4>&p
```

```
$ program2 |&
$ exec 5<&p
$ exec 6>&p
```

```
$ print -u4 hello          #write to program1
$ read -u3 repl1           #read from program1
$ print $repl1
hello - this is from program1
```

```
$ print -u6 howdy          #write to program2
$ read -u5 repl2           #read from program1
$ print $repl2
howdy - this is from program2
```

5-20

Example of using co-processes.

The 'preserve' shell script is started from .profile at login with:

```
~/preserve |&
```

tf = obsolete.120295.3722 on Feb 12, 1995, from pid=3722

When logout occurs, this co-process that was started from .profile at login will be sent the HUP signal.

The signal handler executes to save the trashed files and to cleanup the trashcan directory.

Single quotes are essential in the signal handler to avoid expansion of the * wild card as the signal handler is established with the 'trap' command.

The 'if' statement checks the status return from the 'cp' command.

The file is removed only if 'cp' is successful.

Errors from the 'cp' command are written to stdout which is the channel back to the calling co-process.

The 'print' command writes a message to stdout which is the channel back to the calling co-process.

'save' function

'for' loop will occur for all arguments on the command line.

Those arguments can be full or relative pathnames.

The first 'print' command writes the full pathname of the argument to the function to the co-process.

```
$ ( - command substitution for [[ && print  
[[ $1 != /* ]]- is the first character in $1 a /  
&& - execute the second command only if the  
- first command is successful  
print $PWD/ - the current directory name followed by /
```

If the function is called with a full pathname argument, the command substitution prints nothing. \$file is a full pathname.

The 'read' command reads from the co-process one of the following:

1. error message from the 'cp' command in 'preserve'
2. '\$file saved' success message

```

# preserve - co-process started in .profile. Save
# files in ~/trashcan and delete the original if
# successful. Print a success message or an error
# message to stdout. On exit, archive all files in
# ~/trashcan and delete the saved versions.

# Signal handler -- archive files, compress archive,
# delete saved versions.

tf=obsolete.$(date +%d%m%y)$$      # name for archive

trap 'cd ~/trashcan
      tar -cf ~/$tf * ;\
      compress ~/$tf ;\
      rm * ;\
      exit' TERM HUP INT STOP QUIT

while true      # save the files and delete originals.
do
    read file
    if cp $file ~/trashcan 2>&1 # errors to stdout
    then
        rm $file
        print "$file saved"      # msg to stdout
    fi
done

```

```

# save.fn - Function which writes full pathname of
# files passed as command line arguments to the
# preserve co-process. It reads a message from the
# co-process and prints it to stdout.

function save {
    for fil
    do
        print -p $([[ $fil != /* ]] && print $PWD/) $fil
        read -p stat
        print $stat
    done
}

```


Readability and Maintainability

Here are some stylistic suggestions for writing KornShell scripts:

- Always include a comment at the beginning of the script that tells what task the script was written to perform.

```
# cleanup - script to search my directories and
#   eliminate unwanted, space-wasting files.
...
```

- Organize the script into sections of related commands with a descriptive comment at the beginning of each section.

```
...
# set up unique file names for log files
...
# create background processes
...
# check for errors
...
```

- Explain any obscure syntax with a comment. If it took you more than a second or two to figure out how to do something clever in the script, you won't be able to remember how or why if you have to change it later.

```
# if pathname is relative, prefix pathname with
# the working directory string
path=$( [[ $1 != /* ]] && print $PWD/ )$path
```

- Place a blank line before and after a compound statement. These include **if**, **while**, **until**, **for**, **select**, and **case** statements.

```
id

if [[ -w /etc/passwd ]]
then
    print "have write access to /etc/passwd"
fi

print $# positional parameters - $*
```


- Use spaces or tabs to indent the bodies of compound statements. Use the same indentation level for separate lines of the same logic statement, e.g. **if**, **then**, **fi** in the example below.


```
while true
do
    read file

    if cp $file ~/trashcan
    then
        rm $file
        print "$file saved"
    fi

done
```

- Set up compound statements consistently.

```
if [[ $# -eq 0 ]]
then
    print "Usage: $0 filename"
    exit 1
fi
...
if [[ ! -x $file ]] ; then
    print "$file: Cannot execute"
fi
```

 Choose one or the other

- Avoid combining unrelated commands on a line using the semicolon operator.

```
cd ~/trashcan ; userlist=$(users)      # bad
...
cd $1 ; print "listing of $PWD" ; ls    # ok
```


Performance

Here are a few tips for improving the performance of your KornShell scripts:

1. Use KornShell built-ins rather than external programs whenever possible.
2. Use aliases rather than functions whenever possible
3. Keep the `$ENV` file small. Remember, this file executes for all child processes.
4. Use tracked aliases -- set the *trackall* option.
5. Set the *nolog* option to keep any function definitions out of the history list.
6. Use autoload functions.
7. Avoid creating child processes unnecessarily. Among other techniques, consider:
 - a. Using the `.` (dot) command to execute scripts where possible
 - b. Using `{}` to group commands rather than `()` where possible
8. Define integer variables when the value is to be manipulated rather than displayed.
9. Use the semicolon to group related commands. The shell can process two or three commands after reading a single line from the script file. BEWARE! While this is generally good for performance, it can be very bad for readability.
10. The presence of comment lines seems to have little or no effect on performance of a script, so you can't use that as an excuse not to document what your scripts do.
11. Use the **time** command to verify that your strategies have in fact enhanced performance.

IFS is set for the 'set' command in the 'while' loop.

A complete list of options that can be set is in Bolsky p. 197.

Stdin for the entire 'while' loop is /etc/passwd.

The 'set' command in the loop sets the positional parameters to the fields of the passwd file.

The fields of the passwd file are:

Field 1	username
Field 2	password
Field 3	user id
Field 4	group id
Field 5	personal information
Field 6	home directory
Field 7	shell

When the 'while' loop has finished, we have read every line of /etc/passwd.

The value of 'max' will be the largest user id.

Shell Script Examples

