

TNE30019 – Unix for Telecommunications

Unit Introduction

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019 – Unit Introduction

Teaching Staff

- Convenor: Dr. Jason But
- Lab Instructors: Quoc Khanh Le, Fari Jowkarishasaltaneh
- Tutorial Instructors: Fari Jowkarishasaltaneh

Dr. Jason But

TNE30019 – Unit Introduction

Unit Overview

Motivation

- Key network infrastructure runs Unix, e.g. routers
- Key network services run under Unix, e.g. DNS
- Large fraction of network servers runs Unix

Learning Outcomes

- Use Unix system from command line (and GUI)
- Understand basics of Unix OS
- Configure common network services on Unix
- Basic shell scripting and socket programming
- Design and implement network service
- Document lab work and research assignment

Dr. Jason But

TNE30019 – Unit Introduction

My Reflection – Does the Portfolio Work?

Overall I believe so, despite the large amount of extra work for us, both during and before semester

In the previous system a significant number of students who worked hard enough to merit a HD often had to settle for a D. Similarly there were occasions where students who probably did enough to pass failed to jump all requirements, and students who probably didn't managed to scrape through

The structure of the portfolio leaves me very comfortable that all students who have passed this Unit have definately earnt that pass, all teaching staff believe that all students who did enough to merit a HD will have received one

Dr. Jason But

TNE30019 – Unit Introduction

- Unit Outline
- Lecture Slides
- Grading Information
- Specific Portfolio Task Information
- Tutorial Information
- Discussion Board
- Lets have a look...

Lecture Recordings and Live Online

- Lecture Recordings
 - One-two hours pre-recorded content per week
 - Covers core material
- Live Online Sessions
 - Interactive activities
 - Demonstrations
 - Question and answer sessions
 - Industry Speakers

Industry Speakers

- Compulsory attendance
- Tied to Portfolio tasks

Theory Tasks

- Confirm your theoretical understanding of the material
- Tests, Discussion participation, Report

Practical Tasks

- Confirm your practical ability to build/deploy solutions
- Labs, Project, Programming

Communication Tasks

- Confirm your ability to communicate your learning/discoveries
- Lab Reports, Research Report, Presentation, Reflection

Tutorials

- One hour
- 9 tutorials during semester (*Grand Final Holiday*)
- Preparation will help (see Canvas)
- Some Portfolio tasks assessed in tutorials
- First tutorial covers tech reports and plagiarism

Laboratories

- Two hours
- Build on each other (don't miss any)
- Generally cannot be completed during lab time
- Ten labs during semester

Portfolio Lab Tasks

- Each lab exercise must be at least completed to the **Pass** Level
- You get to choose if you want to do more

Portfolio Lab Reports

- Three labs must be written up with a Lab Report
- Which labs are to be written are listed on Canvas
- Portfolio task sheets specify minimum requirements

Assessment/Grades

Portfolio

- Managed through Doubtfire
<https://doubtfire.ict.swin.edu.au>
- You get to choose what grade you want to get
- Must complete all tasks at a grade level to achieve that grade
- See Doubtfire and Canvas for task and grade information

Issues Keeping Up

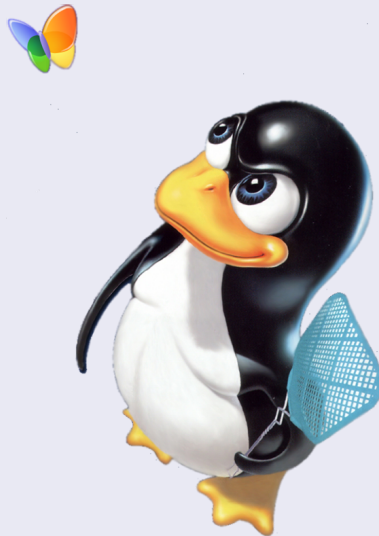
- **Don't wait** – contact us early
- Extensions considered/granted if they occur infrequently
- No deadline for Pass students (except end-semester)

Didn't get what you want

- **Don't give up**
- Come talk to me

Exam

No Exam



TNE30019/TNE80014 – Unix for Telecommunications

History of Unix

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019/TNE80014 – History of Unix

There is more than Windows

- Many(most?) laptop or desktop computers run MS Windows
- Well, these days quite a number run Apple MacOS
- But there are lot's of "other types" of computers
 - Can you identify some?
 - Mainframes
 - Smart phones
 - ...
- Many of those run Unix Operating Systems (OSs)

Dr. Jason But

TNE30019/TNE80014 – History of Unix

History of Unix

Late 1960's/early 1970's

- Multiple platforms
- No common Operating System
- Programmers wrote everything
 - Boot loading code
 - Direct access to hardware
- Lead to development of Operating System software
 - Manage boot-strapping of computer hardware
 - Manage access to underlying hardware through common APIs
 - Manage basic book-keeping tasks of computer

Dr. Jason But

TNE30019/TNE80014 – History of Unix

History of Unix

Dennis Ritchie



Ken Thompson



AT&T Bell Research Labs – 1969

- MultICS (*Multiplexed Information and Computing Service*) project canned – Ritchie and Thompson left without project
- Management decided **NO MORE OS PROJECTS**
- Thompson found PDP-7 sitting unused in corner
- With his wife on holiday Thompson started programming OS

Dr. Jason But

TNE30019/TNE80014 – History of Unix

History of Unix – PDP-7



Source: <http://en.wikipedia.org/w/index.php?title=PDP-7&oldid=619635714>

History of Unix

Name – team in-joke

- Like MultICS, but only one user – Thompson
- UnICS (*Un-multiplexed Information and Computing Service*)
- Name morphed to **UNIX**

Thompson and others created OS with basic tools

- Kernel
- Hardware IO Management
- File management
- Shell
- Programming tools (notably **C** language)

History of Unix

- Thompson and Ritchie needed more processing power, but AT&T would not support any OS development

The Plan

- Proposed to management to buy PDP-11
- Proposed to develop tools to edit and format text
- Mentioned as footnote they also *needed* to write new OS to support those tools
- Management gave green light

The Outcome

- Trial with three typists in patents department – they loved it
- Management were happy, bought newer, better PDP-11
- Unix lived on

History of Unix

Ground Breaking

- Hierarchical file system
- Multi tasking/user
- System administration tools – backup, removable storage
- Multiple programming languages (including C)
- All in only 4,200 lines of code that needed 16kB of RAM

Public Exposure

- Paper published in Communications ACM, 1974
- Multiple requests for copy of OS
- But there was a problem

History of Unix

Problem

- AT&T were bound by decree not to sell products related to telephones or telecommunications (price for legal monopoly)
- AT&T **could not** sell Unix
- Forced to release code for a nominal fee (reproduction costs)
- License stated, no support, no bug fixes

Explosion

- Release contained source for 'C' Compiler – language of OS
- Became OS of choice at Universities and Research Labs
 - Source code and compiler were provided
 - Could develop device-drivers, new versions, new hardware
 - Could explore and experiment with OS implementation

History of Unix

User Revolt

- Users had to provide their own support and banded together
- Thompson and Ritchie wanted to help, but were not allowed to release updated code – *it might be seen as support*
- Magic cookie box – user groups told “Go here, at this time, and you might find something useful”

