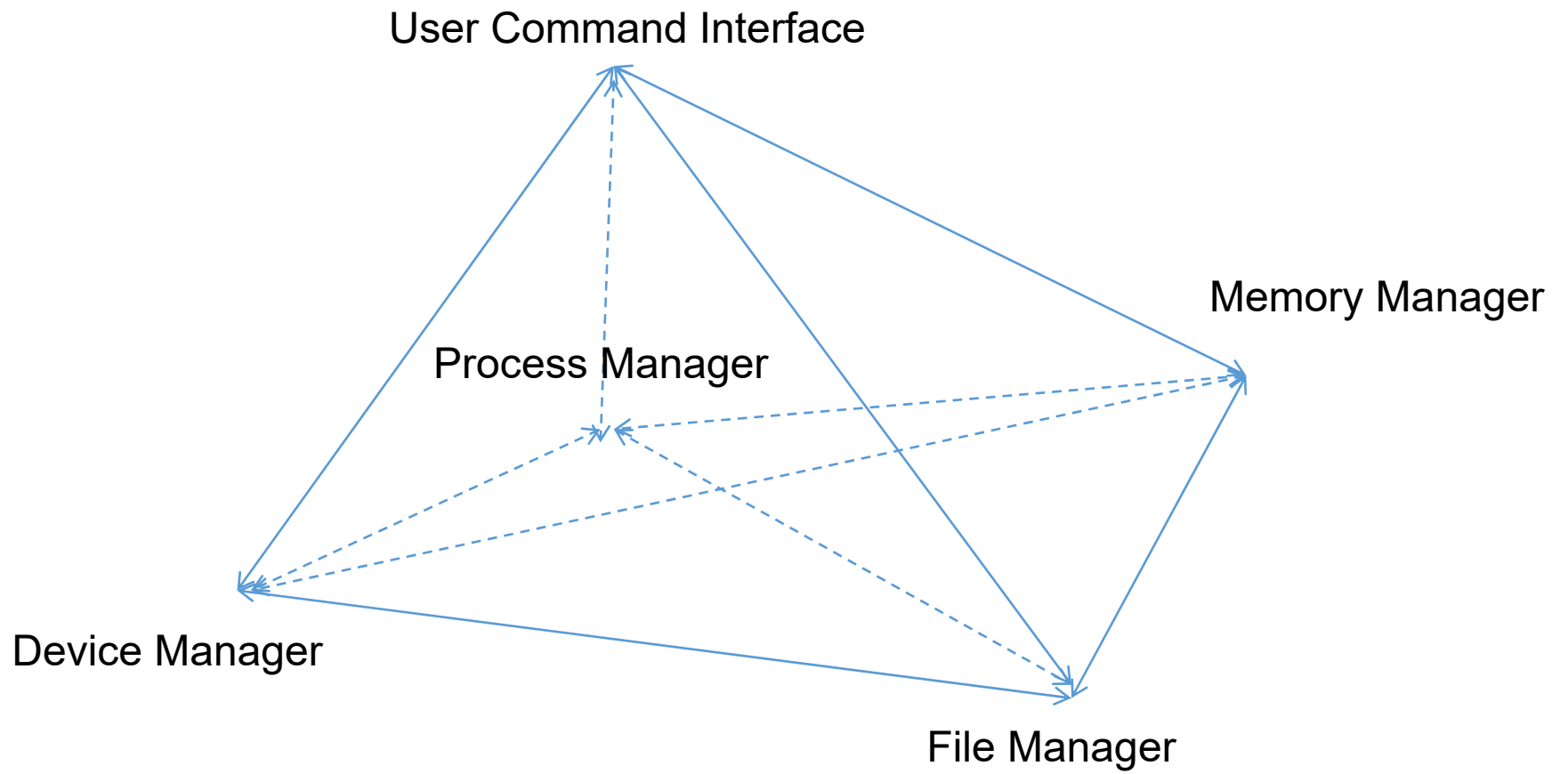


Lecture 2

Operating systems

- An operating system is an executive manager, it's the boss.
- Four essential managers of every operating system:
 - The Memory Manager
 - The Processor Manager
 - The Device Manager
 - The File Manager
- Networked systems have a fifth manager: the Network Manager
- The User Command Interface



Memory Manager

- It checks the validity of requests for memory space
- It allocates a portion of unused memory space to legal request
- It keeps track of who is using which section of memory in a multi-user environment
- It deallocates memory when the time comes to reclaim the memory
- Don't forget, it preserves the space in main memory occupied by the operating system itself

Processor Manager

- It decides how to allocate the CPU.
- It keeps track of the status of each process.
- Once it allocates the processor, it sets up the necessary registers and tables
- It reclaims the processor when the job is finished or when the time is up
 - The Job Scheduler accepts or rejects the incoming jobs
 - The Process Scheduler decides which process gets the CPU and for how long

Device Manager

- It controls the user of devices
- It monitors every device, channel, and control unit
- It tracks the status of each device
- It uses preset policies to determine which process will get a device and for how long
- It allocates and deallocates the devices

File Manager

- It keeps track of every file in the system
- It uses predetermined access policies to enforce restrictions on who has access to which files
- It controls what users are allowed to do with files
- It allocates the resource by opening the file and deallocates it by closing the file

A simplified example

- Let's say someone types a command to execute a program, what will happen?

