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. <u>The kernel :</u>

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

- Accounting information and limits

- Add the process to the ready list

- Initial allocation of memory and resources must be a subset of parent's and be assigned as shared Initial priority of the process can be greater than the parent's

Management of processes in UNIX

How processes are managed after creation in UNIX

1. Suspend - Change process state to suspended
    - A process may suspend only its descendants
    - May include cascaded suspension
    - Stop the process if the process is in running state and save the state of the processor in the process control block
    - If process is already in blocked state, then leave it blocked, else change its state to ready state
    - If need be, call the scheduler to schedule the processor to some other process
2. Activate - Change process state to active
    - Change one of the descendant processes to ready state
    - Add the process to the ready list
3. Destroy - Remove one or more processes
    - Cascaded destruction
    - Only descendant processes may be destroyed

- If the process to be “killed" is running, stop its execution
- Free all the resources currently allocated to the process
- Remove the process control block associated with the killed process

4. Change priority - Set a new priority for the process
    - Change the priority in the process control block
    - Move the process to a different queue to reflect the new priority

3.4.2 Scheduling in UNIX

Scheduler decides the process to run first by using a scheduling algorithm

3.4.2.1 Type of scheduling used in UNIX

Pre-emptibility

In UNIX, Processes in user space are *pre-emptible* - what this means is that a process may have the CPU taken away from it arbitrarily. This is how pre-emptive multitasking works: the scheduling routine will periodically suspend the currently executing process, and possibly schedule another task to run on that CPU. This means that theoretically, a process can be in a situation where it never gets the CPU back. In reality the scheduling code has an interest in fairness and will try to give the CPU to each process with a weak level of fairness, but there are no guarantees

Algorithms are:

- Shortest Remaining Time Scheduling
  - Preemptive version of shortest job next scheduling
  - Preemptive in nature (only at arrival time)
  - Highest priority to process that need least time to complete
  - Priority function P
  - Schedule for execution
  - Average waiting time calculations
- Round-Robin Scheduling
  - Preemptive in nature
  - Preemption based on time slices or time quanta
  - Time quantum between 10 and 100 milliseconds
  - All user processes treated to be at the same priority
  - Ready queue treated as a circular queue

Desirable features of a scheduling algorithm

1. Fairness: Make sure each process gets its fair share of the CPU
2. Efficiency: Keep the CPU busy 100% of the time
3. Response time: Minimize response time for interactive users
4. Turnaround: Minimize the time batch users must wait for output
5. Throughput: Maximize the number of jobs processed per hour

3.5 DEVICE MANAGEMENT

To perform useful functions, processes need access to the peripherals connected to the computer, which are controlled by the kernel through device drivers. For example, to show the user something on the screen, an application would make a request to the kernel, which would forward the request to its display driver, which is then responsible for actually plotting the character/pixel.

3.5.1 Special features of Device management in UNIX

Device drivers run as part of the kernel, either compiled in or as run-time loadable modules. The kernel architectures, Monolithic kernel does this and it have the advantage of speed and efficiency.

- **Device manager**

Device manager will be the interface between the device drivers and the both the rest of the kernel and user applications.

The device manager needs to do two things:

1. Isolate devices drivers from the kernel so that driver writers can worry about interfacing to the hardware and not about interfacing to the kernel
2. Isolate user applications from the hardware so that applications can work on the majority of devices the user might connect to their system

In most operating systems, the device manager is the only part of the kernel that programmers really see. Writing a good interface will make the difference between an efficient and reliable OS which works with a variety of devices and an OS which you spend all your own time writing and debugger drivers for.

Capabilities of device manager

1. Asynchronous I/O: that is, applications will be able to start an I/O operation and continue to run until it terminates.
2. Plug and Play: drivers will be able to be loaded and unloaded as devices are added to and removed from the system. Devices will be detected automatically on system startup, if possible.

- **Drivers**

Because we want our kernel to be plug-and-play capable, it isn't enough for drivers to be added to the kernel at compile time, as Minix and old Linux do. We must be able to load and unload them at run time. This isn't difficult: it just means we have to extend the executable file interface to kernel mode.