Eventually – The Takeover

- Berkeley Unix strain (BSD) became first default Unix
- Regular public releases (updates)
- Berkeley University created
 - New editor **vi**
 - New Shell **cs**
 - New compilers **Pascal, Lisp**
 - Virtual memory
 - TCP/IP stack and sockets API

History of Unix

Legal Issues

- AT&T filed law suit against Berkeley over intellectual property
- Berkeley filed counterclaim against AT&T for license breaches
- Legal uncertainty slowed development of BSD for Intel PCs
- GNU (GNU's Not Unix!) project started developing open-source license-free UNIX

New Kid on the Block

- By 1990 GNU had userspace apps but no kernel in sight
- In 1991 Linus Torvalds released open-source Unix-like OS for Intel PCs – **Linux**
 - New kernel inspired by Tanenbaum's MINIX
 - Userspace applications from GNU
- Since early 2000s support for several new (low-cost) CPUs
- Google released Linux-based **Android** in 2008

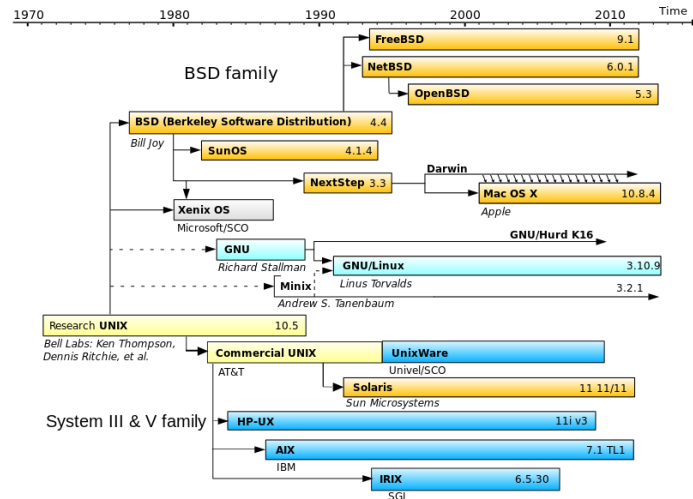
History of Unix

- UNIX fragmented
- Many versions, some licensed, some free
- AT&T Bell still held original license
- Hardware manufacturers were releasing their own versions

For many years

- Predominately used in research institutions
- Used to run large server-type computer systems
 - More stable, scalable, secure

History of Unix



Source: <http://en.wikipedia.org/w/index.php?title=Linux&oldid=619755530>

Unix – Today

BSD

Turned into several products

- OpenBSD
- FreeBSD / PC-BSD
- NetBSD
- Parts of Mac OS X, e.g. network stack



Linux

Huge number of distributions

- Red Hat, Debian, Ubuntu, openSuSE, ...
- Same Linux kernel and core GNU tools
- Different file-system layouts
- Different setup/management tools
- Different user-level applications



Unix – Today

Unix is everywhere

- Many BSD or Linux servers
- Linux fairly common on desktops
- Mac OS X (partially)
- Network routers
- Embedded systems
 - Smart phones
 - Smart TVs
 - Set-top boxes
 - Navigation systems
 - Medical instruments



TNE30019/TNE80014 – Unix for Telecommunications

Operating Systems Basics

Dr. Jason But

Swinburne University

Multi-Tasking

Older computers typically had 1 CPU (1 core)

- Can only run one job at a time
- One program counter
- One memory block
- One CPU Cache

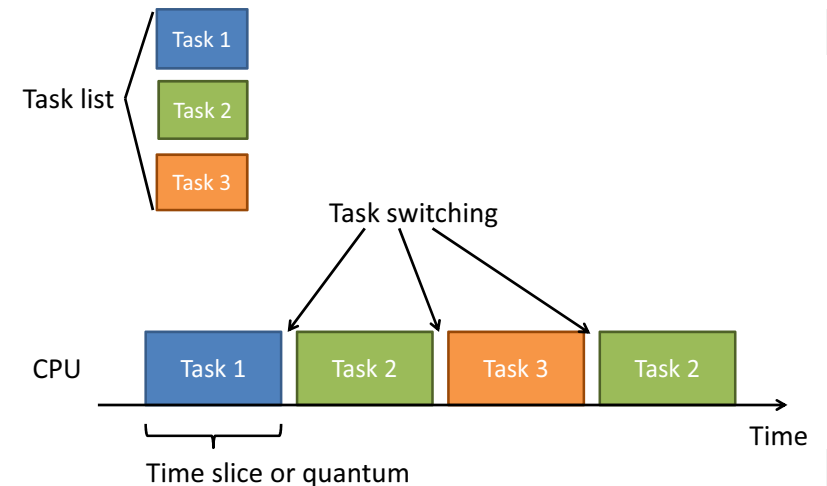
How do we multi-task?

- **Preemptive** task switching is common
- Run one task for a while and then...
- Interrupt and suspend current task
- Run another task

Outline

- Multi-Tasking
- Memory Management
- Kernel vs. User space

Multi-Tasking – Task Switching



Multi-Tasking – Task Switching

Periodically

- Stop CPU
- Save current program's context
- Load next program's context
- Start CPU

Program Context

- Program counter
- CPU registers
- Program stack memory

Multi-Tasking – Interrupts/Exceptions

Some tasks demand immediate attention

- Hardware has received new data
- Errors, e.g. division by zero
- Timer events
- Device or CPU triggers hardware or software **interrupt**

CPU is interrupted

- Pause current task
 - Execute small interrupt handler
 - Task is restarted after interrupt function completed
- Task-switch realised as periodic interrupt that changes context to new task

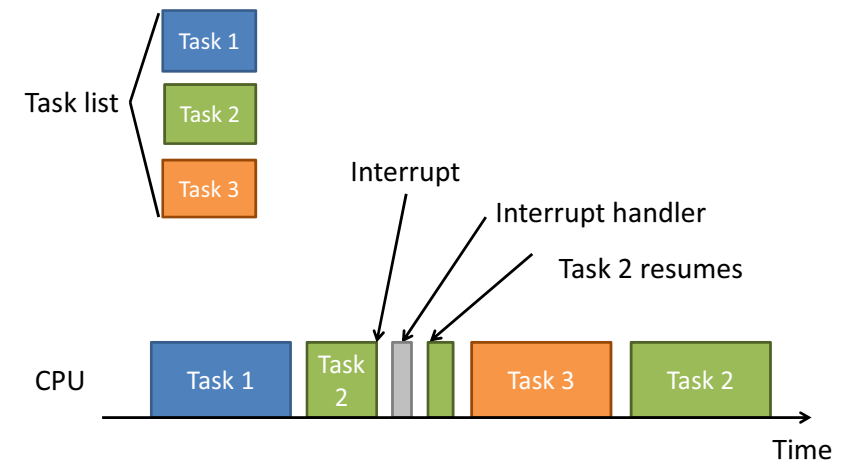
Multi-Tasking – Switching Frequency

Configure task switching frequency

- 100Hz to 1000Hz common default
- Configurable with kernel parameter
- Set parameter in /boot/loader.conf (FreeBSD)
- Recompile kernel (Linux)

- Advantages/disadvantages of low frequency?
- Advantages/disadvantages of high frequency?

Multi-Tasking – Interrupts



Multi-Tasking – Task States

Running Task

Task currently executing, only one task can run on each CPU core

Waiting Task

Task can run, but is not running

Sleeping/blocked Task

- Task wants to wait for some time
- Task cannot run because it is waiting for another task to finish first

Multi-Tasking – Which Task?