```
# uidmax          print the largest uid in /etc/passwd.
integer max=0
IFS=:              # for set command below

set -o noglob      # disable filename expansion in case
                  # there are * in passwd entries.

while read entry   # stdin is /etc/passwd
do
    set $entry     # passwd entry fields become $1,$2,...

    if (( $3 > max ))    # then this uid is larger
    then
        max=$3
    fi

done < /etc/passwd    # stdin for while loop

print "Largest uid in /etc/passwd is $max"
```

The only difference between 'replay_opt' and the example from Day 2, 'replay', is that 'replay_opt' uses 'getopts' to process the command line options, instead of 'case' and 'shift'.

The 'd' and 's' options take a argument.

The leading colon to 'getopts' will not work in ULTRIX.

The leading colon causes the following to be reported.

The :) choice indicates that an option that requires an argument was used wuthout an argument.

The option is \$OPTARG.

The ?) choice indicates that an invalid option has been used.

```

# replay_opt      shell script to invoke a command repeatedly.
#                  The command is specified as a cmd line arg.
# Options:
#      -d          delay interval (default = 5 sec)
#      -s          size in number of lines (default=23 or LINES)
#      -t          show the tail of the display
# getopt is used to process the command options.
Delay=5           # delay interval
Tail=""           # show tail of display
Size=${LINES:-23} # size of display
Usage="Usage: $0 [-d delay] [-s size] [-t] command"

trap 'clear; exit' 1 2 3 15

while getopt :d:s:t option ; do
    case $option in
        d)      Delay=$OPTARG
                ;;
        s)      Size=$OPTARG
                ;;
        t)      Tail=True
                ;;
        :)      print "$OPTARG option requires argument" >&2
                print $Usage >&2
                exit 1
                ;;
        \?)     print "Unrecognized option \"$OPTARG\"" >&2
                exit 1
                ;;
    esac
done

# Shift options from command line
shift $OPTIND-1

# Verify that Delay is numeric
if let $Delay > /dev/null 2>&1
then
    :           # null statement
else
    print "d option argument not an integer" >&2
    exit 1
fi

```



```

# Ensure a command was supplied
if [[ $# -eq 0 ]] ; then
    print "No command supplied" >&2
    print $Usage >&2
    exit 1
fi

# Ensure that command is valid
if eval $* > /dev/null 2>&1 ; then
    :          # null statement
else
    print "$* is not a valid command" >&2
    exit 1
fi

# Run the command repeatedly
while true ; do
    clear
    if [[ -n $Tail ]] ; then
        eval $* | tail -$Size
    else
        eval $* | sed ${Size}q
    fi
    sleep $Delay
done

```

It is suggested that the man page explanation of 'sed' be read carefully.

The 'sed' command has 2 work areas:

pattern space	for the selected line
hold space	as a temporary area where the selected line can be stored.

\$1 is the search pattern.

\$2 is the name of the file to be searched.

/ \$1 / puts a line that has the search pattern into the pattern space.

As 'sed' reads down through the file looking for the search pattern, it executes the 'h' command on each line.

The 'h' command places the pattern space (line just read) into the hold space.

When we reach the line that has the search pattern, the hold space will contain the line prior to the one containing the search pattern.

For this line that contains the search pattern,

x	exchange the pattern space and the hold space This puts the previous line in the pattern space.
p	print the pattern space (the previous line)
x	exchange the pattern space and the hold space This puts the line containing the search pattern in the pattern space.
p	print the pattern space (the line containing the search pattern)
n	read the next line of the file into the pattern space
p	print the pattern space (the line following the one containing the search pattern)

```
# csed          shell script to grep for a string in a file
#               and display several lines about that line
#               containing the string.
```

```
if [[ $# -ne 2 ]] ; then
    print "Usage: $(basename $0) pattern filename" >&2
    exit 1
fi
```

```
sed -n -e "/$1/{x;p;x;p;n;p;}" -e h $2
```

```
# sed reads a line from the file into its pattern space and
# applies the sed command (-e args) to the lines that match.
# As the next line is read, the pattern space is copied into
# the hold space (by the hold command below);
```

```
#
#      x          exchange the contents of the pattern space
#                  and the hold space
#      p          print the contents of the pattern space
#
#      n          replace the pattern space with the next
#                  line of input
#      h          replace the contents of the hold space with
#                  the pattern space
#
#      -n option   suppress the file contents normally
#                  written to standard output
```


Chapter 6: Exercises

Day 1 Lab Exercises

1. Write a shell script that checks for a minimum of two arguments. Print an error message to the standard error file if the user types less than two arguments; otherwise, print all the arguments to the standard output file.

2. Write a shell script which will accept a single file name on the command line. If the file exists, the script will ask the user whether a backup copy of the file should be made. A backup copy with a suffix of *.backup* is created, if requested, and the vi editor is invoked for the file.

3. Write a shell script that will prompt for and read from the standard input file a first name, a last name and a telephone extension. Write these values to a file named **phonelist** and loop until a first name of "quit" is typed.

4. The shell script **dirtest** works incorrectly. Make a copy in your directory and execute it with several pathnames as command line arguments, e.g.

```
$ dirtest /etc /etc/motd /mnt
```

Use shell debugging techniques to find and correct the problems. A listing of **dirtest** follows:

```
#!/bin/ksh
#dirtest - shell script with bugs
for i
    if (-d i)
        print "$i is a directory"
    else
        print "$1 is not a directory"
    endif
end
```

5. Write a shell script that will accept an option (*-r* or *-w*) and a file name argument on the command line. Have the script print an appropriate message to the standard output file indicating that the file does not exist, is readable or is writeable. If an option is not typed on the command line, assume *-r*.

6. Write a shell script that will make a copy with a suffix of *.save* of any file specified on the command line, e.g. **abc** to **abc.save**. Assume that users might specify multiple file names on the command line. If no file names are typed on the command line, make the *.save* copies for all files in the current working directory.

7. Use the **ps(1)**, **cut(1)**, and **sort(1)** commands to print a list of process id's currently in use. Print the list of PID's in increasing numerical order.

Day 2AM Lab Exercises

1. Create a function to change the permissions of a file to 600, invoke **vi** to edit the file and change the file permissions to 400.

2. Create a function to display the line(s) of the **ps ax** output that contain the string specified as an argument, e.g.

