A LECTURE NOTE

ON

OPERATING SYSTEM (II)

(CSC 413)

COURSE LECTURER:

DR. SODIYA A. S.

CHAPTER ONE - UNIX OPERATING SYSTEM

1.0 INTRODUCTION TO UNIX OPERATING SYSTEM

An operating system is the suite of programs which make the computer work.

The UNIX operating system was designed to let a number of programmers access the computer at the same time and share its resources. This real-time sharing of resources makes UNIX one of the most powerful operating systems ever.

Features of UNIX operating system:

- Multitasking capability (UNIX lets a computer do several things at once,)
- Multiuser capability (The computer can take the commands of a number of users -- determined by the design of the computer -- to run programs, access files, and print documents at the same time.)
- Portability (permit to move from one brand of computer to another with a minimum of code changes.)
- Library of application software

COMPONENT OF UNIX O/S :

The UNIX operating system is made up of *three parts*; the kernel, the shell and the programs.

1. The kernel :

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file storage and communications in response to system calls.

An illustration of the way that the shell and the kernel work together, suppose a user types **`rm myfile`** (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **`rm`**, and then requests the kernel, through system calls, to execute the program **`rm`** on **myfile**. When the process **`rm myfile`** has finished running, the shell then returns the UNIX prompt to the user, indicating that it is waiting for further commands.

2. The shell :

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI) which interprets the commands the user types in and then arranges for them to be carried out.

The commands are themselves programs. When they terminate, the shell would give the user another prompt.

The adept user can customize his/her own shell, and users can use different shells on the same machine. As an illustration the shell may be customized with certain features to help the user in inputting commands, filename Completion - By typing part of the name of a command, filename or directory and pressing the [**Tab**] key, the shell will complete the rest of the name automatically. If the shell finds more than one name beginning with the letters that has been typed, it would beep, prompting the user to type a few more letters before pressing the tab key again.

3. Programs

Program consists of the tools and applications that offer additional functionality to the operating system. Typically, tools are grouped into categories for certain functions, such as word processing, business applications, or programming.

2.0 HISTORY

The history of UNIX starts back in 1969, when a small group composed of Ken Thompson, Dennis Ritchie and others started working on the "little-used PDP-7 in a corner" at Bell Lab and what was to become UNIX. For 10years,the development of UNIX proceeded at AT&T in numbered versions. The 1974 version was re-written in C, a major mile stone for the operating system's portability among different systems .The 1975 version was the first to become available outside Bell Lab. It became the basis of the first version of UNIX developed at the university of California Berkely. By 1983, Computer Research Group (CRG), UNIX System Group (USG) and a third group merge to become UNIX System Development Lab and AT&T announces UNIX System V, the first supported release. Installed base was 45,000. The goals of the group were to design an operating system to satisfy the following objectives:

- Simple and elegant
- Written in a high level language rather than assembly language
- Allow re-use of code

The group worked primarily in the high level language in developing the operating system. The first edition was released in 1971, it had an assembler for a PDP-11/20, file system, fork(), roff and ed. It was used for text processing of patent document.

In 1998, X/Open introduced the UNIX 95. In 1995, a branding programme for implementations of the Single UNIX Specification. Novell sold UnixWare business line to SCO and was Digital UNIX introduced in the same year.

In 1999, the UNIX system reaches its 30th anniversary. Linux 2.2 kernel was released. The Open Group and the IEEE commence joint development of a revision to POSIX and the Single UNIX Specification.

In 2003, The core volumes of Version 3 of the Single UNIX Specification were approved as an international standard.

In addition, while initially designed for medium-sized minicomputers, the operating system was soon moved to larger, more powerful mainframe computers. As personal computers grew in popularity, versions of UNIX found their way into these boxes, and a number of companies produce UNIX-based machines for the scientific and programming communities.

UNIX POPULARITY

Many vendors have decide to use UNIX because of the following reasons :

- UNIX is relatively easy to run. Only a very small amount of its codes are written in assembly language. UNIX is nearly the unanimous choice of operating system for computer companies started since 1985. The user benefit which results from this is that UNIX runs on a wide variety of computer systems. Many traditional vendors have made UNIX available on their systems in addition to their proprietary operating systems.

- The application program interface allows many different types of applications to be easily implemented under UNIX without writing assembly language. These applications are relatively portable across multiple vendor hardware platforms. Third party software vendors can save costs by supporting a single UNIX version of their software rather than four completely different vendor specific versions requiring four times the maintenance.
- Vendor-independent networking allows users to easily network multiple systems from many different vendors.

3.0 DESIGN ISSUES OF UNIX

UNIX is a stable, multi-user, multi-tasking system for servers, desktop and laptop. It has a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment.

Everything in UNIX is either a file or a process.

A *process* is an executing program identified by a unique PID (process identifier).

A file is a collection of data. They are created by users using text editors, running compilers etc.

All the files are grouped together in the directory structure.