Examples of such systems include;

Solaris (now Open Solaris) HP/UX



AIX

DEC Unix (now known as True 64)

SGI IRIX (deprecated but still about)



SCO (deprecated but still about)

Mac OS X



Linux (Centos, Ubuntu, Enterprise Redhat)

BSD (Free/Net/Open)

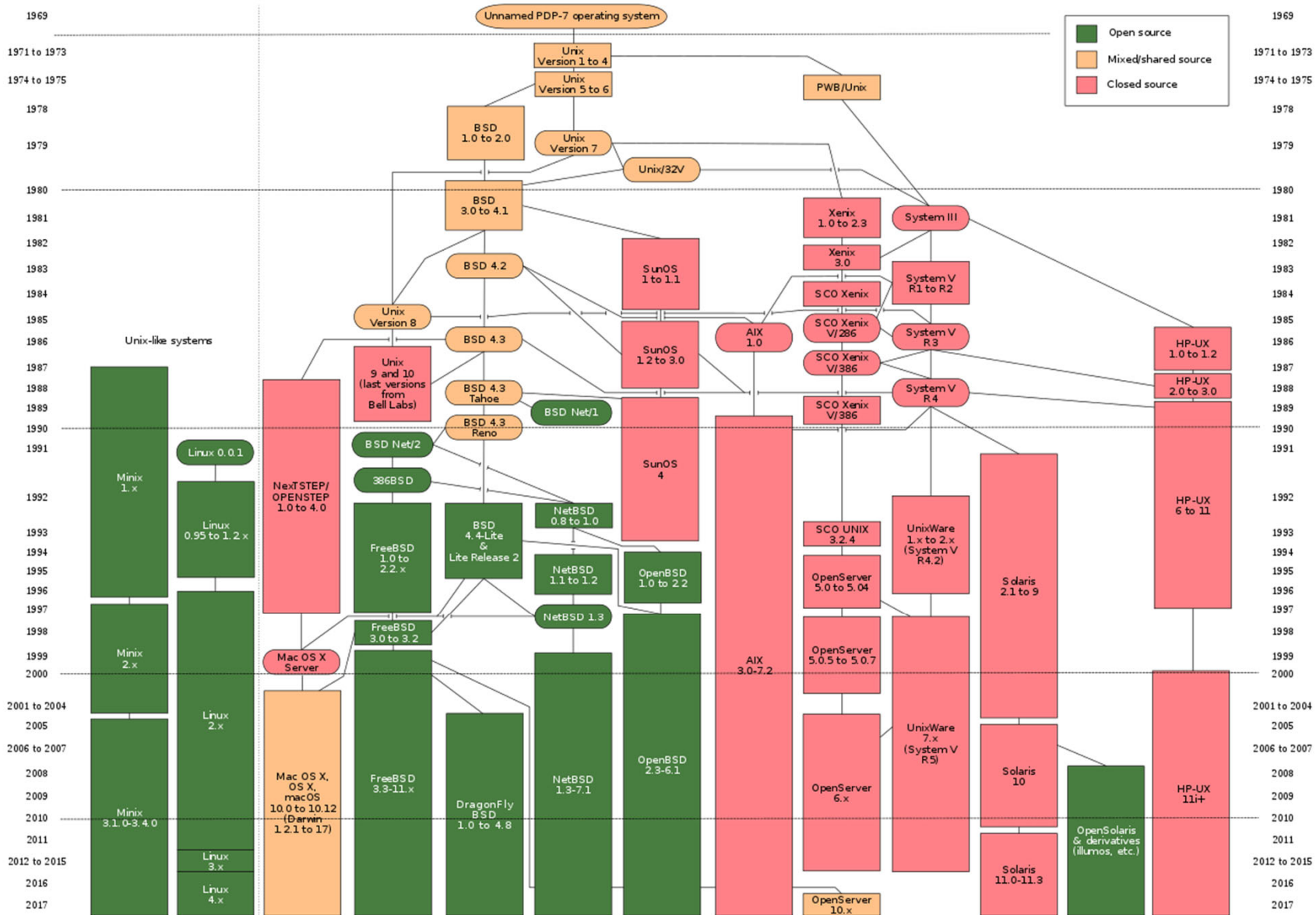


Windows SBS, Server 2008, Server 2003 and even
Windows NT.

VMS, Multics, OS360, OS1100, VM (these are older).

The UNIX operating system

- A portable operating system
- A multi-tasking operating system
- A multi-user computer operating system in a time-sharing configuration
- Provides a standard command-line interface

