OPERATING SYSTEMS FUNDAMENTALS

UNIT I INTRODUCTION **9 hours**

Concept of Operating Systems, OS Services, System Calls, Structure of an OS - Layered, Monolithic, Microkernel Operating Systems, Case study on UNIX and WINDOWS Operating System. KORN SHELL PROGRAMMING: Basic Script Concepts, Expressions, Decisions Making: Selections, Repetition, Special Parameters and Variables, Changing Positional Parameters, Argument Validation, Debugging Scripts.

*****

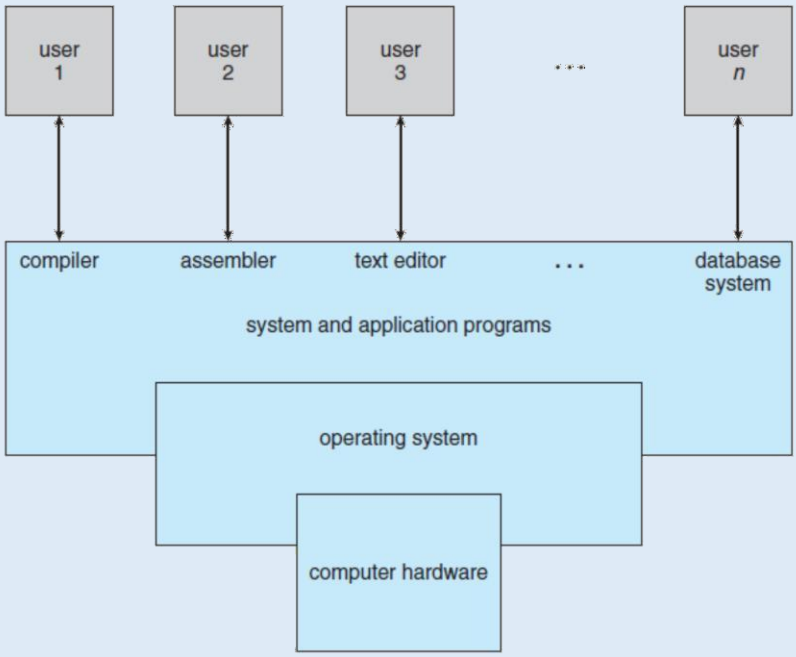
Operating Systems

- An **operating system** is a program that manages a computer's hardware.
- It provides the basics for application program and
- acts as an intermediary between a user and the computer hardware.
- It simply provides an ***environment*** within which other programs can do useful work.
- **Resource allocator** – manages and allocates resources.
- **Control program** – controls the execution of user programs and operations of I/O devices.
- examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, MacOS, VMS, MVS, and VM.
- Mainframe operating systems are designed primarily to optimize utilization of hardware.
- Personal computer (PC) operating systems support complex games, business applications, and everything in between.
- Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs.
- Thus, some operating systems are designed to be ***convenient,*** others to be ***efficient,*** and others to be some combination of the two.

Computer System Components

A computer system can be divided roughly into four components:

1. **Hardware** – provides basic computing resources (CPU, memory, I/O devices).
2. **Operating system** – controls and coordinates the use of the hardware among the various application programs for the various users.
3. **Applications programs** – Define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
4. **Users** (people, machines, other computers).



Abstract view of the components of a computer system

Operating systems can be explored from two viewpoints: i) the user and **ii)** the system

i) User View: From the user's point view,

- The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. In this case, the operating system is designed mostly for **ease of use**, Don't care about **resource utilization.** designed for one user to monopolize its resources
- **In mainframe** or mini-computer, users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization.
- **In workstations** connected to networks of other workstations and **servers**. In which operating system is designed to compromise between individual usability and resource utilization.
- **In smart phone**, user interface for mobile computers generally features a **touch screen**, where the user interacts with the system by pressing and swiping.
- **Embedded computers** in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

ii) System View: From the computer's point of view,

- an operating system as a **resource allocator**. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly.

- an operating system is a **control program** that manages the execution of user programs to prevent errors and improper use of the computer. It is concerned with the operation and control of I/O devices.

Components of OS: OS has two parts.

(1)Kernel.
(2)Shell.

(1) Kernel is an active part of an OS i.e., it is the part of OS running at all times. It is a program which can interact with the hardware. Ex: Device driver, dll files, system files etc.

(2) Shell is called as the command interpreter. It is a set of programs used to interact with the application programs. It is responsible for execution of instructions given to OS (called commands).

Operating system goals

- Execute user programs and make solving user problems easier.
- Make the computer system convenient to use.
- Use the computer hardware in an efficient manner.

OS Services

- Operating systems provide an environment for execution of programs and services to programs and users.

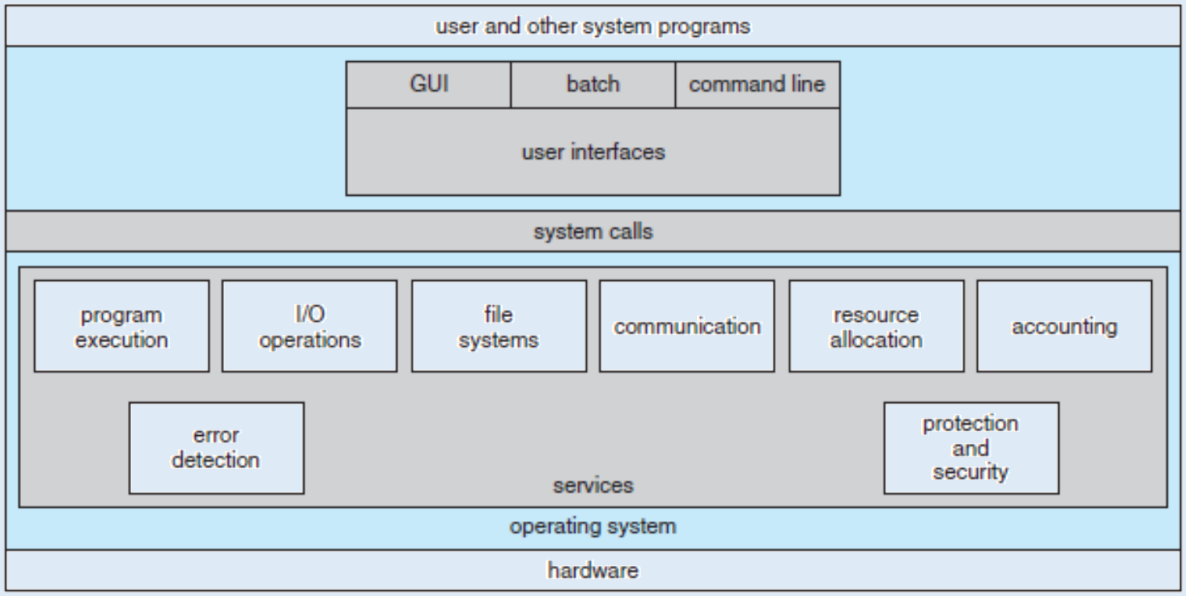
One set of operating-system services provides functions that are helpful to the user:

- ✓ **User interface** - Almost all operating systems have a user interface (UI). It Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch Interface.
- ✓ **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- ✓ **I/O operations** - A running program may require I/O, which may involve a file or an I/O device. So, OS must provide the required I/O
- ✓ **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- ✓ **Communications** – Processes may exchange information, on the same computer or between computers over a network. Communications may be implemented by two methods
  - i) shared memory
  - ii) message passing (packets moved by the OS)
- ✓ **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program.
  - For each type of error, OS should take the appropriate action to ensure correct and consistent computing.

- Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system.

Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing

- ✓ **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them.
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- ✓ **Accounting -** To keep track of which users use how much and what kinds of computer resources
- ✓ **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.

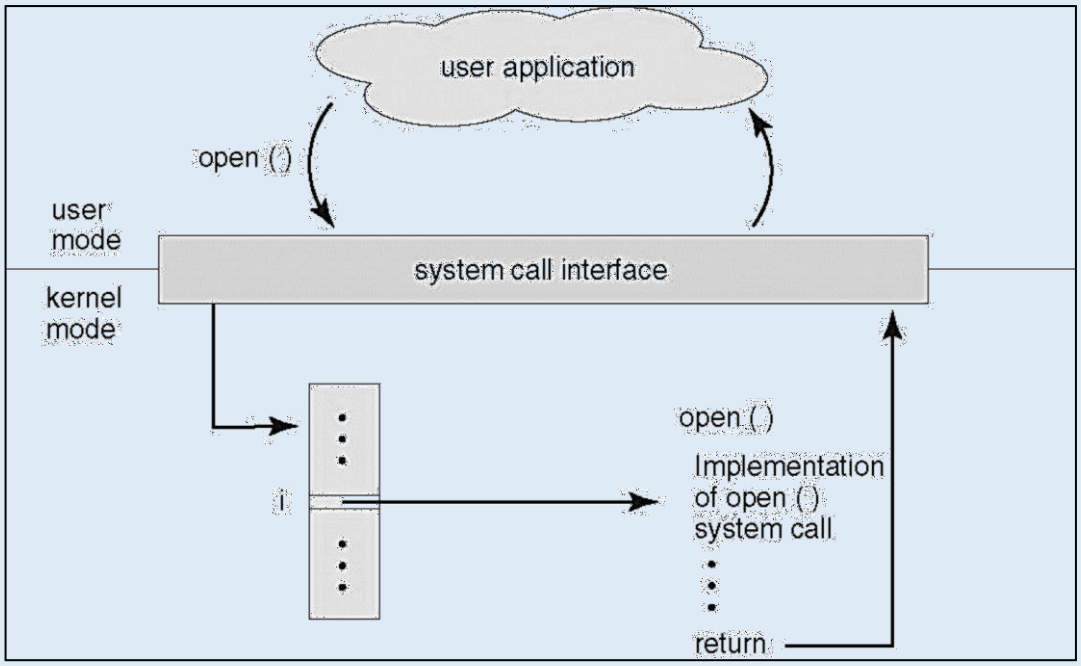


A view of operating system services

System Calls

- System calls provide the interface between a running program and the operating system.
- Generally available as assembly-language instructions.
- Languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C. Bliss, PL/360, PERL)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use

- Three general methods are used to pass parameters between a running program and the operating system.
  - ✓ Pass parameters in *registers*.
  - ✓ Store the parameters in a table or in memory, and the table address is passed as a parameter in a register.
  - ✓ *Push* (store) the parameters onto the *stack* by the program, and *pop* off the stack by operating system.
- Typically, a number associated with each system call
- **System-call interface** maintains a table indexed according to these numbers
- The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler)



The handling of a user application invoking the open() system call.

System Calls Categories

1. Process control

- End, abort
- Load, execute
- Create process, terminate process
- Get process attributes, set process attributes
- Wait event, signal event
- Allocate and free memory

2. File management

- Create file, delete file
- Open, close
- Read, write, reposition
- Get file attributes, set device attributes

3. Device management

- Request device, release device
- Read, write, reposition
- Get device attributes, set device attributes
- Logically attach or detach devices

4. Information maintenance

- Get time or date, set time or date
- Get system data, set system data
- Get process, file, or device attributes
- Set process, file, or device attributes

5. Communications

- Create, delete communication connection
- Send, receive messages
- Transfer status information
- Attach or detach remote devices

6. Protection

- Control access to resources
- Get and set permissions
- Allow and deny user access

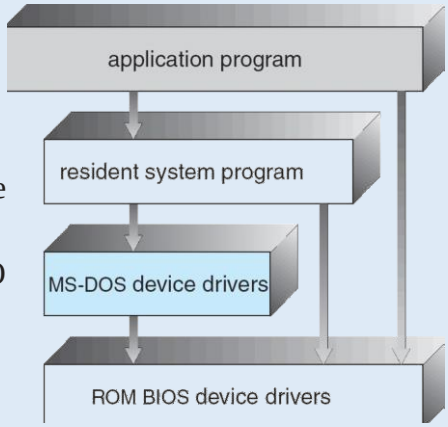
Structure of an OS

- A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.
- partition the task into small components, or modules.
- Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.
- There are 5 different structures based on how components are interconnected into kernel.
    1. Simple Structure
    2. Monolithic Structure

3. Layered Approach
4. Microkernels
5. Modules

Simple Structure

- Many operating systems do not have well-defined structures.
- This type of OS started as small, simple, and limited systems.
- MS-DOS is an example of such a system.
- MS-DOS – written to provide the most functionality in the least space
- Not divided into modules
- In MS-DOS, the interfaces and levels of functionality are not well separated.
- application programs are able to access the basic I/O routines to write directly to the display and disk drives.
- Hence, MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.