```
$ func inetd
```

should print:

```
143 ? I 0:00 /etc/inetd
```

Ensure that any **ps** lines created by the function execution are not printed, e.g. lines containing **grep inetd**.

3. Define the function **llf** and alias **ll** such that after logging out and logging in again, they will be defined differently for shell scripts that are executed by name or as an argument to **ksh**.

For scripts executed by name:

```
llf = 'ls -l $1'
```

```
ll = 'ls -l'
```

For scripts executed as an argument to **ksh**:

```
llf = 'ls -F $1'
```

```
ll = 'ls -F'
```

Test with a shell script containing the lines:

```
ll $1
```

```
llf $1
```

What is the effect if the following line is inserted as the first line of the shell script:

```
#!/bin/ksh
```

4. Write a shell script using **case** which takes one or two command line arguments. If two arguments are typed on the command line, the first is a file name and the second is a directory name. Use **find(1)** to search for a file of that name in the specified directory. If only one argument is typed on the command line, it is a file name to be sought in the current working directory tree.

5. Write a shell script to compile a source file whose name (without the `.c`) is given on the command line. Additional command line arguments must begin with a dash (-) and represent options for the compile command. Note that options could occur before or after the file name, e.g.

```
cc -g -o abc abc.c -lm
```

Day 2PM Lab Exercises

1. Write a .profile shell script for a captive account that will allow the user to execute the following commands ONLY:

ls
vi
exit

Ensure that the user will not be able to terminate the script by typing the interrupt or quit characters.

2. Write a function to print the hexadecimal value of a decimal integer typed as an argument.

3. Write a shell script that takes a positive integer from the command line and does the following:

- a. If the number is 1, stops.
- b. If the number is even, divides by 2 and starts over.
- c. If the number is odd, displays the number in base 10 and base 2, then multiplies the number by 3, adds 1 and starts over.

4. A histogram is a form of a bar graph in which lines of some character (usually an asterisk) represent values in a table. Write a shell script that draws a histogram of values in an array of integers. Each row of the histogram should represent the number in that array element. Test the script with an exported array defined by another script.

5. Write a shell script to replace the **rm** command. The script, before removing the file, will create a compressed copy of the file in the **trashcan** subdirectory of the home directory. The filename in the **trashcan** directory should contain the date and time at which the removal occurred.

6. Write a shell script to **unrm** the compressed file created by the shell script of the preceding exercise. This script should provide a menu listing the date and time of removal of the files of that name that might be unremoved.

Day 3 Lab Exercises

1. Write a shell script which uses **getopts** to convert an octal, decimal, or hexadecimal number given on the command line to octal, decimal or hexadecimal. The **-r** option takes an argument, *o*, *d*, or *x*, to indicate the number base of the result (have the script assume a decimal result if the **-r** option is not passed). The options **-o**, **-d**, or **-x** indicate the number base of the input (have the script assume decimal input if none of these options is passed).

The following lab exercise is intended to be a challenge for you. It illustrates advanced techniques and is not meant to be easy.

2. Write a shell script that will create co-processes to perform remote shell commands to several hosts on the network simultaneously and log the results of the commands in log files (one per remote host) on the local host. The main script should report the pathnames of the log files on exit. The log file names need to be unique and obvious (i.e. the host name should be a part of the log file name). The script executing in each co-process will obtain the name of the host and the name of the log file as command line arguments. The main script will prompt the user for the command to be executed remotely, then pass each command to all the co-processes via the co-process' standard input. Be sure that when the main script exits, it gives co-processes time to finish any incomplete commands.

Appendix A: Common Symbols

List of Symbols

<code>\$1, \$2, ... \$9, \${10}, \${11}, ...</code>	positional parameters specified on command line
<code>\$0</code>	name of shell script
<code>\$\$</code>	process id (PID) of currently executing shell
<code>\$PPID</code>	process id of parent process to current shell
<code>\$#</code>	number of positional parameters
<code>\$*</code>	all positional parameters
<code>\$?</code>	exit status of most recent command
<code>!</code>	process id of last background process
<code>\${var:-default}</code>	value for variable <code>var</code> default value if <code>var</code> is not set
<code>\${#var}</code>	Number of characters in the value of variable <code>var</code>
<code>\${#array[*]}</code>	Number of elements in array named <code>array</code>

Appendix B: Review Tutorial

Appendix B

This appendix is intended to be a reading assignment for those of your students who might be new to the KornShell. It provides more detail about the basic concepts reviewed in chapter 1 than the examples alone. If a student is unsure of these basics, (s)he can review this text after class and (hopefully) get on track for the next day's work.

Introduction

The UNIX user interface is called a *shell*. A shell is a program which runs in the context of an interactive (login) process. Its purpose is to accept commands from the user and to execute other programs based on those commands. This course teaches the advanced features of one of the commonly available shells for UNIX systems, called the KornShell.

The KornShell provides the following basic services for users:

- Redirection of input and output on the command line
- Creation of pipelines from the command line
- Metacharacters used as wildcards and for other purposes
- The keeping of a history of previously executed commands
- Command line editing
- Command line filename completion
- The creation and use of variables
- Background processing and job control

You should be familiar with these basic features of the KornShell before you begin to learn its advanced programming capabilities. In addition, it is necessary that you be able to create and modify files using a UNIX text editor.

If you have not had the time to become acquainted with these basics, this appendix may provide the foundation you need to get the most from the rest of the training.

I/O REDIRECTION:

Redirection of input and output is a feature supported by all of the UNIX shells. It is based on one simple idea: a program is more useful if it can be flexible about where it obtains its input and where it sends its output.

For programs to be flexible, the UNIX operating system provides to the programmer a generic input source and a generic output target. The shell provides to the user of the program an easy way to assign the generic input source or output target to something real.

GENERIC INPUT SOURCE:

The generic input source used by programs that need or want to be flexible is called *standard input*, or *stdin*. This is also the source from which the shell accepts your commands.

GENERIC OUTPUT TARGETS:

There are two targets for output. The first is used by programs to generate their normal output. The name of this output target is *standard output* or *stdout*. This is also the target to which the shell writes its prompt string (the KornShell's default prompt string is \$).

The second output target is used only to produce error messages. The name of this generic target is *diagnostic output*, or more often, *standard error* or *stderr*.

STATUS OF STANDARD FILES AT LOGIN:

When you log in to a UNIX system, a process is created for you which is running your shell program (set up in the password file by the system administrator). The system assigns the standard files so that input comes to the shell from the terminal keyboard and output and error messages go to the terminal screen.

UNIVERSALITY OF INPUT AND OUTPUT:

The UNIX operating system conducts all input and output operations as if they were being carried out to or from a file. This allows all I/O to be initiated in the same way by a program, whether it is being done using a disk file (**/class/user/myfile**), a terminal device (**/dev/tty03**), a magnetic tape (**/dev/rmt0h**), or just about anything else.

Standard Files

Standard input:

The file from which shell accepts commands

Standard output:

The file to which shell sends prompt character (\$)

The file to which most utilities write their output
who, date, ls, . . .

Standard error (diagnostic output):

The file to which shell writes error messages

Status of the standard files at login:

standard input	keyboard
standard output	terminal
standard error	terminal

All input/output operations are conducted through files

/class/user/myfile

/dev/tty03

/dev/rmt0h

REDIRECTING THE STANDARD FILES:

The shell provides special characters which you can use on the command line to assign (redirect) one or more of the standard files to actual files other than the terminal keyboard and screen.

In the example on the facing page, we can see the normal method for executing the **who** command. The information which the command sends to its standard output appears, as expected, on the terminal.

On the next line the **who** command is followed by the right-hand angle bracket character (**>**), then the name of a file. This addition to the **who** command tells the shell to execute the command normally in all respects except for one: rather than using the terminal for the standard output, it uses the file named **whofile**. As you can see, nothing appears on the terminal screen except the prompt for the next command. Using the **cat** command to display the contents of **whofile** shows that the output was indeed redirected there.

REDIRECTING TO AN EXISTING FILE:

Since there can only be one file with a given name in any directory, an existing **whofile** would be destroyed in the example shown. This can be prevented easily if the user turns on the *noclobber* KornShell option. The **set -o** command shown on the facing page does just this, and the next command attempting to redirect the **date** output fails with an appropriate error message.

Remember that *noclobber* applies ONLY to output redirection, not to any other file-related activity. You can override the *noclobber* option on a command-by-command basis using a pipe character in conjunction with the output redirection metacharacter (**>|**) as in the next command shown. Sure enough, **whofile** has the date and time.

In the last few commands in this section the **set +o** command turns off the *noclobber* option. As you can see, the **date** command clobbers **whofile**.

The double angle bracket (**>>**) is used to add to an existing file.

REDIRECTING INPUT:

Redirecting the input of a command from a file uses a similar syntax. The left-hand angle bracket (**<**) in the example causes the **mail** command to obtain the text sent to john not from the keyboard, but from the file named **message**.

Redirection of Standard Output/Input

To redirect output to a new file:

```
$ who
user tty02 Dec 5 11:30
kjd tty04 Dec 5 12:53
sam tty00 Dec 5 13:03

$ who > whofile

$ cat whofile
user tty02 Dec 5 11:30
kjd tty04 Dec 5 12:53
sam tty00 Dec 5 13:03
```

To redirect output to an existing file:

```
$ set -o noclobber
$ date > whofile
whofile: file already exists

$ date >| whofile
$ cat whofile
Thu Feb 1 13:38:36 EST 1990

$ set +o noclobber          # clear noclobber
$ date > whofile
$ cat whofile
Thu Feb 1 14:22:17 EST 1990
```

To append redirected output to an existing output file:

```
$ ls >> file
```

To redirect input:

```
$ mail john < message
```

REDIRECTING STANDARD ERROR:

Standard error can also be redirected as shown in the example to the right. In this instance we have guaranteed that an error will occur by setting the permissions on **file1** so that no one has access to it. Attempting to read the file using **cat** fails as you can see. The use of the output redirection indicator (**>**) preceded by the number **2** causes *stderr* to be redirected just as *stdout* was in a previous example. The number **2** is the file descriptor for *stderr*. *stdin* is descriptor **0** and *stdout* is descriptor **1** (but that's a topic which will be discussed during the lecture).

Redirecting any or all of the standard files can be done for any command as long as the command uses the standard files. The first example in the next section shows standard output redirected to the file **myout**, and at the same time standard error is redirected to the file **myerr**.

To send the output and the error messages from a command to the same file, you could type:

```
shell_command > myout 2> myout
```

But there is a shorthand way to get this done as shown in the next example. The syntax **2>&1** is read, "Make file descriptor **2** a copy of file descriptor **1**." In other words, *stdout* is redirected to **myout**, and *stderr* is redirected the same as *stdout*.

THE NULL DEVICE:

The last example on the facing page uses the null device, **/dev/null**. Like everything else, the null device is a file. When you write to this file, the write operation completes immediately and the data goes into the great beyond, never to be seen nor heard from again. When you read from this file, the read completes immediately, but the only thing you get is an end-of-file (EOF) indication.

Redirection of Standard Error

To redirect standard error:

```
$ chmod 000 file1
$ cat file1
file1: Permission denied

$ cat file1 2> problem
$ cat problem
file1: Permission denied
```

To redirect standard error and standard output:

```
$ shell_command > myout 2> myerr
$ shell_command > myout 2>&1
```

To eliminate any output:

```
$ shell_command > /dev/null 2>&1
```

/dev/null is a special file.

It is the UNIX data sink (bit bucket).

When read, it yields EOF.

When written, it provides an infinite sink.

PIPELINES:

We have seen how *stdin*, *stdout*, and *stderr* can be redirected to files, but what if, somehow, the *stdout* of one command could be connected to the *stdin* of another. If this were possible, commands could be connected to one another to perform more complex tasks than any individual command could do.

Pipelines provide this capability. In the first example to the right, the **who** command is shown for reference, then a pipeline is executed. In this pipeline, the *stdout* of **who** is connected to the *stdin* of the **sort** command. The result, as you can see, is that the output from **who** is sorted before it is displayed.

In the second pipeline example, **who** is *piped* to the **wc** command. This command counts the characters, words, and lines in its input (the **-l** option tells **wc** to count lines only). The input to **wc** is, of course, the output from **who**, so this pipeline is a clever way to determine how many users are on the system -- 3.

REDIRECTING OUTPUT FROM A PIPELINE:

If the last command in a pipeline sends its output to *stdout*, then the output from the whole pipe can be redirected using the same techniques shown previously, as in the next example opposite.

OTHER PIPELINE EXAMPLES:

In the last examples, the output of a long listing is piped to the **grep** command. The **grep** command *filters* the output from the **ls -l** command and only passes through those lines which begin with a lower-case b. The term *filter* describes any command which, like **grep**, both gets its input from *stdin* AND sends its output to *stdout*. a filter can be used in any position in a pipeline: beginning, middle, or end.

Programs such as **who** and **ls**, since they do not use *stdin*, are only allowed to be the first command in a pipeline. Programs like **lpr**, since they do not use *stdout*, are only allowed to be the last command in a pipeline.

The final example shows **ls**, a non-filter, piped to **grep**, a filter, piped to **lpr**, another non-filter.

Pipelines

```
$ who
sam  tty10 Jun 30 10:54
mary tty03 Jun 30 08:13
mgr  tty12 Jun 28 17:34

$ who | sort
mary tty03 Jun 30 08:13
mgr  tty12 Jun 28 17:34
sam  tty10 Jun 30 10:54

$ who | wc -l
3

$ who | sort > people

$ cat people
mary tty03 Jun 30 08:13
mgr  tty12 Jun 28 17:34
sam  tty10 Jun 30 10:54
```

To list all files in `/dev` which are block structured devices:

```
$ ls -l /dev | grep '^b'
brw----- 1 root system 8, 3072 Apr 13 10:19 rz3a
brw----- 1 root system 8, 3073 Apr 13 10:19 rz3b
brw----- 1 root system 8, 3074 Apr 13 10:19 rz3c
brw----- 1 root system 8, 3075 Apr 13 10:19 rz3d
```

To list on the printer all files in `/dev` which are block structured devices:

```
$ ls -l /dev | grep '^b' | lpr
```

WILD CARD METACHARACTERS:

A *metacharacter* is any character to which the shell assigns some special meaning. All of the UNIX shells use many metacharacters and the KornShell is no exception. Among the most commonly used metacharacters are what we call the wild card metacharacters. These have several uses, but they are primarily employed in creating patterns of characters that can be expanded by the shell into a list of filenames.

The table at the top of the opposite page shows the three wild card constructions used most often with an explanation of how each works. The `ls -a` command shows the set of filenames available for matching, and the succeeding commands use the wild cards to create pattern matches from among these files.

So `x?` sets up a pattern matched by any filename that is two characters in length with the first character a lowercase *x*. The pattern `x*` in the next command can be matched by a filename of any length which begins with a lowercase *x*, including the file named *x* (alone).

The pattern `[xyz]*` can be matched by filenames of any length which begin with either an *x*, a *y*, or a *z*. The pattern `[x-z]4*` can be matched by filenames of any length beginning with an *x*, *y*, or *z*, and having a *4* as the second character.

In the pattern `[!a-y]` the *bang* character (!) has the meaning "not". This pattern is matched by any filenames that does not begin with any of the letters *a* through *y*.

THE TILDE METACHARACTER:

The tilde (~) metacharacter is special. It creates a pattern which is matched by your home directory string. Placing another user's login name immediately after the tilde changes the pattern to match that user's home directory.

Wild Cards

*	Matches any string of characters, including a null string
?	Matches exactly ONE of any character
[chars]	Matches exactly ONE of any character among those included between the brackets

```
$ ls -a
.    .login  get.c put.c  x  x2 xfour  z4
..   a      msg.c put.o  x1 x3 y4sale zed
```

```
$ ls x?
x1      x2      x3
```

```
$ ls x*
x        x1        x2        x3        xfour
```

```
$ ls x??
x?? not found
```

```
$ ls [xyz]*
x        x2        xfour  z4
x1       x3        y4sale zed
```

```
$ ls [x-z]4*
y4sale z4
```

```
$ ls [!a-y]*
z4      zed
```

~	Matches user's home directory
~sam	Matches the home directory for user sam

```
$ ls -l ~/srcfile
-rw-r--r-- 1 mary      362 Nov 3 12:13 srcfile
$ cp myfile ~
```

COMMAND HISTORY:

The KornShell keeps a list of your past commands called the *history list*. The **history** command is used to display the 16 most recently executed commands from the history list (the example display has been abbreviated in the interest of conserving space). Note that each command in the command history is assigned a unique number.

When you log in, the KornShell pre-loads your history list with the list of commands saved in the history file. The default name for this file is **.sh_history** in your home directory. The number of commands saved in the file defaults to the last 128 commands you executed in your most recent login session. Later in this course, you will learn how to specify a different file name or a different number of past commands to save if you wish.

The **r** command recalls and executes a previous command from the history. There are several ways to specify the command you wish to re-execute, as shown in the examples to the right:

- If you simply type the **r** command with no argument, the previous command is re-executed.
- If you specify a positive integer *n*, the command with that number is re-executed.
- If you specify a negative integer *-n*, you instruct the **r** command to re-execute the *nth* most recent command.
- If you specify a string, the most recent command that began with that string is re-executed.

THE HISTORY NUMBER:

Because the command number is an important aid to remembering what commands executed when, you might wish to use the history number in your command prompt. The last example on the facing page shows how to do this. The bang (!) character is replaced in the prompt with the current history number. Note that the apostrophes are essential. They cause the shell to defer replacing the ! with the command number until just before the prompt is written to *stdout*.

History of Past Commands

