Chapter 1

Introduction to Systems Programming

Chapter Objectives

To understand introductory concepts used over the duration of the course.

Objectives

For this chapter, the following are the objectives:
- Introduction to Systems Programming.
- Understanding Implementation and Standards in Linux.
- Understanding Architecture and Design of Linux Kernel.

Slide #1-1

Notes

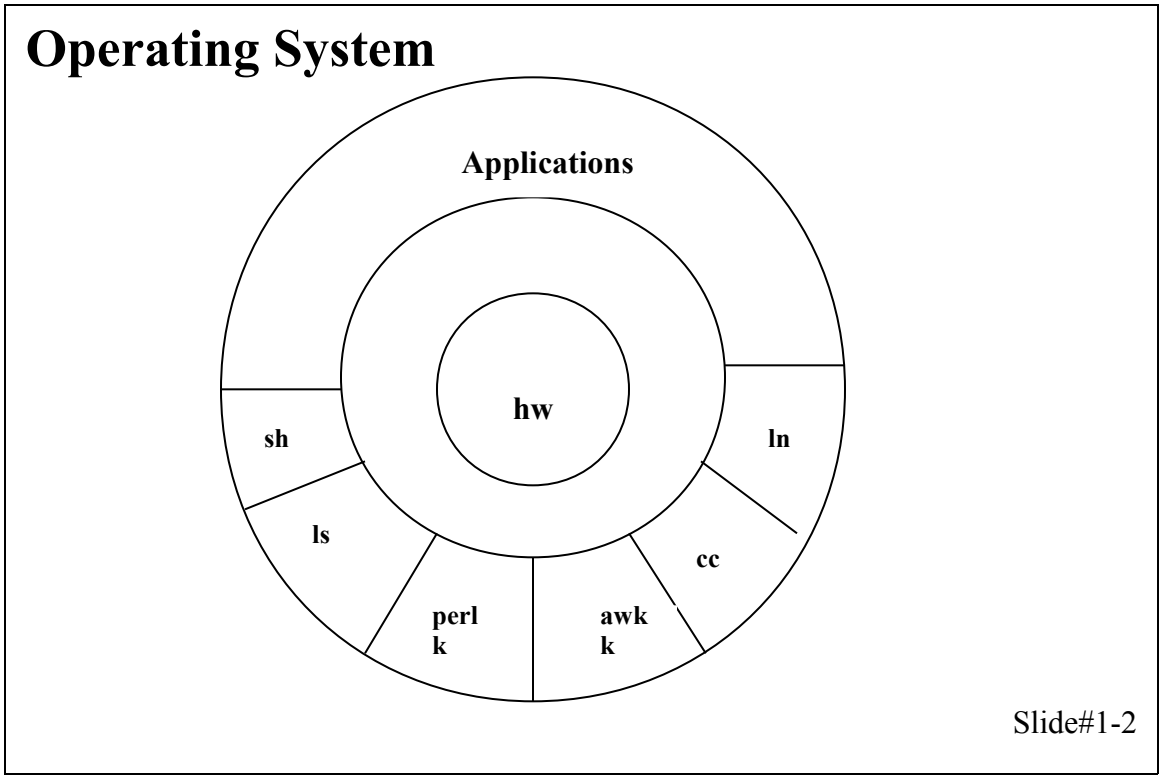
In this chapter, we examine the architectural model of the Linux operating system by examining the physical, memory, process and logical structures involved in running a simple 'C' program.
The objective of this chapter is to provide an understanding of the concepts that are used over the duration of the course.

Chapter Organization

1. **Objective**: Introduction to
   - Linux Systems Programming
   - POSIX and portability.
2. **Description**: The kernel has to track the physical and logical structures in memory for every 'C' program. This chapter provides an introduction to the concepts used in course.
3. **Concepts Covered in Chapter**:
   - UNIX and UNIX Flavors,
   - POSIX and Standardization
   - LINUX Notes
   - UNIX Systems Programming & Conventions
4. **Prior Knowledge**:
   'C' Programming Experience pre-requisite – Data Types, structs and pointers.
   UNIX/LINUX Kernel uses C, and, not C++
5. **Teaching & Learning Strategy**:
   Discussion questions are,
   - Is LINUX UNIX ? … Different Flavors of UNIX and,
   - Attempts at standardization

   - What is the difference between a process and a program?,
   - What hardware/system resources does a process need/access?,
   - What does the kernel do? And,
   - How does the kernel get involved?
6. **Teaching Format**:
   Theory + Homework Assignments
7. **Study Time**: 150 Minutes (Lecture & Theory) +
   ~45 minutes (Homework Assignments)
8. **Chapter References**:
   Stevens; APUE: Ch #1, #2
   Vahalia; UI: Ch #1, #2.1-2.4
   Robbins & Robbins; USP: Ch #1.

Introduction to Systems Programming



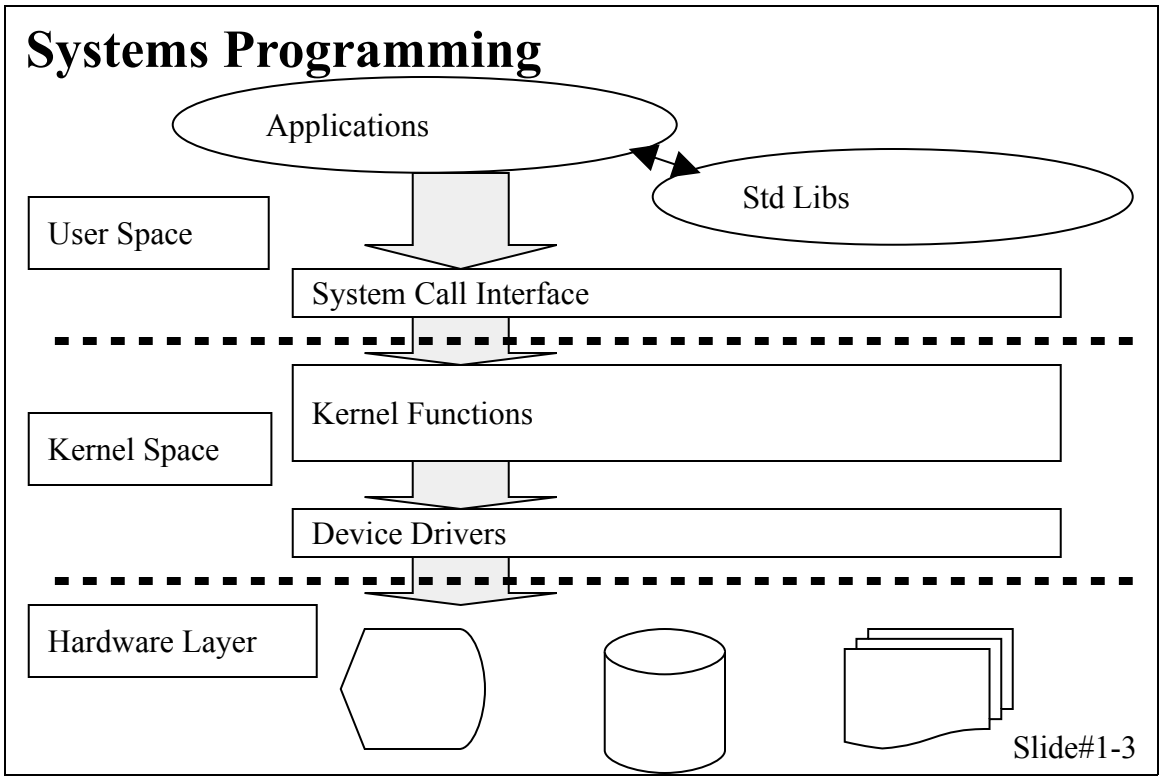
Notes:

1. The architectural model of the Linux operating system includes a number of primary components, which are discussed further in this Chapter.