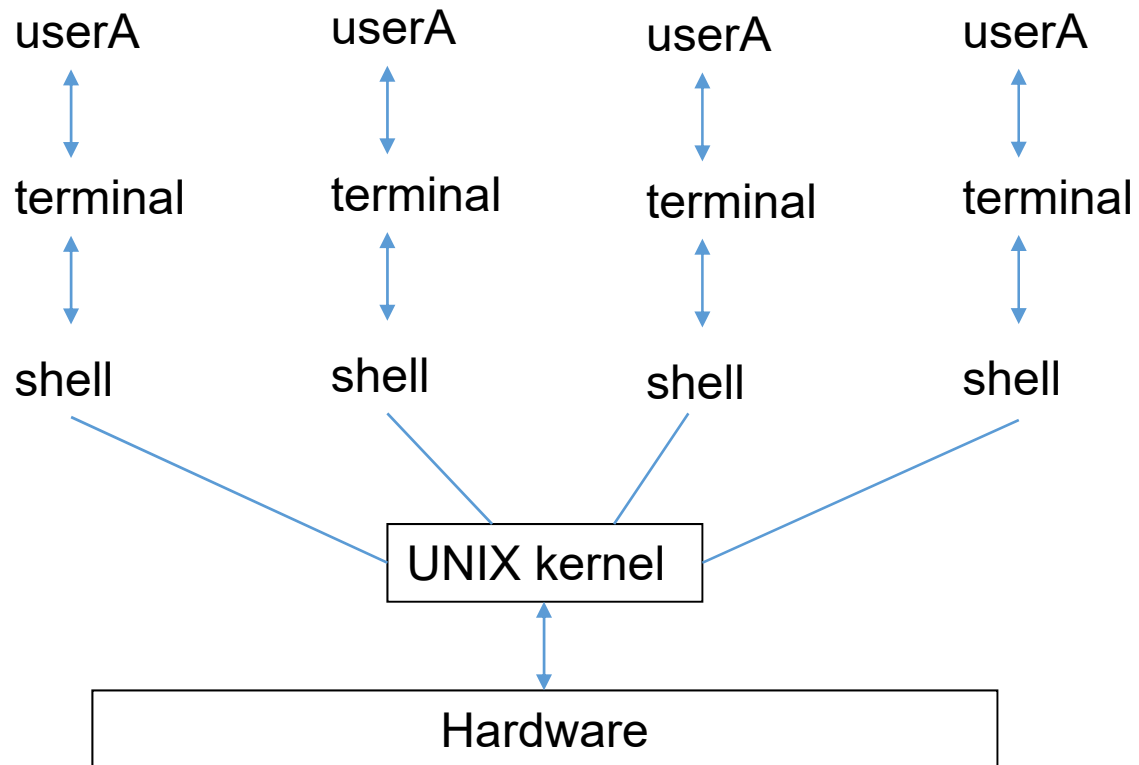


Friction between UNIX and Linux

- Linux was developed in 1991 in Finland
- Linux is a reimplementation and elaboration of the UNIX kernel.
- Linux conforms to the POSIX standard.
- Linux is free, open source, and cooperatively developed.
- Many distributions: Debian, Linux Mint, Red Hat Enterprise, SUSE Linux Enterprise, Ubuntu, etc.
- There are other UNIX flavors like AIX, FreeBSD, HP-UX, Solaris, Unixware

Accessing a UNIX system

- The kernel is a core component that is loaded when you first turn your computer on.
- The kernel loads all other components and serves to centrally control the activities of the computer.
- A user interacts with the kernel through a channel called terminal.
- After logging into a command-line terminal, the user receives a user interface called a shell.



Common UNIX shells

- Bourne shell (sh)
 - POSIX shell (sh)
 - Korn shell (ksh)
 - C shell (csh)
 - BASH shell (bash)
-
- When a user logs into a graphical terminal
 - X Windows is loaded to provide graphical functionality
 - A desktop environment is loaded to standardise the look and feel of X Windows

Basic shell commands for a command-line terminal

- Shell prompt
 - # for the root user
 - \$ for all other users
- Commands indicate the name of the program to execute and are case sensitive
- Options are specific letters that start with a dash (-) and appear after the command name to alter the way the command works.
- Arguments also appear after the command name, yet do not start with a dash. They specify the parameters that the command works upon.

Some common UNIX command

Command	Description
ls	List your files
more filename	Show the first part of a file
emacs filename	Create and edit a file
vi filename	Create and edit a file
mv fn1 fn2	Move a file
cp fn1 fn2	Copy a file
rm filename	Remove a file
diff fn1 fn2	Compare files, and show where they differ
wc filename	How many lines, words, and characters
chmod	Change the permission of a file

Some common UNIX command

Command	Description
gzip filename	Compress a file
gunzip filename	Uncompress a file
gzcat filename	Look at a gzipped file without having to gunzip it
cat filename	Look at the content of the file
lpr filename	Print
mkdir dirname	Make a new directory
cd dirname	Change the directory
pwd	Where you are
grep string fn(s)	Look for the string in the files.

Some common UNIX command

Command	Description
w	Who's logged in and what they are doing
who	Who's logged in and where they are coming from
finger	Display information on system users
last	Give a list of everyone's logins
id	Display the user id and group id
uname -a	Display all information about the system
date	Current date and time
cal	Display the calendar of the current month
exit	Exit the current shell

Some common UNIX command

Command	Description
whoami	Return your username
passwd	Change your password
ps	Display current processes
kill PID	Kill the process with the id you have
quota -v	Show what your disk quota is
echo	Print text to the terminal screen
mail	Send file by email
ssh	Open a secure connection
ftp	Upload or download file
!!	Repeat the previous command

Obtain command help

- It is impractical to memorise the syntax and use of more than 1000 UNIX commands.
- Use the command *man* followed by the command name to retrieve the manual page of that command

man date

- Search the manual pages by keyword

man -k keyword

Manual page section number

Section number	Description
1	Commands that any user may execute
1M	Command that only root user may execute
X1M	X Windows commands that only the root user may execute
2	System calls
3	Library routines and functions
3M	Mathematical functions
3F	Fortran functions
4	File formats
5	Miscellaneous
6	Games
7	Device drivers and interfaces

Shell metacharacters

Metacharacter	Description
\$	Shell variable
&	Background command execution
;	Command termination
< << > >>	I/O redirection
	Command piping
* ? []	Shell wildcards
' " \	Metacharacter quotes
`	Command substitution
() {}	Command grouping

Some examples

- echo my shell is \$SHELL
- echo my shell is '\$SHELL'
- echo my shell is "\$SHELL"
- echo my shell is \SHELL
- echo Today is `date`