Where is task information stored?

Task list also known as run queue

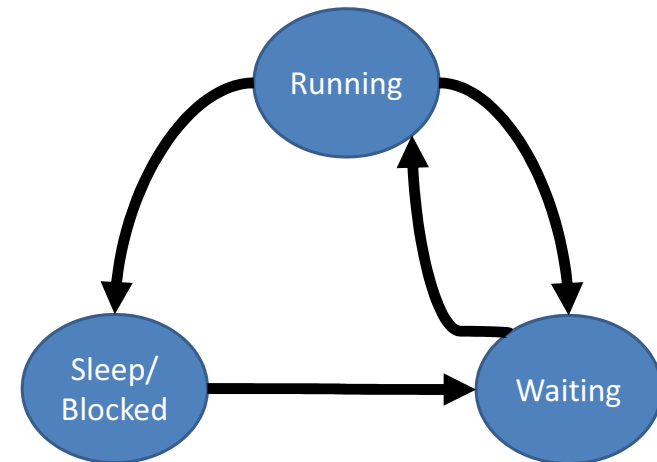
Who chooses which task to run?

Operating system **scheduler**

How does the scheduler choose?

- Task state – running, waiting, sleeping/blocked
- Specified task priority – highest priority first
- Task run-time – how much CPU time task already had

Multi-Tasking – Task States



Memory Management

Virtual Memory

- Pretend there is more memory than there really is
- Actual RAM is window into larger virtual memory space
- Virtual RAM stored on disk

To access memory not in current RAM Window

- Memory page from RAM is updated back to disk
- Required page from disk is loaded into RAM

Layers of caching

- 1 CPU micro-cache caches recently calculated values
- 2 CPU registers cache frequently needed variables
- 3 CPU L1 cache caches most common portion of L2 cache
- 4 CPU L2 cache larger – caches common portion of RAM
- 5 RAM – caches virtual memory on disk

Multi-Tasking – Multi-core Systems

- Task-switching complicated on systems with more than 1 core (or more than 1 CPU)
- True parallel execution
- Shared memory, shared code-space, and shared devices
 - Need synchronization to avoid race conditions
- Per CPU core registers, L1 and L2 cache
 - Migrating tasks between CPU cores reduces cache efficiency
 - Scheduler needs to optimize use of available CPU cores

Kernel Space vs. User Space

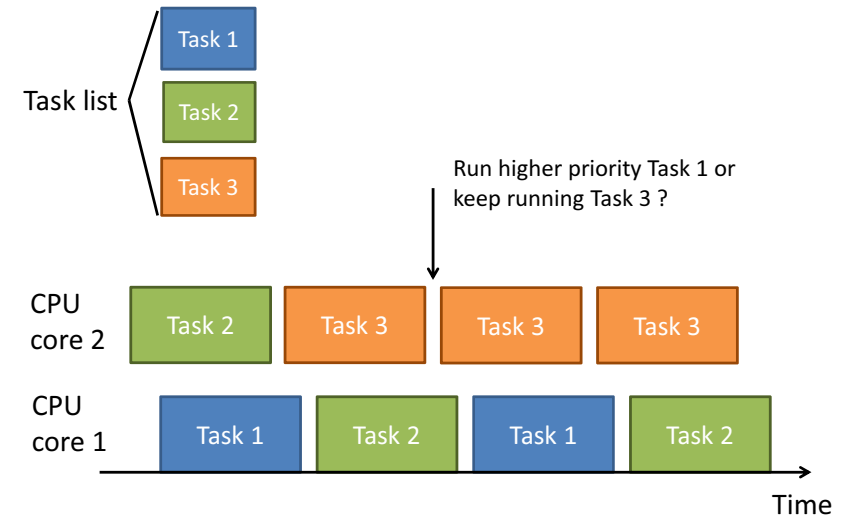
Kernel

- Core of operating system, runs in kernel space
- Full access to memory and all other hardware
- Trusted, well-tested code (hopefully)

User processes (including root's processes)

- Run in user space
- Limited memory access
- Access to system resources via **system calls**
- System call will trigger special interrupt
- This interrupt will execute kernel system call code and return results to user process

Multi-Tasking – Multi-core Systems



TNE30019/TNE80014 – Unix for Telecommunications

The Unix Kernel – Device Drivers

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019/TNE80014 – Device Drivers

Hardware Abstractions

- One major goal of OS
- Abstract differences in underlying hardware
- Hide hardware access details
- Manage access to hardware resources by multiple processes

Example

- All applications have access to video
- Producing video output works same way regardless of actual graphics card installed

Operating System Implementation

Device Drivers

Dr. Jason But

TNE30019/TNE80014 – Device Drivers

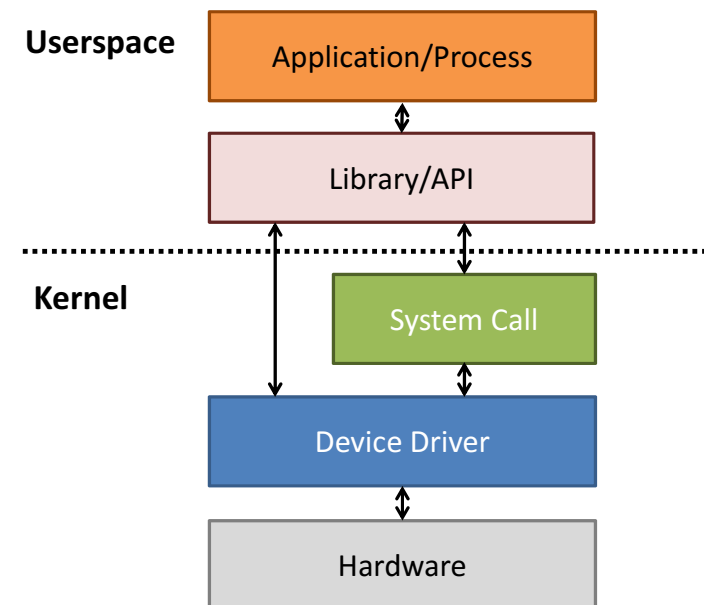
Outline

- Hardware Abstractions
- Unix Device Driver Types
 - Block vs. Character
 - Special devices
 - Compiled vs. Module
 - Network Interface Drivers

Dr. Jason But

TNE30019/TNE80014 – Device Drivers

Device Drivers



Dr. Jason But

TNE30019/TNE80014 – Device Drivers

Device Drivers

- Manage control of hardware device
- Manage sending data to hardware device
- Manage receiving data from hardware device
- Provide per device-type common API for access

Unix Devices

Unix Device Drivers provide access to devices via virtual files located in /dev directory:

- Naming not consistent among different Unix implementations
- Directory is populated as drivers are loaded and started
- Unix devices all fall into two categories – **Block** or **Character**

Block Devices

Equivalent to block of memory with Random Access

- Can read/write to any point in device
- Access restrictions managed by Unix file system permissions
- Access is granted by device driver
 - Whether read and/or write is possible given type of device
 - Whether actual data transfer is valid
- Concurrent processes are blocked until current process has relinquished device