- MS-DOS was also limited by the hardware.
- Intel 8088 for which it was written provides no dual mode and no hardware protection,

Advantages

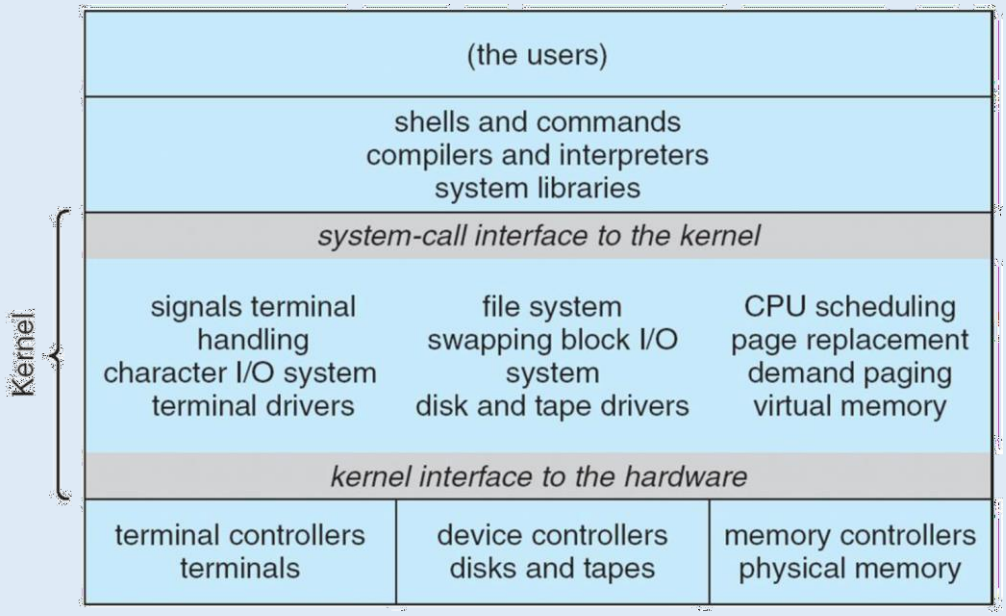
- Easy for kernel developers to develop such operating system.
- It delivers better application performance.

Disadvantages

- The structure is very complicated as no clear boundaries exists between modules
- If any service fails it leads to the failure of the entire system.
- More security issues are always here.

Monolithic Operating System

- The monolithic operating system controls all aspects of the operating system's operation, including file management, memory management, device management, etc.
- The core of an operating system for computers is called the kernel (OS). All other System components are provided with fundamental services by the kernel. The operating system and the hardware use it as their main interface. When an operating system is built into a single piece of hardware, such as a keyboard or mouse, the kernel can directly access all of its resources.
- Working on top of the operating system and under complete command of all hardware, the monolithic kernel performs the role of a virtual computer.
- This is an old operating system that was used in banks to carry out simple tasks like batch processing and time-sharing, which allows numerous users at different terminals to access the Operating System.



Example

- Unix
- Windows 9x series
- Solaris
- IBM – AIX
- BSD

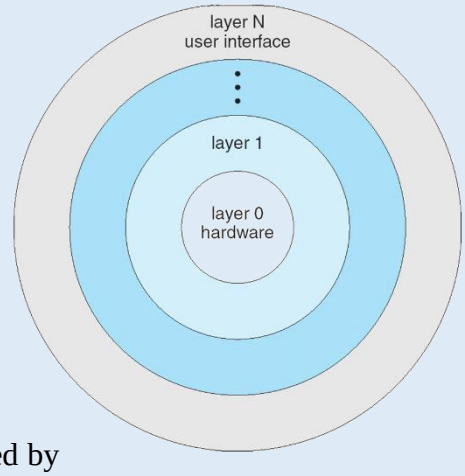
Advantages

- Simple and easy to implement.
- Faster execution due to direct access to all the services.
- A process runs completely in single address space in the monolithic kernel.
- The monolithic kernel is a static single binary file.

Disadvantages

- Addition of new features or removal of obsolete features is very difficult.
- If any service fails in the monolithic kernel → the failure of the entire system.
- Security issues are always there → there is no isolation among various servers present in the kernel

Layered Approach



- The operating system is broken into a number of layers (levels).
- The bottom layer (layer 0) is the hardware; the highest (layer *N*) is the user interface.
- The main advantage of the layered approach is simplicity of construction and debugging.
- This approach simplifies debugging and system verification.
- the design and implementation of the system are simplified.
- Each layer is implemented only with operations provided by lower-level layers.
- each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.
- The major difficulty with the layered approach involves appropriately defining the various layers.
- Because a layer can use only lower-level layers,
- careful planning is necessary

some of Layers are

layer 0 - Hardware
layer 1 – CPU scheduling
layer 2 – Memory Management
layer 3 – Process Management
layer 4 – I/O buffer
layer 5 – User Interface

Advantages

- Easy debugging
- Easy update
- No direct access to hardware

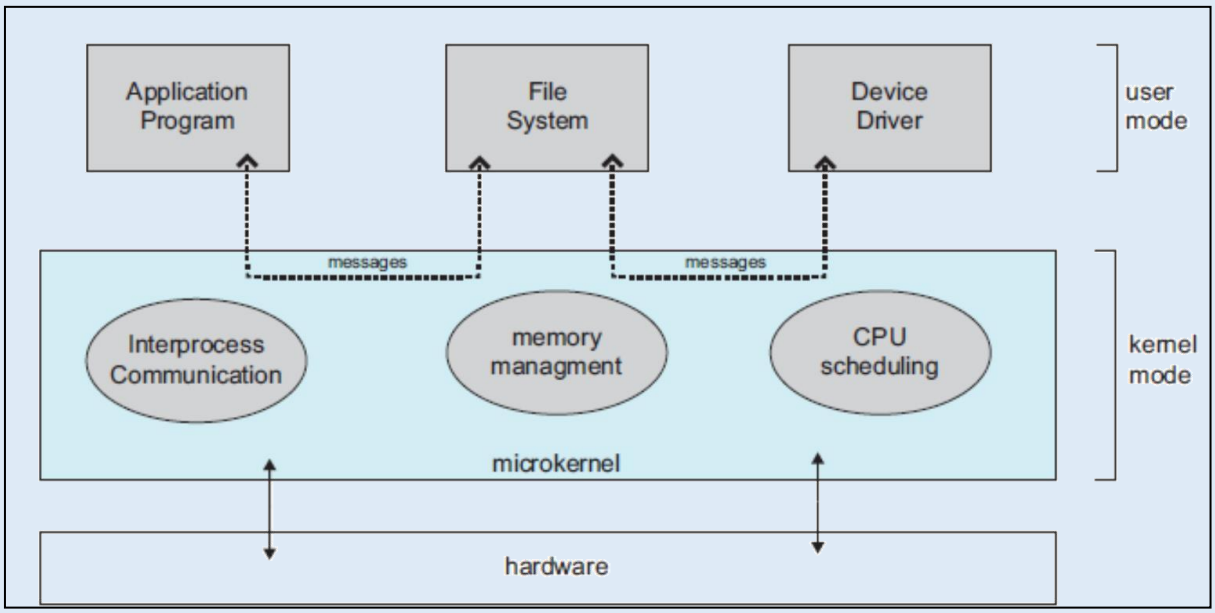
Disadvantages

- Complex and careful implementation.
- Slower in execution
- functionality

Micro kernel Operating System Structure

- This method structures the operating system by removing all nonessential components from the kernel and implementing them as system programs and user-level programs.
- The result is a smaller kernel.
- There is little difficulty here, which services should remain in the kernel and which should be implemented in user space.

- microkernels provide minimal process like CPU scheduling, memory management and a communication facility.
- The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space.
- Communication is provided through **message passing.**



Advantages

- it makes extending the operating system easier without modification of the kernel.
- operating system is easier to portable from one hardware design to another.
- It provides more security and reliability since most services are running as user—rather than kernel—processes.
- If a service fails, the rest of the operating system remains untouched.

Disadvantage

- the performance of microkernels can suffer due to increased system-function overhead.

Case Study on UNIX Operating System

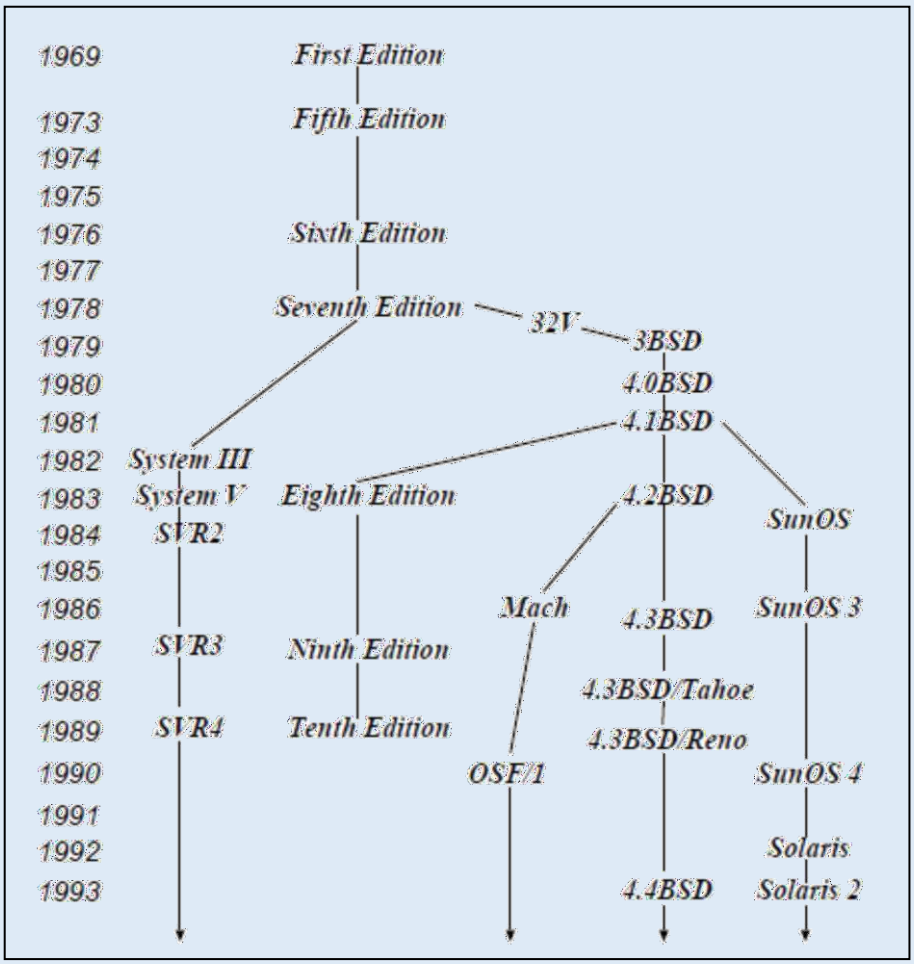
Unix: Introduction

• Unix first developed in 1969 at Bell Labs (Thompson & Ritchie)
• Originally written in PDP-7 asm, but then (1973) rewritten in the 'new' high-level language C
  - ➢ easy to port, alter, read, etc.
• 6th edition ("V6") was widely available (1976).
  – source avail) people could write new tools.
  – nice features of other OSes rolled in promptly.
• By 1978, V7 available (for both the 16-bit PDP-11 and the new 32-bit VAX-11).
• Since then, two main families:

– AT&T: “System V”, currently SVR4.
– Berkeley: “BSD”, currently 4.3BSD/4.4BSD.