- **Interfaces**

Once we've detected the devices installed in the system we need to keep a record of them somewhere. The standard Unix model, employed by Minix and Linux, is to keep directory somewhere in the file system. This directory is filled with special directory entries, directory entries which don't point to any data, each of which refers to a specific device via major and minor device numbers. The major device number specifies the device type or driver to use and the minor number specifies a particular device implemented by that drivers.

3.6 Security

An important kernel design decision is the choice of the abstraction levels where the security mechanisms and policies should be implemented. Kernel security mechanisms play a critical role in supporting security at higher levels.

One approach is to use firmware and kernel support for fault tolerance (see above), and build the security policy for malicious behavior on top of that (adding features such as cryptography mechanisms where necessary), delegating some responsibility to the compiler. Approaches that delegate enforcement of security

policy to the compiler and/or the application level are often called *language-based security*.

The lack of many critical security mechanisms in current mainstream operating systems impedes the implementation of adequate security policies at the application abstraction level. In fact, a common misconception in computer security is that any security policy can be implemented in an application regardless of kernel support.

4.0 ADVANTAGES OF UNIX O/S

- Unix is more flexible and can be installed on many different types of machines, including main-frame computers, supercomputers and micro-computers.

- Unix is more stable and does not go down as often as Windows does, therefore requires less administration and maintenance.

- Unix has greater built-in security and permissions features than Windows.

- Unix possesses much greater processing power than Windows.

- Unix is the leader in serving the Web. About 90% of the Internet relies on Unix operating systems running Apache, the world's most widely used Web server.

- Software upgrades from Microsoft often require the user to purchase new or more hardware or prerequisite software. That is not the case with Unix.

- The mostly free or inexpensive open-source operating systems, such as Linux and BSD, with their flexibility and control, are very attractive to (aspiring) computer

wizards. Many of the smartest programmers are developing state-of-the-art software free of charge for the fast growing "open-source movement”.

- Unix also inspires novel approaches to software design, such as solving problems by interconnecting simpler tools instead of creating large monolithic application programs.

CHAPTER TWO - LINUX

Introduction

What is Linux?
Linux is a UNIX-like operating system that runs on many different computers. Linux was first released in 1991 by its author Linus Torvalds at the University of Helsinki and developed by Linus Torvalds (author) and Andrew Morton. Linux is the operating system *kernel*, which comes with a *distribution* of software The Linux kernel is an operating system kernel used by a family of Unix-like operating system. It started out as a personal computer system used by individuals, and has since gained the support of several large operations such as HP, IBM, and Sun microsystem. It now used mostly as the server operating system. It's a prime example of open source development system. It's written in C

Since then it has grown tremendously in popularity as programmers around the world embraced his project of building a free operating system, adding features, and fixing problems. Linux is portable, which means you'll find versions running on name-brand or clone PCs, Apple Macintoshes, Sun workstations, or Digital Equipment Corporation Alpha-based computers. Linux also comes with source code, so you can change or customize the software to adapt to your needs. Finally, Linux is a great operating system, rich in features adopted from other versions of UNIX. The term Linux distribution is used to refer to the various operating systems that run on top of the Linux kernel. Linux is one of the most prominent examples of free/open source software. Today, the Linux kernel has received contributions from thousands of programmers.

Event Leading To the Creation

The UNIX operating system was conceived and implemented in 1960 and first released in 1970. Its portability and availability caused it to the widely adopted and modified by academic institutions and businesses. In 1983, Richard Stallman started the GNU project with the goal of creating a free UNIX like operating system. As part of the work, he wrote the GNU general public license (GPL). By the early 1990's there was almost enough available software to create a full operating system. However, the GNU kernel called HURD, failed to attract attention from developers leaving GNU incomplete. A solution seemed to appear in form of MINIX. It was released by Andrew S Tanenbaum in 1987, as an operating system, MINIX was not a superb one while source code was available, modification and retribution was restricted. This factors and lack of widely adopted free kernel made Torvalds start is project.

Processes

The concept of a *process* is fundamental to any multiprogramming operating system. A process is usually defined as an instance of a program in execution; thus, if 16 users are running vi at once, there are 16 separate processes (although they can share the same executable code). Processes are often called "tasks" in Linux source code.

