Unit 1
Basic Shell Concepts

Unit Objectives

After completing this unit, you should be able to:

- Describe the AIX shells
- Use the AIX filesystem
- Create a shell script
- Use metacharacters
- Use I/O redirection
- Use pipes and tees
- Group commands
- Run background processes
- Use shell job control
- Use command line recall and editing

Shells

- What is a shell?
  - User interface to AIX
  - Command interpreter
  - Programming language

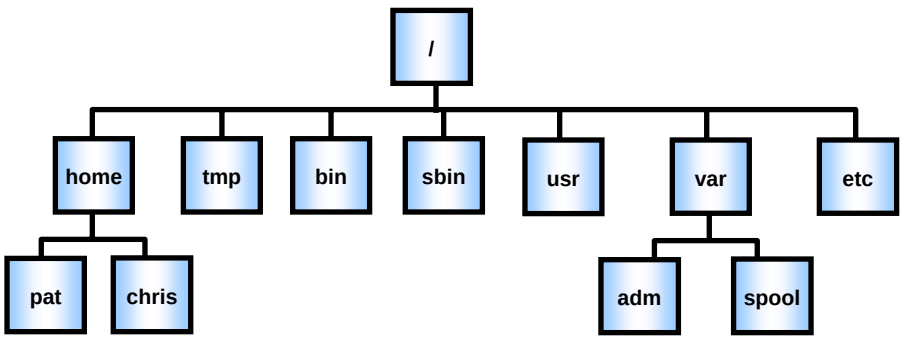
- AIX shells:
  - Bourne - bsh
  - Bourne-Again - bash
  - C - csh
  - Distributed - dsh
  - Korn (88) - ksh
  - Korn (93) - ksh93
  - POSIX - psh
  - Restricted - rsh
  - Trusted - tsh
  - Default - sh (links to ksh in AIX V4 and V5)

This Course

- AIX 5 loads with the 88 Korn Shell, the 93 Korn Shell, the Bourne Shell, and the Bash Shell (and more).
- root may create different users who log into different shells.
- The default shell in AIX 5 is 88 Korn Shell.
- This course focuses on the 88 Korn Shell. The slight differences in the other three shells will be noted on the slide or in the student notes.
- Available logins are:
  - team01, team02, team03, team04, team05
  - bash01, bash02, bash03, bash04, bash05
  - ksh9301, ksh9302, ksh9303, ksh9304, ksh9305
- The password is the same as your login name
- All exercises are located in /home/workshop:
  - They need to be copied into your $HOME

Directories

- The filesystem comprises directories in a hierarchical structure



- Refer to the files and directories with a a full or relative path name

- "." represents current dir, ".." represents parent directory

Basic File Commands

Command	Argument	Function
`mkdir`	`directory`	Create new directory directory
`rmdir`	`directory`	Delete empty directory directory
`rm`	`file`	Remove a file
`rm -r`	`directory`	Delete directory directory and any sub-directories
`ls`	`directory`	Give a listing of directory - many options: l, R, d, a i, t
`pwd`		Print working directory - where you are in the tree right now
`mv`	`old new`	Rename a file or directory - "new" can be a new file name, or a directory in which to place the file
`cp`	`old new`	Copies a file to a new name
`ln`	`name copy`	Creates another name without copying the contents
`cd`	`directory`	Change working directory to directory

A File

- Definition:
  - Collection of data, located on a portion of a disk.
  - Stream of characters or a **byte stream**.

- No structure is imposed on an ordinary file by the operating system.

- Examples:
  - Binary executable code – `/bin/ksh`
  - Text data – `/etc/passwd`
  - C program text – `/home/pat/prog.c`
  - Device special file – `/dev/null`
  - Directory special file – `/home`

```
$ file filename        – to find out which file type
$ strings filename     – if the file type is 'binary'
```

AIX File Names

- Should be descriptive of the content
- Are case-sensitive
- Should use only alphanumeric characters:

  UPPERCASE lowercase digits
  # . @ - _

- Should not begin with "**+**" or "**-**" sign
- Should not contain embedded blanks or tabs
- Should not contain shell "special" characters:

  *** ? > < / ; & ! ~ $ \ |**
  [ ] { } ( ) ` ` ' ' " "

What Is a Shell Script?

