This module:-

- FOR VISITING STUDENTS
  - Is often NOT accepted as a substitute for a basic OS theory course back home, because it isn't!
- Is NOT an introduction to basic OS theory
  - which is now covered in a Sem 2 course
    - Which I used teach
  - Lecturer concerned prefers to do it all
  - Eases coordination between theory and practice
  - Probably not much overlap between the courses

This lecture is a course sampler..

  - So you know what it entails, to help you decide whether it's right for you,if the course is an option
- ...and if it's core don't panic... only simple syntax
- No mind bending concepts...We get there a step at a time...
- Don't expect to understand this collection of slides fully yet : merely a selection of unmodified slides throughout course for illustration
- Comments relate only to CS2503 Operating Systems I course
- Diploma students, with no previous IT used do it,
  - already had made the biggest jump and stretched themselves already more than the course would
  - Even complete newbies will settle in within a few weeks
  - It will all come together in the end!

This module is

- Designed to provide hands on practice in
  - Command line
  - Filesystem
  - Text-processing
    - Regex, awk, sed families etc.
  - Bash scripting
  - For basic requirements: e.g. system monitoring, log processing
- While you may not be able to
  - Design and write advanced complex scripts
- You should not be freaked by scripts and be able to
  - Run and modify even advanced scripts to suit your own requirements

Some of the most...

- boring,
- terse,
- error prone,
- standard-free,
- Incomprehensible syntax, almost

- useful,
- powerful,
- jam-freeing,
- flexible
- Incomparably powerful tools

you're ever liable to encounter
Are here, like all of life!
And they're honed for speed, C, buffers...

Lots done – and lots we'd like to do

- This course drags you through most of the desperate detail that you would not do yourself
  - So you can face almost anything in this area
- But does not do the nice cool interesting stuff
  - Which you can easily read up yourself
- So while it does not look particularly 'cool' interviewers might think you are cool!
- Due to time pressure it omits
  - probably interacting concurrent processes..
  - But even the experts there have issues
  - Git, containers etc. but who knows… time will tell!

'Official' specs:- CS2503 Operating Systems 1

- Module Objective: Students will learn about file system management and scripting in modern operating systems, using Unix as a case study.
- Module Content: Operating Systems from an architectural perspective. The Unix Operating System. Shell scripting. Environment Variables. File protection mechanisms.
- Learning Outcomes: On successful completion of this module, students should be able to:
  - Use the Unix OS (Operating System) at the shell level.
  - Know: Basic file related commands; Input and output redirection; the file protection mechanism; Commonly used Unix utilities; Shell scripting.
  - Understand: File and memory protection mechanisms.

‘Official’ specs for follow up:- CS2506 Operating Systems II

- Module Objective: Students will learn about process and instruction execution and management in modern operating systems; and they will learn about systems programming.
- Module Content: Systems programming: Memory management and pointer manipulation; Large-scale application organization. Libraries. Makefiles. Devices, files and IO. Processes and resources. Scheduling. Device organisation and management. Interrupts. User/system state transitions. Interprocess communication and synchronisation. Operating system threads. Operating system APIs
- Learning Outcomes: On successful completion of this module, students should be able to:
  - Have a good understanding of system programming techniques, and of OS APIs (Application Programming Interfaces) for file I/O, process creation, and interprocess communication.
  - Know: Processes, exit statuses, and process control.
  - Understand: The difference between processes and threads; Critical section; Race conditions; Deadlock and starvation; Mutexes and semaphores; Virtual memory; Interrupts; The process cycle.
  - Learn to: Implement programs and libraries; Create and maintain make files; Dynamically allocate and free memory; use OS APIs for I/O, process creation, and inter-process communication.

Basic Course Overview – just to get going!

- Basic commands
  - Files
  - Directories
  - Access management - modes
  - Disk space management
  - Processes & process management
- Editors
  - Basic ubiquitous :
    - Ed /Sed / Vi(m) / pico/nano
  - GUI
  - IDE
- Tools & Utilities
  - Grep – basically to find strings, either names, contents, but flexible & programmable with regular expressions : a way of specifying string patterns (grep:- global regular expression print)
  - AWK (open source gawk) -
- Extras
  - bash scripting
  - ssh/VPN
- System Administration
  - System configuration
  - User administration
  - Software installation / compiles etc.

No point in reinventing the wheel,
So rather than developing a script,
Check commands and options,
And tools and options
Before attempting to write a script

Then write a script if unavoidable
Simplifying it by Using the powerful
Commands & options, and tools
covered above!