Command input and output

- Your shell can manipulate command input and output. Command input and output are represented by labels known as file descriptors.
 - Standard input (stdin), descriptor number 0, by default refers to the user input typed on the keyboard
 - Standard output (stdout), descriptor number 1, by default refers to the terminal screen
 - Standard error (stderr), descriptor number 2, by default refers to the terminal screen

Redirection

- Redirect standard output and standard error from the terminal screen to a file using the > shell metacharacter followed by the pathname of the file
- The shell assumes standard output in the absence of a numeric file descriptor
- You can also redirect both standard output and standard error to the same file using special notation (see next slide)
- You can redirect a file to the standard input of a command using the < metacharacter

Some examples

- `ls /etc/hosts /etc/h`
- `ls /etc/hosts /etc/h 1>goodoutput`
- `ls /etc/hosts /etc/h 2>badoutput`
- `ls /etc/hosts /etc/h 1>goodoutput 2>&1`
- `ls /etc/hosts /etc/h 1>>goodoutput 2>&1`

- You can also redirect a file to the standard input of a command using the < metacharacter
- Some commands only accept files when they are passed by the shell through standard input.
- The tr command is one such command that can be used to replace characters in a file sent via standard input.

```
tr | L </etc/hosts
```

Pipes

- To connect the stdout of one command to the stdin of another, use the | symbol, commonly known as a pipe
- You can use more than one | metacharacter on the command line to pipe information from one command to another
- Use the *tee* command to write to the standard output and a file
- You can also combine the redirection and pipe

Combine redirection and piping

- You can also combine redirection and piping together
 - Input direction must occur at the beginning of the pipe
 - Output direction must occur at the end of the pipe

- # Filter
- Any command that can take from standard input and give to standard output is called a filter.

Command	Description
cut	Prints selected portions of its input lines
sort sort -r	Sorts lines in a file (ASCII) Reverse sorts lines in a file (ASCII)
wc wc -l wc -w wc -c	Counts the number of lines, words, and characters in a file
pr pr -d	Formats a file for printing Formats a file double spaced
tr	Replaces characters in the text of a file
grep	Displays lines in a file that match a regular expression
nl	Numbers lines in a file

cut

- The *cut* command prints selected portions of its input lines. It's commonly used to extract delimited fields.
- The default delimiter is <Tab>
- You can change delimiters with the `-d` option
- The `-f` options specifies which fields to include in the output
- How to enumerate the system's users?

Sort

Option	Meaning
-b	Ignore leading whitespace
-f	Case insensitive sorting
-k	Specify the columns that form the sort key
-n	Compare fields as integer numbers
-r	Reverse sort order
-t	Set field separator (the default is whitespace)
-u	Output unique records only

How to reverse sort the `/etc/group` file by its group ID?

Exercises

- List /etc page by page
- Replace all lowercase a with uppercase A in /etc/passwd and display the content in the terminal screen page by page
- Replace all lowercase a with uppercase A in /etc/passwd, sort the contents by the first character on each line, and then save the results to /home/jyan/results

Shell variables

- Environment variables are typically set by the system and contain information that the system and programs access regularly
- User-defined variables are created by users
- There are other special variables that are useful when executing commands and creating new files and directories

Environment variables

- Use the *set* command or *env* command to see a list of environment variables and their values
 - HOME: the absolute pathname to the user's home directory
 - PWD: the present working directory in the directory tree
 - PATH: a list of directories to search for executable programs
 - SHELL: the absolute pathname of the current shell
 - LOGNAME: the username of the current user used when logging into the shell
 - TZ: the time zone for the system

User-defined variables

- Variable names are prefixed with a dollar sign when their values are referenced
 - `echo $MYVAR`
- You can surround the variable name with curly braces
 - `echo "I am the ${REV}th in the competition."`
- Variables created in the current shell are only available to the current shell, not to the subshells, unless you export them

Setting variables

- To change the value of a variable, specify the variable name immediately followed by a equal sign (=) and the new value
- In some distributions, you must use the *setenv* command followed by the variable name and the new value.
- Variable names can contain alphanumeric characters, dash (-) or underscore (_)
- They must not start with a number
- They are typically capitalised to follow convention

Other variables

- Other variables are not displayed by `set` or `env` commands, and perform specialised functions
 - `UMASK` is a special variable (discussed later)
- Command aliases are shortcuts to commands stored in special variables that can be created and viewed by the `alias` command
 - `alias dw="data;who"`
 - `alias`

Bash scripting

- Bash scripts automate things you'd otherwise be typing on the command line
 - The first line is known as the “shebang” statement and declares the text file to be a script for interpretation by bash:
 - `#!/bin/bash`
 - Comments start with a hash mark (#)
 - Invoke the shell as an interpreter or turn on the execution bit of the script to run the script
 - No need to set the file's extension but some people use `.sh`

Command-line arguments and functions

- Command-line arguments to a script become variables whose names are numbers
 - \$1 is the first command-line argument, and so on
 - \$0 is the name by which the script was invoked
 - \$# contains the number of command-line arguments that were supplied
 - \$* contains all the arguments at once



UbuntuServer1 [Running] - Oracle VM VirtualBox



File Machine View Input Devices Help

```
abc123@ubuntuserverseu:~$ cat script1
#!/bin/bash
function show_usage {
    echo "Usage: $0 source_dir dest_dir"
    exit 1
}

if [ $# -ne 2 ]; then
    show_usage
else
    printf "Source directory is $1\n"
    printf "Destination directory is $2\n"
fi

abc123@ubuntuserverseu:~$ bash script1
Usage: script1 source_dir dest_dir
abc123@ubuntuserverseu:~$ bash script1 dir1 dir2
Source directory is dir1
Destination directory is dir2
abc123@ubuntuserverseu:~$
```

