Unix System Programming

The “Operating System” and System Calls

Outline

Previously (and Chap 1 & 2 from text)

- Covered Brief UNIX history/interface
- UNIX overview - process, shell, file
- Brief intro to basic file I/O - open(), close(), read(), write(), lseek(), fprintf (library call)
- Week of C

This Week:

- Read Chapter 3
- Administrative: Rock.c
- The Operating System & System Calls
- UNIX history - more on the key players
- Efficiency of read/write
- The File: File pointer, File control/access

Administrative

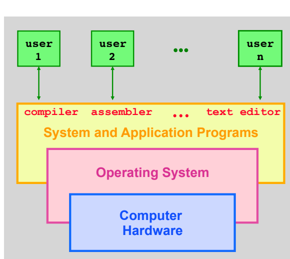
- HW2 posted
- Review `rock.c`

Lets Reflect : What is an Operating System?

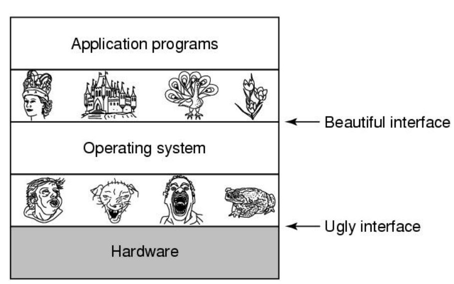
- Software (‘kernel’) that runs at all times
  - » Really, the part of the system that runs in ‘kernel mode’ - or in privileged mode.
    - – As always there are exceptions to this ‘rule’
- Distinguishing what makes up the OS is challenging (some grey areas)
- The Job of the OS is two unrelated functions:
  - » (1) Provide abstractions of resources to the users or applications programs (extends the machine), and
  - » (2) Manage and coordinate hardware resources (resource manager)
    - – CPU, memory, disk, printer

The Bigger Picture

- Operating System
  - » Between Hardware and the Users
  - » Provides an interface/ programming environment for the activities in the system
    - – activities’ (processes) in the system.
      - The application programs
    - – Definition: A *process* is an activity in the system – a running program, an activity that may need ‘services’ (we will cover this concept in detail next week).



The OS provides an *Extended Machine*



- Operating System turn the **ugly** hardware into *beautiful* abstractions.

Example: Resource *Abstraction*

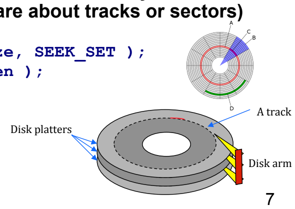
- **Example: Accessing a raw disk really involves :**
  - » specifying the data, the length of data, the disk drive, the track location(s), and the sector location(s) within the corresponding track(s). (150 mph)

```
write( block, len, device, track, sector );
```

- **Problem: But applications don't want to worry about the *complexity* of a disk (don't care about tracks or sectors)**

```
lseek( file, file_size, SEEK_SET );
write( file, text, len );
```

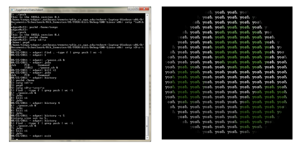
interfaces to OS via system calls



Heads generate a magnetic field that polarize the disk

Shell: Another Level of Abstraction provided to users

- **Provide 'users' with access to the services of the kernel.**
  - » A 'shell' of-course,– illusion of a thin layer of abstraction to the kernel and its services.
- **CLI – command line interface to kernel services (project 1 focus)**
- **GUI - graphical user interface to the kernel**



Person

Shell Abstraction

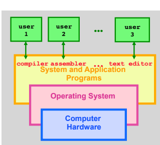
OS Abstraction

Hardware

Key Questions in System Design

How to provide a *beautiful* interface, consider:

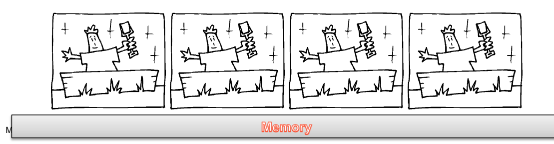
- **What does the OS look like? To the user?**
- **What services does an operating system provide?**
  - » These services need to be provided in a safe manner
    - – E.g., Provision for Safe resource sharing (disk, memory)
    - – What is the mechanism to provide Safety? And why do we nee it?



- Memory Management
- Process Management
- File Management
- I/O System Management
- Protection & Security

Why Safety?: Resource Sharing

- **Example Goal: Protect the OS from other activities and provide protection across activities.**
- **Problem: Activities can crash each other (and crash the OS) unless there is coordination between them.**
- **General Solution: Constrain an activity so it only runs in its own memory environment (e.g., in its own sandbox), and make sure the activity cannot access other sandboxes.**
  - » Sandbox: Address Space (memory space)
    - – It's others memory spaces that the activity can't touch *including the Operating System's address space*



Safety: Resource Sharing

- **Example: Areas of protection:**
  - » Writing to disk (where) – really any form of I/O.
    - – Files, Directories, Socket
  - » Writing / Reading Memory
  - » Creating new processes
- **How do we create (and manage) these 'areas' of protection.**
  - **Let the Kernel Handle it, and for safety it acts in privileged mode to access to hardware broadly**