CS2503 – Systems Environment.

- It’s all about the environment!?
  - GUI vs CLI
  - Shells
  - Editors
  - Modes
  - Directories & Permissions
  - Configurations & Privileges
  - Security & Firewalls
  - Technologies, tools & techniques…

Miscellaneous topics en route - 1

- Dual/ Multi Boot Machines
  - Rationale : - past, present & likely future
  - OS installation
  - Disk Partitioning
- Virtual Machines
  - Rationale : - past, present & likely future
  - Examples : Virtual PC, Virtual Box
- Software installation & Configuration
- Compilation from source – maybe a follow on module
- Code Version control: git, cvs, subversion

Miscellaneous topics en route – 2

- Remote Access
  - Command line .. ssh
  - Desktop ..oodles
- File Transfer
  - Command line …ftp
  - GUI … oodles Filezilla etc.
  - Or use Google Drive, Dropbox, Tonido etc.
    - (avoid if security an issue, hacking risk)
    - can incur unnecessary network load
- Backup & (Disaster) Recovery

Structure & sections

- It’s all Text :
  - programs, settings, filesystem
- Putting
  - the right thing
    - Editors : nano, Kate, ed, sed, vi, vim, emacs,
  - in the right place
    - Filesystem commands
- Now where did I put it?
  - Find, grep, regular expressions (locate files needs system to build index)
- How much time and space is it taking?
  - Account for Disk use / processor use (AWK, ps, top etc.)
- Playing safe
  - Data Integrity … Keep a copy, just in case…
    - Backups
    - Maintenance scripts.

Why all these weird cumbersome editing tools!?

- Think back –
  - No VDU screens - only serial printer teletypes
    - Noisy, unreliable, (paper tears, ink ribbon wears)
    - and materials relatively costly
    - Slow (perhaps 1 letter per 1-2 seconds)
    - So need max info. with least printing…
      - Also less bandwidth required…
        - Although most were wired directly to machine
        - So less comms load on limited machines
        - interrupts are costly, need to flush and reload both code and context for new process
- Therefore
  - Terse input and output & regular expressions
  - Terse powerful commands… 2-letters – 2-finger typists

Unix editors – historical..!

- **ed** : a terminal line editor … 'twould do you in! - don't use normally
  - But it makes you appreciate
    - progress : visual GUI editors
    - & tradition : why things often are the way they were
    - & regular expressions!
- **sed** : stream editor : can change the WORLD
  - can change files on the fly from a program of editing commands – very fast & powerful…
  - great for 'big'
    - edits, if it's right .. Global change of name on a pile of files
    - Eejits, if it's wrong! .. Can it make a big mess!
  - Just sed the world and all will be changed…

    (to quote a tough centurion!)

    (but don't think you are God – that was the devil's downfall !)
- **vi** : visual ;-) …. replaced with vim & variants, none that visual,
  - Like nano on dope

Overview of character processing

- **tr** – basic character substitution – **tr**anslate
  - As an introduction to stream editing…
  - Stream!? … a stream of characters.. e.g. a file
  - as distinct from interactive online editing…
  - Stream editing can be programmed and run in batch (non-interactive) mode … suited for large jobs…
- Regexp – regular expressions – way to specify patters
  - Ubiquitous throughout CS… used everywhere, in
    - interactive editors – ed, vi, vim, emacs … and derivatives… (incl Microsoft) etc.
    - stream editors – sed
    - Command arguments …e.g. ls *.txt
    - SQL queries
    - Language specification …
      - although (E)BNF is used to specify grammars
- **grep** family – find (for filenames) generally uses grep
  - **g**lobal **r**egular **e**xpression **p**rint'' – but find is more meaningful!

Regular Expressions

- You can use and even administer Unix systems without understanding *regular expressions* but you will be doing things the hard way
- *Regular expressions* are endemic to Unix
  - vi, ed, sed, and emacs
  - awk, Tcl, Perl and Python, SQL … even search engines
  - grep, egrep, fgrep
- *Regular expressions* descend from a fundamental concept in Computer Science called *regular grammars*, from *finite automata* theory
  - Just a fancy way of defining & validating a sequence of tokens … like algebra

So What Is a Regular Expression?