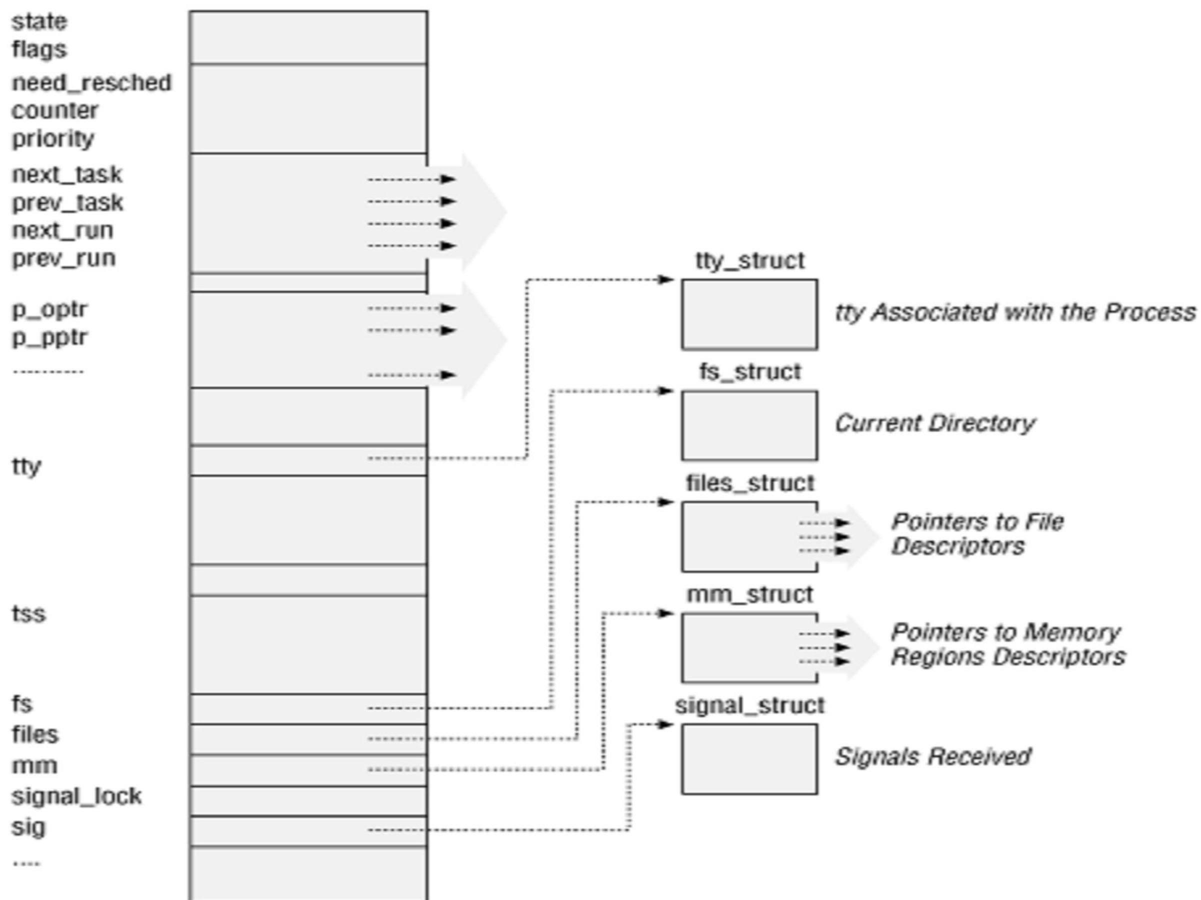
Properties of processes

- Static
- Dynamic

Process Descriptor

In order to manage processes, the kernel must have a clear picture of what each process is doing. It must know, for instance, the process's priority, whether it is running on the CPU or blocked on some event, what address space has been assigned to it, which files it is allowed to address, and so on. This is the role of the *process descriptor,* that is, of a task_struct type structure whose fields contain all the information related to a single process. As the repository of so much information, the process descriptor is rather complex. Not only does it contain many fields itself, but some contain pointers to other data structures that, in turn, contain pointers to other structures. The figure below describes the Linux process descriptor schematically.

Figure 1 The Linux Process Descriptor



The five data structures on the right side of the figure refer to specific resources owned by the process. These resources will be covered in future chapters. This chapter will focus on two types of fields that refer to the process state and to process parent/child relationships.