2. System or Server: Hardware, and, the Operating System.
   2.1. Hardware: consists of CPU, Memory, and, devices .. disks, networks etc.
   2.2. Operating System: : Runs hardware - also, known as OS.
      2.2.1. consists of kernel and, applications .. that interact with the Kernel.

3. Linux Kernel is a resource manager.
   3.1. Several resources e.g. CPU-Time, Memory and Disk, available on a system.
   3.2. Not all of them are used, by a running program. (Some programs are CPU-intensive, some are memory-intensive, yet others are IO intensive.)
   3.3. kernel regulates access to physical resources amongst users on a system.
      3.3.1. The Shell provides an interface into the Linux kernel.
      3.3.2. Various Linux shell interfaces available: .. sh, csh, ksh, bash etc.
      3.3.3. The Linux shell interface is a program, running on the system.

In Ritchie's Words, "*UNIX is simple and coherent, but it takes a genius (or at any rate, a programmer) to understand and appreciate its simplicity*". (Vahalia, #1.3.2)



Notes

1. What is Systems Programming?
   (adapted from http://www.ee.ic.ac.uk/docs/software/unix/programming/sys/sysprog.html)
   1.1. Historically, a system programmer created general purpose utilities (e.g. editors, assemblers) and system libraries containing useful routines.
      1.1.1. Often it involved building new services into OS (e.g. new device drivers).
      1.1.2. The activity often required specialized knowledge of the underlying operating system and employed services not normally used by the applications programmer.
   1.2. These days a systems programmer probably creates a program or library visible to all users, however it does not have all features needed by all programmers.
      1.2.1. A systems programmer might create a utility which needs to make efficient use of the available resources, i.e. multiple process creation, synchronization of programs etc..
      1.2.2. A systems programmer can use libraries of subroutines (described in section 2 and 3 of the on-line manual) used in system or application programs, refer man 2 intro
      1.2.3. Additionally, there is a reliable description of the functions and the system level data-structures and global constants available, can be found in the "world-readable" header files (usually in the directories /usr/include and /usr/include/sys).

Linux Implementation and Standards

UNIX

1) Different flavors of UNIX .. Solaris, HP-UX, IBM AIX ..
2) UNIX is "technically" a specification
   .. Single UNIX Specification
3) Linux is termed "UNIX-like".

Slide#1.4

Notes:

1. What is UNIX?
   (sourced from http://www.unix.org/what_is_unix/)
   1.1. UNIX is now a single stable specification to be used to develop portable applications that run on systems conforming to the Single UNIX Specification.
      1.1.1. The UNIX specification has been separated from its licensed source-code product.
      1.1.2. UNIX is now no longer just the operating system product from AT&T (later, Novell), documented by the System V Interface Definition (SVID), controlled and licensed from a single point.
      1.1.3. Neither is it a collection of slightly different products from different vendors, each extended in slightly different ways.
   1.2. With the Single UNIX Specification, there is now a single, open, consensus specification that defines a product.
      1.2.1. There is also the UNIX mark, or brand, that is used to identify those products that conform to the Single UNIX specification.
      1.2.2. Both the specification and the trade mark are now managed and held in trust for the industry by X/Open Company.
   1.3. Historical List of Platform Vendors Supporting Single UNIX Specification
      Acer; Amdahl; Apple; AT&T GIS; Bull; Convex; Cray; Data General; Compaq; Encore; 88 Open; Fuji Xerox; Fujitsu Ossi; Hal; Hewlett-Packard; Hitachi; IBM; ICL; Matsushita; Mips ABI; Mitsubishi; Motorola; NEC; Novell/USL; Oki; Olivetti; OSF; PowerOpen; Precision RISC; Pyramid; SCO; Sequent; Sequoia; Sharp; Siemens-

Nixdorf; Silicon Graphics; Sony; Sparc International; Stratus; Sun Microsystems; Tadpole; Tandem; Thompson/Cetia; Toshiba; Unisys; Wang Labs.

1.4. Different Flavors of UNIX (Historical Note)
Many different variants and Flavors .. a decent list can be found at
http://www.ugu.com/sui/ugu/show?ugu.flavors

2. What is LINUX?

(sourced from http://www.linuxpowered.com/LDP/FAQ/Linux-FAQ/general.html)

2.1. In a general sense, Linux is a clone of the operating system Unix, written from scratch by Linus Torvalds with assistance from a loosely-knit team of developers across the Net. It aims towards POSIX and Single UNIX Specification compliance.

2.2. Linux has all the features you would expect in a modern fully-fledged Unix, including true multitasking, virtual memory, shared libraries, demand loading, shared copy-on-write executables, proper memory management, and TCP/IP networking.

2.3. Linux was first developed for 32-bit x86-based PCs (386 or higher). These days it also runs on (at least) the Compaq Alpha AXP, Sun SPARC and UltraSPARC, Motorola 68000, PowerPC, PowerPC64, ARM, Hitachi SuperH, IBM S/390, MIPS, HP PA-RISC, Intel IA-64, DEC VAX, AMD x86-64 and CRIS architectures.

2.4. Linux is easily portable to most general-purpose 32- or 64-bit architectures as long as they have a paged memory management unit (PMMU) and a port of the GNU C compiler (gcc).

2.4.1. Linux has also been ported to a number of architectures without a PMMU, although functionality is then obviously somewhat limited.

2.4.2. See the uClinux project for more info.

3. More on .. What is LINUX? [2]
(sourced from http://www.tldp.org/LDP/sag/html/sag.html)

3.1. The name "Linux" is used to refer to the operating system kernel, system software, and application software, collectively, as "Linux", and that convention is used here .. in this document.

3.1.1. At the lowest level, every Linux system is based on the Linux kernel.

3.1.2. The Linux kernel has all of the features of a modern operating system: true multitasking, threads, virtual memory, shared libraries, demand loading, shared, copy-on-write executables, proper memory management, loadable device driver modules, video frame buffering, and TCP/IP networking.

3.2. Most often, the name "Linux" is used to refer to the Linux Operating System.

3.2.1. An OS includes the kernel, but also adds various utility programs and shells.

3.2.2. Some people honor the request of Richard Stallman and the GNU Project, and call the Linux OS GNU/Linux, because a good number of these utility programs were written by the GNU folks.

3.3. Finally, software companies (and sometimes volunteer groups) add on lots of extra software, like the XFree86 X Window System, Gnome, KDE, games and many other applications. These software compilations which are based on the Linux OS are called Linux distributions.

4. Is Linux Unix?
   (adapted from http://www.linuxpowered.com/LDP/FAQ/Linux-FAQ/general.html)

   4.1. Officially an operating system is not allowed to be called a Unix until it passes the Open Group's certification tests, and supports the necessary API's.

      4.1.1. Certification involves large fees, so it is not allowed to be called Unix.

      4.1.2. Certification really doesn't mean very much anyway.

      4.1.3. Very few of commercial operating systems pass the Open Group tests.

   4.2. Unofficially, Linux is very similar to the UNIX operating system, and for many purposes is equivalent.

      4.2.1. Linux the kernel, is an operating system kernel that behaves and performs similarly to the UNIX operating system from AT&T Bell Labs.

      4.2.2. Linux is often called a "Unix-like" operating system.
      For more information, see http://www.unix-systems.org/what_is_unix.html.

POSIX and UNIX Standardization

1. Many different variants/flavors of UNIX, since late-1980's.
2. Standardization .. initiated by large-user communities/orgs,
   provides compatibility of
   - behavior, and
   - call signature
3. POSIX
4. Standards: Strict Conformance vs General Compliance

Portability ? Mac OS, Win xx, UNIX
.. across hardware platforms ?
.. and, need for standards ?