- A *regular expression* is simply a description of a pattern that describes a set of possible characters in an input string : avoids full listing of all possibilities
- Weve already seen some simple examples of *regular expressions* (known as *regex* from here on)
  - In vi when searching :/c[aou]t searches for cat, cot, or cut… the /tells it to search for the following string…
  - In the shell
    - ls *.txt

      » but in string regexp c* implies 0 or more c's
    - cat chapter? … jams all chapters together …. Order !?
    - cp Week[1234].pdf /home/urid/Oct

Regex Examples

- Variable names in C
  - One alphabetic,

    followed by one or more alphanumeric
  - [a-zA-Z_] [a-zA-Z_0-9]*
  - [[:alpha:]_] [[:alnum:]_]*
- Dollar amount with optional cents
  - \$[0-9]+(\.[0-9][0-9])?
- Time of day
  - (1[012]|[1-9]):[0-5][0-9] (am|pm)
- e.g. search for all instances of am,

  and get ham, spam & damn!

  - regex can select!

ed Tradition & historical precedence

NB: don't try to learn ed, either now or later,
But it's a good intro. To see how & why regex were/are used

Principles still
- applicable to modern systems
- retained on some.

Regular expressions ubiquitous / endemic

Buffers : originally saved
- Original copy from accidental changes…
  - So trivial to revert to saved option
- Space on low memory machines :
  - only edits small part of file at a time!

ed - line editor : basic … but helps with basics

- File is copied into buffer, /tmp/ed.*
- so buffer refers to copy of file being edited, with changes made
  - being written and saved using w
  - or simply abandoned without saving by using q - quit.
- Default position is set to last line.
- If filename does not exist, may give warning and create it, automatically entering insert mode, for first line.
- Two modes
  - Command mode - works on the lines e.g. d for delete etc.
    - ^C generally interrupts current command
  - Insert mode:
    - i, a - inserts / appends text before / after current line
    - Stopped by . alone on a single line

Tradition
- Buffers : save
  - Original copy from accidental changes…revert to saved option
  - Space on low memory machines : only edits a small part of the file at a time!

Other Substitute Examples

- s/cat/dog/
  - Substitute dog for the first occurrence of cat in *pattern space*
- s/Tom/Dick/2
  - Substitutes Dick for the second occurrence of Tom in the *pattern space*
- s/wood/plastic/p
  - Substitutes plastic for the first occurrence of wood and outputs (prints) *pattern space*
- s/Mr/Dr/g
  - Substitutes Dr for every occurrence of Mr in *pattern space*

sed, The **S**tream **Ed**itor

- Does it on the fly, e.g. if a company merges or changes its name, all references everywhere changed in a ~line
- *sed* is descended from our friend, *ed*
  - Both operate on files one line at a time
  - Both use a similar command format
    - *[address] operation [argument]*
  - *both* can use command scripts
    - *ed filename <script_file*
- *sed* cannot be used interactively, unlike ed
- will only take commands from a script or command line
- All input to comes from standard input and goes to standard output, which can be redirected, and output must be redirected for changes to be saved.
- There is also the option to supply an edit filename containing edit scripts on the command line

Visual Editor – vi (& vim –vi improved)

- vi is a screen editor, the unit of change is *character* rather than a line
- vi is found on most ~~Unix~~ installations, even Win
- Unlike MS-Word, WordPerfect, or other word processors, *vi* files consist of plain ASCII text
  - There are no special formatting codes
- However, *vi* is so feature rich that
  - it appears daunting to learn
  - takes considerable study and practice to master
  - If used by a master, is very fast 'poetry in motion'!?
- Vim – can function as a full programming IDE

Kate - (KDE Advanced Text Editor)

- Big plus –
  - Easy to use
  - Gives 3 basic viewing options (last 2 optional)
    - The editor window
    - The filesystem (to the left)
    - A CLI terminal on the bottom
  - multi document editor, based on a rewritten version of the kwrite editing widget of KDE.
- multi-view editor which allows user to view
  - several instances of the same document with all instances being synced,
  - more files at the same time for easy reference or simultaneous editing.
- The terminal emulation and sidebar are docked windows
  - that can be removed/reinserted in the main window, according to preference.

nano (the Linux pico)

- Simple graphical editor
- Very easy to learn
- Shows command options at bottom.
- Configurable for basic programming
- Available via SSH...
- Will do most things fine.
- Yerrra 'twill do!
- *** as of 9/9/2018 version 3 released ***
  - Reads files 70% faster
  - Handles ASCII nearly twice as fast
  - Handy keybinding :
    Ctrl+(Shift)+Del – del next/(prev) word

Grep ... Finding things... find & grep

- grep comes from the ed search command
  '**g**lobal **r**egular **e**xpression **p**rint' or g\re\p
- This was such a useful command that it was written as a standalone utility
- There are two other non-POSIX variants of grep
  - *egrep* extended grep
  - *fgrep* fixed grep
- *grep* is the answer to the moments where you know you want a the file that contains a specific phrase but you can't remember it's name
- Most GUI's provide a 'find' interface for grep.