<u>Process State</u>

As its name implies, the 'state' field of the process descriptor describes what is currently happening to the process. It consists of an array of flags, each of which describes a possible process state. In the current Linux version these states are mutually exclusive, and hence exactly one flag of state is set; the remaining flags are cleared. The following are thepossible process states:

TASK_RUNNING

The process is either executing on the CPU or waiting to be executed.

TASK_INTERRUPTIBLE

The process is suspended (sleeping) until some condition becomes true. Raising a hardware interrupt, releasing a system resource the process is waiting for, or delivering a signal are examples of conditions that might wake up the process, that is, put its state back to TASK_RUNNING.

TASK_UNINTERRUPTIBLE

Like the previous state, except that delivering a signal to the sleeping process leaves its state unchanged. This process state is seldom used. It is valuable, however, under certain specific conditions in which a process must wait until a given event occurs without being interrupted. For instance, this state may be used when a process opens a device file and the corresponding device driver starts probing for a corresponding hardware device. The device driver must not be interrupted until the probing is complete, or the hardware device could be left in an unpredictable state.

TASK_STOPPED

Process execution has been stopped: the process enters this state after receiving a SIGSTOP, SIGTSTP, SIGTTIN, or SIGTTOU signal. When a process is being monitored by another (such as when a debugger executes a ptrace( ) system call to monitor a test program), any signal may put the process in the TASK_STOPPED state.

TASK_ZOMBIE

Process execution is terminated, but the parent process has not yet issued a wait( )-like system call (wait2( ), wait3( ), wait4( ), or waitpid( )) to return information about the dead process. Before the wait( )-like call is issued, the kernel cannot discard the data contained in the dead process descriptor because the parent could need it

Identifying A Process

Any Unix-like operating system, on the other hand, allows users to identify processes by means of a number called the *Process ID* (or *PID*). The PID is a 32-bit unsigned integer stored in the PID field of the process descriptor. PIDs are

numbered sequentially: the PID of a newly created process is normally the PID of the previously created process incremented by one. However, for compatibility with traditional Unix systems developed for 16-bit hardware platforms, the maximum PID number allowed on Linux is 32767. When the kernel creates the 32768th process in the system, it must start recycling the lower unused PIDs.

Memory Management

The memory management subsystem is one of the most important parts of the operating system. Since the early days of computing, there has been a need for more memory than exists physically in a system. Strategies have been developed to overcome this limitation and the most successful of these is virtual memory. Virtual memory makes the system appear to have more memory than is physically present by sharing it among competing processes as they need it. Virtual memory does more than just make your computer's memory go farther. The memory management subsystem provides:

Large Address Spaces

The operating system makes the system appear as if it has a larger amount of memory than it actually has. The virtual memory can be many times larger than the physical memory in the system.

Protection

Each process in the system has its own virtual address space. These virtual address spaces are completely separate from each other and so a process running one application cannot affect another. Also, the hardware virtual memory mechanisms allow areas of memory to be protected against writing. This protects code and data from being overwritten by rogue applications.

Memory Mapping

Memory mapping is used to map image and data files into a process' address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process.

Fair Physical Memory Allocation

The memory management subsystem allows each running process in the system a fair share of the physical memory of the system.

Shared Virtual Memory

Although virtual memory allows processes to have separate (virtual) address spaces, there are times when you need processes to share memory. For example there could be several processes in the system running the bash command shell. Rather than have several copies of bash, one in each process's virtual address space, it is better to have only one copy in physical memory and all of the processes running bash share it. Dynamic libraries are another common example of executing code shared between several processes.

Shared memory can also be used as an Inter Process Communication (IPC) mechanism, with two or more processes exchanging information via memory common to all of them. Linux supports the Unix System V shared memory IPC.