       Right Ctrl

Variable scope

- Variables are global within a script
- Functions can create their own local variables with a local declaration
 - local variablename

What is the output



The screenshot shows a terminal window titled "UbuntuServer1 [Running] - Oracle VM VirtualBox". The terminal displays the following commands and output:

```
abc123@ubuntuserverseu:~$ cat scopetest
#!/bin/bash

function localiser {
    echo "In function localiser, a starts as '$a'"
    local a
    echo "After local declaration, a is '$a'"
    a="localiser version"
    echo "Leaving localiser, a is '$a'"
}

a="test"
echo "Before calling localiser, a is '$a'"
localiser
echo "After calling localiser, a is '$a'"
abc123@ubuntuserverseu:~$
```

Control flow

- if-then and if-then-else
- The terminator for an if statement is fi
- The elif keyword to mean “else if”
- See the example in the next slide. Both the peculiar [] syntax for comparisons and the integer comparison operators (e.g., -eq) are inherited from the original Bourne Shell’s channelling of /bin/test.
- The brackets actually invoke test and are not a syntactic requirement of the if statement

UbuntuServer1 [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

```
abc123@ubuntuserverseu:~$ cat script2
#!/bin/bash
function show_usage {
    echo "Usage: $0 source_dir dest_dir"
    exit 1
}

if [ $# -ne 2 ]; then
    show_usage
elif [ -d $1 ] && [ -d $2 ]; then
    printf "Source directory is $1\n"
    printf "Destination directory is $2\n"
else
    show_usage
fi

abc123@ubuntuserverseu:~$ bash script2 aaa bbb
Usage: script2 source_dir dest_dir
abc123@ubuntuserverseu:~$ _
```

Elementary bash comparison operators

String	Numeric	True if
<code>x = y</code>	<code>x -eq y</code>	x is equal to y
<code>x != y</code>	<code>x -ne y</code>	x is not equal to y
<code>x < y</code>	<code>x -lt y</code>	x is less than y
<code>x <= y</code>	<code>x -le y</code>	x is less than or equal to y
<code>x > y</code>	<code>x -gt y</code>	x is greater than y
<code>x >= y</code>	<code>x -ge y</code>	x is greater than or equal to y
<code>-n x</code>	-	x is not null
<code>-z x</code>	-	x is null

bash file evaluation operators

Operator	True if
-d file	file exists and is a directory
-e file	file exists
-f file	file exists and is a regular file
-r file	You have read permission on the file
-s file	file exists and is not empty
-w file	You have write permission on the file
file1 -nt file2	file1 is newer than file2
file1 -ot file2	file1 is older than file2

Loops

- bash's for...in construct takes some action for a group of values
- Any whitespace-separated list of things, including the contents of a variable, works as a target of for...in
- for loop

```
for (( i=0; i < $count; i++)); do
    ...
done
```
- while loop
- See examples in the next a few slides

```
UbuntuServer1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
abc123@ubuntuserverseu:~$ cat script3
#!/bin/bash

suffix=BACKUP--`date +%Y%m%d`

for script in script*; do
    newname="$script.$suffix"
    echo "Copying $script to $newname..."
    cp $script $newname
done
abc123@ubuntuserverseu:~$ bash script3
Copying script1 to script1.BACKUP--20191124...
Copying script2 to script2.BACKUP--20191124...
Copying script3 to script3.BACKUP--20191124...
abc123@ubuntuserverseu:~$ ls
scopetest  script1.BACKUP--20191124  script2.BACKUP--20191124  script3.BACKUP--20191124
script1    script2                   script3
abc123@ubuntuserverseu:~$
abc123@ubuntuserverseu:~$
```



UbuntuServer1 [Running] - Oracle VM VirtualBox



File Machine View Input Devices Help

```
abc123@ubuntuserverseu:~$ cat script4
#!/bin/bash
```

```
exec 0<$1
while read line; do
    echo "$line"
done
```

```
abc123@ubuntuserverseu:~$ bash script4 script3
#!/bin/bash
```

```
suffix=BACKUP--`date +%Y%m%d`
```