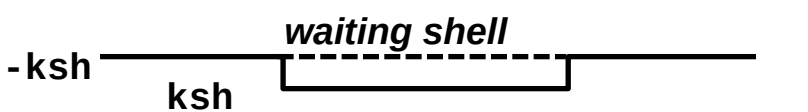
- A readable text file which can be edited with a text editor
  - **`/usr/bin/vi shell_prog`**
- Anything that you can do from the shell prompt
- A program, containing:
  - System commands
  - Variable assignments
  - Flow control syntax
  - Shell commands
- And comments
  - **`#!/bin/ksh`** is not a comment if **`#!`** is in the first position

Invoking Shells

`$ ksh` begins a new 88 Korn shell, interrupting the current one

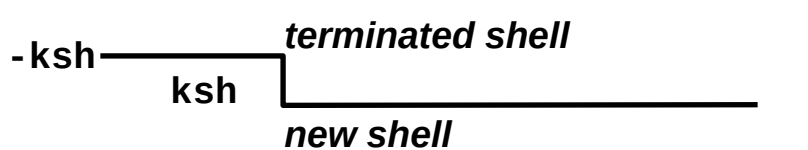
`$ ksh -c commands` runs commands in a shell

`$ ksh -r` starts a restricted shell



terminates the current shell and replaces with new shell

`$ exec ksh`

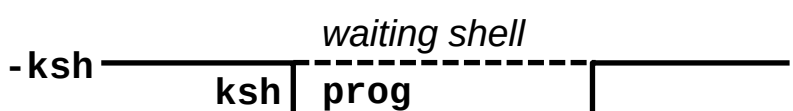


Invoking Scripts

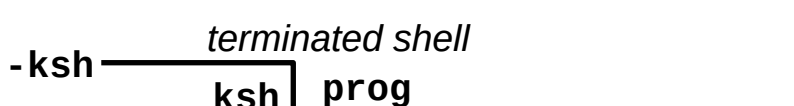
$. prog (sourced) in current shell environment

$ ksh prog run **prog** in a new Korn shell

$ prog (or ./prog) run in a new shell if **prog** is executable

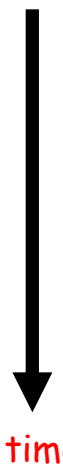


$ exec prog run **prog** in a new shell to replace the current one



Korn Shell Configuration Files

Invoking the Korn Shell sources:



`/etc/environment`	Sourced by all AIX processes
`/etc/profile`	Sourced by login shells
`.profile` **`.exrc (.vimrc)`**	Login shells source these files in the user's home directory
`$ENV` **`.kshrc (.bashrc)`**	A resource file listed in the ENV environment variable will be sourced by the shell

Each new **explicit** Korn shell sources the ENV file again

* If using CDE, **`.dtprofile`** must be changed to force an execute of **`.profile`**. If using bash, please refer to student notes.

What Are Metacharacters?

- Characters with special meaning
  - Three types
    - Wildcard (or expansion)
    - Shell
    - Quoting
  - Shell processes metacharacters before executing a command
  - There are many different shell metacharacters
  - Metacharacters can be mixed
- Wildcard metacharacters can be turned off by shell options