3.1 An Abstract Model of Virtual Memory

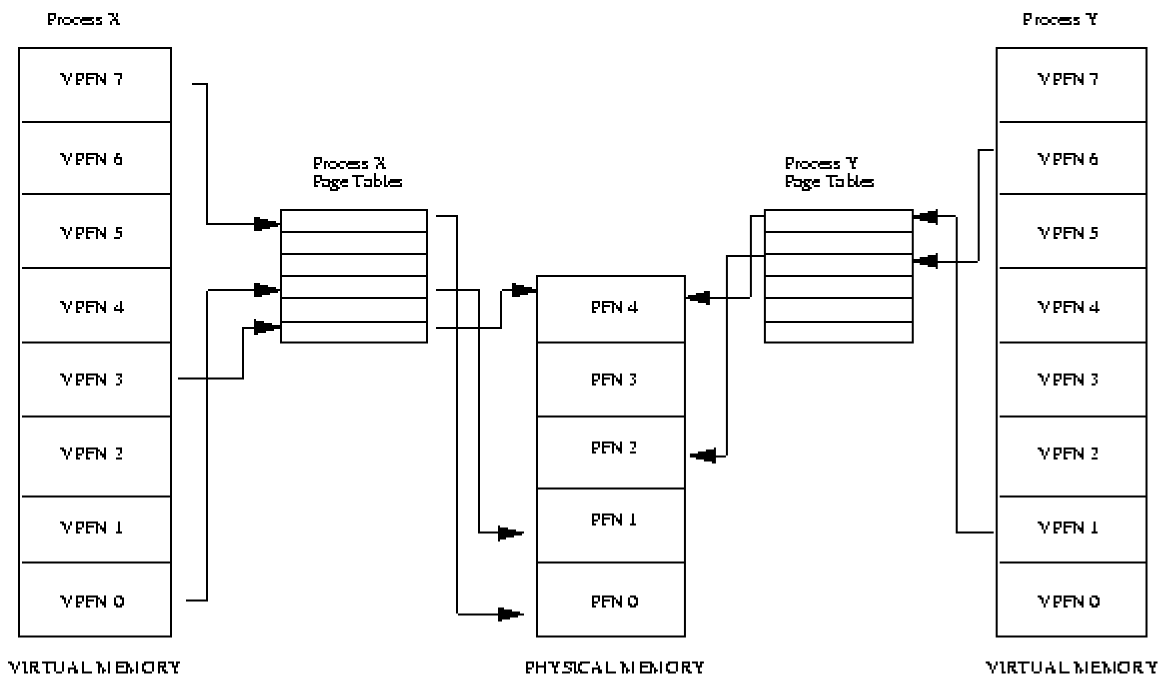


Figure 3.1: Abstract model of Virtual to Physical address mapping

Before considering the methods that Linux uses to support virtual memory it is useful to consider an abstract model that is not cluttered by too much detail.

As the processor executes a program it reads an instruction from memory and decodes it. In decoding the instruction it may need to fetch or store the contents of a location in memory. The processor then executes the instruction and moves onto the next instruction in the program. In this way the processor is always accessing memory either to fetch instructions or to fetch and store data.

In a virtual memory system all of these addresses are virtual addresses and not physical addresses. These virtual addresses are converted into physical addresses by the processor based on information held in a set of tables maintained by the operating system.

To make this translation easier, virtual and physical memory are divided into handy sized chunks called *pages*. These pages are all the same size, they need not be but if they were not, the system would be very hard to administer. Linux on Alpha AXP systems uses 8 Kbyte pages and on Intel x86 systems it uses 4 Kbyte pages. Each of these pages is given a unique number; the page frame number (PFN).

In this paged model, a virtual address is composed of two parts; an offset and a virtual page frame number. If the page size is 4 Kbytes, bits 11:0 of the virtual address contain the offset and bits 12 and above are the virtual page frame number. Each time the processor encounters a virtual address it must extract the offset and the virtual page frame number. The processor must translate the virtual page frame number into a physical one and then access the location at the correct offset into that physical page. To do this the processor uses *page tables*.

Interrupts And Exceptions

An *interrupt* is usually defined as an event that alters the sequence of instructions executed by a processor. Such events correspond to electrical signals generated by hardware circuits both inside and outside of the CPU chip.
Interrupts are often divided into *synchronous* and *asynchronous* interrupts:

• *Synchronous* interrupts are produced by the CPU control unit while executing instructions and are called synchronous because the control unit issues them only after terminating the execution of an instruction.