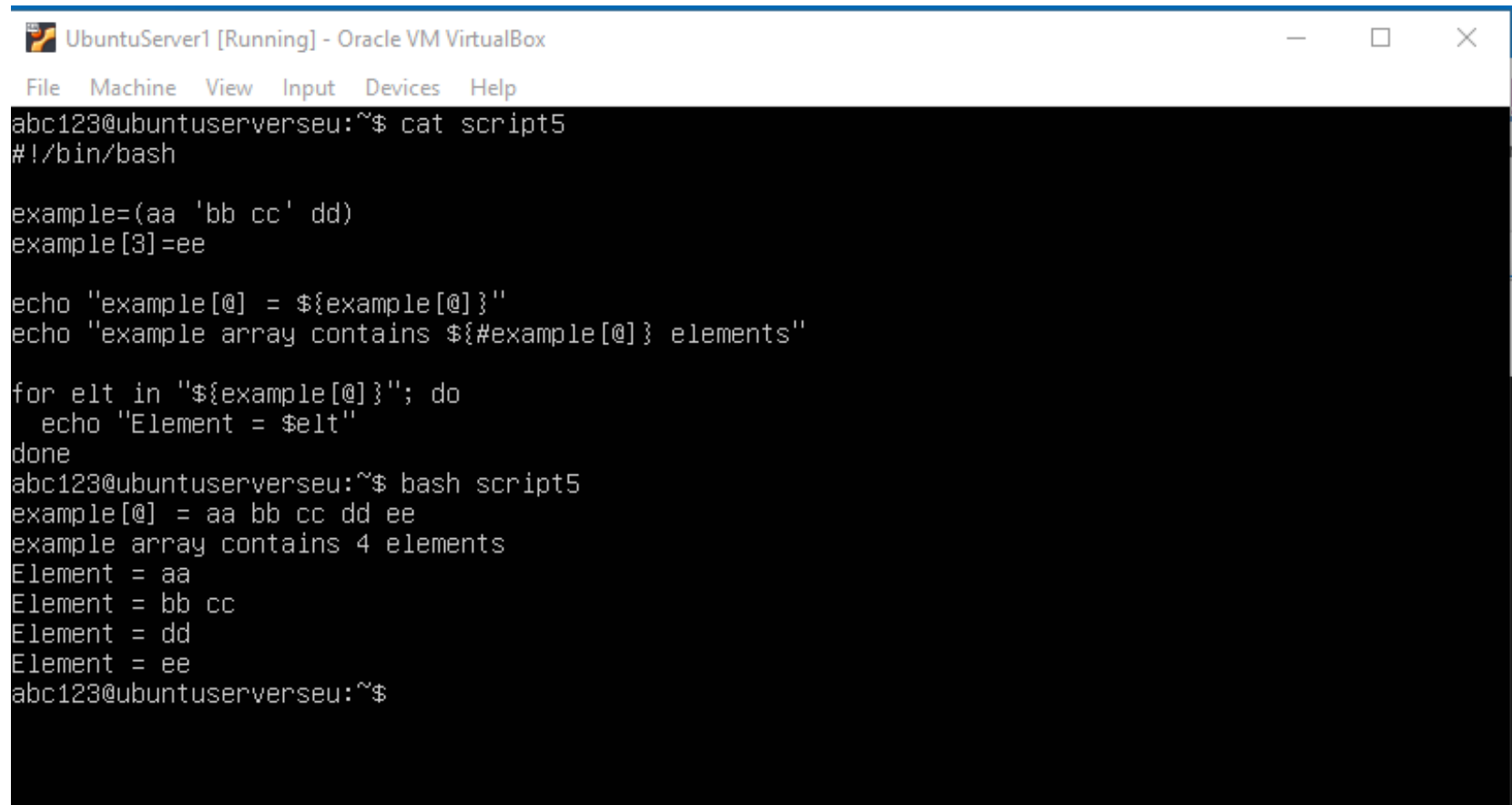
```
for script in script*; do
newname="$script.$suffix"
echo "Copying $script to $newname..."
cp $script $newname
done
```

```
abc123@ubuntuserverseu:~$
```

Arrays

- Arrays are not often used in bash.
- Literal arrays are delimited by parentheses, and the elements are separated by whitespace
 - `example=(aa bb cc dd)`
- Use `${array_name[subscript]}` to access individual elements
 - `echo ${example[1]}`
- The subscripts `*` and `@` refer to the array as a whole
 - `${example[@]}`
 - `${#example[@]}` yields the number of elements in the array

What is the output?



```
UbuntuServer1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
abc123@ubuntuserverseu:~$ cat script5
#!/bin/bash

example=(aa 'bb cc' dd)
example[3]=ee

echo "example[@] = ${example[@]}"
echo "example array contains ${#example[@]} elements"

for elt in "${example[@]}"; do
    echo "Element = $elt"
done
abc123@ubuntuserverseu:~$ bash script5
example[@] = aa bb cc dd ee
example array contains 4 elements
Element = aa
Element = bb cc
Element = dd
Element = ee
abc123@ubuntuserverseu:~$
```

Regular expressions

- Regular expressions enable you to specify a certain pattern of text within a text document.
- Similar to wildcard metacharacters but interpreted by a text tool program.
- More regular expression metacharacters are available than wildcard metacharacters.

Symbol	Description
.	Matches any one character
*	Matches zero or more occurrences of the previous character.
?	Matches zero or one occurrence of the previous character.
+	Matches one or more occurrences of the previous character.
[chars]	Matches any character from a given set, e.g., [a-z]
[^chars]	Matches any character not in a given set, e.g., [^a-z]
()	Matches either the element to its left or to its right.

Symbol	Description
<code>^</code>	Matches the beginning of a line.
<code>\$</code>	Matches the end of a line.
<code>\w</code>	Matches any word character. [a-zA-Z0-9_]
<code>\s</code>	Matches any whitespace character.
<code>\d</code>	Matches any digit. [0-9]
<code>{n}</code>	Matches exactly n instances of the preceding element.
<code>{min,}</code>	Matches at least min instances of the preceding element.
<code>{min,max}</code>	Matches any number of instances of the preceding element from min to max.