Examples

Hard disk, USB stick, CD/DVD, graphics card

Block Devices – FreeBSD

SATA Disks

/dev/ada0
/dev/ada1s2
/dev/ad1as2b

Video

/dev/agpgart

CD/DVD Devices

/dev/acd0

SCSI Disks

/dev/da1
/dev/da2s1

Block Devices – Disks

- Disk drives are treated as block devices
 - Large block of memory where data can be written/read
 - File system needed to manage individual directories/files
 - Formatting involves creating empty file system on device
- Any driver that implements a block device can
 - Have file system installed
 - Be treated as disk
- Only root can directly access disk block device and then only if it is not mounted (being used)
 - Direct access to block device can destroy file system and make disk unreadable
 - Used for disk-recovery

Block Devices – RAM Disks

RAM Disk Driver

- Requests block of memory (RAM/virtual memory)
- Provides block driver for that memory block (`/dev/md?`)
- This block device can be formatted and used as disk
- Not a real disk – just area of memory that *looks* like disk
 - Very fast
 - Convenient for storing frequently-used files
 - Useful if there is no disk (diskless machines, CD-bootable OS)
- For main kernel (above driver) there is **NO** difference between
 - Hard disk, floppy disk, CD/DVD, USB key/disk, RAM disk
 - Only difference is which file system is installed

Character Devices

Serial input/output – data is sent or received sequentially

- Data written cannot be retrieved
- Data read cannot be read again
- Access restrictions managed by Unix file system permissions
- Access is granted by device driver
 - Whether read and/or write is possible given type of device
 - Whether actual data transfer is valid
- Concurrent processes are blocked until current process has relinquished device

Examples

Keyboard, Mouse, Printer

Character Devices – FreeBSD

Keyboard

`/dev/kbd0`

Mouse

`/dev/sysmouse`

Serial Ports

`/dev/cuaa0`

Virtual Consoles

`/dev/tty1`

Special (character) devices – FreeBSD/Linux

Discard all input

`/dev/null`

Provides zero (ASCII 0x00) characters

`/dev/zero`

Provides random bits

`/dev/random`

Device Drivers – Compiled Into Kernel

Driver implementation part of main kernel file

- Makes sense for drivers needed on all platforms or if there are not many choices of hardware
- May be faster since code is compiled into main kernel and not accessed via dynamic link
- Main kernel is larger
- Often used for standard devices
 - Keyboard, serial port, parallel port

Device Drivers – Module Control

- Needed modules will be automatically loaded, but often need manual control

FreeBSD

- Load kernel module: `kldload`
- Unload kernel module: `kldunload`
- List loaded modules: `kldstat`

Linux

- Load kernel module: `modprobe`, `insmod`
- Unload kernel module: `rmmmod`
- List loaded modules: `lsmod`

Device Drivers – Linkable Modules

Dynamically loaded at system boot (*or later*)

- Makes sense for drivers where hardware is very diverse
- Device in `/dev` is dynamically created as driver is loaded
- May be slower since calls to driver involve dynamic module links which take time to resolve
- Main kernel is smaller and loads faster
- More flexible as you can change hardware configuration without re-compiling main kernel
- Some configuration file tells kernel which modules to load
- Often used for non-standardised devices
 - USB, network cards, graphics cards

Network Interface Devices

- Network devices are special – exposed through network interface configuration
- Naming scheme: interface **name** followed by **number**
- For BSD Unixes name is based on device driver
- For Linux name is based on type, hardware bus and address

Example network device names

FreeBSD Intel EtherExpress Fast Ethernet – `fxp0`
FreeBSD Intel EtherExpress Gigabit Ethernet – `em0`
Linux Intel EtherExpress Gigabit Ethernet – `enp1s0`

Network Device Properties

- Can't write directly to network interface devices
- Kernel provides API to access network devices
- Network devices are configured via `ifconfig` command

Runtime configuration

- Some behaviour can be configured at runtime by changing variables with `sysctl`
 - Example: show TCP related variables with `sysctl net.inet.tcp`
- `sysctl` also has some read-only variables (show state)
- On Linux we also have `/proc` file system to view kernel information or configure things
 - Example: show CPU details with `cat /proc/cpuinfo`

Not everything can be configured during runtime

- If module we can unload, reconfigure, reload module
 - On Linux edit files in `/etc/modprobe.d/`
 - On FreeBSD edit `/boot/loader.conf` and `/etc/rc.conf`
- Otherwise have to reconfigure, recompile kernel, and reboot

TNE30019/TNE80014 – Unix for Telecommunications

The Unix Kernel – File Systems & Permissions

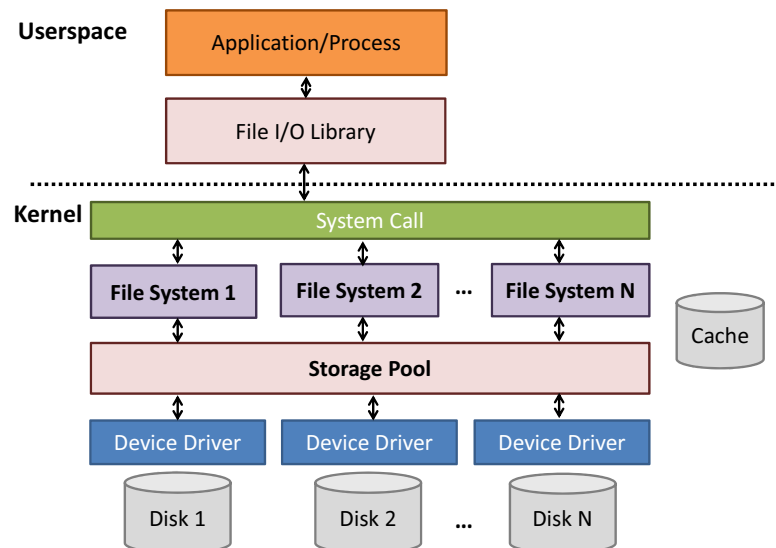
Dr. Jason But

Swinburne University

Outline

- Supported File Systems
- Global File System
- Unix File System Structure
- File Names
- Multi-User System
- File System Permissions

Block Devices – File Systems



File System Formats

FAT32

Windows 95/98 or earlier, Windows floppy disks, USB sticks
Driver provided with most UNIX systems

NTFS

Windows XP, Vista, 7, 8
Read-only driver provided with most UNIX systems

ext2/ext3/ext4

Linux Extended File System 2/3/4

ZFS

Current standard BSD File System

Many others

Accessible on Unix if file-system driver is available

Unix Global File System

- MS-DOS / Windows world uses multiple disks
 - Each disk is named and independent (e.g. **c:**, **d:**)
- In Unix there is only one virtual disk
- Different disks are “mounted” and form directory tree in global or “root” file system (root is /)

Example:

- Floppy disk drive is mounted to `/mnt/floppy`
- All files in `/mnt/floppy` are on floppy disk – copying file to this directory copies it to floppy disk
- Floppy disk can be mounted to any directory (not just `/mnt/floppy`)
- Un-mounting `/mnt/floppy` forces OS to write cached data to disk and allow you to eject it

Unix File System Structure

`/bin`
System binaries

`/boot`
Kernel and boot files

`/dev`
Device driver interfaces

`/etc`
System and service configuration files

`/home`
User home directories

Unix File System Structure

`/lib`
System libraries

`/mnt`
Standard place for mount points

`/proc`
System process information

`/root`
Root (super-user) home directory

`/sbin`
System super-user binaries
Only executable by root users

Unix File System Structure

`/usr` – Not system stuff

- `/usr/bin` Regular user binaries
- `/usr/lib` Regular user libraries
- `/usr/local` Non-standard stuff (machine specific)
- `/usr/sbin` Super-user binaries
- `/usr/share` Common shared files
- `/usr/src` System source code