• Standardisation efforts (e.g. POSIX, X/OPEN) to homogenise.
• Best known “UNIX” today is probably linux, but also get FreeBSD, NetBSD, and (commercially) Solaris, OSF/1, IRIX, and Tru64.

Unix Family Tree (Simplified)



Design Features

Ritchie and Thompson writing in CACM, July 74, identified the following (new) features of UNIX:

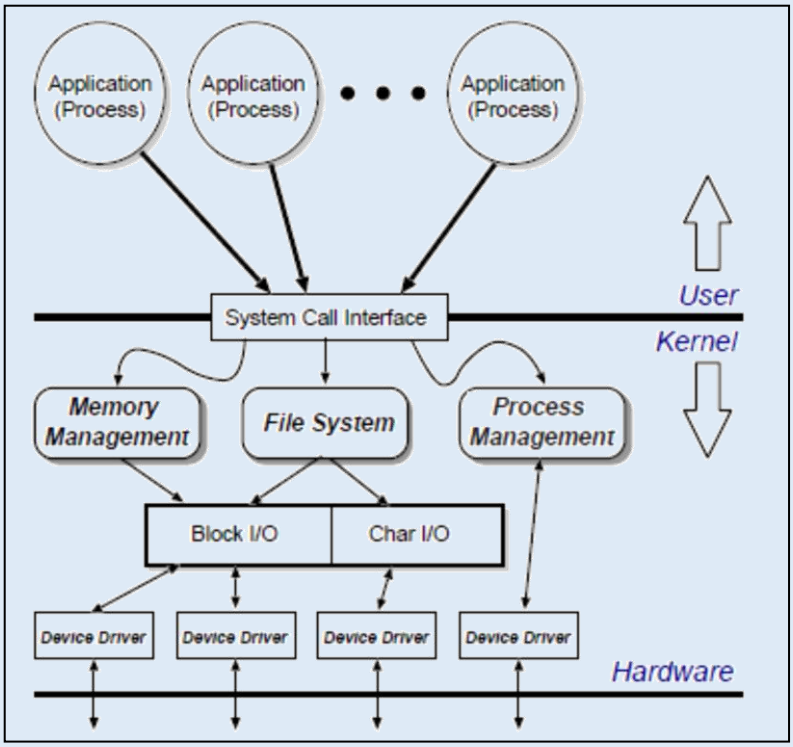
1. A hierarchical file system incorporating demountable volumes.
2. Compatible file, device and inter-process I/O.
3. The ability to initiate asynchronous processes.
4. System command language selectable on a per-user basis.
5. Over 100 subsystems including a dozen languages.
6. A high degree of portability.

Features which were not included:

• real time
• multiprocessor support

Fixing the above is pretty hard.

Structural Overview



- Clear separation between user and kernel portions.
- Processes are unit of scheduling and protection.
- All I/O looks like operations on files.

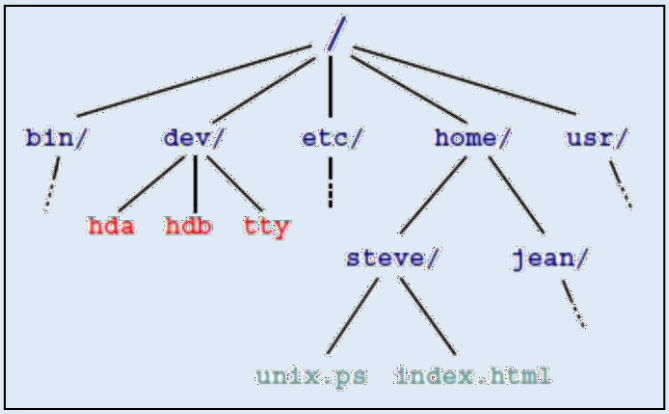
File Abstraction

- A file is an unstructured sequence of bytes.
- Represented in user-space by a file descriptor (fd)
- Operations on files are:
  - fd = open (pathname, mode)
  - fd = creat(pathname, mode)
  - bytes = read(fd, buffer, nbytes)
  - count = write(fd, buffer, nbytes)
  - reply = seek(fd, offset, whence)
  - reply = close(fd)
- Devices represented by special files:
  - support above operations, although perhaps with bizarre semantics.
  - also have ioctl's: allow access to device-specific functionality.
- Hierarchical structure supported by directory files.

Directory Hierarchy

- Directories map names to files (and directories).
- Have distinguished root directory called '/'
- Fully qualified pathnames) perform traversal from root.
- Every directory has '.' and '..' entries: refer to self and parent respectively.

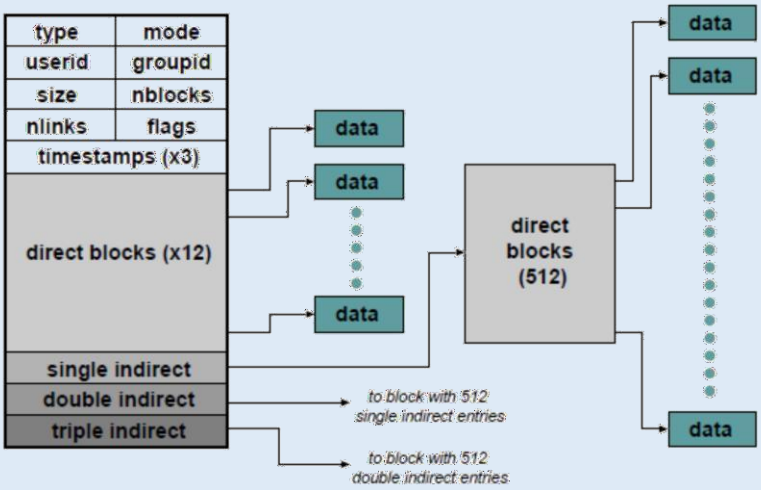
• Shortcut: current working directory (cwd).
• In addition shell provides access to home directory as ~username (e.g. ~steve/)



Aside: Password File

• /etc/passwd holds list of password entries.
• Each entry roughly of the form:
user-name:encrypted-passwd:home-directory:shell
• Use one-way function to encrypt passwords.
– i.e. a function which is easy to compute in one direction, but has a hard to compute inverse (e.g. person to phone-number lookup).
• To login:
1. Get user name
2. Get password
3. Encrypt password
4. Check against version in /etc/password
5. If ok, instantiate login shell.
• Publicly readable since lots of useful info there.
• Problem: off-line attack.
• Solution: shadow passwords (/etc/shadow)

File System Implementation



• In kernel, a file is represented by a data structure called an index-node or i-node.

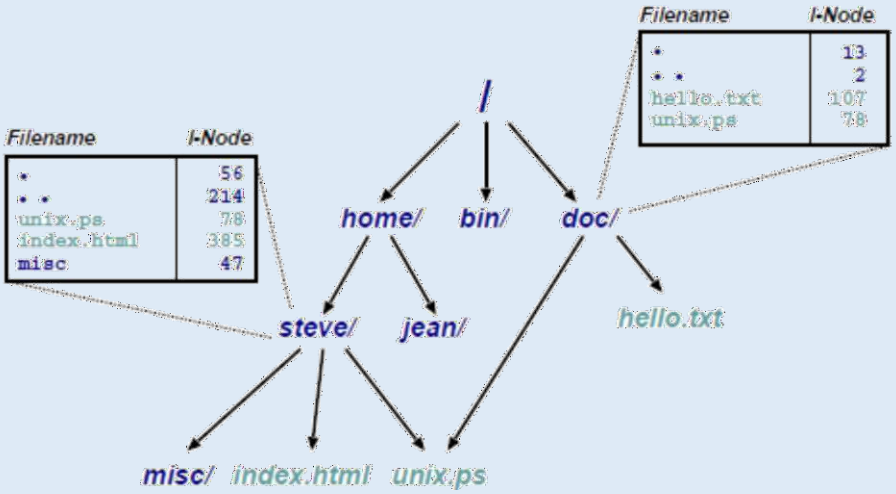
• Holds file meta-data:

a) Owner, permissions, reference count, etc.

b) Location on disk of actual data (file contents).

• Question: Where is the filename kept?

Directories and Links



• Directory is a file which maps filenames to i-nodes.

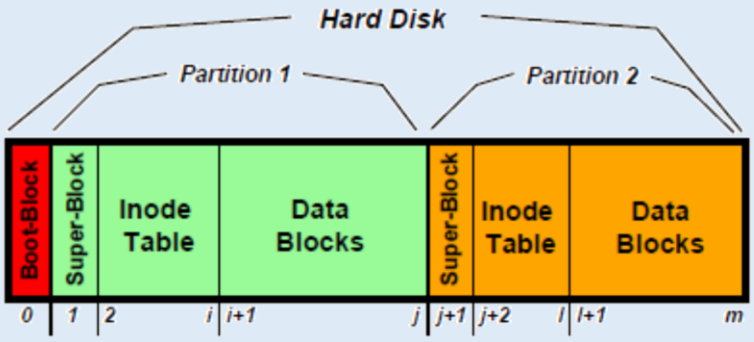
• An instance of a file in a directory is a (hard) link.

• (this is why have reference count in i-node).

• Directories can have at most 1 (real) link. Why?

• Also get soft- or symbolic-links: a 'normal' file which contains a filename.

On-Disk Structures



• A disk is made up of a boot block followed by one or more partitions.

• (a partition is a contiguous range of N fixed-size blocks of size k for some N, k).

• A Unix file-system resides within a partition.

• The file-system superblock contains info such as:

– number of blocks in file-system

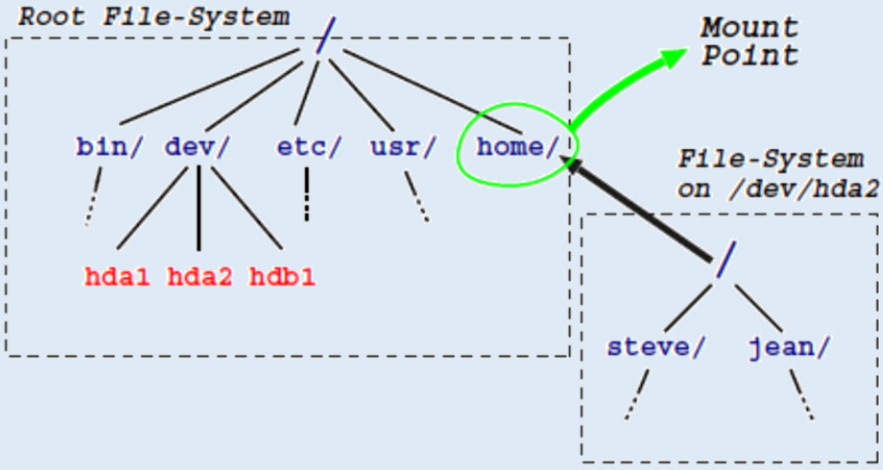
– number of free blocks in file-system

– start of the free-block list

– start of the free-inode list.