Bash Scripting

- Basically use a minimalist language
  - With basic
    - Input, output,
    - Character processing
    - Flow of control statements
- To issue OS commands.

Do not expect to understand all now, they are merely for illustration.

vim first_script.sh – passing values...

```
#! /bin/bash
greeting="Hi!"
echo -e " \n\n"
echo " ******** ************ ************ ************ ******** ********"
echo " ********      $greeting - We're now up and running!?     ********"
echo " ******** ************ ************ ************ ******** ********"
echo -e " \n\n"

echo -e "The script name is:- \t\t $0"
echo -e "The parameter list is:- \t $*"

echo -e "\nParameters :-"
echo -e "First:- \t $1"
echo -e "Second:- \t $2"
echo -e "Third:- \t $3"

echo -e "\n\n\nHere are some environment variables"
echo -e "HOME directory          \t $HOME"
echo -e "PATH            \n$PATH \n\n"
exit 0
```


bash first_script.sh a b c d

```
******** ************ ************ ************ ******** ********
********      Hi! - We're now up and running!?      ********
******** ************ ************ ************ ******** ********
The script name is:-    first_script.sh
The parameter list is:-     a b c d

Parameters :-
First:-         a
Second:-        b
Third:-         c

Here are some environment variables
HOME directory          /users/csdipact2012/jsad1
PATH
/users/csdipact2012/jsad1/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/usr/games
```

Environment Variables : convenient, can be reference by name throughout commands/programs

HOME - self explanatory
PATH – sequence (determines order, first met is taken) of directories which the shell looks for an executable command or program (including ones you write or modify),

Software engineering

- Complexity management : Divide and conquer
- Early approaches; suitable for simple programs
  - Input -> process -> output ... one liners etc.
  - Algorithms + Data Structures = programs
  - But chaos still resulted from
    - Intertwined code : spaghetti code
    - Unprotected data operations : stealthy mistakes on data
- Resolved by further separation ..
  - Object oriented
    - separate problem into interacting objects
    - Objects encapsulate data with permitted operations to avoid errors
  - Model View Controller – further separation & demarcation of function (basically a higher level version of Input -> process -> output)
    - Each function can be changed without affecting the others,
      - provided agreed uniform interfaces are used
-

Snakes and ladders – like Software!?

- This is just a simple illustration of the main control flow statements using bash commands and syntax,
- With examples of alternate (often dense) syntax and implementations for same
- Don't look for any deeper meaning or relevance…

  (similarities with software development is entirely accidental...)
- Most people are familiar with the game or something like it…
  - For convenience & speed for playing,
    - this 'board' is limited to 10 rather than 100 squares.
- Syntax colouring is … well from an older system
- code is an image, for easy scaling.
- And don't expect to understand the code now… just a sample