Slide#1-5

Notes:

1. Limits
   1.1. Each implementation has limits or "magic" numbers, which can be
      1.1.1. constants .. .. hard-coded into the programs, or,
      1.1.2. variable/changeable .. .. default/operating values could have been determined using ad-hoc techniques.
   1.2. Changeable Limits, can further be classified as
      1.2.1. Compile-Time: limits are specified in header file <limits.h>
      1.2.2. Run-Time: obtained dynamically .. sysconf(), pathconf, fpathconf()

2. Standardization
   2.1. Standardization can be classified, depending on organization providing the standards:
      2.1.1. International Standards Organization .. .. ISO/IEC 9945:2002
      2.1.2. IEEE Standards .. .. IEEE .. POSIX
      2.1.3. National Body Standards .. .. ANSI (American Natl. Stds. Inst.), BSI (British Stds. Inst.)
      2.1.4. Vendor Groups .. .. The Open Group .. Single UNIX Specification, Version 3.
      2.1.5. Government Standards .. .. FIPS
   2.2. Version 3 of Single UNIX Specification unites IEEE POSIX, The Open Group, industry efforts.
   2.3. Standards are continually evolving.

3. POSIX
   (sourced from the POSIX FAQ at http://www.opengroup.org/austin/papers/posix-paper1.txt)
   3.1. is a family of standards,
   3.2. an acronym for Portable Operating System Interface.
   3.3. still evolving .. Standards are continually evolving, and, POSIX is no exception.
   3.4. a registered trademark of the IEEE, and
   3.5. POSIX.1
      3.5.1. is a well-known POSIX standard - IEEE Std 1003.1
      3.5.2. POSIX.1 is about source code portability.
      3.5.3. not a coding implementation nor an operating system, but,
      3.5.4. specifies APIs (i.e. at the source level),
      3.5.5. provides standard definition and programming interface, guaranteed across supported systems - for app. programmer.
   3.6. POSIX is not limited to the UNIX environment.
      e.g. Win NT/2k/XP, MAC OS 8 + …
   3.7. The name POSIX was suggested by Richard Stallman.
      The following quote appears in the Introduction to POSIX.1:
      3.7.1. "The name POSIX was suggested by Richard Stallman. It is expected to be pronounced pahz-icks as in positive, not poh-six, or other variations. The pronounciation has been published in an attempt to promulgate a standardized way of referring to a standard operating system interface".

4. Family of POSIX Standards
   (sourced from: POSIX Projects and Status [04 Dec 2003] at
   http://www.pasc.org/standing/sd11.html#statustable)

Standard	Description
POSIX.0	Guide to POSIX Open Systems Environment. An overview of POSIX - It is not a standard similar to POSIX.1.
POSIX.1	Systems API (C Language)
POSIX.2	Shell and tools (IEEE-approved standard)
POSIX.3	Testing and verification
POSIX.4	Real-time and threads
POSIX.5	ADA binding to POSIX.1
POSIX.6	System security
POSIX.7	System administration
POSIX.8	Networking Transparent File Access

	Protocol-independent Network Interface Remote Procedure Calls (RPC) Open System Interconnect, - Protocol-Dependent Application Interfaces
POSIX.9	FORTRAN binding to POSIX.1
POSIX.10	Super-computing Application Environment Profile (AEP)
POSIX.11	Transaction Processing AEP
POSIX.13	Real-time and Embedded Systems AEP
POSIX.15	Batch Queuing Extensions. This standard has been approved as 1003.2d.
POSIX.26	Device Control API

5. POSIX is heavily influenced by UNIX® -- and in the latest revision now merges with The Open Group's Base Specifications which comprise the core of the Single UNIX Specification -- in the mid eighties there was a plethora of UNIX operating systems, most of which had names ending in X (e.g. HPUX, AIX, PNX, Xenix, etc), and that certainly influenced the naming decision.

6. What is POSIX.1?

   6.1. POSIX standard 1003.1. known for short as POSIX.1

   6.2. It provides source code portability.. It specifies application programming interfaces (APIs) at the source level

   6.3. The major sections of POSIX.1 are definitions, utilities (awk, grep, ps, vi, etc.), headers (unistd.h, sys/select.h and other C headers), threads, networking, realtime, internationalization, maths functions etc., in total .. over 1350 interfaces. It is neither a code implementation nor an operating system, but a standard definition of a programming interface that those systems supporting the specification guarantee to provide to the application programmer. Therefore, as earlier, POSIX is not limited to UNIX environment. e.g. Win NT/2k/XP, MAC OS 8 + …

7. Application Compliance to POSIX.1
   For POSIX.1, there are four categories of compliance, ranging from a very strict compliance to a very loose compliance. The various categories are outlined in the following subsections:

   7.1. Strictly conforming POSIX.1 applications
   This is the strictest level of application conformance.
   Applications at this level should be able to move across implementations with just a recompilation. A strictly conforming POSIX.1 application requires only the facilities described in the POSIX.1 standard and applicable language standards. At the time of this writing, the only language interface that has been standardized for POSIX.1 is the C language interface. (A strictly conforming POSIX application can use 110 calls from the standard C libraries.)

7.2. ISO/IEC-conforming POSIX.1 applications.
This level of conformance is not as strict as the previous one for two reasons. First, it allows a POSIX.1 application to make use of other ISO or IEC standards. Second, it allows POSIX.1 applications within this level to require options or limit values beyond the minimum. For example, such an application could require that the implementation support filenames of at least 16 characters. The POSIX.1 minimum is 14 characters. An ISO/IEC-conforming POSIX.1 application is one that uses only the facilities described in ISO/IEC 9945-1 and approved conforming language bindings for ISO or IEC standards. This type of application must include a statement of conformance that documents all options and limit dependencies, and all other ISO or IEC standards used.

7.3. Applications conforming to POSIX.1 and <National Body>
This level of conformance is not as strict as the previous, as it allows a <National Body>–conforming POSIX application to use calls from the ANSI or the BSI standard set of calls.A <National Body>–conforming POSIX.1 application differs from an ISO/IEC-conforming POSIX.1 application because this type of application may also use specific standards of a single ISO/IEC organization, such as the American National Standards Institute (ANSI) or British Standards Institute (BSI). This type of application must include a statement of conformance that documents all options and limit dependencies, and all other <National Body> standards used.

7.4. POSIX.1-conforming applications that use extensions.
A conforming POSIX.1 application using extensions, is an application that differs from a conforming POSIX.1 application only because it uses nonstandard facilities that are consistent with ISO/IEC 9945-1. Such an application must fully document its requirements for these extended facilities.

8. Understanding POSIX Compatibility

(sourced from http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarwbgen/html/msdn_posix.asp)

8.1. POSIX Compliance and Windows NT/2k/XP

8.2. POSIX is not limited to the UNIX environment.

8.3. It can also be implemented on non-UNIX operating systems, as was done with the IEEE Standard 1003.1-1990 (POSIX.1) Implementations on Virtual Memory System (VMS), Multiprogramming Executive (MPE), and the Convergent Technology Operating System (CTOS).

9. Understanding Linux POSIX Compatibility

9.1. Linux meets the requirements of strict POSIX conformance.
For a good description of POSIX options and their status in Linux and glibc see URL:
http://people.redhat.com/~drepper/posix-option-groups.html

9.2. The IEEE and The Open Group have announced an extension to POSIX Certification to cover the Latest POSIX Standard.

9.3. The IEEE-SA Standards Board approved two new POSIX real-time standards on December 10, 2003:

9.3.1. REVISION P1003.13/D3 (C/PA) Standard for Information Technology -Standardized Application Environment Profile - POSIX Realtime and Embedded Application Support (AEP)

9.3.2. NEW P1003.26/D4 (C/PA) Standard for Information Technology - Portable Operating System Interface (POSIX) - Part 26: Device Control Application Program Interface (API) [C Language]

9.3.3. The new standards were developed by the System Services Realtime Working Group (SSWG-RT).A future phase will cover the Realtime POSIX profiles.

9.4. Notes on Linux POSIX Compatibility

9.4.1. Linux is often termed POSIX-like, Mostly-POSIX .. similarly imprecise terms. Magazine articles sometimes refer to Linux as POSIX Compliant.
Linux is developed with POSIX standards objective.