```
$ history
9 cat letter.mail
10 date
11 rm mmdc.c
12 ls -l /class/user/save
13 chmod a-x mbox
14 history
```

```
$ r 10
date
Fri Feb 2 12:18:17 EST 1990
```

```
$ r -4
ls -l /class/user/save
-rwxr-xr-- 1 user          580 Jul 5 11:38 save
```

```
$ r cat
cat letter.mail
Don't forget lunch on Friday.
Susan
```

```
$ r ch
chmod a-x mbox
```

Include command number in prompt string:

```
$ PS1='!$ '
18$
```

COMMAND LINE EDITING:

The **r** command discussed in the previous examples allows you to re-execute a previous command, but not to change it. The KornShell's command line editing feature provides you with the capability to:

1. Search the history list
2. Choose a command for execution, and
3. Optionally edit the command before you place it into execution.

To enable the command line editing feature, you must turn on either the *vi* or *emacs* shell options. The **set -o vi** command example on the facing page chooses a vi-like command editor (as you might expect, turning on the *emacs* option chooses an emacs-like command editor).

Once you have chosen an editing style, you can recall past commands and edit them. The next examples show the vi-like interface.

The ESCAPE key starts the vi-like command editor. The **k** command fetches the next command back in the command history. Subsequent **k** commands step back into the command history.

Whichever editor interface you choose, you will have available a subset of the editing commands available to you on the command line.

Among the most useful vi-style commands are:

- **k** and **j** Select the next command; **k** by decreasing command number, **j** by increasing command number.
- **h** and **l** Move the cursor left (**h**) or right (**l**).
- **x** Delete the character under the cursor.
- **istring**<esc> Insert text until ESCAPE is typed.
- **Rstring**<esc> Replace text until ESCAPE is typed.

FILENAME COMPLETION:

To save typing, you can take advantage of the filename completion capability of the KornShell. To use this feature, type enough of the filename so that you have a unique set of characters as in the first example, then type ESCAPE and a backslash (\). The KornShell automatically completes the name. If the beginning that you typed is not unique, the shell fills in what it can and waits for you to add to the name as in the second example. The name can be a relative pathname or a full pathname as in the last example.

Editing Past Commands and Filename Completion

```
$ set -o vi
$ <esc>k          #retrieve latest command
$ _set -o vi      #cursor at beginning of line
k                #retrieve next latest command
$ _chmod a-x mbox #cursor at beginning of line
^C              #terminate history scan
```

vi commands:

h j k l	Move cursor (left, down, up, right)
x	Delete character
istring<esc>	Insert <i>string</i>
Rstring<esc>	Overwrite with <i>string</i>

Filename Completion:

```
$ ls
comm edit files make machine shell

$ lpr ed<esc>\
lpr edit

$ chmod 755 m<esc>\
chmod 755 ma
chmod 755 mak<esc>\
chmod 755 make

$ grep sam /etc/pas<esc>\
grep sam /etc/passwd
sam:A34fjU76Fcv5g:203:0:Samuel Adams:/usr/users/sam:/bin/ksh
```

KORNSHELL VARIABLES:

A KornShell variable is a character string to which is assigned a value. That value is another character string. When the shell encounters a variable on the command line or in a shell script, it is replaced with its value and the value is used.

Variable names can contain letters, numbers, and the underscore (`_`) character. Variable names are not allowed to begin with a number. Upper and lower case characters are distinct.

The first several examples on the opposite page show how to define a variable. The command is a simple equality statement with the variable name on the left of the equal sign and its value on the right. Spaces are not allowed around the equal sign.

USING VARIABLES:

The third and successive examples show how variables are identified to the KornShell once they have been defined. The dollar sign (`$`) metacharacter is the indicator that what follows is the name of a variable.

The **print** command displays its arguments at *stdout* and is the method you will use to show the value of a variable. To see all variables that you have defined and their meanings, type the **set** command with no arguments.

Shell Variables

```
$ VAR1=Brown
$ VAR2=stone
$ CONCAT=$VAR1$VAR2
```

```
$ print "$VAR1"
Brown
```

```
$ print "$VAR2"
stone
```

```
$ print "$CONCAT"
Brownstone
```

KornShell variables are stored as ASCII strings.

Variable names **MUST** begin with a letter or underscore, but may contain digits, letters or underscores.

SPECIAL KORNSHELL VARIABLES:

The variables on this page are all special to the KornShell. The table on the facing page lists their meanings and uses.

Special KornShell Variables

HOME	Home directory: default argument for cd
PS1	Primary prompt string: default = "\$ "
PS2	Secondary prompt string: default = "> "
HISTSIZE	Size of the history list: default = 128
HISTFILE	File in which history is saved: default = \$HOME/.sh_history
PWD	Present working directory set by cd command
OLDPWD	Previous working directory set by cd command
IFS	Internal field separator: default = whitespace
MAIL	Users mail file: if set, user is informed by shell of arrival of mail every \$MAILCHECK seconds
TERM	Terminal type
PPID	Parent process ID
TMOUT	A number of seconds: if a command is not issued to the shell in this time period, the shell exits
TMOUT=0	Inhibits this automatic logoff feature
RANDOM	A random integer from 1 to 32767
PATH	A list of directories to be searched for a command name

CHILD PROCESSES:

When the KornShell executes a command, it does so by creating a *child* process. The child is a duplicate of its parent in most respects. The child runs the program or shell script called for by the command while the parent process waits. When the program or script finishes executing, the child process is terminated and the parent is free to prompt for the next command.

EXPORTING VARIABLES TO CHILD PROCESSES:

A child process is *almost* identical to its parent. One feature that it does not share is a knowledge of all the variables that have been defined.

In the first example to the right is a script which simply uses the **print** command to display the value of a variable named **var**. In the right-hand part of the example, **var** is defined with a value of **12345**, but when the child process is created to execute the script, nothing is displayed. Clearly, the child is ignorant of the value of the **var** variable.

In the left-hand part of the example, the variable is defined as before, but this time, the **export** command is used to make the value of this variable available to all subsequent child processes. Now, when a child is created to run **shscript**, the child obviously knows the value of **var**.

The next example shows that if a new value is given to **var**, the child processes created subsequently will have access to the new meaning. This tells us that exporting variables is a task performed anew by the shell for each child it creates.

defining and exporting a variable can be done using a single **export** command as shown in the next example.

The final example uses the **export** command used with no arguments to display all variables which are currently being exported.

Exporting Variables into Child Processes