`/var` – Run time information storage

- `/var/cron` Cron job information
- `/var/log` System log files
- `/var/run` Runtime information (process ID files)

Unix File System Structure

`/usr/local` – Non-standard stuff (machine specific)

`/usr/local/bin` Non-standard binaries

`/usr/local/lib` Non-standard user libraries

`/usr/local/etc` Non-standard configuration files

`/usr/local/sbin` Non-standard super-user binaries

`/usr/local/share` Non-standard shared files

`/usr/local/src` Non-standard source code

Unix File System – Files

File Names

- Maximum length 255 bytes (commonly)
- No limit on path name length (commonly)
- As many periods as you like
- No extensions required to signify executable

File Owners

- One owner – tied to single numeric user id (e.g. **root** = 0)
- One group – tied to numeric group id (e.g. **wheel** = 0)

Links

Symbolic link Like a Windows shortcut (aka soft link)
File contains path to another file/directory it links to

Hard link File pointing to another file's location on disk

Unix File System – Partitions

Partition

- Part of physical/logical disk
- Each partition can have its own file system
- Special partition for swap

Traditionally often multiple partitions

- For example, separate partitions for `/`, `/home` and `/var`
- File system corruption contained to partition
- Less chance of full file system problem (logging to `/var`)
- Easier upgrading, e.g. `/home` can be left untouched

Single partition

- Easier to setup
- More efficient disk use

Unix File System – Symbolic vs. Hard Link

Create link

```
ln [-s] <target_name> <link_name>
```

Directory Entries `/home/user/`

Type	Name	Inode
File	- a.txt	1234
Hard	- b.txt	1234
Symb l	- c.txt	2500
		[...]

Files on disk

This is a simple example text file.

`/home/user/a.txt`

Unix File System – Navigating the Tree

Show files in current directory

```
ls
```

Show current working directory

```
pwd
```

Change directory

```
cd <path>
```

Two types of paths

- **Absolute**: Specific directory/file in tree
- **Relative**: Directory/file relative to working directory

Multi-User Concept

- Concept of many users
 - Some tasks can **only** be performed by certain users (privileges)
 - Some files can **only** be accessed by certain users
- User groups allow managing of privileges for multiple users
- Security through permissions
 - Regular users cannot access memory or processes of other users
 - Regular users can only access devices or files if permitted
 - Regular users cannot change ownership/permissions of devices or other user's files
 - Only system administrator (aka **root**) can access all processes, files, unmount file systems, change all permissions, etc.

Multi-User – Changing Users

You can log out and log in as different user... cumbersome

su (substitute user)

```
su <user_name>
```

sudo (substitute user to **do** something)

- Allow subsets of users to execute commands as another user (usually root)
- File /etc/sudoers lists users and commands they can execute
- Huge amount of configurability compared to setuid/setgid
- When is it useful?
 - Give users administrative rights without need to be root
 - Allow users to execute few privileged applications

Unix File System – Permissions

- Each file has permission set for three sets of users
 - User What owner can do with file
 - Group What users in owner group can do
 - Others What everybody else can do
- Allowable permission bits
 - Read Specified user(s) can access file
 - Write Specified user(s) can modify file
 - Execute File is executable by specified user(s)
- Special permissions bits for executable files
 - setuid File is executed as if owner executed it
 - setgid File is executed as if group member executed it
 - sticky If set only owner of directory can rename and remove files

Unix File System – Permission-related Commands

Show permissions

```
ls -l
```

Show groups

groups or id

Change permissions

```
chmod <permissions> <files>
```

Change owner

```
chown <owner> <files>
```

Change group

```
chgrp <group> <files>
```

Unix File System – File Permissions Example

```
[szander@myhost]$ ls -l /bin/ls
-r-xr-xr-x 1 root wheel 30464 Oct 19 2011 /bin/ls
```

Annotations for the command output:

- Permission bits: `-r-xr-xr-x`
- Link count (hard links): `1`
- User/owner: `root`
- Group: `wheel`
- Size and Date: `30464 Oct 19 2011`
- Type of file: `/bin/ls`

Type of file (- = regular file, d = directory, l = soft link)

Unix File System – File Permissions Example

```
[szander@myhost]$ ls -l /bin/ls
-r-xr-xr-x 1 root wheel 30464 Oct 19 2011 /bin/ls
```

Annotations for the command output:

- `root` is owner belongs to `wheel` = root group (FreeBSD)
- `Others` can read and execute
- `Group members` can read and execute
- `User/Owner` can read and execute

Unix File System – File Permissions Example

```
[szander@myhost]$ ls -l /dev/bpf
crw-rw-r-- 1 root wheel 0, 11 Jul 7 16:43 /dev/bpf
```

Annotations for the command output:

- `root` is owner belongs to `wheel` = root group (FreeBSD)
- `Others` cannot read, write or execute
- `Group members` cannot read, write or execute
- `User/Owner` can read and write

TNE30019/TNE80014 – Unix for Telecommunications

Compiling and Installing the Kernel

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019/TNE80014 – Compiling the Kernel

Why Recompile Kernel?

- Distributions come with pre-compiled kernel
- Many users don't bother, but there are some good reasons
- Because it's cool
- Because you need smallest possible kernel for embedded device
- Because you need to modify kernel config (e.g. HZ=1000)
- Because somebody coded useful patch or new driver not (yet) part of official kernel distribution

Dr. Jason But

TNE30019/TNE80014 – Compiling the Kernel

Outline

- Compilation Process
- Configure Kernel
- Compile Kernel
- Install New Kernel
- System Recovery

Dr. Jason But

TNE30019/TNE80014 – Compiling the Kernel

Recompiling the Kernel

- FreeBSD kernel source code lives in `/usr/src/sys`
- Linux kernel source code lives in `/usr/src/linux`
- Kernel is written in C (and CPU-specific assembler)

Step-by-step process

- Configure kernel
- Compile kernel and modules
- Install kernel
- Update boot manager (optional)

Dr. Jason But

TNE30019/TNE80014 – Compiling the Kernel

Compilation process

- C source files are compiled to object files using C compiler (**gcc** – GNU C Compiler)
- Object files are linked into one executable using linker (**ld**)
 - Main kernel
 - Module objects
- How do we manage compilation of tens of thousands of files without missing one?
- How do we make small code change and rebuild kernel without compiling everything?

make and Compiling

- We need some automated means of compiling kernel

make

- Tool that uses project configuration file(s) (Makefile)
 - Specifies which files need to be compiled prior to linking
 - Checks object files against source files and only compiles source **IF** it is newer (changed since last compile)
 - Manages linking
- Kernel is built by typing “make”

FreeBSD Configuring and Building

- Kernel configurations in `/usr/src/sys/<platform>/conf`
- For 64-bit PCs `<platform>` is `amd64`
- Configuration file name: `<config_name>`

Compiling

```
cd /usr/src
make buildkernel KERNCONF=<config_name>
```

Installing

```
cd /usr/src
make installkernel KERNCONF=<config_name>
```

- This will move the old directory to `/boot/kernel.old`
- Copy newly compiled kernel to `/boot/kernel`