9.4.2. UNIX, is used only to describe products that pass the X/Open validation suites. POSIX does not have similar restrictions.

9.4.3. Any reference to POSIX without indicating which POSIX standard it refers to, or been certified for, is not useful.

10. Understanding Windows NT/2k/XP POSIX Compatibility

(sourced from Understanding Windows NT POSIX Compatibility at http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarwbgen/html/msdn_posix.asp)

10.1. As earlier, POSIX Compliance and Windows NT/2k/XP POSIX is not limited to the UNIX environment. It can also be implemented on non-UNIX operating systems, as was done with the IEEE Standard 1003.1-1990 (POSIX.1) Implementations on Virtual Memory System (VMS), Multiprogramming Executive (MPE), and the Convergent Technology Operating System (CTOS).

10.2. Win NT/2k/XP can provide strict POSIX conformance ..

10.3. e.g. NTFS provides a several features to support the Portable Operating System Interface (POSIX) standard, which is defined by the Institute of Electrical and Electronic Engineers (IEEE) standard 1003.1-1990 (also known as ISO/IEC 9945-1:1990). NTFS includes the following POSIX-compliant features.

- **Case-sensitive naming** For example, POSIX interprets README.TXT, Readme.txt, and readme.txt as separate files.
- **Hard links** A file can have more than one name. This allows two different file names, which can be in different folders on the same volume, to point to the same data.
- **Additional time stamps** for when the file was last accessed or modified.

10.4. The POSIX subsystem included with Windows NT and Windows 2000 is not included with Windows XP Professional. A new subsystem supporting the broad functionality found on most UNIX systems beyond the POSIX.1 standard is shipped as part of Interix 2.2. The Interix subsystem can be certified to the NIST FIPS 151-2 POSIX Conformance Test Suite.

⇒ For more information about Interix 2.2, see the Microsoft Interix 2.2 link on the Web Resources page http://www.microsoft.com/windows/reskits/webresources.

Linux Kernel Architecture and Design

The Kernel

The Kernel
- is a **resource manager**,
- loaded into main memory at boot-up, and resident thereafter,
- provides Multi-User, Multi-Tasking (Multi-Processing, and/or Multi-Threading) capabilities.

The kernel does
- Process management
- Memory Management
- File Systems
- Device Control
- Networking

Notes

1. What Features of Linux OS

   The Linux Operating System supports

   1.1. Multi-User .. is the ability to support multiple users simultaneously.

      1.1.1. vs. Single-User .. No Login Control, No Passwords. e.g. MS-DOS

   1.2. Multi-Process .. is the ability to run multiple processes simultaneously. The system switches between the processes, but to the user, it appears that they are all running concurrently.

      1.2.1. vs. Single-Process .. No Job Control, No Scheduling e.g. MS-DOS

      1.2.2. on Single-CPU Systems ..

         Cooperative MP .. voluntarily relinquish control of CPU .. e.g. Microsoft Windows, Win 95, MAC OS 8.

         Pre-emptive MP .. external authority can take control of CPU

      1.2.3. on Multi-CPU Systems.

         Asymmetric Multi-Processing .. one CPU handles all device interrupts .. early variants of UNIX on early MP systems

         Symmetric Multi-Processing .. each CPU handles its own device interrupts .. most current UNIX and UNIX-like Operating Systems on modern MP systems.

   1.3. Multi-Threading .. is the ability to support multiple execution threads.

      1.3.1. vs. Single-Threaded .. may not be Re-Entrant.

1.3.2. same ideas as Multi-Processing, but, within the scope of a Single-Process

The Kernel .. 2

The kernel does
- Process management
- Memory Management
- File Systems
- Device Control
- Networking

Slide#1-6

Notes

1. The Kernel .. as a Resource Manager.
   1.1. The *kernel* is a program in charge of handling all resource requests, for processes that could be are executing concurrently on a Unix System.
   1.2. Typically, the kernel does
      1.2.1. Process management .. Create/destroy processes, IPC, Scheduling, also transparency of abstraction for several processes for single or multiple CPUs.
      1.2.2. Memory management .. management/policy .. memory resource, virtual address
      1.2.3. Filesystems .. abstraction over non-uniform or unstructured hardware.
      1.2.4. Device control .. Device management and access to resources. Almost every system operation is eventually an access to a resource.
      1.2.5. Networking .. Networking management tasks is done by the kernel, because most network operations are not specific to a process: incoming packets are asynchronous events. Packet collecting, identifying, dispatching, delivering, routing and address resolution issues are implemented within the kernel.
   1.3. Most modern UNIX's including Linux have the ability to extend the set of features offered by the kernel, at runtime. This means that functionality can be added to the kernel while the system is up and running. Each piece of code added to the kernel at runtime is called a module.
   1.4. Linux kernel offers support for quite a few different types (or classes) of modules, including, device drivers.
   1.5. Each module is made up of object code (not linked into a complete executable) that can be dynamically linked to the running kernel and can be unlinked.

Program vs. Process

Program
- is a file
- could be a source code listing or an executable.

Process
- is a running instance of a program
- executes instructions .. either CPU ops or access resources

Slide#1-7

Notes:

1. Program: The file on disk – either binary, or source.
2. Process: A running instance of a program.
   2.1. Multiple copies of the same program could be running simultaneously.
   2.2. Each is a running instance of the program. Each is a "process".
3. Process View
   3.1. A "running" process on CPU, could either be doing
      3.1.1. CPU operations .. Process is running in USER space: Currently executing instructions on CPU e.g. MOVE/COPY from "local" memory locations, COMPUTE-operations etc. USER time consumed by the process.
      3.1.2. accessing resource .. Process is running in KERNEL space .. e.g. access disk file. Kernel accesses resources.. on behalf of the process. Instructions requiring access to common system resource are mostly IO operations .. e.g. read(), write() etc. A process needs to access a resource, the process is running in kernel-space. Kernel is accessing a resource on behalf of process (from a process accounting perspective). Upon completion of the kernel-mode operation, the process may switch back to the user-space or, user-mode operation. %SYS time consumed by the process.
4. Process Accounting Summary.
   4.1. CPU-time consumed in USER-space, is reported under USER .. for the process.
   4.2. CPU-time consumed by kernel (for a process), is reported under - SYS .. for process.

Process .. System Calls vs stdlib functions

System calls
.. used for accessing kernel functions and resources

stdlib functions (or, stdlib calls)
.. used for routine tasks
.. may be wrappers on system calls .. e.g. getchar() over read()
.. may not access any system call .. e.g. strlen(), atoi()

Slide#1-8

Notes

1. System Calls: A process accesses a resource, using system calls.
   1.1. System Calls are well-defined call interfaces through which programs request services from kernel representing a set of entry points into the kernel and documented in Section 2 of man pages - the UNIX Programmers Manual .
   e.g. man 2 chmod
   1.2. System Calls are a set of well-defined function calls, that provide entry-points into the kernel, for kernel-related tasks.

2. vs. stdlib function calls
   2.1. A process may do routine tasks using stdlib functions .. e.g. strcpy()
   2.2. Most system calls are mapped into equivalent function calls in the C std libs.
      2.2.1. documented in Section 3 of man pages - the UNIX Programmers Manual .
      2.2.2. A process can call the stdlib function using the standard C calling sequence. This stdlib function in turn invokes the system call, to access the resource.
      2.2.3. System calls may be wrapped into functions in the C stdlibs. It is important to note that while most System Calls are wrapped into stdlib functions, there are stdlib functions that do not call any System Calls (e.g. strcpy() and atoi()).
      2.2.4. Some system calls that are wrapped into stdlib functions include:
      open(): fopen(); close(): fclose(); read():getchar(), getc(), gets(), fgets() .. write(): putchar(), printf(); vs. Unix Utilities in /bin
   2.3. utilities such as chmod, chown map directly on to syscalls .. chmod() and chown()

Process .. Process Execution Mode

Execution Mode
- user mode: limited access to hardware and resources.
- kernel mode: full access to resources (via the Kernel).