• *Asynchronous* interrupts are generated by other hardware devices at arbitrary times with respect to the CPU clock signals. Intel 80x86 microprocessor manuals designate synchronous and asynchronous interrupts as *exceptions* and *interrupts*, respectively. We'll adopt this classification, although we'll
occasionally use the term "interrupt signal" to designate both types together (synchronous as well as asynchronous). Interrupts are issued by interval timers and I/O devices; for instance, the arrival of a keystroke from a user sets off an interrupt. Exceptions, on the other hand, are caused either by programming errors or by anomalous conditions that must be handled by the kernel. In the first case, the kernel handles the exception by delivering to the current process one of the signals familiar to every Unix programmer. In the second case, the kernel performs all the steps needed to recover from the anomalous condition, such as a page fault or a request (via an int instruction) for a kernel service.

The Role of Interrupt Signals

As the name suggests, interrupt signals provide a way to divert the processor to code outside the normal flow of control. When an interrupt signal arrives, the CPU must stop what it's currently doing and switch to a new activity; it does this by saving the current value of the program counter (i.e., the content of the eip and cs registers) in the Kernel Mode stack and by placing an address related to the interrupt type into the program counter. There is a key difference between interrupt handling and process switching: the code executed by an interrupt or by an exception handler is not a process. Rather, it is a kernel control path that runs on behalf of the same process that was running when the interrupt occurred. As a kernel control path, the interrupt handler is lighter than a process (it has less context and requires less time to set up or tear down).

Interrupt handling is one of the most sensitive tasks performed by the kernel, since it must satisfy the following constraints:

• Interrupts can come at any time, when the kernel may want to finish something else it was trying to do. The kernel's goal is therefore to get the interrupt out of the way as soon as possible and defer as much processing as it can. For instance, suppose a block of data has arrived on a network line. When the hardware interrupts the kernel, it could simply mark the presence of data, give the processor back to whatever was running before, and do the rest of the processing later (like moving the data into a buffer where its recipient process can find it and restarting the process). The activities that the kernel needs to perform in response to an interrupt are thus divided into two parts: a *top half* that the kernel executes right

away and a *bottom half* that is left for later. The kernel keeps a queue pointing to all the functions that represent bottom halves waiting to be executed and pulls them off the queue to execute them at particular points in processing.

• Since interrupts can come at any time, the kernel might be handling one of them while another one (of a different type) occurs. This should be allowed as much as possible since it keeps the I/O devices busy. As a result, the interrupt handlers must be coded so that the corresponding kernel control paths can be executed in a nested manner. When the last kernel control path terminates, the kernel must be able to resume execution of the interrupted process or switch to another process if the interrupt signal has caused a rescheduling activity.

• Although the kernel may accept a new interrupt signal while handling a previous one, some critical regions exist inside the kernel code where interrupts must be disabled. Such critical regions must be limited as much as possible since, according to the previous requirement, the kernel, and in particular the interrupt handlers, should run most of the time with the interrupts enabled.

Interrupts and Exceptions

The Intel documentation classifies interrupts and exceptions as follows:
• Interrupts:

Maskable interrupts

Sent to the INTR pin of the microprocessor. They can be disabled by clearing the IF flag of the eflags register. All IRQs issued by I/O devices give rise to maskable interrupts.

Nonmaskable interrupts

Sent to the NMI (Nonmaskable Interrupts) pin of the microprocessor. They are not disabled by clearing the IF flag. Only a few critical events, such as hardware failures,
give rise to nonmaskable interrupts.
• Exceptions:

Processor-detected exceptions

Generated when the CPU detects an anomalous condition while executing an instruction. These are further divided into three groups, depending on the value of the eip register that is saved on the Kernel Mode stack when the CPU control unit raises the exception:

Faults

The saved value of eip is the address of the instruction that caused the fault, and hence that instruction can be resumed when the exception handler terminates. Resuming the same instruction is necessary whenever the handler is able to correct the anomalous condition that caused the exception.

Traps

The saved value of eip is the address of the instruction that should be executed after the one that caused the trap. A trap is triggered only when there is no need to re-execute the instruction that was terminated. The main use of traps is for debugging purposes: the role of the interrupt signal in this case is to notify the debugger that a specific instruction has been executed (for instance, a breakpoint has been reached within a program). Once the user has examined the data provided by the debugger, she may ask that execution of the debugged program resume starting from the next instruction.