Snakes and ladders
block structured pseudocode algorithm

```
Starting position (pos =1)        (Start position, should it be 0?))
While not at end position         (10 for easy example, 100 for real)
    throw a die  from [1-6] or 999 to exit      (invite user to 'throw a die')
    case      (validate input!)
    [1-6] valid input, move by throw pos += throw
    999 break out of loop and finish      (emergency exit no…999!)
    otherwise : invalid input : ignore & loop again
    endcase
    if pos exceeds end            (gotta stay on the board!)
    then retreat from end by excess
    endif
    if (pos==3) or (pos==6)       (3 or 6, take a tree of 4 sticks)
    then jump forward (via ladder) by 4
    endif
    if pos is a multiple of 4     (divisible by 4, goto the floor!)
    then jump back (via snake) by 3
    endif
endwhile
```

```
#!/bin/bash
# THIS A DEMO OF COMPACT CRYPTIC CODING WITH CONCISE LOGIC EXPRESSIONS
# - AND IS BEST AVOIDED FOR CLEAR EASILY MAINTAINABLE CODE
# IT IS OF NO BENEFIT, OTHER THAN DEMONSTRATING THE USE OF CONTROL FLOW AND LOGIC TESTS IN BASH
# The example is a corny die cast case of Snakes and Ladders - without either!
# and stops at 10 rather than 100 to save time and patience!
#
# note that \followed immediately by RETURN key, escapes the RETURN key permitting line continuation.

pos=1 ;                 # starting position
while (( 10 - $pos ))   # the perverse test for zero, which fails to 1 when ((expr evaluates to zer0))
                        # Safer to use the less confusing [[ "$pos" -ne 10 ]] or test "$pos" -ne 10
do
        printf "\n*** Give another throw of the die!?  ***\n"
        echo -e "\n*** Either a number from [1-6] ... \n ...OR 999 to escape! ***"
        read throw
        case "$throw" in

                # checks valid range for die throw using case pattern regexp range [1-6])
                # checks numbers using both methods: expr and $(( ... ))
                [1-6])  pos=`expr $pos + $throw` ; over=$(( $pos - 10 )) ;;

                # breaks out of main loop to finish
                999)    echo -e "\n\n\n Bye! And have a nice die! \n\n\n" ; break;;

                # ignore faulty input for rest of this pass
                *)      echo -e "\n***  Some die!? it threw a: $throw" ; continue
        esac

                # if the throw value would exceed the end of the board, just count back the excess
        [[ $over -gt 0 ]] && pos=$(( 10 - $over )) \
                && echo -e "\n Overshot - bounce back!" || echo -e "\nStepping out -"

                # ladder - if divisible by 3, go up 4, except for 9 (i.e. 3 or 6)
        [[ $pos -eq 3 ]] || [[ $pos -eq 6 ]] && pos=$(( $pos + 4 )) \
                && echo -e "^^^^ a multiple of 3 - except for 9 - go up by 4! ^^^^\n"

                # snake - slide back 3 towards the floor if pos evenly divisible by 4
        ! (( $pos % 4 )) && pos=$(( $pos - 3 )) \
                && echo -e "vvv a multiple of 4, go 3 to the floor! vvv\n"

                # state current position
        echo -e "\nNow at : $pos"

done
printf "\n******************\n    Finished!    \n******************\n"

exit 0
```


The main loop – expanded for clarity!

```
#!/bin/bash

pos=1 ;                 # starting position

# the perverse test for zero, which fails to 1 when ((expr evaluates to zer0))
# Safer to use the less confusing [[ "$pos" -ne 10 ]] or test "$pos" -ne 10

while (( 10 - $pos ))
do

        # loop body ... consisting of
        # case statement processing inputs
        # followed by logic handling overshoot, snakes and ladders!

done
printf "\n******************\n    Finished!    \n******************\n"

exit 0
```


First part of loop body using case with break & continue

```
printf "\n*** Give another throw of the die!?  ***\n"
echo -e "\n*** Either a number from [1-6] ... \n ...OR 999 to escape! ***"
read throw
case "$throw" in

        # checks valid range for throw of die ...
        # ...using case pattern regexp range [1-6])
        # checks numbers using both methods: expr and $(( ... ))
        [1-6])  pos=`expr $pos + $throw`
                over=$(( $pos - 10 )) ;;

                # breaks out of main loop to finish
        999)    echo -e "\n\n\n Bye! And have a nice die! \n\n\n"
                break;;

                # ignore faulty input for rest of this pass
        *)      echo -e "\n***  Some die!? it threw a: $throw"
                continue
esac
```


Second part of loop body – logic for overshoot, snakes & ladders

```
# if the throw value would exceed the end of the board, just count back the excess
[[ $over -gt 0 ]] && pos=$(( 10 - $over )) \
        && echo -e "\n Overshot - bounce back!" || echo -e "\nStepping out -"

        # ladder - if divisible by 3, go up 4, except for 9 (i.e. 3 or 6)
[[ $pos -eq 3 ]] || [[ $pos -eq 6 ]] && pos=$(( $pos + 4 )) \
        && echo -e "^^^^ a multiple of 3 - except for 9 - go up by 4! ^^^^\n"

        # snake - slide back 3 towards the floor if pos evenly divisible by 4
! (( $pos % 4 )) && pos=$(( $pos - 3 )) \
        && echo -e "vvv a multiple of 4, go 3 to the floor! vvv\n"

        # state current position
echo -e "\nNow at : $pos"
```