```

$ cat shscript
print $var

$ var=12345
$ export var
$ shscript
12345
$

$ var=12345
$ shscript
$
```

Export the value of the variable into all subsequently executed commands or shells.

```

$ var=abcde
$ shscript
abcde
```

If an exported variable is modified, the new value is exported.

```

$ export var=12345    #define & export
$ export              #list environment
ENV=~/.environ
HOME=/usr/users/sam
LOGNAME=sam
MAIL=/usr/mail/spool/sam
PATH=/usr/users/sam/bin:/bin:/usr/bin:/etc:.
TERM=vt100
TZ=EST05EDT
var=12345
```

FOREGROUND AND BACKGROUND PROCESSES:

You might have a program which takes twenty minutes to generate a report. You can start that program as a background job, leaving the terminal and keyboard free for you to do something else while the report is cooking.

A background job is a normal child process except that the parent does not wait for the child to terminate before prompting for the next command.

The ampersand (&) character following a command tells the shell to execute the program in a background process (second example).

There are two commands commonly used to keep track of the status of background processes, the **ps** command and the **jobs** command.

The **ps** command displays information about processes in general. Options to **ps** are used to select the types of processes and the level of information you wish to see. The normal display shows process id, execution state, terminal, elapsed time, and command. The **x** option selects your parent process and all its children. The left-hand side of the third example shows the parent process running the shell (**ksh**), the child executing **ps**, and the background child executing **grep**.

The **jobs** command displays minimal information about background jobs only. The right-hand side of the third example shows the output from **jobs**. The display consists of the job number, job status, and command.

When a background job completes, a message appears on *stderr*.

Note the difference in the displays in the next set of commands. This example shows a pipeline running in the background. There are two new entries in the **ps** display, one for each child participating in the pipeline, but there is still only one entry in the **jobs** display -- the child processes are treated as a unit.

The last set of examples show the long form of the displays for **ps** and **jobs**. Added to the **ps** display are fields showing process flags, user id, parent process id, and priority information (PRI and NI). The only enhancement to the **jobs** display is the addition of process id.

Foreground and Background Processes

```
$ ps x
```

```
PID STAT TT TIME CMD
2680 S    01 0:00 - (ksh)
3204 R    01 0:00 ps x
```

```
$ grep '^B' file > report &
[1] 4750
```

```
$ ps x
```

```
PID STAT TT TIME CMD
2680 S    01 0:00 - (ksh)
4751 R    01 0:05 ps x
4750 R    01 0:01 grep ^B
```

```
$ jobs
```

```
[1] + Running grep ^B file>report
```

```
[1] Done grep ^B file > report
```

```
$ grep '^B' file | lpr &
```

```
[1] 4791
```

```
$ ps x
```

```
PID STAT TT TIME CMD
2680 S    01 0:00 - (ksh)
4792 R    01 0:07 ps x
4790 R    01 0:07 grep ^B
4791 R    01 0:01 lpr
```

```
$ jobs
```

```
[1] + Running grep ^B file | lpr
```

To obtain a long listing of process status:

```
$ grep '^B' file > report &
[1] 1706
```

```
$ ps lx
```

```
F      UID PID   PPID  PRI  NI   STAT  TT   TIME  COMMAND
10808001 10 1704     1   15   0    S    01   0:00  - (ksh)
10008101 10 1706   1704   46   0    R    01   0:00  grep '^B'
10008101 10 1708   1704   43   0    R    01   0:00  ps lx
```

```
$ jobs -l
```

```
[1] + 1706 Running      grep ^B file > report
```

JOB CONTROL:

Suppose you executed the report program without the ampersand metacharacter (in other words, as a foreground job). After waiting ten minutes for the command prompt to reappear, you realize your mistake. You have better things to do than watch the cursor blink, but you don't want to start the report program over again -- you've already wasted more than ten minutes. Solution: stop the job and resume it in the background!

The first example opposite shows the **sort** pipeline being stopped and resumed as a background job. The CONTROL-Z keystroke is the stop character. Note that the **jobs** command shows the stopped job. It's not in the foreground nor in the background. The **bg** command resumes the stopped job in the background.

The next command starts another background job. Note the second field in the **jobs** display. One of the jobs is marked with a +, the other with a -. If there were a third or fourth job, this field would be blank for them. The notes under this example explain the significance of these markers.

Jobs can be moved into the foreground using **fg** either from the background or the stopped condition.

Finally, suppose you had finished everything you needed to do except printing the report file, but the report program is still cooking in the background, and you are late for a luncheon engagement. The last example on the facing page uses the **wait** command to synchronize the foreground activities with a background job or jobs. You can use this knowledge to have your parent process wait for the report to cook in the background, then you can type the appropriate command to get the hardcopy of the report and leave for the restaurant, confident that the printer will not be activated until the report is finished.

To move a foreground job to the background:

```
$ sort abc.dat | lpr
^Z
Stopped

$ jobs
[1] + Stopped sort abc.dat | lpr

$ bg %1
[1] sort abc.dat | lpr &

$ grep '^B' file > report &
[2] 264

$ jobs
[2] + Running grep ^B file > report
[1] - Running sort abc.dat | lpr
```

+ identifies the current job. This job will be the default job for commands like **bg**, etc.

- identifies the previous job. This job may have been the current job at one time, and will be the current job when the current job finishes.

To move a background job into the foreground:

```
$ fg %2
lpr report
$
```

To wait for a background process to complete:

```
$ wait %1
ls
$ a.tmp cat.tmp file klm.tmp rpt.c xyzv.tmp
```

JOB CONTROL (CONTINUED):

The ultimate control over a background job is the power of life and death. The first example opposite shows the **kill** command being employed to terminate a job running in the background. The second example shows the **stop** command used to stop a background job. You may be excused for some confusion at this point because terminate and stop are usually regarded as synonyms. In this case, however, their meanings are different. The meaning of terminate is about the same as it had in the Arnold Schwarzenegger movies. The meaning of stop is more like the one it had in Driver's Ed class.

A question that sometimes arises at this point is, "Why **stop**? Why not just type CONTROL-Z as on the previous page?" The answer is that the CONTROL-Z keystroke can only affect the job running in the foreground. To have the same effect on a background job requires another method, the **stop** command is that method.

The table at the bottom of the facing page shows the different ways you can refer to background and stopped jobs in commands like **fg**, **bg**, **stop**, and **kill**.

To terminate a background job:

```
$ kill %1
```

To stop a background job:

```
$ stop %2
```

Job references for the **fg**, **bg**, **wait**, and **kill** commands:

%n	Job number
%str	Job whose command begins with <i>str</i>
%?str	Job whose command contains <i>str</i>
%+	Current job
%%	
%-	Previous job

STANDARD FILES FOR BACKGROUND JOBS:

The examples on the facing page show commands and techniques you have already been introduced to, but the purpose here is to explore what happens to *stdin* and *stdout* for a background job.

The behavior shown in these examples only applies when *stdin* or *stdout* are connected to the default files, the keyboard and screen of your terminal. If redirection is being used, the situations shown in the examples do not apply.

TERMINAL INPUT:

In the first example, an editing session is started, and a command is given to the editor which will probably take a long time to complete. The foreground process running the `ed` editor is then stopped and the user executes several other commands in the foreground. The question is, what will happen when the editor has completed the task the user gave it and prompts for the next command to be entered at *stdin*?

The answer seen in the example is that the background process is stopped. Background jobs have no access to *stdin*. If they attempt to read from *stdin*, they must stop. How can such a stopped job be resumed? By bringing it into the foreground.

TERMINAL OUTPUT:

What about *stdout*? The next example shows what happens. A command is started in the background, and when it attempts to send its display to *stdout*, it succeeds.

There is a problem with this behavior. You may not wish to see the display at this time. You may be attempting to use the terminal for other important purposes, and the interjection of the display output from a background job might disrupt your attempt to show one of your colleagues how something works, or disrupt some test you are conducting.

The ability of a background job to access the terminal is actually under the control of the terminal driver. You can use the terminal characteristic *tostop* to signal the driver how to handle terminal output from a background job. *tostop* means terminal output **stop** and it applies only to background jobs. The command `stty tostop` turns on this characteristic. `stty -tostop` turns it off.

Input / Output in Background Processes

Standard Input:

```
$ ed bigfile
120000
1,$s/this/that/
^Z
Stopped

$ bg
[1] ed bigfile &
$
...some foreground commands...
```

```
[1] + Stopped (tty input) ed bigfile

$ fg
ed bigfile
w
120000
q
$
```

Standard Output:

```
$ ls -l /class/user/save &
[1] 372
$ -rwxr-xr-- 1 user          580 Jul 5 11:38 save

$ stty tostop

$ ls -l /class/user/save &
[1] 373
$ <return>
[1] + Stopped (tty output) ls -l /class/user/save

$ fg
ls -l /class/user/save
-rwxr-xr-- 1 user          580 Jul 5 11:38 save
$
```

EDITING TEXT FILES:

This page has nothing whatsoever to do with KornShell features, but a novice shell user might also be unfamiliar with the editor which is available on the systems that will be used during class. That editor is **vi**.

To invoke the vi editor, use the command in the example at right.

The vi editor gets its command input using the keys on the typewriter keyboard -- the QWERTY keyboard as it is sometimes called. The vi editor is beginning to show its age: it makes minimal use of such "advanced" terminal features as arrow keys, editing keys, function keys, or the auxiliary keypad, all of which have been around for the last ten years at least. But it seems always to be there, no matter what the implementation of UNIX you happen to be working on.

The following vi commands will give you enough facility to get you started creating and modifying the shell script files you will be asked to develop during this class:

h j k l	Cursor movement keys:	h	Cursor left
		j	Cursor down
		k	Cursor up
		l	Cursor right
x	Gobble key - eats the character the cursor is on		
dd	Delete the whole line the cursor is on		
itext<esc>	Insert <i>text</i> at cursor - existing text moves to right		
atext<esc>	Append <i>text</i> after cursor - existing text makes room		
J	Join the next line to the end of the current line		
:r file	Read - insert contents of <i>file</i> after the current line		
:wq	Write contents of file and quit the editor		
:q!	Quit the editor without writing contents of file		

Also helpful might be these commands:

.	Repeat previous command
u	Undo previous command

Basic vi Editing

\$ vi textfile

The UNIX operating system developed from the Computing Science Research Group at Bell Laboratories in Jersey. It is said that the creators of the UNIX system has this objective in mind:

to create a computing environment
where they themselves
could comfortably and effectively pursue their own work
programming research.

Until 1980, the UNIX system was mostly confined to
an environment consisting of university computer science
departments and research and development organizations.

~
~
~
~

"textfile" 13 lines, 504 characters

x

dd

iNew <ESC>

aNew <ESC>

J

:r newfile

:wq

:q!