Slide#1-9

Notes

1. Process Execution Modes .. User Mode versus Kernel Mode
   1.1. user mode: Applications running with limited access to hardware and resources. This is true even if the application has root privileges. (Ring 3 on x86).
   1.2. kernel mode: Kernel running with full access to resources.
      1.2.1. Drivers and Modules have kernel privileges. (Ring 0 on x86).
      1.2.2. Control of Execution is transferred from the user space to the kernel space through system calls, implemented using traps or exceptions, or, hardware interrupts.
      1.2.3. The mode is the state for each individual cpu in an SMP system, as each CPU could potentially be in a different execution mode.

2. In what mode(s) would the following code execute?

```
int main() {while(1);}
```

3. How about the following code?

```
int main() {while(putchar(getchar()));}
```

Process .. Process Execution Time

Execution time for a process
- Elapsed time,
- CPU time, in two parts
.. User CPU, and,
.. System CPU time.

Slide#1-10

Notes:

1. UNIX Time Values
   1.1. Historically, UNIX maintains three values for measuring the execution time of a process,.
      1.1.1. Elapsed Time: Also called wall clock time, it represents the amount of time a process takes to run. Its value could depend on the other processes being run on the system.
      1.1.2. User CPU Time: User CPU time is the CPU time for the user instructions.
      1.1.3. System CPU time: System CPU time is the CPU time taken by the kernel, when it executes on behalf of a process. For example, the time spent within the kernel - performing the read()or write() is charged to the process.
   1.2. The sum of User CPU time and System CPU time is often called CPU time.

2. What time(s) are meaningful for the following code?
   ```
   int main() {while(1);}
   ```

3. And, for the following code?
   ```
   int main() {while(putchar(getchar()));}
   ```
   Hint: CPU Time? Elapsed Time?

Process Architecture – Address Space

"runs" in memory.

in its own address space
- Code, and
- Data

Slide#1-11

Notes:

1. Data: Globals, string constants, and initialized variables; Heap, and, Stack.

2. Primitive Data Types .. char, short, int, float, long
   System Data Types .. time_t, size_t, mode_t ..
   Length in bytes GNU C/C++ data types on 64-bit addressing architectures.

Datatype	Length	Notes
char	1 byte (8 bits)	a. char can be signed or unsigned .. **CPU dependent** e.g. in **Solaris 5.6** /usr/include/sys/isa_defs.h .. on sparc/i386 .. char is signed. .. on PowerPC .. char is unsigned. b. Java uses Unicode UCS2 char semantics .. char is 2 bytes
short	2 bytes (16 bits)	.. both 32-bit and 64-bit architectures.
int	4 bytes (32 bits)	.. both 32-bit and 64-bit architectures.
float	4 bytes (32 bits)	
long	8 bytes (64 bits)	Was 4 bytes (32 bits) on 32-bit addressing architectures
long long	8 bytes (64 bits)	Can become 16 bytes (128 bits)
double	8 bytes (64 bits)	
long	8 bytes (64 bits)	Can become 10 bytes (80 bits)

double		
size_t	8 bytes (64 bits)	size of(sizeof())
*	8 bytes (64 bits)	PointerType - Was 4 bytes (32 bits) on 32-bit addressing architectures
* + 0	8 bytes (64 bits)	PointerType Arithmetic

3. Data Types and Limits

   (values associated with numerical constants - specified in 32-bit addressing <limits.h> header file)

Datatype	**Meaning**	**Value**
CHAR_BIT	Number of bits - in a **char**.	8
SCHAR_MAX	Maximum value - in a **signed char**.	127
UCHAR_MAX	Maximum value - in **unsigned char**.	255U
CHAR_MAX	Maximum value - in a **char**.	SCHAR_MAX
SHRT_MAX	Maximum value - in a **short**.	32767
WORD_BIT	Number of bits - in a word or **int**.	32
INT_MAX	Maximum value - in an **int**.	2147483647
UINT_MAX	Maximum value - in an **unsigned int**.	4294967295U
SSIZE_MAX	Maximum value - in type **ssize_t**.	INT_MAX
LONG_BIT	Number of bits - in a **long**.	32
LONG_MAX	Maximum value - in a **long**.	2147483647L
ULONG_MAX	Maximum value - in an **unsigned long**.	4294967295UL
LLONG_MAX	Maximum value - in a **long long**.	9223372036854775807L
ULLONG_MAX	Maximum value - in an **unsigned long long**.	18446744073709551615UL

4. Check limits on your system. If available, at the UNIX shell prompt, type `getconf -a`

5. Check the limits on your system.

```
#include <stdio.h>
#include <limits.h>

int main() {
#if  defined(_CHAR_IS_SIGNED)
        printf("signed char,");
#elif defined(_CHAR_IS_UNSIGNED)
```

```
        printf("unsigned char,");
#endif

printf("
-sizeof(short)=%d,
-sizeof(int)=%d,
-sizeof(long int)=%d,
-sizeof(long)=%d,
-sizeof(float)=%d,
-sizeof(long long)=%d\n",
sizeof(short), sizeof(int), sizeof(long int),
sizeof(long), sizeof(float),sizeof(long long));
}
```

6. Primitive System Data Types (sourced from Stevens Sec #2.7 - Pg 45)

datatype	Meaning	Datatype
clock_t	counter of clock ticks	clock_t
dev_t	device numbers	dev_t
fd_set	file descriptor sets	fd_set
fpos_t	file position	fpos_t
gid_t	numeric group IDs	gid_t
ino_t	i-node numbers	ino_t
mode_t	file type , file creation mode	mode_t
nlink_t	link counts for directory entries	nlink_t
off_t	file sizes and offsets(signed)	off_t
pid_t	process IDs and process group IDs(signed)	pid_t
ptrdiff_t	result of subtracting two pointers(signed)	ptrdiff_t
rlim_t	resource limits	rlim_t
sig_atomic_t	data type that can be accessed atomically	sig_atomic_t
sigset_t	signal set	sigset_t
size_t	sizes of objects(such as strings)(unsigned)	size_t
ssize_t	functions that return a count of bytes(signed)(read,write)	ssize_t
time_t	counter of seconds of calendar time	time_t
uid_t	numeric user IDs	uid_t
wchar_t	can represent all distinct character codes	wchar_t

Kernel .. Execution Context

Executes routine “housekeeping” tasks, or
Executes on behalf of a process.

Slide#1-12

Notes:

1. Kernel is running in System Context .. “housekeeping” tasks .. maintaining clock, servicing interrupts.
   Note: Some UNIX kernel’s charge interrupts to a process, while others do not.