Linux Configuring and Building

- Sometimes distribution-specific steps
- Here generic steps for vanilla kernel (www.kernel.org)

Preparing

```
cd /usr/src/linux
make mrproper (only if need cleanup from previous compile)
make oldconfig or make menuconfig
```

Compiling

```
make
```

Linux Installing

- Kernel lives in `/boot/vmlinuz-<kernel-version>`
- Modules live under `/lib/modules/<kernel-version>/`

Installing

```
make modules_install  
make install
```

- This will put new kernel into `/boot`
- This will put new modules into `/lib/modules/`

Create initial RAM disk (if make install doesn't)

```
cd /boot  
mkinitrd -o initrd.img-<kernel-version>  
<kernel-version>
```

What if you “Stuff it Up”™



Source: <http://www.quickmeme.com/meme/3qmy44>

Rebooting

- **Current kernel will still be running**
- On FreeBSD/Linux, when program runs, its entire code is in memory – you can delete program but it will still run

Running your new kernel

- Update boot manager if needed (depends on boot manager)
- Reboot system
`shutdown -r now` or `reboot`
- New kernel will automatically load

What if you “Stuff it Up”™

- Your new kernel doesn't work
- System won't boot
- You need to recover old kernel

Recovery Process

- 1 Boot recovery system
 - Good kernel on hard disk selectable in boot manager (if exists)
 - Good kernel from CD/DVD image
- 2 Mount local hard disk so you have access to files
 - FreeBSD: rename `/boot/kernel.old` to `/boot/kernel`
 - Linux: relink `vmlinuz` and `initrd` to good kernel and good `initrd` and update boot manager if necessary
- 3 Reboot

TNE30019/TNE80014 – Unix for Telecommunications

System Bootup

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019/TNE80014 – System Bootup

System Startup – UEFI/BIOS

- Initial OS to manage hardware (UEFI/BIOS)
- Firmware on motherboard
- Loaded after turning on computer
- Has basic functionality to access hardware
- Knows bootable hardware devices
 - User can define device order in setup
- Knows how to hand over to hardware for booting
 - Can start boot loader from hard disk
 - Can start PXE stack on network card

Dr. Jason But

TNE30019/TNE80014 – System Bootup

Outline

- How Unix System Starts Up
- What Happens after System Boots
- Startup Scripts
- The login Process
- The shell

Dr. Jason But

TNE30019/TNE80014 – System Bootup

System Startup – Boot Loader

- How does an OS load?
- **Boot Loader** is small program that loads and runs OS kernel
- Needs to know where kernel is stored on disk
- Often provides some means to
 - Select which kernel to boot
 - Set parameters for kernel to boot

Boot Loader Functions

- 1 Loads kernel executable
- 2 Starts kernel

Dr. Jason But

TNE30019/TNE80014 – System Bootup

System Startup – Kernel

- Is the actual OS

Kernel Startup Functions

- 1 Run system checks, e.g memory test
- 2 Setup memory management, CPU, interrupt handling
- 3 Detect hardware devices
- 4 Load initial RAM disk (Linux)
- 5 Load device drivers
 - Drivers create devices in /dev
 - Configures hardware ready for use
- 6 Start idle process, scheduler and other key kernel processes (**PID=0**)
- 7 Mount root file system
- 8 Launch `init` – process initialisation daemon (**PID=1**)

System Startup – Run Levels

- Unix has concept of booting into different run-levels
- Different start-up scripts/services are run depending on chosen run-level
- Eight runlevels, three of which are “standard”

Standard and common run-levels

- 0 Halt (standard)
- 1 Single-user mode (standard)
- 2 Local multi-user console (common)
- 3 Multi-user console (common)
- 5 Multi-user graphical desktop (common)
- 6 Reboot (standard)

System Startup – init

- `init` is first process and manages startup of system processes
- `/etc/inittab` is the configuration file for `init`
 - Specifies default run-level
 - If no default run-level `init` will prompt user
- `init` launches `rc` with run-level as parameter
- `init` starts virtual terminals
- On Linux, big push to replace `init` with `systemd`

rc script

Shell script to start basic services, daemons and applications

Linux `/etc/init.d/rc`

FreeBSD `/etc/rc`

System Startup – rc

- This is where you can customise system
- Script decides what processes to launch

FreeBSD

- Runs scripts in `/etc/rc.d` with the `start` flag
 - Example: `/etc/rc.d/ntpd start` – Launches `ntpd` daemon
- Scripts written so processes are launched in correct order
- Variables configured in `/etc/rc.conf` tell scripts in `/etc/rc.d` whether to actually launch that process and command line parameters to use

Linux

- Directories maintained for each run level – location platform specific
- Each directory contains sym links to scripts to execute

System Shutdown

- Signal is sent to `init` to terminate
- On FreeBSD `init` runs `/etc/rc.shutdown`
 - Script tries to stop all processes launched by `rc`
- On Linux `init` executes scripts in `rc0.d` or `rc6.d`

`init` – Final Shutdown

After all services stopped

- Terminate any remaining processes
- Flush disks
- Terminate and shuts down system

The login Process

- Virtual terminals started by `init` will launch `login` application
- This is runnable process (`/bin/login`)

`login` process

- Displays the “login:” prompt
- Prompts for username and password
- Checks username and password against database
- If user is properly authenticated, grants access

`login` is **NOT** user interface

- Sets environment variables
- Launches yet another process – the **shell**
- Shell provides user interface

Your system is booted ... and then

User Interaction

- Console: keyboard + mouse + display
- On servers user interaction is typically done via network connections (remote console)
- On clients need console for user interaction
- Console access on servers is also useful in emergencies

Unix systems output to **console**

- Traditionally terminal connected to serial port
- Nowadays console is tied to virtual terminal (`/dev/ttyv0`)
- Graphics card driver ties this terminal to display

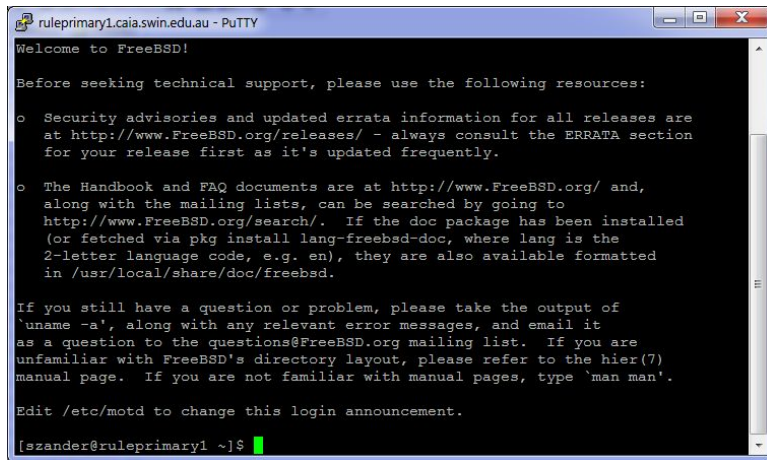
The Shell

- Shell is basically command interpreter
 - Prompt user for input
 - Execute user's commands
 - Display output
- Which shell `login` starts defined in user database

Many different shells with different functionality

`sh` Bourne shell
`bash` Bourne-again shell
`csh` C shell
`tcsh` TENEX C shell
`ksh` Korne shell
`zsh` Z shell