Examples

- `^\d{4}$`
- `Letter[^1238]`
- `(mother|father)`
- `M[ou]’?am+[ae]r ([Aeae]l[-])?[GKQ]h?[aeu]+([dhz][dhz]?){1,2}af[iy]`

The grep command

- Global Regular Expression Print searches for information using regular expressions
- It displays the lines of text that match extended regular expressions.
 - `grep "hello" file1`
 - `grep -v "hello" file1`
 - `grep -i "hello" file1`
 - `grep "^|" file1`

Perl programming

- Perl is a scripting language that offers vastly more power than bash
- It is a better choice for systems administration than traditional programming language (the other choice is Python)
- Perl can do more in fewer lines of code, with less painful debugging
- Perl's catch phrase is that "there's more than one way to do it"

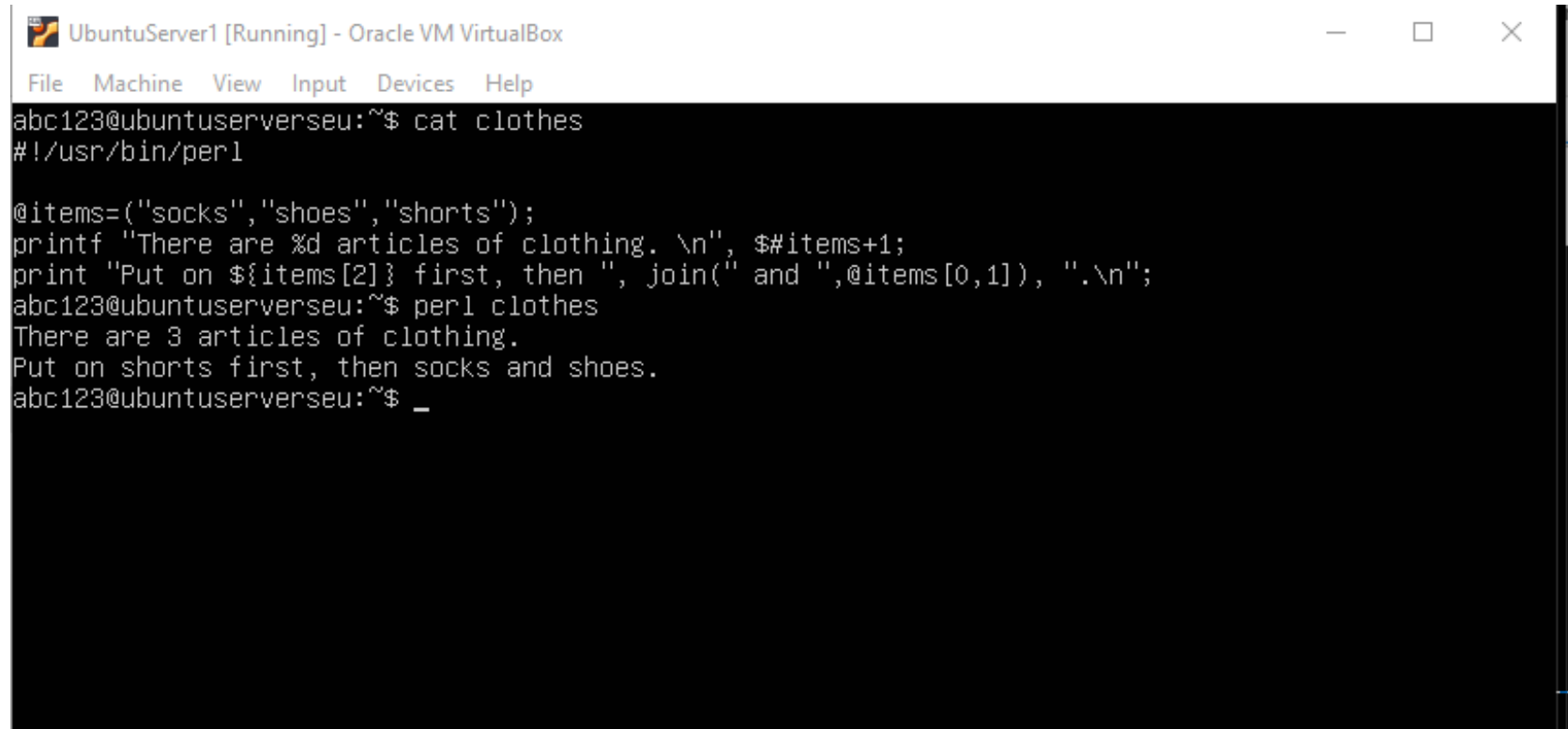
Perl programming

- Perl statements are separated by semicolons
- Comments start with a hash mark (#)
- Blocks of statements are enclosed in curly braces
- Lines in a perl script are not shell commands; they are perl codes
- The first line is known as the “shebang” statement and declares the text file to be a script for interpretation by perl:
 - `#!/usr/bin/perl`

Variables and arrays

- Perl has three fundamental data types:
 - Scalars: scalar variables start with \$
 - \$a="this is an example";
 - \$count=0;
 - Arrays: array variables start with @
 - @items=("socks", "shoes", "shorts");
 - Array subscripting begins at zero, the index of the highest element is \$#items
 - The array @ARGV contains the script's command-line arguments
 - Hashes (aka associative arrays): hash variables start with %

What is the output?



The screenshot shows a terminal window titled "UbuntuServer1 [Running] - Oracle VM VirtualBox". The terminal content is as follows:

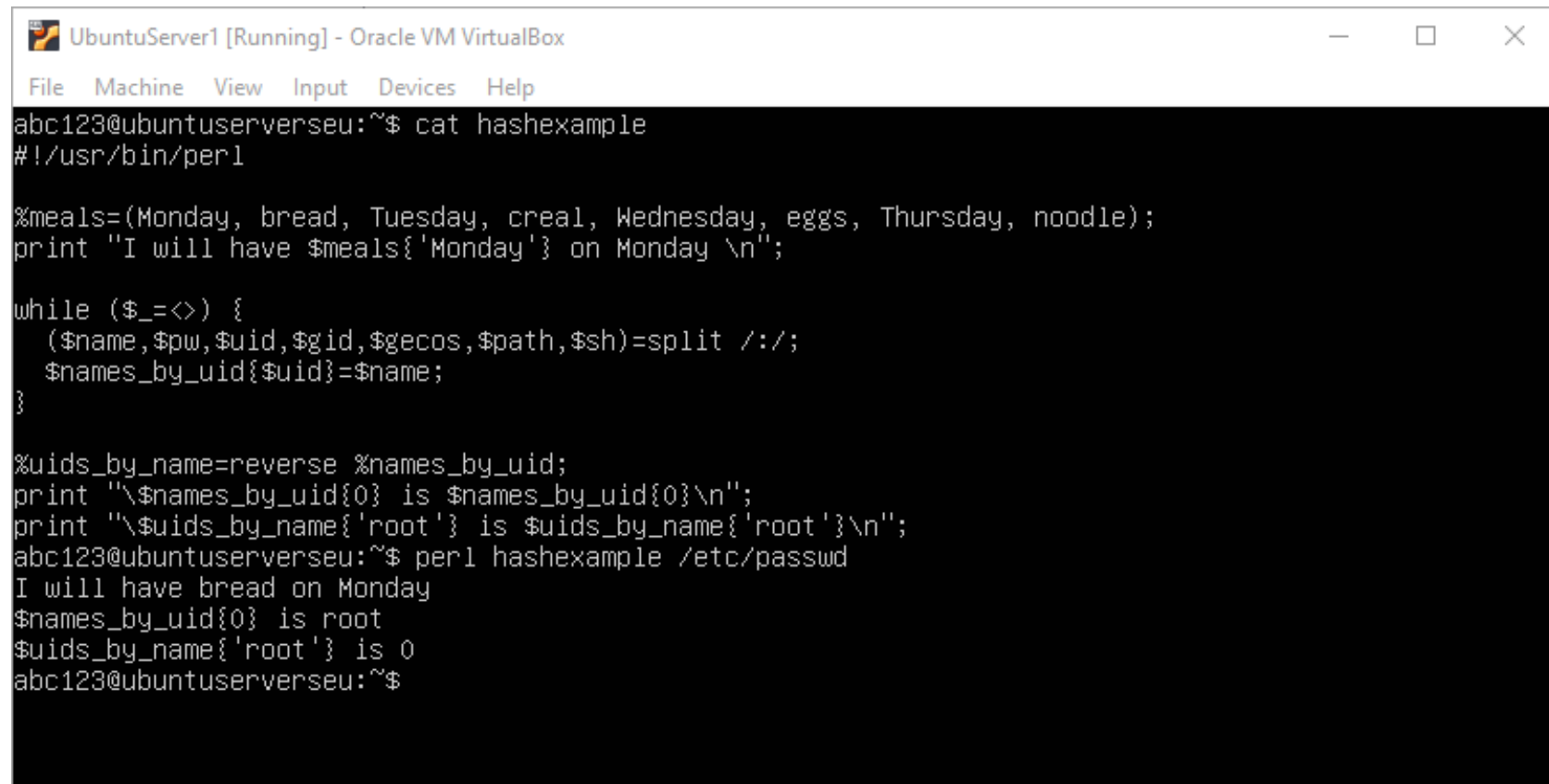
```
File Machine View Input Devices Help
abc123@ubuntuserverseu:~$ cat clothes
#!/usr/bin/perl

@items=("socks","shoes","shorts");
printf "There are %d articles of clothing. \n", $#items+1;
print "Put on ${items[2]} first, then ", join(" and ",@items[0,1]), ".\n";
abc123@ubuntuserverseu:~$ perl clothes
There are 3 articles of clothing.
Put on shorts first, then socks and shoes.
abc123@ubuntuserverseu:~$ _
```

Hashes

- A hash represents a set of key/value pairs.
- In other words, a hash is an array whose subscripts are arbitrary scalar values
- Subscripting is indicated with curly braces rather than square brackets
 - `$myhash{'ron'}`

What is the output?



```
UbuntuServer1 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
abc123@ubuntuserverseu:~$ cat hashexample
#!/usr/bin/perl

%meals=(Monday, bread, Tuesday, creal, Wednesday, eggs, Thursday, noodle);
print "I will have $meals{'Monday'} on Monday \n";

while ($_=<>) {
    ($name,$pw,$uid,$gid,$gecos,$path,$sh)=split /:/;
    $names_by_uid{$uid}=$name;
}

%uids_by_name=reverse %names_by_uid;
print "\$names_by_uid{0} is $names_by_uid{0}\n";
print "\$uids_by_name{'root'} is $uids_by_name{'root'}\n";
abc123@ubuntuserverseu:~$ perl hashexample /etc/passwd
I will have bread on Monday
$names_by_uid{0} is root
$uids_by_name{'root'} is 0
abc123@ubuntuserverseu:~$
```


Elementary Perl comparison operators

String	Numeric	True if
<code>x eq y</code>	<code>x = y</code>	x is equal to y
<code>x ne y</code>	<code>x != y</code>	x is not equal to y
<code>x lt y</code>	<code>x < y</code>	x is less than y
<code>x le y</code>	<code>x <= y</code>	x is less than or equal to y
<code>x gt y</code>	<code>x > y</code>	x is greater than y
<code>x ge y</code>	<code>x >= y</code>	x is greater than or equal to y

You get all the file-testing operators shown in Slide 48 except for the `-nt` and `-ot` operators

Control flow

- Control flow in Perl is very similar to that in bash
- if construct has no then keyword or terminating word

Input and output

- When you open a file for reading or writing, you define a filehandle to identify the channel
 - `open($fh, '<', $inputfilename) || die "Couldn't open file.";`
 - `open($fh, '>', $outputfilename) || die "Couldn't open file.";`

Questions?