2. Kernel is running in Process Context .. accessing resources, on behalf of a process.

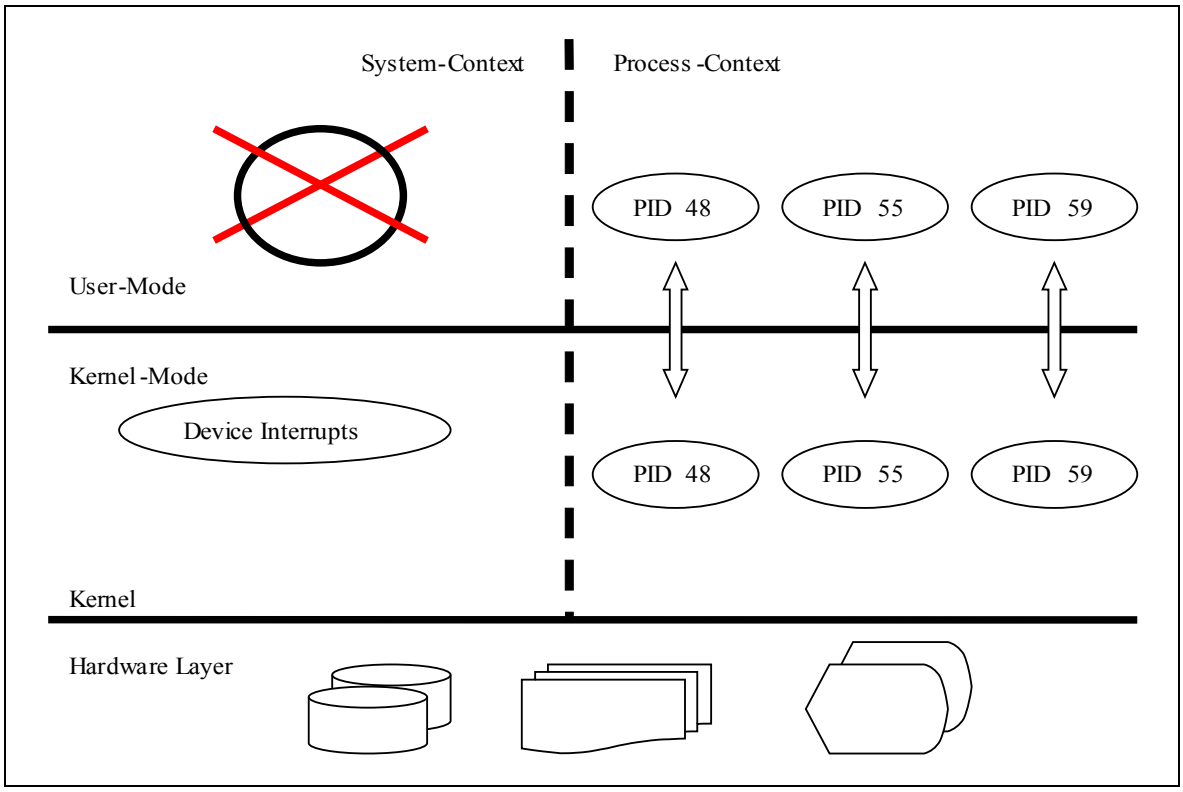
3. If the kernel is executing code, it is in the kernel mode, and, it has full kernel privileges.

4. In the Process context, the kernel is executing code on behalf of a user process - as the user process made a system call request to access a resource. The system call caused an entry into the kernel, at a well-defined entry point. Examples are
   A call to getpid() or, getcwd()
   A read() or a write(), on a file - a resource to which, the kernel regulates access.
   A memory request.

5. In a process context, the kernel keeps, for each process, an entry in the process table. The process table is usually an array or a linked list of structures. From the structure for a given process, you can easily find a pointer to the internal fd table for that process. In the process context, is a structure called task_struct.

6. In Linux, the process table is an array (called "task") of struct task_struct's, and includes a pointer to a struct files_struct, which has the fd array (look at /usr/include/linux/sched.h for details).
   The process table entry is also sometimes called the process descriptor.
   In traditional UNIX’s, the process table is a linked list of struct proc's, which include a pointer to the u_area, which has info about the fds (look at /usr/include/sys/proc.h).

7. In a system context, the kernel is in a kernel thread, and is executing routine maintenance tasks.
   The kernel may be
   servicing an interrupt, or,
   in a system task – such as running the process scheduler,
   executing a task queued up via a kernel timer, a task queue or a tasklet.
   The process context is undefined, and may have incorrect values.
   The context of a kernel thread is lighter than that of a process. Therefore, swapping in and out of kernel threads is easier and faster.



Notes:

1. Mode Switch versus Context Switch.
   1.1. Uresh Vahalia's model of user vs kernel modes, process vs system context.
   1.2. Mode Switch .. A process can be executing, either in User Space, or in, Kernel Space. i.e. to execute a system call, a process switches to the kernel mode.
   1.3. Context Switch .. The kernel can be executing, either in Process Context, or in System Context. i.e. to execute a system call, the kernel is running in Process Context (.. within the Context of a Process).

Kernel .. Types of Memory Addressing

memory address .. refers to the contents of a memory location.

memory location addressing in three ways:
- Physical Addresses
- Linear Addresses
- Logical Addresses

memory can be divided into .. kernel is a program ..
- Kernel Code, and
  - Kernel Data area .. Kernel Globals, Kernel Heap, and, Kernel Stack.

Slide#1-14

Notes:

1. Physical Address
   1.1. Addresses memory locations in memory chips, at the hardware level.
   1.2. (corresponds to electrical signals sent along the address pins of a CPU to the memory bus.)

2. Linear Address (also, known as Virtual Address)
   2.1. Addresses memory locations within program address space, at a process/task level
   2.2. The linear-to-physical address conversion is handled by the paging unit.

3. Logical Address (in x86 architectures)
   3.1. Used for addressing memory locations in machine language instructions, at the OS level (e.g. in x86 processor architecture, the 64k segmentation notation that divides the programs into segments (each logical address consists of the segment address and the offset address within that segment.)).
   3.2. The logical-to-linear address conversion is handled by the segmentation unit.

4. Memory Management Notes

   Memory Management relates to the rules that govern memory addresses.
   - physical addresses relate to addressing memory on memory chips
   - virtual addresses relate to address memory independent of the physical address.

   Therefore, memory requirements or size for a process - can exceed the physical memory size, and yet, load and execute.

Memory Management Notes .. Process View

User Programs will not have all of a fixed amount of physical memory available on a system.

When a program is compiled,

- the compiler generates object file(s) from source file(s).
- the linker links object file(s) with appropriate libraries, and, generates an executable program file.

When a program starts executing

- The executable program has virtual address offsets that, are mapped to physical memory locations when a program is run. The address mapping / translation occurs before loading the executable program into memory, so that actual instructions/data are accessed.
- the kernel sets up data structures - proc, virtual address space, per-process structures for the process, and, makes/schedules the process for running.
- When the process gets running, it switches to user mode and runs in user mode.
- In due course, the process would need access to a page that's not in main memory (when for instance, the pages in its working set are not currently in main memory). This event is called a **page fault**.
    - At a page fault, the kernel puts the process to sleep, switches from user mode (process context) to kernel mode (process context), and attempts to load the page that the process had requested.
        - The kernel searches for the page by locating the virtual address in the per-process region.
        - It then looks at segments (text, data, or other) per-process region to find the actual region that contains the information necessary to read in the page.
    - The kernel needs a free page in which to load the process's requested page. If there are no free pages, the kernel must either page or swap out pages to make room for the new page request.
    - Once there is some free space, the kernel pages in a block of pages from disk. This block contains the requested page plus additional pages that may be used by the process.
    - Finally the kernel establishes the permissions and sets the protections for the newly loaded pages. The kernel wakes the process and switches back to user mode so the process can begin using the requested page.
    - Pages are not brought into memory until the process requests them for execution.
    - This is main concept of a **demand paging system**.

Memory Management Notes .. kernel view

The kernel is always resident in main memory.

- A kernel's configuration includes tunables and limits for kernel tables, daemons, device drivers loaded.
- The memory left over is **available memory**. Most systems display memory statistics during boot time. If a system does not have much available memory, the kernel can be reconfigured as a smaller kernel.

Each process has a well-defined structure.

- The kernel uses specific control data structures to manage the process during its execution. An important data structure is the virtual address space .. *vas* or *as* in <*vas.h*> or <*as.h*>.
- A virtual address space exists for each process and maps to specific segments of a process: text (code), data, u_area, user, and kernel stacks; shared memory; shared library; and memory mapped file segments.
  - Per-process regions protect and maintain the pages mapped to the segments.
  - Each segment has a corresponding virtual address space segment as well.
  - Processes share their text segment.
  - Shared libraries allow programs to access commonly-used code at runtime. Memory requirements of executing programs are reduced. Only one copy of the code is required. Each program accesses shared library code, only when necessary.
  - The u_area contains information used by the kernel, and is a fixed size.
  - The pointer to the user stack is contained in the u_area, and, its size changes during its execution.
  - The process's initialized and uninitialized (BSS) data is in the data segment.
  - Memory mapping allows files to be accessed using memory instructions.
  - Shared memory segments are used by a process to share data with other processes.