We Can Start Working Now



A screenshot of a PuTTY window titled 'ruleprimary1.caia.swin.edu.au - PuTTY'. The window shows the FreeBSD login screen. The text displayed is as follows:

```
Welcome to FreeBSD!

Before seeking technical support, please use the following resources:

o Security advisories and updated errata information for all releases are
  at http://www.FreeBSD.org/releases/ - always consult the ERRATA section
  for your release first as it's updated frequently.

o The Handbook and FAQ documents are at http://www.FreeBSD.org/ and,
  along with the mailing lists, can be searched by going to
  http://www.FreeBSD.org/search/. If the doc package has been installed
  (or fetched via pkg install lang-freebsd-doc, where lang is the
  2-letter language code, e.g. en), they are also available formatted
  in /usr/local/share/doc/freebsd.

If you still have a question or problem, please take the output of
`uname -a`, along with any relevant error messages, and email it
as a question to the questions@FreeBSD.org mailing list. If you are
unfamiliar with FreeBSD's directory layout, please refer to the hier(7)
manual page. If you are not familiar with manual pages, type `man man`.

Edit /etc/motd to change this login announcement.

[szander@ruleprimary1 ~]$
```

TNE30019/TNE80014 – Unix for Telecommunications

Presenting a User Interface to the World

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

Multiple Terminals

- Console is default terminal
- OS will echo any system messages to console
- Even on servers, people realised advantages of multiple terminals
- Unix systems run multiple virtual terminals (BSD: 8, Linux: 6)

Applications running on virtual terminal

- Take input from `stdin` on virtual terminal
- Output `stdout` to virtual terminal device

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

Outline

- Console and multiple consoles/terminals
- Virtual Terminals
- GUI on Unix

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

Virtual Terminals

- On BSD `/dev/ttyv0` through `/dev/ttyv7` (Linux `/dev/tty0` through `/dev/tty5`)
- On BSD switch between terminals using Alt-F1 to Alt-F8 (Linux Alt-F1 through Alt-F6)
- Use Ctrl-Alt-F? when switching from GUI

Switching virtual terminals will

- Connect keyboard to `stdin` on currently selected terminal
- Connect monitor/display to `stdout` on currently selected virtual terminal
- Input to other virtual terminals will block until they are enabled and user enters input
- Output to other virtual terminals will be buffered until they are enabled and then displayed to user

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

Virtual Terminals

- Virtual terminals are not restricted to keyboard and monitor connected to computer
- Other terminals: **telnet** or **ssh**

telnet/ssh network daemon will

- Accept connection
- Create virtual terminal tied to network connection
- Input from remote computer will be redirected to `stdin`
- `stdout` will be redirected to remote computer
- login process will be started on new virtual terminal
- login process will(may) start shell on virtual terminal
- User types and sees on their local computer but commands are executed on remote computer

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

Virtual Terminals

- Similar process with terminal window under Unix GUI
- Virtual terminal is created running instance of shell
- No login process – user is already logged on
- When terminal program (**xterm**) is active input is redirected from keyboard to virtual terminal
- Output from virtual terminal is buffered and displayed by X-Windows Manager

Virtual Terminals are an abstraction of a generic console

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

X-Windows – Server

- **X Window System** is Unix's GUI
- Based on generic modular approach of Unix applications

X Server responsible for

- Managing display of information
- Managing input from input devices to multiple windows
- **NOT**
 - Look and feel
 - Audio

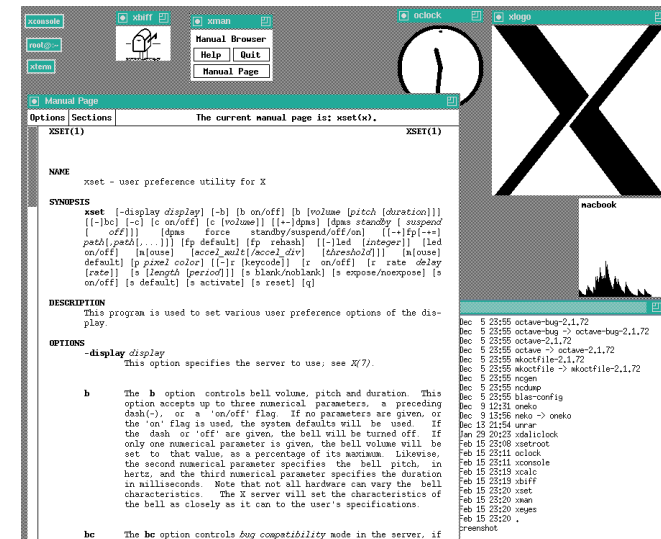
X Display Manager dictates the look

- Many different display managers
 - KDE
 - Gnome
 - Enlightenment
 - ...

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

X-Windows – Historic Window Manager twm

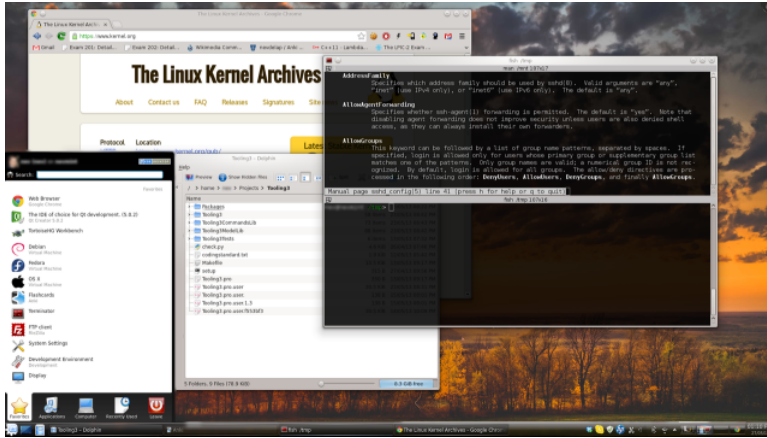


http://en.wikipedia.org/w/index.php?title=X_Window_System&oldid=617939694

Dr. Jason But

TNE30019/TNE80014 – User Input/Output

X-Windows – Modern Window Manager KDE



http://en.wikipedia.org/w/index.php?title=X_Window_System&oldid=617939694

X-Windows – Server Configuration

- X.Org project provides open source X Window System

/etc/X11/xorg.conf

- Defines hardware configuration of machine
 - Specifies graphics card
 - Keyboard and mouse type
 - Specifies available display resolutions
 - Creates “screen” for X to use
-
- Recent versions of Xorg are *configuration free*

X-Windows – Client/Server Model

X-Windows designed as networkable GUI from start

- More flexible and generic
 - Run graphical applications on multiple different machines and have all display on one screen
 - Traditionally allows applications on central server – users access server to run applications
 - CPU intensive applications could be run on more powerful workstations
-
- In general users use **client** to connect to remote **server**
 - Need to **invert** this paradigm for X-Windows

X-Windows – Client/Server Model

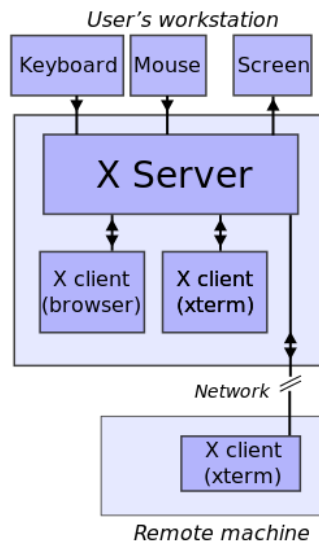
Server

- Manages display
- Since display is where user is – server runs on local computer

Client

- Application run by user
 - Talks to server to receive input and display output
 - Runs on remote computer
-
- Desktop use – both server and client run on local machine and talk to each other using **localhost**

X-Windows – Client/Server Model



http://en.wikipedia.org/w/index.php?title=X_Window_System&oldid=617939694

TNE30019/TNE80014 – Unix for Telecommunications

The Unix Shell and CLI

Dr. Jason But

Swinburne University

Dr. Jason But

TNE30019/TNE80014 – The Unix Shell and CLI

What is the Shell?