Bash – for snakes and ladders!

```
#!/bin/bash
pos=1 ;                 # starting position
while (( 10 - $pos ))   # perverse test, which fails when ((expr evaluates to zer0))
do
        printf "\n*** Give another throw of the die!?  ***\n"
        echo -e "\n*** Either a number from [1-6] ... \n ...OR 999 to escape! ***"
        read throw
        case "$throw" in
                [1-6])  pos=`expr $pos + $throw` ; over=$(( $pos - 10 )) ;;
                999)    echo -e "\n\n\n Bye! And have a nice die! \n\n\n" ; break;;
                *)      echo -e "\n***  Some die!? it threw a: $throw" ; continue
        esac
        [[ $over -gt 0 ]] && pos=$(( 10 - $over )) \
                && echo -e "\n Overshot - bounce back!" || echo -e "\nStepping out -"
        [[ $pos -eq 3 ]] || [[ $pos -eq 6 ]] && pos=$(( $pos + 4 )) \
                && echo -e "^^^^ a multiple of 3 - except for 9 - go up by 4! ^^^^\n"
        ! (( $pos % 4 )) && pos=$(( $pos - 3 )) \
                && echo -e "vvv a multiple of 4, go 3 to the floor! vvv\n"
        echo -e "\nNow at : $pos"
done
printf "\n*****************\n    Finished!    \n*****************\n"
exit 0
```

A simple contrived example for illustration of :-

- Control structures : while, case, break, continue, cryptic if then && etc.
- Logical test [[...]]
- Numerical evaluation and test ((....))

This programming style is far too dense, but compact for presentation.

Disk space administration

These commands can be used to determine or identify files/ directories with substantial disk use

- either intrinsically with options etc.

(which vary with bash Linux/BSD versions)

- Or additionally with pipes using sort and grep,

- du – displays disk block usage statistics
- df – displays info on disk free..

Using grep and sort, simply with list directory (ls) can select files of a specific type and sort them by size / modification date, so you can decide which ones to archive or delete.

Check manual (man) for more information and usage examples.

ls -1 | grep 'whatever' | sort –(field no e.g. date or size)

Or variations with intrinsic command options.

Disk device nearly full!

```
#! /bin/sh
df -kl | grep -iv filesystem |  awk '{  print $6" "$5} '| while
 read LINE; do
  PERC=`echo $LINE | cut -d"%" -f1 | awk '{  print $2 } '`
  if [ $PERC -gt 98 ]; then
    echo "${ PERC} % ALERT"  | mail -s "${ LINE}  on `hostname` is
 almost full"  admin@rocket.ugu.com
  fi
done
```

Df –kl display free disk space, in k, and locally mounted only..e.g.

```
Filesystem   1024-blocks      Used Available Capacity  Mounted on
/dev/disk0s2   488050672 305645696 182148976    63%    /
```

grep –ignore case, an inverted match to 'filesystem' (i.e. cut the header line with filesystem)

awk prints 6th (mount point) and 5th (% used) fields separated by a space, to give: 63% /

… and pipe all that stuff to a loop, which keeps reading, while there is a line to read…

... echoes the line through a pipe to cut field1, (**-f1**) to the field delimit character (**-d"%"**)

…(a perverse cut, d is not for delete, but denoting '%' as the delimit char for field1 = / 63)

... pipes that to awk, just to extract field 2, now the numeric value of the %, without the '%'!

It then sends an alert if this value is –gt than 98, which is dangerously high.

Perverse, & inefficient script example from a book, to show how to 'bash' them together!?

Log monitoring script – alert on new error!

Grep finds lines with ERROR in the logfile (choose one to monitor) & writes to temporary file
Which is compared with the previous copy for differences, which are notified to admin@...
The changed file then becomes the new standard for comparison with any further ERRORs!

```
#! /bin/sh
touch /tmp/sys.old
while [ 1 ]
do
  grep ERROR /var/adm/SYSLOG > /tmp/sys.new
  FOUND=`diff /usr/tmp/sys.new /tmp/sys.old`

  if [ -n "$FOUND" ];
  then
     mail -s "ALERT ERROR" admin@madmen.com < /tmp/sys.new
     mv /tmp/sys.new /tmp/sys.old
  else
     sleep 10
  fi

done
```

Backups

Various Options

- Entire filesystems / partitions
  - dump/restore
  - dd – disk duplicate (bit by bit)
- Selective:
  - Tar - tape archive, simple, stable, ubiquitous & updated : widely used
  - Cpio – great but modifies file (create & access) times
  - Rsync – (remote sync of files) incremental copy across the network

Then the easy bits... time permitting

- Mostly done with GUI & tools, but have no fear of CLI for
  - Installation
  - Configuration
  - Management
- Virtual machines
  - Programs which behave as machines
  - So you can load and mess with operating systems and applications, without (with less risk of) compromising your real work machine!?