Aborts

A serious error occurred; the control unit is in trouble, and it may be unable to store a meaningful value in the eip register. Aborts are caused by hardware failures or by invalid values in system tables. The interrupt signal sent by the control unit is an emergency signal used to switch control to the corresponding abort exception handler. This handler has no choice but to force the affected process to terminate.

Programmed exceptions

Occur at the request of the programmer. They are triggered by int or int3 instructions; the 'into' (check for overflow) and 'bound' (check on address bound) instructions also give rise to a programmed exception when the condition they are checking is not true. Programmed exceptions are handled by the control unit as traps; they are often called *software interrupts*. Such exceptions have two common uses: to implement system calls, and to notify a debugger of a specific event.
Linux uses two types of descriptors:

Interrupt gates & trap gates.
Trap gate: Trap gates are used for activating exception handlers.

Interrupt gate: Cannot be accessed by user mode progs

<u>The Linux Booting Process</u>

In most cases, the Linux kernel is loaded from a hard disk, and a two-stage boot loader is required. The most commonly used Linux boot loader on Intel systems is named LILO (Linux Loader); corresponding programs exist for other architectures. LILO may be installed either on the MBR, replacing the small program that loads the boot sector of the active partition, or in the boot sector of a (usually active) disk partition. In both cases, the final result is the same: when the loader is executed at boot time, the user may choose which operating system to load. The LILO boot loader is broken into two parts, since otherwise it would be too large to fit into the MBR. The MBR or the partition boot sector includes a small boot loader, which is loaded into RAM starting from address 0x00007c00 by the BIOS. This small program moves itself to the address 0x0009a000, sets up the Real Mode stack (ranging from 0x0009b000 to 0x0009a200), and loads the second part of the LILO boot loader into RAM starting from address 0x0009b000. In turn, this latter program reads a map of available operating systems from disk and offers the user a prompt so she can choose one of them. Finally, after the user has chosen the kernel to be loaded (or let a time-out elapse so that LILO chooses a default), the boot loader may either copy the boot sector of the corresponding partition into RAM and execute it or directly copy the kernel image into RAM. Assuming that a Linux kernel image must be booted, the LILO boot loader, which relies on BIOS routines, performs essentially the same operations as the boot loader integrated into the kernel image described in the previous section about floppy disks. The loader displays the "Loading Linux" message; then it copies the integrated boot loader of the kernel image to address 0x00090000, the setup( ) code to address 0x00090200, and the rest of the kernel image to address 0x00010000 or 0x00100000. Then it jumps to the setup( ) code.

<u>The setup( ) functions</u>
1. Invokes a BIOS procedure to find out the amount of RAM available in the system.

2. Sets the keyboard repeat delay and rate. (When the user keeps a key pressed past a certain amount of time, the keyboard device sends the corresponding keycode over and over to the CPU.)

3. Initializes the video adapter card.

4. Reinitializes the disk controller and determines the hard disk parameters.

5. Checks for an IBM Micro Channel bus (MCA).

6. Checks for a PS/2 pointing device (bus mouse).

7. Checks for Advanced Power Management (APM) BIOS support.

8. If the kernel image was loaded low in RAM (at physical address 0x00010000), moves it to physical address 0x00001000. Conversely, if the kernel image was loaded high in RAM, does not move it. This step is necessary because, in order to be able to store the kernel image on a floppy disk and to save time while booting, the kernel image stored on disk is compressed, and the decompression routine needs some free space to use as a temporary buffer following the kernel image in RAM.

9. Sets up a provisional Interrupt Descriptor Table (IDT) and a provisional Global Descriptor Table (GDT).

10. Resets the floating point unit (FPU), if any.

11. Reprograms the Programmable Interrupt Controller (PIC) and maps the 16 hardware interrupts (IRQ lines) to the range of vectors from 32 to 47. The kernel must perform this step because the BIOS erroneously maps the hardware interrupts in the range from to 15, which is already used for CPU exceptions (see Section 4.2.3 in Chapter 4).