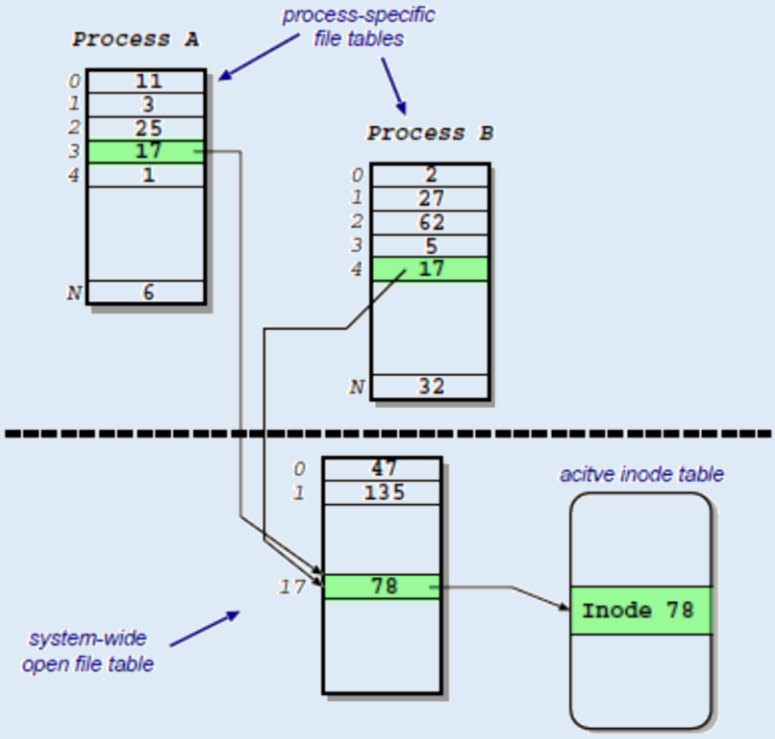
– various bookkeeping information.

Mounting File-Systems



- Entire file-systems can be mounted on an existing directory in an already mounted filesystem.
- At very start, only '/' exists ➔ need to mount a root file-system.
- Subsequently can mount other file-systems, e.g. mount("/dev/hda2", "/home", options)
- Provides a unified name-space: e.g. access /home/steve/ directly.
- Cannot have hard links across mount points: why?
- What about soft links?

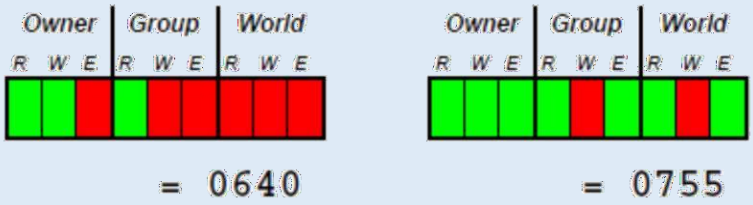
In-Memory Tables



- Recall process sees files as file descriptors
- In implementation these are just indices into a process-specific open file table.
- Entries point to system-wide open file table. Why?

• These in turn point to (in memory) inode table.

Access Control



• Access control information held in each inode.
• Three bits for each of owner, group and world: { read, write and execute }
• Question: What do these mean for directories?
• In addition have setuid and setgid bits:
  – normally processes inherit permissions of invoking user.
  – setuid/setgid allow the user to "become" someone else when running a particular program.
  – e.g. prof owns both executable test (0711 and setuid), and score file (0600)
    ==> any user can run it.
    ==> it can update score file.
    ==> but users can't cheat.
• Question: and what do these mean for directories?

Consistency Issues

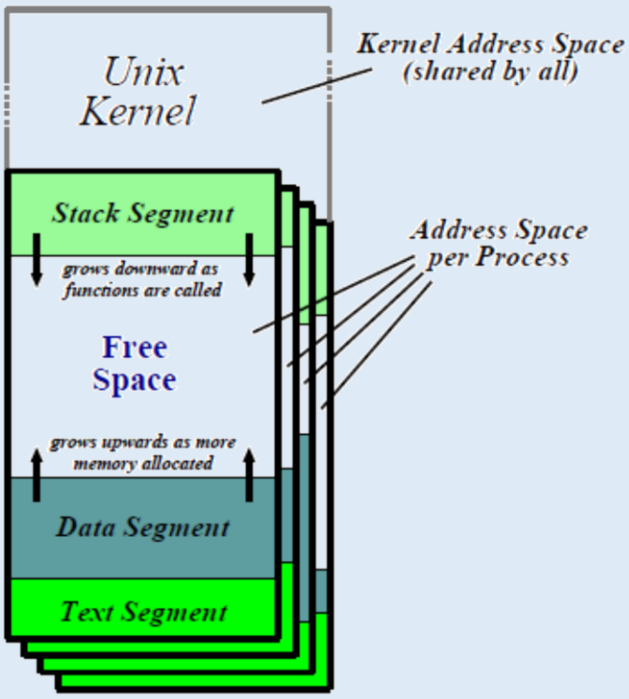
• To delete a file, use the unlink system call.
• From the shell, this is rm <filename>
• Procedure is:
  1. check if user has sufficient permissions on the file (must have write access).
  2. check if user has sufficient permissions on the directory (must have write access).
  3. if ok, remove entry from directory.
  4. Decrement reference count on inode.
  5. if now zero:
     a. free data blocks.
     b. free inode.
• If the system crashes: must check entire file-system:
  – check if any block unreferenced.
  – check if any block double referenced.
• (We'll see more on this later)

Unix File-System: Summary

• Files are unstructured byte streams.
• Everything is a file: 'normal' files, directories, symbolic links, special files.
• Hierarchy built from root ('/').
• Unified name-space (multiple file-systems may be mounted on any leaf directory).

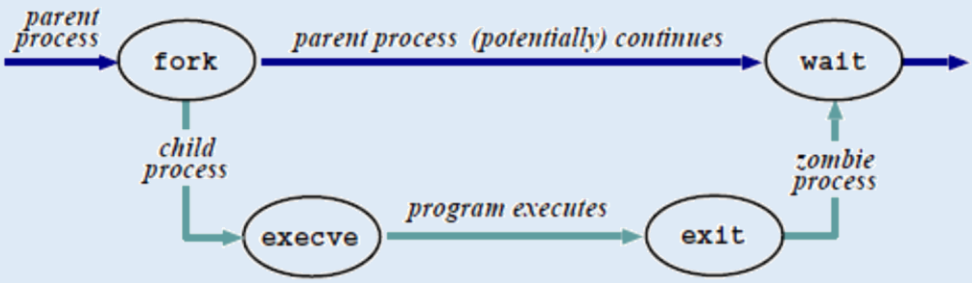
• Low-level implementation based around inodes.
• Disk contains list of inodes (along with, of course, actual data blocks).
• Processes see file descriptors: small integers which map to system file table.
• Permissions for owner, group and everyone else.
• Setuid/setgid allow for more flexible control.
• Care needed to ensure consistency.

Unix Processes



• Recall: a process is a program in execution.
• Have three segments: text, data and stack.
• Unix processes are heavyweight.

Unix Process Dynamics



• Process represented by a process id (pid)
• Hierarchical scheme: parents create children.
• Four basic primitives:
  – pid = fork ()
  – reply = execve(pathname, argv, envp)

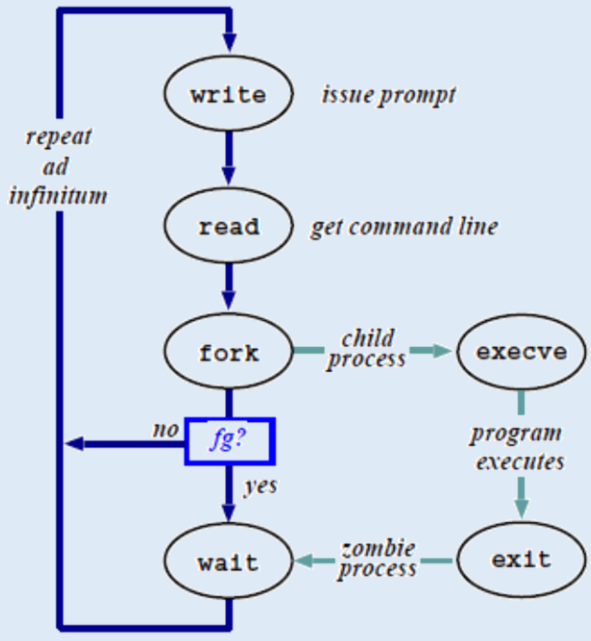
– exit(status)
– pid = wait (status)

• fork() nearly always followed by exec()
==> vfork() and/or COW.

Start of Day

• Kernel (/vmunix) loaded from disk (how?) and execution starts.
• Root file-system mounted.
• Process 1 (/etc/init) hand-crafted.
• init reads file /etc/inittab and for each entry:
1. opens terminal special file (e.g. /dev/tty0)
2. duplicates the resulting fd twice.
3. forks an /etc/tty process.
• each tty process next:
1. initialises the terminal
2. outputs the string "login:" & waits for input
3. execve()'s /bin/login
• login then:
1. outputs "password:" & waits for input
2. encrypts password and checks it against /etc/passwd.
3. if ok, sets uid & gid, and execve()'s shell.
• Patriarch init resurrects /etc/tty on exit.

The Shell



• The shell just a process like everything else.
• Uses path (= list of directories to search) for convenience.
• Conventionally '&' specifies run in background.
• Parsing stage (omitted) can do lots. . .

Shell Examples

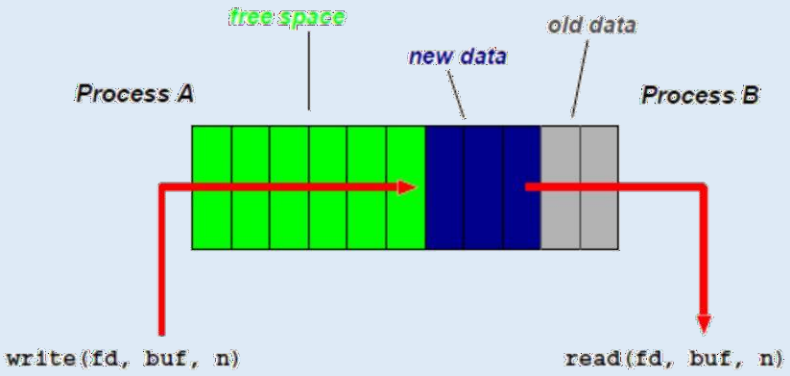
```
# pwd
/home/steve
# ls -F
IRAM.micro.ps                gnome_sizes         prog-nc.ps
Mail/                        ica.tgz             rafe/
OSDI99_self_paging.ps.gz     lectures/           rio107/
TeX/                         linbot-1.0/         src/
adag.pdf                     manual.ps           store.ps.gz
docs/                        past-papers/        wolfson/
emacs-lisp/                  pbosch/             xeno_prop/
fs.html                      pepsi_logo.tif
# cd src/
# pwd
/home/steve/src
# ls -F
cdq/          emacs-20.3.tar.gz  misc/      read_mem.c
emacs-20.3/   ispell/            read_mem*  rio007.tgz
# wc read_mem.c
     95     225    2262 read_mem.c
# ls -lF r*
-rwxrwxr-x   1 steve  user    34956 Mar 21  1999 read_mem*
-rw-rw-r--   1 steve  user     2262 Mar 21  1999 read_mem.c
-rw-------   1 steve  user    28953 Aug 27 17:40 rio007.tgz
# ls -l /usr/bin/X11/xterm
-rwxr-xr-x   2 root   system 164328 Sep 24 18:21 /usr/bin/X11/xterm*
```

- Prompt is '#'.
- Use man to find out about commands.
- User friendly?