Wildcard Metacharacters

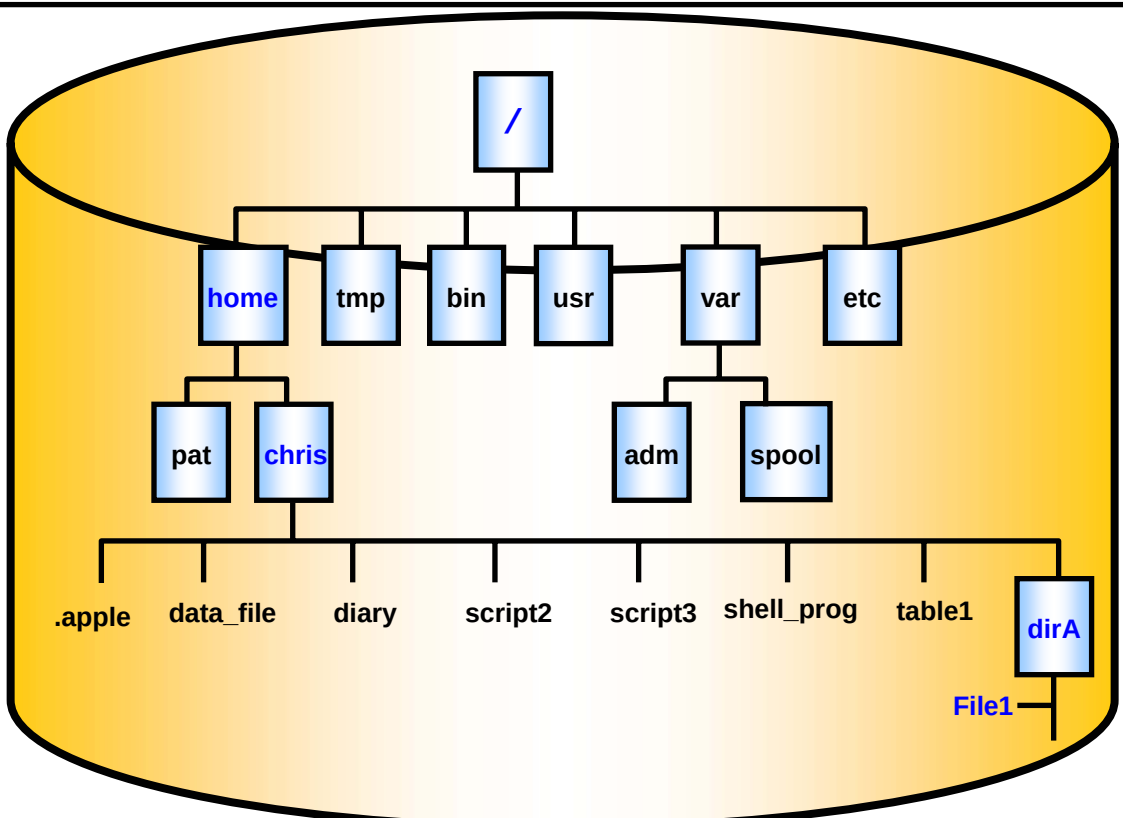
Metacharacters that form patterns that are expanded into matching filenames from the current directory:

`*`	-	Match any number of any characters
`?`	-	Match any single character
`[abc]`	-	Match a single character from the bracketed list
`[!az]`	-	Match any single character except those listed
`[a-z]`	-	Inclusive range for a list

Character Equivalence Classes can be used in place of range lists, to avoid National Language collation problems:

`[[:upper:]]`	-	Range list of all ***upper case*** letters
`[[:lower:]]`	-	All lower case letters: ***a, b, c,... z***
`[[:digit:]]`	-	Digits: ***0, 1, 2,... 9***
`[[:space:]]`	-	Spacing characters: ***tab, space***, and so forth

Sample Directory



Expansion Examples

`$ rm d*y`	removes the **diary** file
`$ file script*`	identifies **script2** and **script3**
`$ head script[345]`	displays the top lines of **script3**
`$ more script[3-6]`	displays **script3** screen by screen
`$ tail script[!12]`	displays the last lines of **script3**

Now, it's your turn...

```
$ touch ?a*
$ pg [st][ah]*
$ ls d*
$ lpr [a-z]*t[0-9]
```

More Shell Metacharacters

The Korn shell can match multiple patterns

`*(pattern|pattern...)`	zero or more occurrences
`?(pattern|pattern...)`	zero or one occurrence
`+(pattern|pattern...)`	one or more occurrences
`@(pattern|pattern...)`	exactly one occurrence
`!(pattern&pattern...)`	anything except

One or more patterns, separated with "**|**" for "or", "**&**" for "and"

Examples:

`*([0-9])`	0 or more ***consecutive*** digits
`?(warning)`	0 or 1 occurrence of "***warning***"
`+([[:upper:]]|[a-z])`	1 or more ***consecutive*** letters
`@([0-9]|abc)`	1 digit or "***abc***"
`!(err*|fail*)`	Word cannot start with "***err***" or "***fail***"

Quoting Metacharacters

Stops normal shell metacharacter processing, including metacharacter expansion

- To form strings

"double quotes"	remove the special meaning of all shell metacharacters ***except*** for the **$**, **`**(backquote), and ****

- To form literal strings

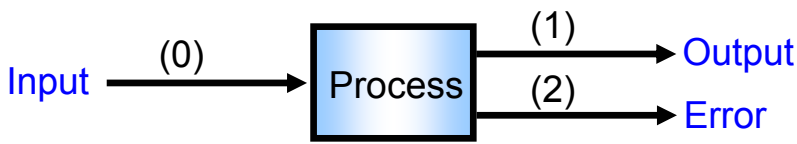
'single quotes'	remove any special meaning of the characters ***within*** the single quotes

- For a literal character

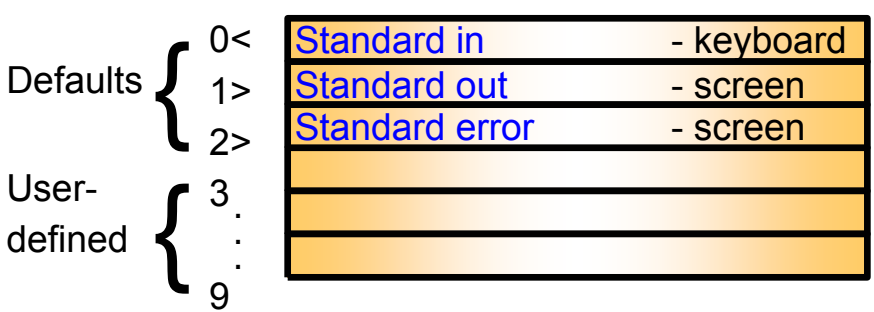
\character	removes the special meaning of the character ***immediately following*** the ****

Process I/O

- Every process has a file descriptor table associated with it



File descriptor table



Input Redirection

Redirecting standard input from a file: **<**

command < filename

```
$ mail marty
Subject: Hello
A letter to see if you are still with us.
<Ctrl-d>
$ _
$ mail -s "Hello" marty < letter
$ _
```

Input may also be given inline. This is called a ***HERE*** document.

```
command << END
text
…
END
```

Output Redirection

Redirecting standard output to a file: **>**

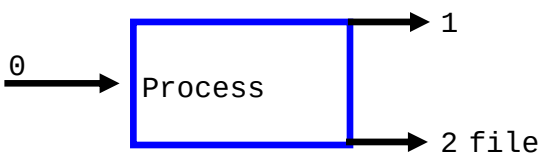
command > filename

```
$ ls /home/chris
data_file script2 script3 shell_prog table1
$ _

$ ls /home/chris > listing
$ _
```

Redirecting standard error output to a file: **2>**

command 2> filename

```
$ cat /home/chris/printout
cat: 0652-050 Cannot open printout.
$ _

$ cat /home/chris/printout 2> errors
$ _
```



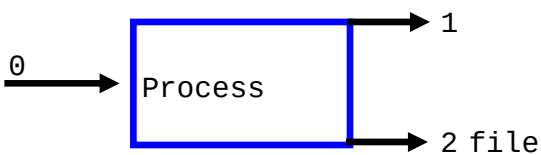
Output Appending

Appending standard output to a file: **>>**
command >> filename

```
$ wc -l /home/chris/script3
    42 /home/chris/script3
$ _

$ wc -l /home/chris/script3 >> line_count
$ _
```

Appending standard error output to a file: **2>>**
command 2>> filename

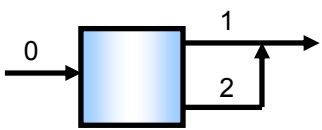
```
$ wc -c /home/chris/characters
wc: 0652-755 Cannot open characters.
$ _

$ wc -w /home/chris/words/ 2>> errors
$ _
```