3.1 The design of the unix operating system

Memory management Policies:

- Allocating swap space
- Freeing swap space
- Swapping
- Demand paging

MEMORY

Primary memory is a precious resource that frequently cannot contain all active processes in the system

The memory management system decides which processes should reside (at least partially) in main memory

It monitors the amount of available primary memory and may periodically write processes to a secondary device called the swap device to provide more space in primary memory

At a later time, the kernel reads the data from swap device back to main memory

UNIX Memory Management Policies

- Swapping
  - Easy to implement
  - Less system overhead
- Demand Paging
  - Greater flexibility
  - Swapping

The swap device is a block device in a configurable section of a disk

Kernel allocates contiguous space on the swap device without fragmentation

It maintains free space of the swap device in an in-core table, called map

The kernel treats each unit of the swap map as group of disk blocks

As kernel allocates and frees resources, it updates the map accordingly

Algorithm: Allocate Swap Space

- malloc(address_of_map, number_of_unit)
  - for (every map entry)

- if (current map entry can fit requested units)
  - if (requested units == number of units in entry)
    - » Delete entry from map
  - else
    - » Adjust start address of entry
  - return original address of entry
- return -1

Swapping Process Out

- Memory → Swap device
- Kernel swap out when it needs memory

1. When fork() called for allocate child process
2. When called for increase the size of process
3. When process become larger by growth of its stack
4. Previously swapped out process want to swap in but not enough memory

The kernel must gather the page addresses of data at primary memory to be swapped out Kernel copies the physical memory assigned to a process to the allocated space on the swap device.The mapping between physical memory and swap device is kept in page table entry Demand Paging Not all page of process resides in memory

When a process accesses a page that is not part of its working set, it incurs a page fault.

The kernel suspends the execution of the process until it reads the page into memory and makes it accessible to the process

3.3 INTERRUPT HANDLERS

[Macro]

system: with-enabled-interrupts *specs &rest body*

This macro should be called with a list of signal specifications, *specs*. Each element of *specs* should be a list of two elements: the first should be the Unix signal for which a handler should be established, the second should be a function to be called when the signal is received One or more signal handlers can be established in this way. with-enabled-interrupts establishes the correct signal handlers and then executes the forms in *body*. The forms are executed in an unwind-protect so that the state of the signal handlers will be restored to what it was before the with-enabled-interrupts was entered. A signal handler function specified as NIL will set the Unix signal handler to the default which is normally either to ignore the signal or to cause a core dump depending on the particular signal.

It is sometimes necessary to execute a piece a code that can not be interrupted. This macro the forms in *body* with interrupts disabled. Note that the Unix

interrupts are not actually disabled, rather they are queued until after *body* has finished executing.

When executing an interrupt handler, the system disables interrupts, as if the handler was wrapped in a without-interrupts. The macro with-interrupts can be used to enable interrupts while the forms in *body* are evaluated. This is useful if *body* is going to enter a break loop or do some long computation that might need to be interrupt

For some interrupts, such as SIGTSTP (suspend the Lisp process and return to the Unix shell) it is necessary to leave Hemlock and then return to it. This macro executes the forms in *body* after exiting Hemlock. When *body* has been executed, control is returned to Hemlock.

[Function]

This function establishes *function* as the handler for *signal*.

Unless you want to establish a global signal handler, you should use the macro with-enabled-interrupts to temporarily establish a signal handler. enable-interrupt returns the old function associated with the signal.

[Function]

system: ignore-interrupt *signal*

Ignore-interrupt sets the Unix signal mechanism to ignore *signal* which means that the Lisp process will never see the signal. Ignore-interrupt returns the old function associated with the signal or *nil* if none is currently defined.

Default-interrupt can be used to tell the Unix signal mechanism to perform the default action for *signal*.

3.4 UNIX PROCESS SCHEDULING

There is the need for processes on a system to occasionally request services from the kernel. Some older operating systems had a *rendezvous* style of providing these services - the process would request a service and wait at a particular point, until a kernel task came along and serviced the request on behalf of the process.

UNIX works very differently. Rather than having kernel tasks service the requests of a process, the process itself enters *kernel space*. This means that rather than the process waiting "outside" the kernel; it enters the kernel itself (i.e. the process will start executing kernel code for itself).

When a process invokes a system call, the hardware is switched to the kernel settings. At this point, the process will be executing code from the kernel image.

The Kernel in UNIX

- Controls the execution of processes by allowing their creation, termination, communication.
- Schedules processes fairly for execution on CPU
- Allocates main memory for an executing process

- Allocates secondary memory for efficient storage and retrieval of user data
- Allows controlled peripheral device access to processes

3.4.1 Basic operations on processes in UNIX

Creation of processes in UNIX

- Establish a new process
- Assign a new unique process identifier (PID) to the new process

- Allocate memory to the process for all elements of process image, including private user address space and stack; the values can possibly come from the parent process; set up any linkages, and then, allocate space for process control block

- Create a new process control block corresponding to the above PID and add it to the process table; initialize di erent values in there such as parent PID, list of children (initialized to null), program counter (set to program entry point), system stack pointer (set to de ne the process stack boundaries)

- Initial CPU state, typically initialized to Ready or Ready, suspend Add the process id of new process to the list of children of the creating (parent) process

- r0. Initial allocation of resources

- k0. Initial priority of the process