Standard I/O

- Every process has three fds on creation:
  - stdin: where to read input from.
  - stdout: where to send output.
  - stderr: where to send diagnostics.
- Normally inherited from parent, but shell allows redirection to/from a file, e.g.:
  - ls >listing.txt
  - ls >&listing.txt
  - sh <commands.sh.
- Actual file not always appropriate; e.g. consider:
  ls >temp.txt;
  wc <temp.txt >results
- Pipeline is better (e.g. ls | wc >results)
- Most Unix commands are filters, i.e. read from stdin and output to stdout ==> can build almost arbitrarily complex command lines.
- Redirection can cause some buffering subtleties.

Pipes



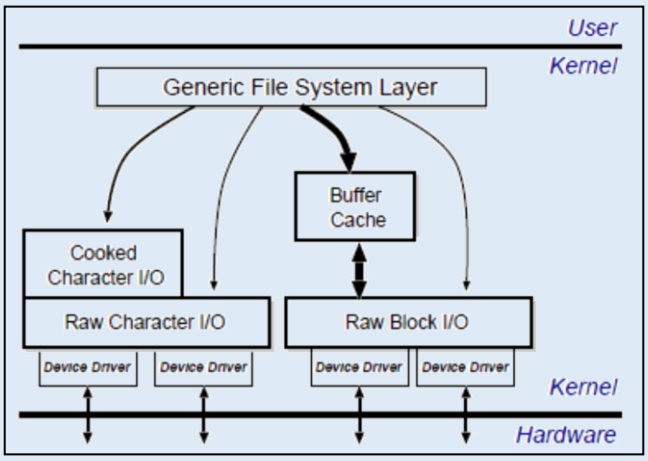
• One of the basic Unix IPC schemes.
• Logically consists of a pair of fds, one for each 'end' of the pipe.
• e.g. reply = pipe(int fds[2])
• Concept of "full" and "empty" pipes.
• Only allows communication between processes with a common ancestor (why?).
• Named pipes address this. . .

Signals

• Problem: pipes need planning) use signals.
• Similar to a (software) interrupt.
• Examples:
  – SIGINT : user hit Ctrl-C.
  – SIGSEGV : program error.
  – SIGCHLD : a death in the family. . .
  – SIGTERM : . . . or closer to home.
• Unix allows processes to catch signals.
• e.g. Job control:
  – SIGTTIN, SIGTTOU sent to bg processes
  – SIGCONT turns bg to fg.
  – SIGSTOP does the reverse.
• Cannot catch SIGKILL (hence kill -9)
• Signals can also be used for timers, window resize, process tracing, . . .

I/O Implementation

• Recall:
  – everything accessed via the file system.
  – two broad categories: block and char.
• Low-level stuff gory and machine dependent ==> ignore.
• Character I/O is low rate but complex ==> most code in the "cooked" interface.
• Block I/O simpler but performance matters ==> emphasis on the buffer cache.



The Buffer Cache

• Basic idea: keep copy of some parts of disk in memory for speed.
• On read do:
1. Locate relevant blocks (from inode)
2. Check if in buffer cache.
3. If not, read from disk into memory.
4. Return data from buffer cache.
• On write do same first three, and then update version in cache, not on disk.
• "Typically" prevents 85% of implied disk transfers.
• Question: when does data actually hit disk?
• Answer: call sync every 30 seconds to flush dirty buffers to disk.
• Can cache metadata too — problems?

Unix Process Scheduling

• Priorities 0–127; user processes _ PUSER = 50.
• Round robin within priorities, quantum 100ms.
• Priorities are based on usage and nice value, i.e.

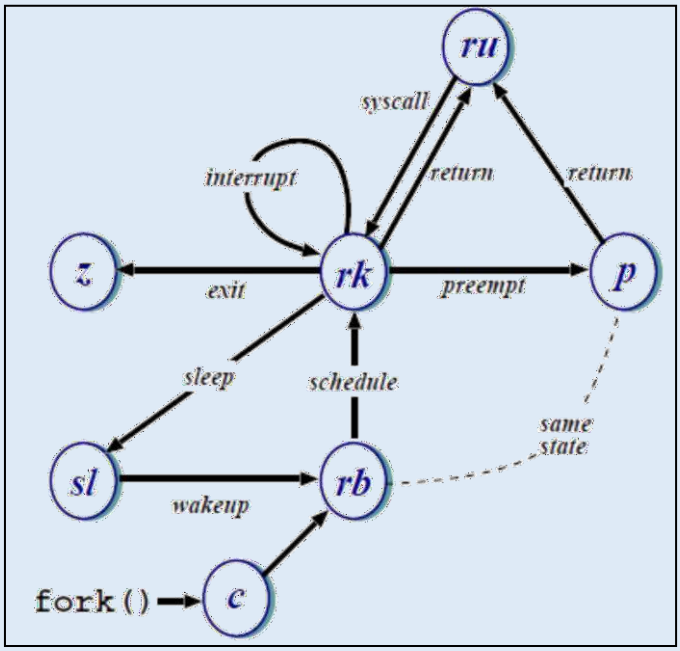
$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{4} + 2 \times nice_j$$

gives the priority of process j at the beginning of interval i where:

$$CPU_j(i) = \frac{2 \times load_j}{(2 \times load_j) + 1} CPU_j(i-1) + nice_j$$

and $nice_j$ is a (partially) user controllable adjustment parameter € [−20, 20].
• $load_j$ is the sampled average length of the run queue in which process j resides, over the last minute of operation
• so if e.g. load is 1 ==> approximately 90% of 1 seconds CPU usage will be "forgotten" within 5 seconds.

Unix Process States



ru = running (user-mode) rk = running (kernel-mode)
z = zombie p = pre-empted
sl = sleeping rb = runnable
c = created

Summary

• Main Unix features are:

– file abstraction
- a file is an unstructured sequence of bytes
- (not really true for device and directory files)

– hierarchical namespace
- directed acyclic graph (if exclude soft links)
- can recursively mount filesystems

– heavy-weight processes

– IPC: pipes & signals

– I/O: block and character

– dynamic priority scheduling
- base priority level for all processes
- priority is lowered if process gets to run
- over time, the past is forgotten

• But Unix V7 had inflexible IPC, inefficient memory management, and poor kernel concurrency.

• Later versions address these issues.

Case study on WINDOWS Operating System

Windows NT: **Evolution**

• Feb 2000: NT 5.0 aka Windows 2000
  – borrows from windows 98 look 'n feel
  – both server and workstation versions, latter of which starts to get wider use
  – big push to finally kill DOS/Win 9x family (but fails due to internal politicking)
• Windows XP (NT 5.1) launched October 2001
  – home and professional ==> finally kills win 9x.
  – various "editions" (media center, 64-bit) & service packs (SP1, SP2, SP3)
• Server product Windows Server 2003 (NT 5.2) released 2003
  – basically the same modulo registry tweaks, support contract and of course cost
  – a plethora of editions. . .
• Windows Vista (NT 6.0) limped onto the scene Q4 2006
  – new Aero UI, new WinFX API
  – missing Longhorn bits like WinFS, Msh
• Windows Server 2008 (also based on NT 6.0, but good) landed Feb 2008
• Windows 7 (NT 6.1 for now) released October 2009. . .

NT Design Principles

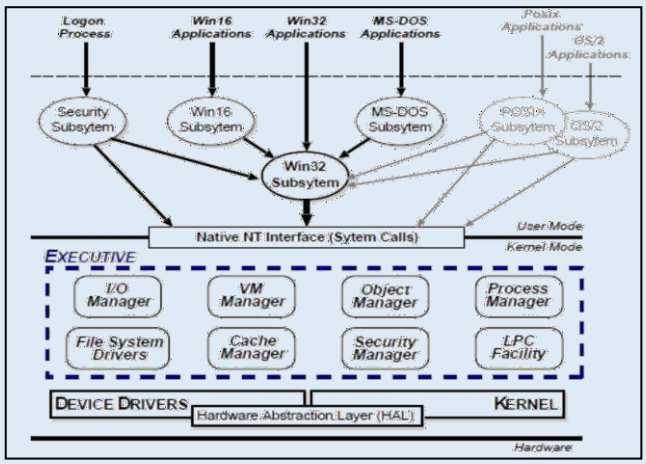
Key goals for the system were:

• portability
• security
• POSIX compliance
• multiprocessor support
• extensibility
• international support
• compatibility with MS-DOS/Windows applications

This led to the development of a system which was:

• written in high-level languages (C and C++)
• based around a micro-kernel, and
• constructed in a layered/modular fashion.

Structural Overview



• Kernel Mode: HAL, Kernel, & Executive
• User Mode: environmental subsystems, protection subsystem

HAL

• Layer of software (HAL.DLL) which hides details of underlying hardware
• e.g. low-level interrupt mechanisms, DMA controllers, multiprocessor communication mechanisms
• Several HALs exist with same interface but different implementation (often vendor-specific, e.g. for large cc-NUMA machines)

Kernel

• Foundation for the executive and the subsystems
• Execution is never preempted.
• Four main responsibilities:
    1. CPU scheduling
    2. interrupt and exception handling
    3. low-level processor synchronisation
    4. recovery after a power failure
• Kernel is objected-oriented; all objects are either dispatcher objects (active or temporal things) or control objects (everything else)

Processes and Threads

NT splits the "virtual processor" into two parts:
1. A process is the unit of resource ownership.
Each process has:
    • a security token,
    • a virtual address space,
    • a set of resources (object handles), and
    • one or more threads.
2. A thread are the unit of dispatching.
Each thread has:
    • a scheduling state (ready, running, etc.),
    • other scheduling parameters (priority, etc),
    • a context slot, and
    • (generally) an associated process.

Threads are:
• co-operative: all threads in a process share address space & object handles.
• lightweight: require less work to create/delete than processes (mainly due to shared virtual address space).

CPU Scheduling

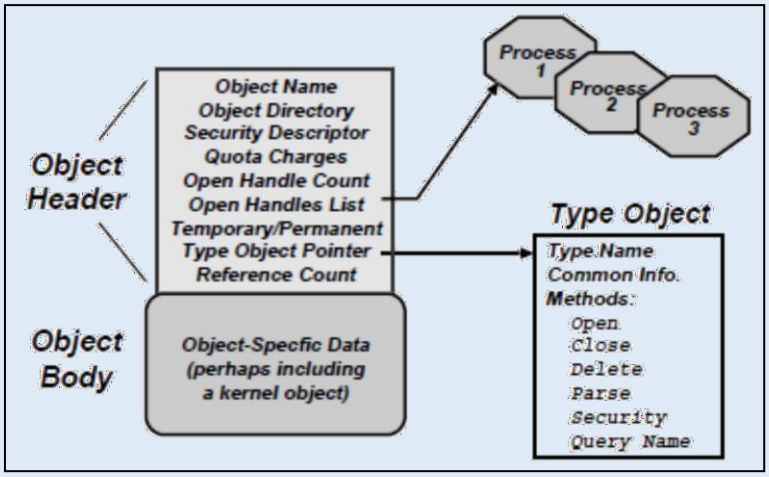
• Hybrid static/dynamic priority scheduling:
    – Priorities 16–31: "real time" (static priority).
    – Priorities 1–15: "variable" (dynamic) priority.
    – (priority 0 is reserved for zero page thread)
• Default quantum 2 ticks (~20ms) on Workstation, 12 ticks (~120ms) on Server.
• Threads have base and current (>=base) priorities.

– On return from I/O, current priority is boosted by driver-specific amount.
– Subsequently, current priority decays by 1 after each completed quantum.
– Also get boost for GUI threads awaiting input: current priority boosted to 14 for one quantum (but quantum also doubled)
– Yes, this is true.