Association

File descriptors can be joined, so that they output to the same place

```
command > file 2>&1
```



Redirects standard error to join with standard out

What do you think these command do?

```
$ cat message_file 2>&1 > errors_file
$ cat message_file 1>&2
```

Setting I/O or File Descriptors

The built-in shell command **`exec`** allows you to

- Open
- Associate
- Close

file descriptors

`$ exec n>of`	Opens output file descriptor ***n*** to file "***of***"
`$ exec n<if`	Opens input file descriptor ***n*** to read file "***if***"
`$ exec m>&n`	Associates output file descriptor ***m*** with ***n***
`$ exec m<&n`	Associates input file descriptor ***m*** with ***n***
`$ exec n>&-`	Closes output file descriptor ***n***
`$ exec n<&-`	Closes input file descriptor ***n***

Setting I/O Descriptor Examples

To open file descriptor 3 for output to Lee's ***out*** file and file descriptor 4 to Lee's ***err*** file

```
$ exec 3> /home/lee/out
$ exec 4> /home/lee/err
$ date >&3
$ ls /home/lee 2>&4
```

To associate output to file descriptor 3 with file descriptor 4

```
$ exec 3>&4
$ wc -l /home/lee/script3 >&3
$ wc -l /home/lee/table1 >&4
```

To close file descriptors 3 and 4

```
$ exec 3>&-
$ exec 4>&-
```

Pipes

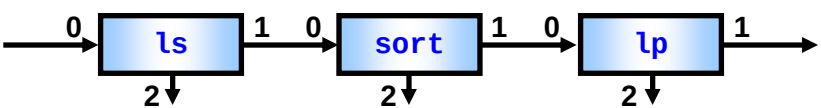
Commands can be joined, so one inputs into the next

```
command1 | command2 | command3
```

Gives a command ***pipeline***

```
$ ls /home/robin | sort -r  |  lp
```

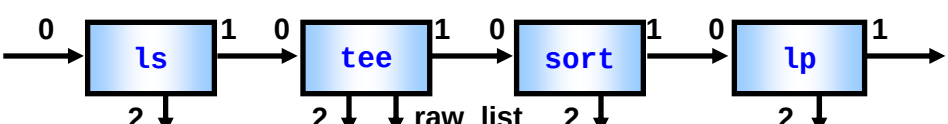
sorts the file list into reverse order, and prints it



Pipelines may have a branch using the ***tee*** command file descriptor

```
$ ls /home/francis | tee raw_list | sort -r | lp
```

saves the unsorted list in the file raw_list



Command Grouping { } and ( )

To combine the output of several commands: **{ }** or **( )**

```
{  command  ;  command  ...  ;  }
```

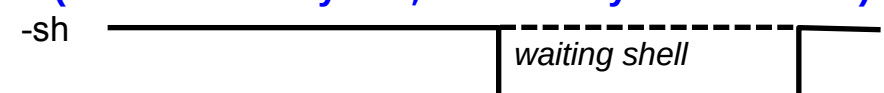
- Runs commands in the current shell
- Directory (or environment) changes remain in effect


```
# { cd /home/lynn ; chown lynn:bin s* ;}
```

```
(  command  ;  command  ...  )
```

- Does not change your current environment

```
# ( cd /home/lynn ; chown lynn:bin d* )
```



This leaves the working directory unchanged on completion

Background Processing

Execute command in the background: **&**

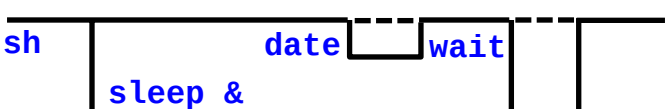
```
$ sleep 999  &
```

Waiting for the end...

```
$ date
Mon Dec 31 11:59:59 EST 2007
$ wait
```

When all background processes have finished

```
$ _
```



Shell Job Control

The shell assigns job numbers to background or suspended processes

- The **`jobs`** command lists your current shell processes and their job ids
- **`<Ctrl-z>`** suspends the current foreground job
- **`bg`** runs a suspended job in background
- **`fg`** brings to foreground a suspended or background job
- Jobs can be stopped with the **`kill`** command
- The **`disown`** command can be used in ksh93