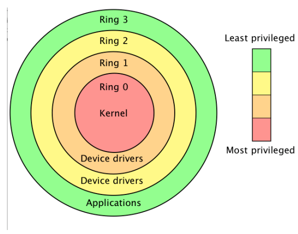
Protection Implementation: *"Dual Mode" Operations*

***How** does the OS prevent arbitrary programs (run by arbitrary users) from invoking accidental or malicious calls to halt the operating system or modify memory such as the master boot sector?*

- **General Idea: The OS is omnipotent and everything else isn't - as simple as that**
  - » Utilize Two modes CPU operation (provided by hardware)
    - – Kernel Mode – Anything goes – access everywhere (unrestricted access) to the underlying hardware.
      - In this mode can execute any CPU instruction and reference any memory access
    - – User Mode – Activity can only access state within its own address space (for example - web browsers, calculators, compilers, JVM, word from microsoft, power point, etc run in user mode).

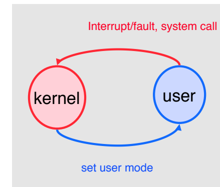
Hardware: Different modes of protection (>2 Intel)

- Hardware provides different mode ‘bits’ of protection – where at the lowest level – ring 0 – anything goes, unrestricted mode (trusted kernel runs here).
  - – Intel x86 architecture provides multiple levels of protection:



Hardware: Example Dual-Mode Operation

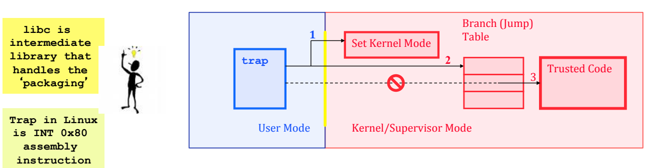
- Mode bit (0 or 1) provided by hardware
  - » Provides ability to distinguish when system is running user code or kernel code
  - » Mode 1 : normal when address space is ‘limited’
  - » Mode 0 : Kernel mode more privileged.
- Mode bit switches
  - » at ‘interrupt’ (trap) to kernel, or
  - » when returning from a trap set back to user mode



- Question: What is the mechanism from the point of view of a process to access kernel functions (e.g., it wants to write to disk)? ….

Mechanics of “System Calls” (e.g., Intel’s trap())

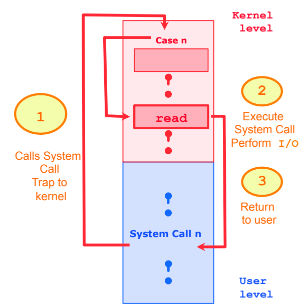
- System Call: Mechanism for user activities (user processes) to access kernel functions.
- Example: UNIX implements system calls (‘request calls’) via the *trap()* instruction (system call, e.g., `read()` contains the trap instruction, internally).



- When the control returns to the user code the CPU is switched back to User Mode.

Example: I/O “System” Calls

- All I/O instructions are privileged instructions.
- Must ensure that a user program could never gain control of the computer in kernel mode
  - » Avoid a user program that, as part of its execution, stores a “new address” in the interrupt vector.
    - – libc

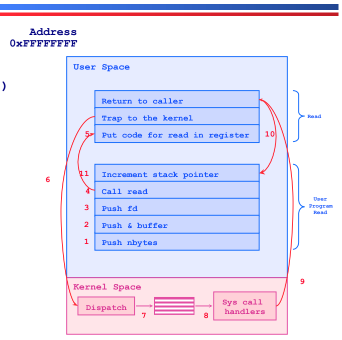


System call to perform I/O Read

UNIX – details - Steps in Making a System Call

P44-45 tannenbaum

- Consider the UNIX `read` “system” call (via a library routine)
  - » `count = read( fd, buffer, nbytes )`
  - » reads `nbytes` of data from a file (given a file descriptor `fd`) into a `buffer`
- 11 steps:
  - » 1-3: push parameters onto stack
  - » 4: calls routine
  - » 5: code for read placed in register
    - – Actual system call # goes into EAX register
    - – Args goes into other registers (e.g, EBX and ECX)
  - » 6: trap to OS
    - – INT 0x80 assembly instruction I in LINUX
  - » 7-8: OS saves state, calls the appropriate handler (read)
  - » 9-10: return control back to user program
  - » 11: pop parameters off stack



Art of picking Registers; http://www.swansontec.com/sregisters.html

System Calls Triva

- Linux has 319 different system calls (2.6)
- Free BSD ‘almost’ 330.

Types of System Calls

- Process control
  - » `fork, execv, waitpid, exit, abort`
- File management (will cover first)
  - » `open, close, read, write`
- Device management
  - » `request device, read, write`
- Information maintenance
  - » get time, get date, get process attributes
- Communications
  - » message passing: send and receive messages,
    - – create/delete communication connections
  - » Shared memory map memory segments

Library Calls

- System call wrappers

Library Routines: Higher Level of Abstraction to System Calls

- Provide another level of abstraction to system calls to
  - » improve portability and
  - » easy of programming
- Standard POSIX C-Library (UNIX) (stdlib, stdio):
  - » C program invoking `printf()` library call, which calls write() system call
- Win 32 API for Windows
- JVM