• On Workstation also get quantum stretching:
– ". . . performance boost for the foreground application" (window with focus)
– fg thread gets double or triple quantum.

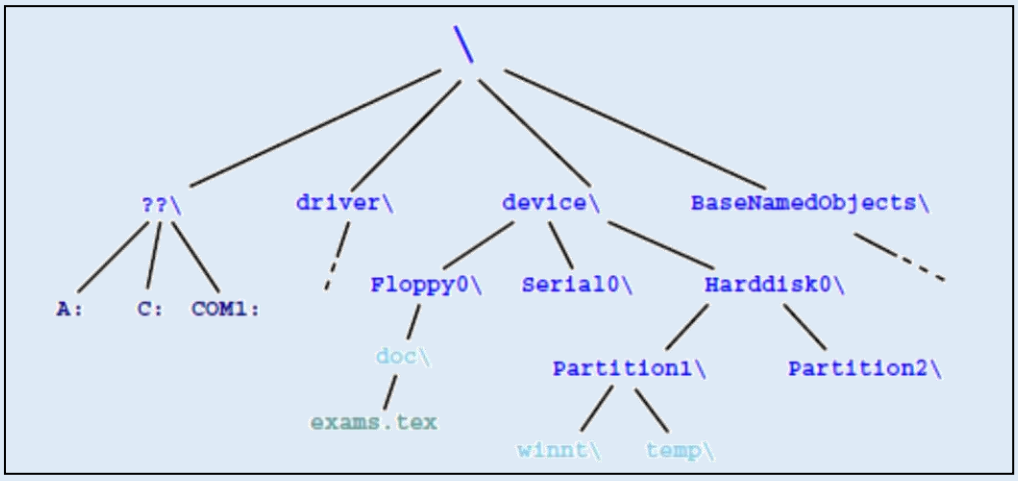
• If no runnable thread, dispatch 'idle' thread (which executes DPCs).

Object Manager



• Every resource in NT is represented by an object
• The Object Manager (part of the Executive) is responsible for:
– creating objects and object handles
– performing security checks
– tracking which processes are using each object
• Typical operation:
– handle = open(objectname, accessmode)
– result = service(handle, arguments)

Object Namespace



• Recall: objects (optionally) have a name
• Object Manger manages a hierarchical namespace:
  – shared between all processes ==> sharing
  – implemented via directory objects
  – each object protected by an access control list.
  – naming domains (using parse) mean file-system namespaces can be integrated
• Also get symbolic link objects: allow multiple names (aliases) for the same object.
• Modified view presented at API level. . .

Process Manager

• Provides services for creating, deleting, and using threads and processes.
• Very flexible:
  – no built in concept of parent/child relationships or process hierarchies
  – processes and threads treated orthogonally.

==> can support Posix, OS/2 and Win32 models.

Virtual Memory Manager

• NT employs paged virtual memory management
• The VMM provides processes with services to:
  – allocate and free virtual memory
  – modify per-page protections
• Can also share portions of memory:
  – use section objects (~software segments)
  – section objects are either based (specific base address) or non-based (floating)
  – also used for memory-mapped files

Security Reference Manager

• NT's object-oriented nature enables a uniform mechanism for runtime access and audit checks
  – everytime a process opens handle to an object, check process's security token and object's ACL
  – compare with Unix (file-system, networking, window system, shared memory)

Local Procedure Call Facility

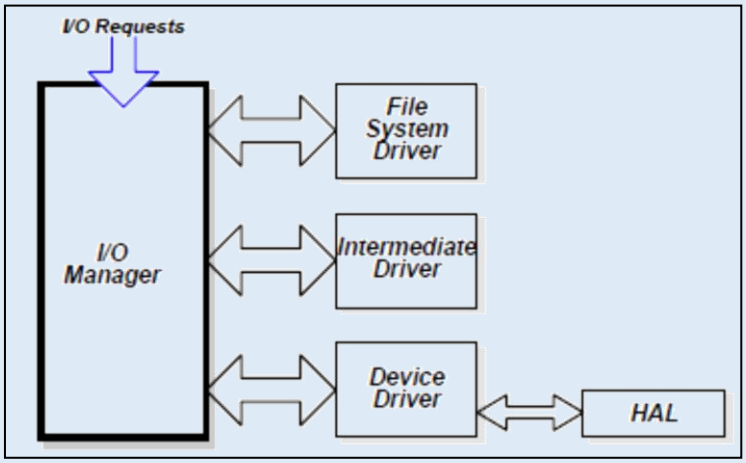
• LPC (or IPC) passes requests and results between client and server processes within a single machine.
• Used to request services from the various NT environmental subsystems.
• Three variants of LPC channels:
  1. small messages (_ 256 bytes): copy messages between processes
  2. zero copy: avoid copying large messages by pointing to a shared memory section object created for the channel.
  3. quick LPC: used by the graphical display portions of the Win32 subsystem.

I/O Manager

• The I/O Manager is responsible for:
  – file systems
  – cache management
  – device drivers
• Basic model is asynchronous:
  – each I/O operation explicitly split into a request and a response

– an I/O Request Packet (IRP) used to hold parameters, results, etc.

• File-system & device drivers are stackable. . .



Cache Manager

• Cache Manager caches "virtual blocks":

– viz. keeps track of cache "lines" as offsets within a file rather than a volume.

– disk layout & volume concept abstracted away.

- no translation required for cache hit.
- can get more intelligent prefetching

• Completely unified cache:

– cache "lines" all live in the virtual address space.

– decouples physical & virtual cache systems: e.g.

- virtually cache in 256K blocks,
- physically cluster up to 64K.

– NT virtual memory manager responsible for actually doing the I/O.

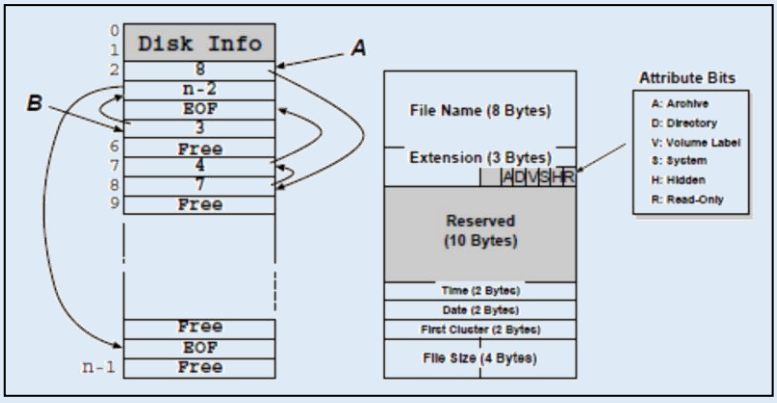
– so lots of FS cache when VM system lightly loaded, little when system thrashing

• NT also provides some user control:

– if specify temporary attrib when creating file ==> data will never be flushed to disk unless absolutely necessary.

– if specify write through attrib when opening a file ==> all writes will synchronously complete.

File Systems: FAT16



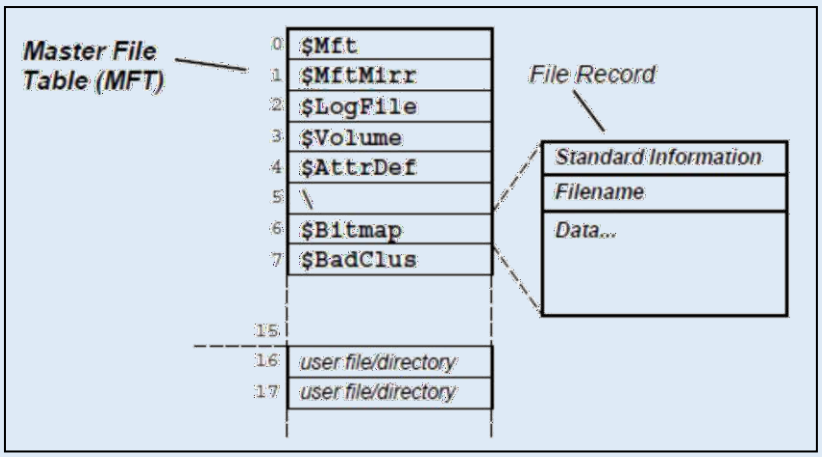
• A file is a linked list of clusters (= a set of 2^n contiguous disk blocks, n >= 0)
• Each entry in the FAT contains either:
  – the index of another entry within the FAT, or
  – a special value EOF meaning "end of file", or
  – a special value Free meaning "free".
• Directory entries contain index into the FAT
• FAT16 could only handle partitions up to (2^{16} × c) bytes ==> max 2Gb partition with 32K clusters (and big cluster size is bad)

File Systems: FAT32

• Obvious extension: instead of using 2 bytes per entry, FAT32 uses 4 bytes
  ➢ can support e.g. 8Gb partition with 4K clusters
• Further enhancements with FAT32 include:
  – can locate the root directory anywhere on the partition (in FAT16, the root directory had to immediately follow the FAT(s)).
  – can use the backup copy of the FAT instead of the default (more fault tolerant)
  – improved support for demand paged executables (consider the 4K default cluster size . . .).
• VFAT on top of FAT32 adds long name support and internationalization:
  – names now unicode strings of up to 256 characters.
  – want to keep same directory entry structure for compatibility with e.g. DOS
  ➢ use multiple directory entries to contain successive parts of name.
  – abuse V attribute to avoid listing these

Still pretty primitive. . .

File-Systems: NTFS



• Fundamental structure of NTFS is a volume:
  – based on a logical disk partition
  – may occupy a portion of a disk, and entire disk, or span across several disks.
• NTFS stores all file records in a special file called the Master File Table (MFT).
• The MFT is indexed by a file reference: a 64-bit unique identifier for a file
• A file itself is a structured object consisting of set of attribute/value pairs of variable length…

NTFS: Recovery

• To aid recovery, all file system data structure updates are performed inside transactions:

– before a data structure is altered, the transaction writes a log record that contains redo and undo information.

– after the data structure has been changed, a commit record is written to the log to signify that the transaction succeeded.

– after a crash, the file system can be restored to a consistent state by processing the log records.

• Does not guarantee that all the user file data can be recovered after a crash — just that metadata files will reflect some prior consistent state.

• The log is stored in the third metadata file at the beginning of the volume ($Logfile)

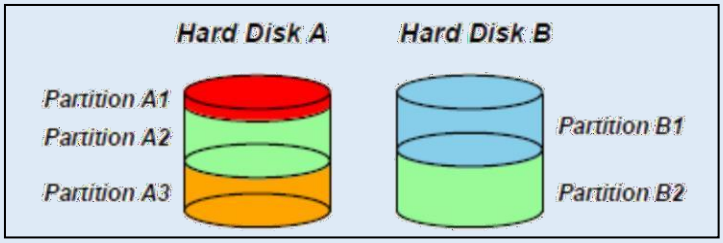
– in fact, NT has a generic log file service

➢ could in principle be used by e.g. database

• Overall makes for far quicker recovery after crash

• (modern Unix fs [ext3, xfs] use similar scheme)

NTFS: Fault Tolerance



• FtDisk driver allows multiple partitions be combined into a logical volume:

– e.g. logically concatenate multiple disks to form a large logical volume

– based on the concept of RAID = Redundant Array of Inexpensive Disks:

– e.g. RAID level 0: interleave multiple partitions round-robin to form a stripe set:

*logical block 0 ! block 0 of partition A2, logical block 1 → block 0 of partition B2, logical block 2 → block 1 of partition A2, etc

– e.g. RAID level 1 increases robustness by using a mirror set: two equally sized partitions on two disks with identical data contents.