Notes on Linux Kernel
(sourced from The Linux Kernel by David Rusling at http://www.tldp.org/LDP/tlk/tlk.html)
The operating system must keep a lot of information about the current state of the system. As things happen within the system these data structures must be changed to reflect the current reality. For example, a new process might be created when a user logs onto the system. The kernel must create a data structure representing the new process and link it with the data structures representing all of the other processes in the system.

Mostly these data structures exist in physical memory and are accessible only by the kernel and its subsystems. Data structures contain data and pointers; addresses of other data structures or the addresses of routines. Taken all together, the data structures used by the Linux kernel can look very confusing. Every data structure has a purpose and although some are used by several kernel subsystems, they are more simple than they appear at first sight.

Understanding the Linux kernel hinges on understanding its data structures and the use that the various functions within the Linux kernel makes of them. This book bases its description of the Linux kernel on its data structures. It talks about each kernel subsystem in terms of its algorithms, its methods of getting things done, and their usage of the kernel's data structures.

Linked Lists
Linux uses a number of software engineering techniques to link together its data structures. On a lot of occasions it uses *linked* or *chained* data structures. If each data structure describes a single instance or occurance of something, for example a process or a network device, the kernel must be able to find all of the instances. In a linked list a root pointer contains the address of the first data structure, or *element*, in the list and each data structure contains a pointer to the next element in the list. The last element's next pointer would be 0 or NULL to show that it is the end of the list. In a *doubly linked* list each element contains both a pointer to the next element in the list but also a pointer to the previous element in the list. Using doubly linked lists makes it easier to add or remove elements from the middle of list although you do need more memory accesses. This is a typical operating system trade off: memory accesses versus CPU cycles.

Hash Tables
Linked lists are handy ways of tying data structures together but navigating linked lists can be inefficient. If you were searching for a particular element, you might easily have to look at the whole list before you find the one that you need. Linux uses another technique, *hashing* to get around this restriction. A *hash table* is an *array* or *vector* of pointers. An array, or vector, is simply a set of things coming one after another in memory. A bookshelf could be said to be an array of books. Arrays are accessed by an *index*, the index is an offset into the array. Taking the bookshelf analogy a little further, you could describe each book by its position on the shelf; you might ask for the 5th book.

A hash table is an array of pointers to data structures and its index is derived from information in those data structures. If you had data structures describing the population of a village then you could use a person's age as an index. To find a particular person's data you could use their age as an index into the population hash table and then follow the pointer to the data structure containing the person's details. Unfortunately many people in the village are likely to have the same age and so the hash table pointer becomes a pointer to a chain or list of data structures each

describing people of the same age. However, searching these shorter chains is still faster than searching all of the data structures.

As a hash table speeds up access to commonly used data structures, Linux often uses hash tables to implement *caches*. Caches are handy information that needs to be accessed quickly and are usually a subset of the full set of information available. Data structures are put into a cache and kept there because the kernel often accesses them. There is a drawback to caches in that they are more complex to use and maintain than simple linked lists or hash tables. If the data structure can be found in the cache (this is known as a *cache hit*, then all well and good. If it cannot then all of the relevant data structures must be searched and, if the data structure exists at all, it must be added into the cache. In adding new data structures into the cache an old cache entry may need discarding. Linux must decide which one to discard, the danger being that the discarded data structure may be the next one that Linux needs.

Abstract Interfaces

The Linux kernel often abstracts its interfaces. An interface is a collection of routines and data structures which operate in a particular way. For example all network device drivers have to provide certain routines in which particular data structures are operated on. This way there can be generic layers of code using the services (interfaces) of lower layers of specific code. The network layer is generic and it is supported by device specific code that conforms to a standard interface.

Often these lower layers *register* themselves with the upper layer at boot time. This registration usually involves adding a data structure to a linked list. For example each filesystem built into the kernel registers itself with the kernel at boot time or, if you are using modules, when the filesystem is first used. You can see which filesystems have registered themselves by looking at the file `/proc/filesystems`. The registration data structure often includes pointers to functions. These are the addresses of software functions that perform particular tasks. Again, using filesystem registration as an example, the data structure that each filesystem passes to the Linux kernel as it registers includes the address of a filesystem specfic routine which must be called whenever that filesystem is mounted.

System Call Conventions

Function Return Value:
- 0 and any +ve value is good
- -1 denotes error

Global Variable: errno
Or, strerror(errno);

Slide #2-4

Notes

1. 0 and any +ve value is good. -1 denotes error.
2. errno
   The global variable errno will be set if an error occurred. errno will not be cleared if a subsequent system call has succeeded.
3. strerr(errno)
   The function strerror() takes an error number and returns the constant describing the nature of the error.
4. Proper Error Handling and Debugging

task_struct

- u_area in traditional UNIX
kernel maintained area on a per-process basis.

Notes:

1. task_struct
   1.1. Information about the process, such as files open, root, and current directories, arguments to current system call and process text, stack and data sizes i.e. it contains process attributes .. e.g. PID, UID, EUID, GID, file descriptor
   1.2. a pointer to the process table entry containing information required for scheduling the process, such as scheduling priorities.
   1.3. file descriptor table containing information required for scheduling a process, such as scheduling priorities.
   1.4. the kernel stack for the process(some implementations exclude the kernel stack from u_area) .. needed for system calls made by the process in kernel mode. The kernel stack is empty while the process is in user mode.
   1.5. In Linux, current - a pointer to struct task_struct is equivalent to u area. The current pointer refers to the user process currently executing. For example, within the kernel, the following statement prints the process ID and the command name of the current process by accessing certain fields in struct task_struct:

```
printk("The process is \"%s\" (pid %i)\n",
current->comm, current->pid);
```

Chapter Summary

This chapter, provided:

- A quick introduction to Linux OS.
- An overview of program and process architecture.
- A description of some fundamental terms that would be encountered over the rest of the course.

Suggested Further Reading & Web References

Some further reading ..

1. Unix System Calls Programming
   a) An Introduction to System Calls (http://www.linux-mag.com/1999-05/compile_01.html)
2. UNIX
   a) What is UNIX? (http://www.unix.org/what_is_unix/)
   - What is The Open Group, OSF, X/OPEN? (http://www.unix.org/what_is_unix/history_timeline.html)
   - "Flavors" of UNIX? (http://www.unix.org/what_is_unix/flavors_of_unix.html)
   b) The Open Group, Single UNIX Spec (http://www.unix.org/version3/)
   c) Flavors of UNIX (http://www.ugu.com/sui/ugu/show?ugu.flavors)
3. LINUX:
   - What is Linux? (http://www.unix.org/what_is_unix/flavors_of_unix.html#linux)
   - Is LINUX UNIX? (http://www.tldp.org/FAQ/Linux-FAQ/general.html#is-linux-unix)
   - POSIX conformance and LINUX (http://people.redhat.com/~drepper/posix-option-groups.html)
   - The LINUX Kernel Notes (http://www.tldp.org/LDP/tlk/tlk.html)
   - LINUX source Browser (http://lxr.linux.no/source/)

Chapter 1
An Introduction to System Calls Programming
Terms & Concepts Worksheet
Table #1 ("Basic")

1. Program.	Source/EXE. Refers to a "file".
2. Process.	instance of a running program.
3. Address Space	addressable memory for a process **Code** (or, text) .. memory section containing executable code. **Data** .. memory section for variables. - Stack .. memory at run-time for local, parameter variables; and, func. call ovhd's. - Heap .. dynamically allocated `malloc()` memory. - Globals .. memory for Initialized, Global, and Static Variables; and, String Constants.
4. Kernel	Is also a Program. - has its own Address Space with its own Code+Data: Stack, Heap and Global Sections.