Shell – definition

Command-line interpreter providing user interface for Unix OS

Choice of shell is configurable (user account details)

`sh` Bourne shell
`bash` Bourne-again shell
`csh` C shell
`tcsh` TENEX C shell

Dr. Jason But

TNE30019/TNE80014 – The Unix Shell and CLI

Outline

- What is the Shell?
- Process Management
- Re-directing Input/Output
- Piping Input/Output
- Shell Scripting

Dr. Jason But

TNE30019/TNE80014 – The Unix Shell and CLI

The Shell

- Process like everything else
- Interpreted programming language
- Commands/instructions are received from input pipe – `stdin`
- Output is delivered to output pipe – `stdout`
- Error output is delivered to error pipe – `stderr`
- Provides environment variables to store values, e.g. `env`
- Provides basic prompt – configurable using shell variables
- Provides built-in commands to perform basic tasks
 - `cd`
 - `pwd`
 - `exit`
 - ...

Dr. Jason But

TNE30019/TNE80014 – The Unix Shell and CLI

The Shell Process

- As process the shell has **process ID** (positive integer number)

Launching Processes

- When you run program, shell launches new process with new process ID
- New process has shell as its parent process
 - Parent process ID = shell process ID
- When shell terminates, all child processes are also terminated
- Often child processes are run in **foreground**
 - Shell process blocks until child process finishes
- You can launch shell from within shell
 - New shell replaces current shell
 - Current shell blocks until new shell terminates

Redirecting Output

- Like all other devices `stdin`, `stdout` and `stderr` live in `/dev`
- Always points to current process/virtual terminal
- Input/output can be redirected to file on filesystem

```
myprog > ./myfile
```

Run myprog and redirect output from `stdout` to file `./myfile`

```
myprog >> ./myfile
```

Redirect output from `stdout`, **appending** it to `./myfile`

```
myprog < ./commands > ./myfile
```

Run program taking input from `./commands` as if it were typed at keyboard, redirect output from `stdout` to `./myfile`

```
myprog > ./myfile 2>&1
```

Redirect output from `stdout` and `stderr` to `./myfile`

Background Processes

- Processes can be run in **background** with “&” after command
- Both new and parent process run concurrently
- If parent process is shell – can continue issuing commands

Managing Processes

- `Ctrl-C`
 - Kills currently running process
- `Ctrl-Z`
 - Suspends currently running process
- `jobs`
 - Lists currently running and suspended *child* processes
- `fg`
 - Resumes specified/last process and moves it to foreground
 - Shell process blocks until foregrounded process finishes
- `bg`
 - Resumes specified job and moves it to background

Piping Output

- Unix “invented” concept of **pipes**
- Also available in newer Windows shells
- Many programs accept input from character device (`stdin`) and direct output to character device (`stdout`)
- Should be able to direct output from one process directly to input of another
 - This is called piping
 - `stdout` of **process 1** is piped (*instead of displayed*) to `stdin` of **process 2** (*instead of reading from the keyboard*)

Piping Output

```
myprog1 | myprog2 | myprog3
```

- Shell launches three separate processes concurrently
- stdout output of myprog1 is treated as input by myprog2
- stdout output of myprog2 is treated as input by myprog3
- myprog3 outputs to stdout

```
Example: cat ./myfile | less
```

- Show file page by page

```
Example: find / | grep foo | wc -l
```

- Runs find to display every single file on filesystem
- Output is piped to grep, which outputs only lines with “foo”
- Output piped to wc, which outputs number of lines input
- Result is count of files with word “foo” in filename

Grouping Processes

```
myprog1 ; myprog2
```

Execute myprog1, when it finishes execute myprog2

```
myprog1 && myprog2
```

Execute myprog1, if it terminates normally (no error code) then execute myprog2

```
myprog1 || myprog2
```

Execute myprog1, if it terminates abnormally (error code) then execute myprog2

- Last two are handy when writing shell programs
- **Do not confuse last with piping**

Piping Output

- Pipes can be extremely powerful
- Generic tools can be built and re-assembled as pipeline to perform complicated tasks
- True modular programming
- Many good standard tools provided with Unix
 - grep, wc, sed, find, gzip, pstopdf, wget
 - and many more...

Shell Scripts

- Shell is much more than interactive console for users

Shell is programming language interpreter

- Variables
- Functions with parameter passing
- Loops
- Conditionals

- Available on command line, but difficult to use
- Power of shell shows when program is pre-written

Writing and running Shell Scripts

- ASCII (text) – use your favourite text editor
- /bin/sh script_name (Bourne shell script)
- /usr/local/bin/bash script_name (BASH shell script)

Shell Scripts

Making Shell Scripts executable

- Executable bit(s) must be set
- First line must start with ASCII characters “#!”
- Rest of first line defines interpreter used to execute script
- Process launcher will launch specified interpreter with script as argument

Example: myprog

```
#!/bin/sh
find ${1} | grep "${2}" | wc -l
```

- Run with ./myprog <dir> <text>
- Equivalent to running /bin/sh myprog <dir> <text>

Running Scripts In Shell

- Not limited to standard shells
- Can be used with any interpreter

PERL Programming Language

```
#!/usr/bin/perl
```

PYTHON Programming Language

```
#!/usr/local/bin/python
```

PHP Programming Language

```
#!/usr/local/bin/php
```

- Normally Web scripting but can be used for shell scripting

Shell Script Variables

Command line arguments

- Accessible with \$<number>
- \$0 is the script name itself
- \$1 is the first command line argument
- ...

Variables

- Syntax depends on shell (here Bourne shell)
- Assignment: WORD=\$1
- Use: \$WORD or \${WORD}

Example: myprog

```
#!/bin/sh
WORD=$1
find / | grep "$WORD" | wc -l
```

Fun With Quotes In Shell Scripts

Single quotes ' '

Preserves literal value of characters in quotes

```
$ echo '$SHELL'
$SHELL
```

Double quotes " "

Preserves literal value of characters in quotes, except \$, \, \, !

```
$ echo "$SHELL"
/usr/local/bin/bash
```

Back quotes/ticks ` `

Everything in back quotes is executed before main command

```
$ echo `basename $SHELL`
bash
```

Example Shell Script

```
#!/bin/sh
# List all .html files and copy first lines
# of each file into one line of text file

FILE_HEADS=$1

FILE_LIST=""`ls *.html`"
echo FILE_LIST: ${FILE_LIST}

RESULT=""
for FILE in ${FILE_LIST} ; do
    FIRST_LINES=`head -2 ${FILE}`
    RESULT="${RESULT}${FIRST_LINES}\n"
done

echo -e ${RESULT} > ${FILE_HEADS}
```