– (other more complex RAID levels also exist)

• FtDisk can also handle sector sparing where the underlying SCSI disk supports it

• (if not, NTFS supports cluster remapping in software)

NTFS: Other Features

• Security:

– security derived from the NT object model.

– each file object has a security descriptor attribute stored in its MFT record.

– this atrribute holds the access token of file owner plus an access control list

• Compression:

– NTFS can divide a file's data into compression units (sets of 16 contiguous clusters in the file)

– NTFS also has support for sparse files

- clusters with all zeros not actually allocated or stored on disk.

- instead, gaps are left in the sequences of VCNs kept in the file record
- when reading a file, gaps cause NTFS to zero-fill that portion of the caller's buffer.

• Encryption:

– Use symmetric key to encrypt files; file attribute holds this key encrypted with user public key

– Not really that useful: private key pretty easy to obtain; and administrator can bypass entire thing anyhow.

Environmental Subsystems

• User-mode processes layered over the native NT executive services to enable NT to run programs developed for other operating systems.

• NT uses the Win32 subsystem as the main operating environment

– Win32 is used to start all processes.

– Also provides all the keyboard, mouse and graphical display capabilities.

• MS-DOS environment is provided by a Win32 application called the virtual dos machine (VDM), a user-mode process that is paged and dispatched like any other NT thread.

– Uses virtual 8086 mode, so not 100% compatible

• 16-Bit Windows Environment:

– Provided by a VDM that incorporates Windows on Windows

– Provides the Windows 3.1 kernel routines and stub routings for window manager and GDI functions.

• The POSIX subsystem is designed to run POSIX applications following the POSIX.1 standard which is based on the UNIX model.

Summary

• Main Windows NT features are:

– layered/modular architecture:

– generic use of objects throughout

– multi-threaded processes

– multiprocessor support

– asynchronous I/O subsystem

– NTFS filing system (vastly superior to FAT32)

– preemptive priority-based scheduling

• Design essentially more advanced than Unix.

• Implementation of lower levels (HAL, kernel & executive) actually rather decent.

• But: has historically been crippled by

– almost exclusive use of Win32 API

– legacy device drivers (e.g. VXDs)

– lack of demand for "advanced" features

– "feature interaction", aka huge swathes of complex poorly implemented user-space code written by idiots

• Continues to evolve. . .

KORN SHELL PROGRAMMING

Basic Script Concepts

shell

- A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.
- Shell is an environment in which we can run our commands, programs, and shell scripts.
- A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

  i. The Bourne Shell
  ii. The C Shell
  iii. The Korn Shell
  iv. The GNU Bourne-Again Shell

Shell Prompt

The prompt, $, which is called the command prompt, is issued by the shell. While the prompt is displayed, we can type a command.

Shell Types:

In Unix, there are two major types of shells −

i. Bourne shell − If we are using a Bourne-type shell, the $ character is the default prompt.

ii. C shell − If we are using a C-type shell, the % character is the default prompt.

The Bourne Shell has the following subcategories −

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow −

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

Shell Scripts

- The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by # sign, describing the steps.
- Shell scripts and functions are both interpreted. This means they are not compiled.

Creating a Shell Script

• A shell script is an executable file which is executed by the shell line-by-line. It can contain the following:

– UNIX commands
– shell programming statements
– comments

• Create using editor of choice
• Can include a #! construct in first line of script to override login shell

– #!/bin/sh uses Bourne shell to execute script
– #!/bin/csh uses C shell to execute script
– etc...

Executing a Shell Script

There are 3 ways to execute a shell script:

1."dot" method
$. scriptname
2."just the name" method
$ scriptname
3.in the background
$ scriptname &

• Method 1 runs the command as if you typed them in on the command line
• Note that methods 2 and 3 require:

– execute permission for scriptname
chmod +x scriptname
– current directory (.) must be in PATH or else must use
./scriptname

Variables

Korn shell includes the following types of variables:

– User-defined variables
– Special shell variables

• User-defined variables can be declared, read and changed from the command line or from within a shell script.
• A variable name can consist of the following:

– letters
– digits
– underscore character
– first character of a variable name must be a letter or an underscore character

- A variable can be made read-only using the readonly command: $ readonly variable_name

Assigning Variable Names

= Operator:

Enter the name that you have chosen for the variable followed by an equal sign and then the value that you want to store in the variable.

```
$ my_card=ace
$ print $my_card
ace

$ my_name="Sree Vishnu"
$ print $my_name
Sree Vishnu
```

read command:

Reads a line from standard input and stores the value(s) entered in the variable name(s) following the read command.

```
$ read fname lname
Sree Vishnu

$ print $fname
Sree

$ print $fname $lname
Sree Vishnu
```

Null Variables

A variable can be set to a null value, even if previously assigned, using any of the following methods. There should be no spaces preceding or following the equal sign.

```
$ name=
$ name="
$ name=""
$ unset varname
```

Then the shell will indicate an error when an undefined variable is encountered.

Operators & Expressions

There are various operators supported by each shell. Bourne shell (default shell) uses the following operators −

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers −

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result −

Total value : 4

The following points need to be considered while adding −

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between **‘ ‘**, called the backtick.

Arithmetic Operators & Expressions

The following arithmetic operators are supported by Bourne Shell. Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	**Description**	**Example**
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2

% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators & Expressions

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	**Description**	**Example**
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.

-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a <= $b]** is correct whereas, **[$a <= $b]** is incorrect.

Example

Here is an example which uses all the relational operators −

```
#!/bin/sh

a=10
b=20

if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi
```

```
if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

The above script will generate the following result −

10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or equal to b

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then −

Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

The following string operators are supported by Bourne Shell. Assume variable **a** holds "abc" and variable **b** holds "efg" then −

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" the size of which is 100 bytes and has **read**, **write** and **execute** permission on −

Show Examples

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.

-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

Decisions Making

There may be a situation when you need to adopt one path out of the given two paths. The conditional statements that allow program to make correct decisions and perform the right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

- The **if...else** statement
- The **case...esac** statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement −

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

Most of the if statements check relations using relational.

if...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

The *Shell expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false* then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

Example

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will generate the following result −

```
a is not equal to b
```

if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows −

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

Upon execution, you will receive the following result −

```
a is not equal to b
```

if...elif...else...fi statement

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
```

```
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
elif [ $a -gt $b ]
then
   echo "a is greater than b"
elif [ $a -lt $b ]
then
   echo "a is less than b"
else
   echo "None of the condition met"
fi
```

Upon execution, you will receive the following result −

```
a is less than b
```

The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here −

- case...esac statement

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

Syntax

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
  pattern1)
    Statement(s) to be executed if pattern1 matches
    ;;
  pattern2)
    Statement(s) to be executed if pattern2 matches
    ;;
  pattern3)
    Statement(s) to be executed if pattern3 matches
    ;;
  *)
   Default condition to be executed
   ;;
esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command ;; indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
  "apple") echo "Apple pie is quite tasty."
  ;;
  "banana") echo "I like banana nut bread."
```

```
  ;;
  "kiwi") echo "New Zealand is famous for kiwi."
  ;;
esac
```

Upon execution, you will receive the following result −

New Zealand is famous for kiwi.

loop

A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

- The while loop
- The for loop
- The until loop
- The select loop

You will use different loops based on the situation. For example, the **while** loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, **while** and **for** loops are available in most of the other programming languages like **C**, **C++** and **PERL**, etc.

The while loop

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
   Statement(s) to be executed if command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine −

```
#!/bin/sh
```

```
a=0

while [ $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

Each time this loop executes, the variable **a** is checked to see whether it has a value that is less than 10. If the value of **a** is less than 10, this test condition has an exit status of 0. In this case, the current value of **a** is displayed and later **a** is incremented by 1.

The for loop

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses the **for** loop to span through the given list of numbers −

```
#!/bin/sh

for var in 0 1 2 3 4 5 6 7 8 9
```

```
do
   echo $var
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
6
7
8
9
```

The until loop

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

```
until command
do
   Statement(s) to be executed until command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

Example

Here is a simple example that uses the until loop to display the numbers zero to nine −

```
#!/bin/sh

a=0

until [ ! $a -lt 10 ]
do
   echo $a
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

0
1
2
3
4
5
6
7
8
9

The select loop

The **select** loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

Syntax

```
select var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and **word1** to **wordN** are sequences of characters separated by spaces (words). Each time the **for** loop executes, the value of the variable var is set to the next word in the list of words, **word1** to **wordN**.

For every selection, a set of commands will be executed within the loop. This loop was introduced in **ksh** and has been adapted into bash. It is not available in **sh**.

Example

Here is a simple example to let the user select a drink of choice −

```
#!/bin/ksh

select DRINK in tea cofee water juice appe all none
do
  case $DRINK in
    tea|cofee|water|all)
      echo "Go to canteen"
      ;;
    juice|appe)
      echo "Available at home"
    ;;
```

```
    none)
      break
    ;;
    *) echo "ERROR: Invalid selection"
    ;;
  esac
done
```

The menu presented by the select loop looks like the following −

```
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
#? juice
Available at home
#? none
$
```

You can change the prompt displayed by the select loop by altering the variable PS3 as follows −

```
$PS3 = "Please make a selection => " ; export PS3
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
Please make a selection => juice
Available at home
Please make a selection => none
$
```

Functions

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual tasks when needed.

Using functions to perform repetitive tasks is an excellent way to create **code reuse**. This is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax −

```
function_name () {
   list of commands
}
```

The name of your function is **function_name**, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example

Following example shows the use of function −

```
#!/bin/sh

# Define your function here
Hello () {
   echo "Hello World"
}

# Invoke your function
Hello
```

Upon execution, you will receive the following output −

```
$./test.sh
Hello World
```

Pass Parameters to a Function

You can define a function that will accept parameters while calling the function. These parameters would be represented by **$1**, **$2** and so on.

Following is an example where we pass two parameters *Zara* and *Ali* and then we capture and print these parameters in the function.