12. Switches the CPU from Real Mode to Protected Mode by setting the PE bit in the cr0 status register. The provisional kernel page tables contained in swapper_pg_dir and pg0 identically map the linear addresses to the same physical addresses. Therefore, the transition from Real Mode to Protected Mode goes smoothly.

13. Jumps to the startup_32( ) assembly language function.

The startup_32( ) Functions

There are two different startup_32( ) functions; the one we refer to here is coded in the *arch/i386/boot/compressed/head.S* file. After setup( ) terminates, the function has been moved either to physical address 0x00100000 or to physical address 0x00001000, depending on whether the kernel image was loaded high or low in RAM.

This function performs the following operations:

1. Initializes the segmentation registers and a provisional stack.

2. Fills the area of uninitialized data of the kernel identified by the _edata and _end symbols with zeros.

3. Invokes the decompress_kernel( ) function to decompress the kernel image. The "Uncompressing Linux . . . " message is displayed first. After the kernel image has been decompressed, the "O K, booting the kernel." message is shown. If the kernel image was loaded low, the decompressed kernel is placed at physical address 0x00100000. Otherwise, if the kernel image was loaded high, the decompressed kernel is placed in a temporary buffer located after the compressed image. The decompressed image is then moved into its final position, which starts at physical address 0x00100000.

4. Jumps to physical address 0x00100000. The decompressed kernel image begins with another startup_32( ) function included in the *arch/i386/kernel/head.S* file. Using the same name for both the functions does not create any problems (besides confusing our readers), since both functions are executed by jumping to their initial physical addresses.

The second startup_32( ) function essentially sets up the execution environment for the first Linux process (process 0). The function performs the following operations:

1. Initializes the segmentation registers with their final values.

2. Sets up the Kernel Mode stack for process.

3. Invokes setup_idt( ) to fill the IDT with null interrupt handlers.

4. Puts the system parameters obtained from the BIOS and the parameters passed to the operating system into the first page frame.

5. Identifies the model of the processor.

6. Loads the gdtr and idtr registers with the addresses of the GDT and IDT tables.

7. Jumps to the start_kernel( ) function.

A.5 Modern Age: The start_kernel( ) Function

The start_kernel( ) function completes the initialization of the Linux kernel. Nearly every kernel component is initialized by this function; we mention just a few of them:

- The page tables are initialized by invoking the paging_init( ) function.
- The page descriptors are initialized by the mem_init( ) function
- The final initialization of the IDT is performed by invoking trap_init( ) and init_IRQ( ).
- The slab allocator is initialized by the kmem_cache_init( ) and kmem_cache_sizes_init( ) functions.
- The system date and time are initialized by the time_init( ) function (see
- The kernel thread for process 1 is created by invoking the kernel_thread( ) function. In turn, this kernel thread creates the other kernel threads and executes the */sbin/init* program.

Device Management(Managing I/O Devices)

The aim of this section is to illustrate the overall organization of device drivers in Linux.

I/O ARCHITECTURE

In order to make a computer work properly, data paths must be provided that let information flow between CPU(s), RAM, and the score of I/O devices that can be connected nowadays to a personal computer. These data paths, which are denoted collectively as the *bus*, act as the primary communication channel inside the

computer. Several types of buses, such as the ISA, EISA, PCI, and MCA, are currently in use. In this section we'll discuss the functional characteristics common to all PC architectures, without giving details about a specific bus type.
In fact, what is commonly denoted as bus consists of three specialized buses:

Data bus

A group of lines that transfers data in parallel. The Pentium has a 64-bit-wide data bus.

Address bus

A group of lines that transmits an address in parallel. The Pentium has a 32-bit-wide address bus.

Control bus

A group of lines that transmits control information to the connected circuits. The Pentium makes use of control lines to specify, for instance, whether the bus is used to allow data transfers between a processor and the RAM or alternatively between a processor and an I/O device. Control lines also determine whether a read or a write transfer must be performed. When the bus connects the CPU to an I/O device, it is called an *I/O bus*. In this case, Intel 80x86 microprocessors use 16 out of the 32 address lines to address I/O devices and 8, 16, or 32 out of the 64 data lines to transfer data. The I/O bus, in turn, is connected to each I/O Understanding the Linux Kernel 344 device by means of a hierarchy of hardware components including up to three elements: I/O ports, interfaces, and device controllers. architecture.