Table #2 ("Kernel")

5. Kernel.	Kernel is a Resource Manager, and it manages the functions of a. Process Management .. creating, destroying and IPC b. Memory Management .. virtual/physical addressing for address space c. Filesystems .. file abstraction – layers structured filesystem over unstructured hw. d. Device Control .. device control operations performed by code specific to device. e. Networking .. Packet collection, identification & dispatch; routing & addr resolution
6. Some "Multi" Terminology.	a. Multi-User .. vs Single User operation e.g. MS-DOS b. Multi-Processing .. vs Single Processing e.g. MS-DOS Also, Cooperative vs Pre-emptive Multi processing; Single CPU vs Multiple CPU; c. Multi-Threading .. vs Single thread of execution e.g. MS-DOS - Re-entrancy and multiple entry points.
7. Some More Terminology.	a. Concurrency .. simultaneous operation; share resources at similar times. b. Asynchronous Operation .. Non-Blocked Processing; No Wait Operation c. Communication .. with other processes either on same system or remote ..

Table #3 ("Mode Vs Context")

8. Mode.	is in reference to a process. - User Mode: Process is doing *useful* work. e.g. computation. Time spent by process in User Mode is the #CPU seconds in "%**usr**" - Kernel Mode: Process sleeping? Kernel doing work on its behalf. Typically overhead. Time spent by process in Kernel Mode is the #CPU seconds in "%**sys**"
9. Context	is in reference to the kernel - Process Context: Kernel doing work for a "process". - System Context: Kernel doing routine maintenance for "System".
10. Mode Switch vs Context Switch	Process Switches between User Mode and Kernel Mode. Kernel Switches between Process Context and System Context. though, often they're used interchangeably.

Table #4 (System Calls, stdlib calls and UNIX cmd utilities)

11. syscalls	System Calls a. for doing "kernel" tasks. b. entry points into the kernel .. open(), close, read(), write() .. c. usually "wrapped" into std lib calls .. fopen(), fclose(), getchar(), putchar() ..
12. stdlib calls.	a. facilitate encapsulation of "routine" tasks b. may be "wrapped" on system calls .. fopen(), fclose(), getchar(), putchar() c. may not have system call equivalent .. strlen(), strcpy()
13. UNIX shell cmd utilities	a. Equivalent UNIX shell cmd utilities .. chmod .. chmod();chown;pwd .. getcwd()

Table #5 ("u area")

14. u area.	user area a. kernel-maintained area for each process b. contains **process** attributes .. e.g. PID, UID, EUID, GID, file descriptor c. may contain "kernel" stack (or ptr) .. separate stack area in "u area" .. used by syscalls – empty upon return from syscall.

Table #6 ("UNIX .. need for standard")

15. Standards.	Source Code Compatibility. Facilitate balancing of a. Architectural Limits and Magic Numbers .. for optimum performance and b. Ease of Portability and Maintenance .. for interoperability
16. POSIX.	a. Family of Standards .. various standards b. POSIX 1003.1 .. standardizes system call interface

Chapter 1

An Introduction to System Calls Programming

Assignment Questions

1.1 Write a simple program that will run 100% user CPU time.

1.2 Stevens: Ex #1.5, #1.6

a) If the calendar time is stored as a signed 32-bit integer, in what year will it overflow. (UNIX Calendar Time Overflow)

b) If the process time is stored as a signed 32-bit integer, and if the system counts 100 ticks per second, after how many days will the value overflow?

1.3 Stevens: Ex #5.4 Pg 144

The following code works correctly on some machines, but not on others.
What could be the problem? **(Hint**: Signature for getchar() is ***int*** *getchar*();)

```
#include <stdio.h>
int  main(void) {
     char c;
     while ( (c=getchar())  != EOF )
          putchar(c);
}
```

1.4 Determine the limit of maximum number of open files for a process.

a) experimentally, and,

b) using the sysconf() call, and, the _SC_OPEN_MAX argument.

1.5 Determine the #clock ticks per second, on your system,

a) using 'getconf' cmd-line/shell utility (if available on your system), and,

b) using sysconf() call.

Optional .. *Not for Credit!* ..

1.6 a) What is the output of the following program?

b) Explain how its output is derived.

c) Explain the basic concepts involved in its process memory address.

```
#include <stdio.h>

void fn(char *str) {
  if (*str) {
      fn(str+6);
      putchar(*str-1);
  }
  return;
}

int main(void) {
  /*  Some, World Currencies */
  char *str="\"YeN! cEnT! pEsO! KrOnEr! pEnCe! LeMpIrA! pUlS! Hi!!\"";

  fn(str);
  putchar('\n');

  return;
}
```

Optional .. *Not for Credit!* ..

1.7 The following program is sourced from http://www0.us.ioccc.org/1984/anonymous.c

a) What is the output of the following program?

b) Explain how its output is derived.

c) Explain the basic concepts involved in its process memory address.

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hell\
o, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---j,i/i);}
```

Chapter Review Questions

1.1 Terms Review: (One–Liners)
a) Program b) Process c) System Calls d) Standard Library e) POSIX

1.2 Explain the term “address space” in Linux.

1.3 Explain the role of the kernel in Linux.

1.4 Explain the difference between Mode Switch and Context Switch.

1.5 Explain POSIX and its relevance to Linux.

Chapter 1

An Introduction to System Calls Programming

Assignment Hints

#1.1 Hint: Perpetual Loop ? User CPU Time is 100% of Elapsed Time.

#1.2.a Hint: The epoch started on Jan 1, 1970 midnight UTC.
Time is a signed quantity and is incremented once every second.
Consider 32 bit signed ints.

#1.3. Hint: char belongs to the 'int' family consisting of signed or unsigned quantities .. and EOF evaluates to -1 …

#1.4 and #1.5 .. None.
#1.6 Hint: fn() traverses recursively to the end of the string ..
#1.7 Hint: Concepts: array notation a[i] is equiv. to "Test"[i] .. "Test'+i .. i["Test"] .. i+"Test"

Chapter 1

An Introduction to System Calls Programming

#1.1 User CPU Time is 100% of Elapsed Time.

```
#include <stdio.h>

int main() {
     while (1);
}
```

#1.2.a and #1.2.b .. ok .. here's the answer ..

1	numbers in a signed int	1024*1024*1024*2	2147483648
2	less 1	1	1
3	#seconds in a signed int	(#1) less (#2)	2147483647
4	#seconds per day	24*3600	86400
5	signed-int/#secs in years	(#3) / 365.25	68.04965039
#1.2	**UNIX Calendar Time overflow**	**1970 + (#5)**	**Jan 19, 2038**
7	#increments per second	100	100
8	#seconds to overflow	(#3) / (#7)	21474836.47
#1.3	**Process Time overflow (#days)**	**(#8) / (#4)**	**248.55**

Chapter 1

An Introduction to System Calls Programming

Useful Links

1. Linux Documentation and Links

   1.1. Most information for Linux Kernel documentation is directly from the **kernel sources**.
   Sometimes, it is the only documentation.

   1.2. **/usr/src/linux/Documentation** and **kernel-docs.txt**, are handy references.

   1.3. **Excellent Resource for Linux**.. http://www.linuxlinks.com/

   1.4. **Kernel Source Browser** http://lxr.linux.no/source

   1.5. **Getting a Linux Distribution** .. http://www.linux.org.uk/LinuxFTP.html

---

   1.6. **Some Vendor Docs …**

      1.6.1. http://www.intel.com/cd/ids/developer/asmo-na/eng/os/linux/index.htm

      1.6.2. http://www.kernelhacking.org/links/index.htm

   1.7. **Some More …**

      1.7.1. http://www.linux.org/docs/index.html

      1.7.2. http://www.kernelhacking.org/links/index.htm

      1.7.3. http://www.faqs.org/faqs/linux/howto/index/