```
#!/bin/sh
```

```
# Define your function here
Hello () {
  echo "Hello World $1 $2"
}

# Invoke your function
Hello Zara Ali
```

Upon execution, you will receive the following result −

```
$./test.sh
Hello World Zara Ali
```

Returning Values from Functions

If you execute an **exit** command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the **return** command whose syntax is as follows −

```
return code
```

Here **code** can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example

Following function returns a value 10 −

```
#!/bin/sh

# Define your function here
Hello () {
  echo "Hello World $1 $2"
  return 10
}

# Invoke your function
Hello Zara Ali

# Capture value returnd by last command
ret=$?

echo "Return value is $ret"
```

Upon execution, you will receive the following result −

$./test.sh
Hello World Zara Ali
Return value is 10

Nested Functions

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a ***recursive function***.

Following example demonstrates nesting of two functions −

```
#!/bin/sh

# Calling one function from another
number_one () {
  echo "This is the first function speaking..."
  number_two
}

number_two () {
  echo "This is now the second function speaking..."
}

# Calling function one.
number_one
```

Upon execution, you will receive the following result −

This is the first function speaking...
This is now the second function speaking...

Function Call from Prompt

You can put definitions for commonly used functions inside your ***.profile***. These definitions will be available whenever you log in and you can use them at the command prompt.

Alternatively, you can group the definitions in a file, say ***test.sh***, and then execute the file in the current shell by typing −

$. test.sh

This has the effect of causing functions defined inside ***test.sh*** to be read and defined to the current shell as follows −

$ number_one
This is the first function speaking...
This is now the second function speaking...
$

To remove the definition of a function from the shell, use the unset command with the **.f** option. This command is also used to remove the definition of a variable to the shell.

$ unset -f function_name

Special Shell Variables and Positional Parameters

The **Special parameters** are the read-only variables that are predefined and maintained by the shell. Now let's see what are the Special parameters in the bash shell.

S.No	Special Parameters	Description
1	$#	This parameter represents the number of arguments passed to the shell script.
2	$0	This parameter represents the script name.
3	$i	This parameter represents the i^{th} argument passed to the shell script like $1, $2
4	$*	This parameter gives all arguments passed to the shell script separated by the space.
5	$!	This parameter gives PID of the last background running process.
6	$?	This parameter represents the exit status of the last command that executed. The 0 code represents success and 1 represents failure.
7	$_	This parameter gives the last argument provided to the previous command that executed.
8	$$	This parameter gives the PID of the current shell.
9	$@	This parameter holds all argument passed to the script and treat them as an array. It is similar to the $* parameter
10	$-	This parameter represents the current flags set in your shell .himBH are the flags in bash shell. Where: H – histexpand m – monitor h – hashall B – braceexpand i – interactive

Example Program: // File name: code.sh

```
echo "Number of argument passed: $#"
echo "Script name is $0"
echo "The 2nd argument passed is: $2"
echo "Arguments passed to script are: $*"
echo "Exit status of last command that executed:$?" #This is the previous command for $_
echo "Last argument provide to previous command:$_"
echo "PID of current shell is: $$"
echo "Flags are set in the shell: $-"
```

Now let's see the output of the above script:

```
┌─nishant at krishna in ~/codes/bash
└─λ ./code.sh gfg whoami 9                                                    0 (0.002s)
Number of argument passed: 3
Script name is ./code.sh
The 2th argument passed is: whoami
Arguments passed to script are: gfg whoami 9
Exit status of last command that executed:0
Last argument provide to previous command:Exit status of last command that executed:0
PID of current shell is: 14023
Flags are set in the shell: hB
```

• The variable $* contains the string of all arguments (positional parameters) on the command line.

$ cat args
print The arguments are: $*
$ args bob dave
The arguments are: bob dave

• The variable $@ is the same as $* except when enclosed in double quotes. Then, each argument contained in $@ is double quoted.

$ cat args
print The arguments are: "$@"
$ args bob dave
The arguments are: bob dave
$ args "bob dave"
The arguments are: bob dave

• **Positional parameters** refer to the individual arguments on the command line. The positional parameters available are referenced as follows:

$1 $2 $3 $4 $5 $6 $7 $8 $9

• The parameter $1 contains the first argument, $2 the second argument, and so on.

$ cat parms
print Arg 1 is: $1
print Arg 2 is: $2
print Arg 3 is: $3
print Arg 4 is: $4
print Arg 5 is: $5

$ parms Space, the final frontier
Arg 1 is: Space,
Arg 2 is: the
Arg 3 is: final
Arg 4 is: frontier
Arg 5 is:

$ parms "Space, the final frontier"
Arg 1 is: Space, the final frontier

Changing Positional Parameters

Use the set command to change positional parameters. It replaces existing positional parameters with new values.

```
$ cat newpos
print starting args are $*
print number of args is $#
print arg 1 is $1
print arg 2 is $2
set 3 4
print new args are $*
print number of args is $#
print arg 1 is $1
print arg 2 is $2
```

Output

```
$ newpos 1 2
starting args are 1 2
number of args is 2
arg 1 is 1
arg 2 is 2
new args are 3 4
number of args is 2
arg 1 is 3
arg 2 is 4
```

• What if there are more than 9 arguments?

```
$ cat ten_args
print arg 10 is $10
print arg 10 is ${10}
```

Output

```
$ ten_args a b c d e f g h i j
arg 10 is a0
arg 10 is j
```

• Use the shift command to perform a shift of the positional parameters n positions to the left. (Default n=1).

• The variables $#,$* and $@ also change with the shift command.

• Once a shift is performed, the first parameter is discarded. The $# variable is decremented, and the $* and $@ variables are updated.

Example of shift:

```
$ cat shift_it
print $#: $0 $*
shift
print $#: $0 $*
shift
print $#: $0 $*
shift
print $#: $0 $*
```

Output

```
$ shift_it 1 2 3 4 5 6 7 8 9 0 a b
12: shift_it 1 2 3 4 5 6 7 8 9 0 a b
11: shift_it 2 3 4 5 6 7 8 9 0 a b
10: shift_it 3 4 5 6 7 8 9 0 a b
9: shift_it 4 5 6 7 8 9 0 a b
```

```
$ cat ten_args
arg1=$1
shift
print $arg1 $*
```

```
$ ten_args 1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
```

Argument Validation

Example 1 division

Consider the calculator example. If the user enters 0 as the divisor, the division operation will be something over 0 (which we know from mathematics that it is an illegal operation). If no validation is place, the bash interpreter will try to execute the division operation, and … will report a division by zero error.

The solution is to validate the user's input to assure the divisor is not 0. Otherwise, a friendly error message is printed to the user.
Consider the following script that performs the required validation:

Program # file name : division.sh

```
if [ $2 -eq 0 ]; then
        echo "Illegal Division by zero Attempt!!"
        exit 1
else
        echo "$1 / $2 = $(($1/$2))"
fi
```

Output

```
elcot@boss:~/mitsos$ vi division.sh
elcot@boss:~/mitsos$ sh division.sh 12 3
12/3 = 4
elcot@boss:~/mitsos$ sh division.sh 12 0
Illegal division by zero attempt
elcot@boss:~/mitsos$ sh division.sh 0 3
0/3 = 0
elcot@boss:~/mitsos$
```

Example 2 calculate factorial of given number

#This script accepts a positive integer from user and calculate its factorial

```
if [ "$1" -eq "$1" ]
then
        if [ $1 -ge 0 ]
        then
                result=1
                for i in `seq $1`
                do
                result=$((result \* $i))
                done
                echo "factorial of $1 is $result"
        else
                echo "enter positive number"
                exit 2
        fi
else
        echo "enter an integer"
        exit 1
fi
```

Output

```
elcot@boss:~/mitsos$ vi fact.sh
elcot@boss:~/mitsos$ sh fact.sh 4
factorial of 4 is 24
elcot@boss:~/mitsos$ sh fact.sh -5
enter positive number
elcot@boss:~/mitsos$ sh fact.sh hi
fact.sh: 1: [: Illegal number: hi
enter an integer
elcot@boss:~/mitsos$
```

Validating the Correct Number of Inputs

What if your script is expecting two arguments and the user enters only one, or didn't provide any input at all?

Again, there should be a validation step.

#This script accepts a positive integer from user and calculate its factorial

```
if [ $# -ne 1 ]
then
        echo "Error: one argument expected"
        exit 3
else
        if [ "$1" -eq "$1" ]
        then
                if [ $1 -ge 0 ]
                then
                        result=1
                        for i in `seq $1`
                        do
                        result=$((result \* $i))
                        done
                        echo "factorial of $1 is $result"
                else
                        echo "enter positive number"
                        exit 2
                fi
        else
                echo "enter an integer"
                exit 1
        fi
fi
```

```
elcot@boss:~/mitsos$ vi fact.sh
elcot@boss:~/mitsos$ sh fact.sh 2 3
Error: one argument expected
elcot@boss:~/mitsos$ sh fact.sh
Error: one argument expected
elcot@boss:~/mitsos$ sh fact.sh 5
factorial of 5 is 120
elcot@boss:~/mitsos$ sh fact.sh hi
fact.sh: 6: [: Illegal number: hi
enter an integer
elcot@boss:~/mitsos$
```

Debugging Scripts

- Debugging is terminology that means the process of identifying errors and possibly resolving them in computer hardware or software.
- Just like all the other programming/coding languages debugging is crucial in bash/shell scripting in order to understand the flow of the program and resolve any errors if they occur.
  1. Using bash options (-x, -n, -v):
  2. Using set -x inside script for the specific script.

1. Using bash options (-x, -n, -v)

a) **-x:** It helps in tracing command output before executing them, when we run the script using this option we get each line of code traced before it is executed and its respective output printed in the terminal.

Example program

```
# Take input from the user
echo -n "Enter a number"
read n

# Calculate remainder using mod (%) operator
remainder=$(( $n % 2 ))

# Odd or Even Check
if [ $remainder -eq 0 ]
then
 echo "You have entered $n -- which is an Even number"
else
 echo "You have entered $n -- which is an Odd number"
fi
```

Output:

```
elcot@boss:~$ vi oddeven.sh
elcot@boss:~$ sh -x oddeven.sh
+ echo Enter a number
Enter a number
+ read n
43
+ remainder=1
+ [ 1 -eq 0 ]
+ echo you have entered 43 -- which is odd number
you have entered 43 -- which is odd number
elcot@boss:~$ sh -x oddeven.sh
+ echo Enter a number
Enter a number
+ read n
66
+ remainder=0
+ [ 0 -eq 0 ]
+ echo you have entered 66 -- which is even number
you have entered 66 -- which is even number
elcot@boss:~$
```

b) **-n:** It helps in checking the syntax errors if there are any in the shell script prior to executing it.

Example program

```
# Take input from the user
echo -n "Enter a number"
read n

# Calculate remainder using mod (%) operator
remainder=$(( $n % 2 ))

# Odd or Even Check
if [ $remainder -eq 0 ]

echo "You have entered $n -- which is an Even number”
else
 echo "You have entered $n -- which is an Odd number"
fi
```

Output

```
elcot@boss:~$ vi oddeven.sh
elcot@boss:~$ sh -n oddeven.sh
oddeven.sh: 9: oddeven.sh: Syntax error: "else" unexpected (expecting "then")
elcot@boss:~$ vi oddeven.sh
elcot@boss:~$ sh -n oddeven.sh
elcot@boss:~$ sh oddeven.sh
Enter a number
78
you have entered 78 -- which is even number
elcot@boss:~$ sh oddeven.sh
Enter a number
51
you have entered 51 -- which is odd number
elcot@boss:~$
```

When the script is executed with -n option output shows the exact line number location of the line of code where it got a syntax error. Then when the error is fixed and run it again with the same option. This shows that the script is syntax-errors-free.

c) **-v:** If there is a need to read the comments and entire script while the script is executing, then -v option helps us achieve this goal.

```
elcot@boss:~$ vi oddeven.sh
elcot@boss:~$ sh -v oddeven.sh
echo "Enter a number"
Enter a number
read n
65

remainder=$(( $n % 2 ))

if [ $remainder -eq 0 ]
then
        echo "you have entered $n -- which is even number"
else
        echo "you have entered $n -- which is odd number"
fi
you have entered 65 -- which is odd number
elcot@boss:~$
```

One slight drawback with this is there is a need to perform debug checks on the entire script, if the script is huge it is quite tough/tedious.

Using set -x inside script for the specific script.

-x : if we want to debug/trace only a certain logical part of OddEven.sh Script and not the whole script. We can **set -x** flag before the code we want to debug and then set it again back to **+x** where tracing of code needs to be disabled.

Example program

```
# Take input from the user
echo -n "Enter a number"
read n

# Enabling trace
set -x

# Calculate remainder using mod (%) operator
remainder=$(( $n % 2 ))

# Disabling trace
set +x

# Odd or Even Check
if [ $remainder -eq 0 ]
then
 echo "You have entered $n -- which is an Even number"
else
 echo "You have entered $n -- which is an Odd number"
fi
```

Output:

```
elcot@boss:~$ vi oddeven.sh
elcot@boss:~$ sh oddeven.sh
Enter a number
76
+ remainder=0
+ set +x
you have entered 76 -- which is even number
elcot@boss:~$
```

Here in this example only the logical bit of OddEven.sh Script is being debugged. Only those outputs are traced in terminal output where the set -x option is specified.