`kill`, **`fg`** and **`bg`** work with the following arguments:

`pid`	process ID
`%job_id`	job ID
`%%` `%+`	current job
`%-`	previous job
`%command`	match a command name
`%?string`	match string in command line

Job Control Example

```
$ cc -o RUNME program_in.c
```
... After some time running this long compilation...
```
Ctrl-z
[2] + 5692 Stopped (SIGTSTP)    cc -o RUNME program_in.c
$ jobs
+ [2] Stopped (SIGTSTP)         cc -o RUNME program_in.c
- [1] Running                   sleep 999 &
$ bg %+
[2] cc -o RUNME program_in.c
$ jobs
+ [2] Running                   cc -o RUNME program_in.c
- [1] Running                   sleep 999 &
$ kill %cc
[2] + 5692 Terminated          cc -o RUNME program_in.c
$ fg %1
sleep 999
$ _
```
Completing the sleep in the foreground...
```
$ jobs
$ _
```

Command Substitution

Command substitution allows you to use the output of a command or group of commands:

- In a variable assignment
- In part of an argument list

Bourne, Korn, and Bash `variable=`command``

-- or --

Korn and Bash `variable=$(command)`

Nesting is possible but can be ***EXTREMELY*** confusing:

```
var=`cmd1 \`cmd2 \\\`cmd3\\\` \` `
```

-- or --

```
var=$(cmd1 $(cmd2 $(cmd3) ) )
```

Command Substitution Examples (1 of 2)

Here is command substitution in action...

```
$ d=$(date)
$ print $d
Fri Feb 29 02:29:00 EST 2008
$ _
$ print "Contents of a file" > tmp_file
$ c=`cat tmp_file`
$ r=$( < tmp_file )
$ print "Cat: $c"
Cat: Contents of a file
$ print "<: $r"
<: Contents of a file
$ _
```

no command, no Sub-Shell

Command Substitution Examples (2 of 2)

Can you explain ***exactly*** what is happening here?

```
$ print "Most recent file: $(ls -t | head -1)"
Most recent file: tmp_file
$ _
$ print "Today is $(date)"
Today is Sat July 07 07:07:07 EDT 2007
$ _
```

Command Line Editing and Recall

vi option for the Korn Shell and emacs for the Bash Shell give:

- Command line editing
- Command recall

```
$ set -o vi or set -o emacs
```

For vi simply press **ESC** to enter editing mode:

- **h** to move the cursor left
- **l** to move the cursor right
- **-** or **k** fetches commands from the history file
- **+** or **j** if you go too far back
- Plus other **vi** commands to perform line editing

For emacs:

- Arrows work, DELete and BackSpace work, else **<Ctrl-b>**, **<Ctrl-f>**, **<Ctrl-d>**
- Up arrow to fetch previous command
  (Check out the Student Notes for more fun stuff!)

Checkpoint

1. What type of file is **`/dev/tty3`**?
2. How could we find out a file type?
3. How can we get **`.kshrc`** to run in an explicit Korn Shell?
4. How can we specify the first character in a file name to be uppercase?
5. How can we ignore error messages from a command?
6. How do you make the normal output of a command appear as error output?
7. How can we group commands, in order to redirect the standard output from all of them?
8. What will **`kill 1`** do?
9. If you have submitted a job to run in foreground, how could you move it to background?
10. How would you set up a command line recall facility?

Unit Summary

- AIX shells
- Hierarchical file-system
- Filenames and types
- Shell scripts
- Invoking shells
- Shell metacharacters: Expansion and quoting
- Redirection -- **<** and **<<** input, **>** and **>>** output, **2>** and **2>>** error
- Setting file descriptors
- Pipes and tees
- Command grouping
- Background processes
- Shell job control
- Shell command editing