Structure and Syscalls

Chapters 2.1, 2.2, 2.3, 2.4, 2.7, 2.8

# OS Services

* The OS usually provides the following systems to a user:
  + **User Interface**: Usually a **GUI**, or a **touchscreen** interface for handhelds. **Command line interface** (**CLI**)uses text commands
  + **Program Execution**: Loading programs into memory and executing it normally or abnormally
  + **I/O Operations**
  + **File-System Manipulation**:
    - Involves reading and writing files
    - Also involves creating, deleting, and searching files by name
    - Can involve security and permission management
  + **Communications**: where processes communicate with one another. Can be implemented by:
    - **Shared memory**: two or more processes write to a shared section of memory
    - **Message Passing**: packets of into in predefined formats are moved between processes
  + **Error Detection**: for each type, OS must take appropriate action to keep system running.
* System processes controlled by the OS include:
  + **Resource Allocation**: allocating resources to multiple processes
  + **Logging**: recording resource usage from different processes
  + **Protection and Security**: Protecting separate processes from interfering with one another and controlling the use of shared information. Also must remain secure from outsiders.

# User and Operating-System Interface

## Command Interpreters

* Usually treated as a special program running when a process is initiated, or when a user first logs on.
* Main function: get and execute next user-specified command
* Two approaches to interpreting command:
  + interpreter itself contains the command, and jumps to specified section of code, setting up the parameters and making the appropriate system call
  + uses the command to identify, load and run the file with the given parameters
    - e.g. rm file.txt will find the file rm and execute it with parameter file.txt

## Graphical UI

* Involves a mouse-based window-and-menu system characterised by a **desktop** metaphor.
* User moves mouse to click on **icons**, representing programs or directories (**folders**

## Touchscreen UI

* Users make **gestures** like swiping and pressing on the screen, which are used to open applications.
* iPad and iPhone use **Springboard** interface

# System Calls

* Provide interface allowing a user to request services from the operating system.
* Usually available as functions written in C or C++, though low level tasks can be written using assembly

## Example

* Graphical user interface, text, chat or text message

  Description automatically generatedHere is an example where we copy the contents of one file to another, and the series of system calls associated with it:

## APIs (Application Programming Interface)

* To handle system calls, developers design programs according to an **API**.
* These are a set of functions available to an application programmer, including input and output parameters.
* These are accessed via a library of code, provided by the OS e.g. C uses libc
* APIs increase **portability** such that any system supporting the API can run its functions, though a lot of API’s functions correlate strongly with the OS system calls.

### RTE (Run-Time Environment)

* **Diagram

  Description automatically generated**This is the full suite of software needed to execute applications written in a given programming language.
* This includes compilers or interpreters or other software such as libraries and loaders.
* RTE provide **system-call interfaces**, linking a software to the available system calls
* Typically when an API function is invoked, the relevant system call can be found using a number-indexed table as shown on the right.
* Remember: status is always returned after the system call is invoked

### Passing parameters to OS

* Three general methods used
  + Simplest approach: pass parameters in registers.
  + If there are more parameters that registers, then parameters are stored in block or table in memory, with its address passed into the register.
  + Third approach: API pushes parameters onto **stack**, and then popped by OS. This does not limit number of length of parameters being passed.

## Types of System Calls

* Six categories in total

### Process Control

* Processes:
  + Create and Terminate process
  + Load, Execute
  + Get and Set process attributes: includes process’s priority, max execution time etc.
  + Wait and Signal event
  + Allocate and Free memory
* If a System call normally or abnormally (causes error trap) terminates a program, the dump of memory can be written to a log file for examination from a **debugger**, which attempts to identify **bugs** to help the programmer.
* In either case, control is transferred to invoking command interpreter for next command to be read (GUI’s might use pop-up window to ask user for guidance).
* Errors can be broken into levels, allowing the combination of abnormal and normal termination, and use the error level to determine the next action automatically.
* Data can be **locked** by a process to disable other processes from accessing the data simultaneously.

### File Management

* Processes:
  + Create and Delete file
  + Open and Close files
  + Read, Write and Reposition files
  + Get and Set device attributes: includes file name and type, protection codes, accounting info etc.
  + Perhaps moving and coping data

### Device Management

* Processes:
  + Request and Release device
  + Read, Write and Reposition data on device
  + Get and Set device attributes: includes file name and type, protection codes, accounting info etc.
  + Perhaps moving and coping data
  + Logically Attach or Detach devices
* Very similar to file management hence OS such as UNIX merge functionality into “**device structure**”.

### Information Maintenance

* Processes:
  + Get and Set time or date
  + Get and Set system data e.g. version number, OS type, memory dump etc.
  + Get and Set process, file, or device attributes
* Exist simply for the purpose of transferring information

### Communications

* Processes:
  + Create and Delete communication Connection
  + Send and Receive messages
  + Transfer status information
  + Attach or Detach remote devices

#### Message passing model

* Communicating processes exchange info via messages via a common mailbox.
* Processes are uniquely identified via a translation of **process name,** done by get\_hostid and get\_processid syscalls
* Host name and IP address are used if processes are from two connected systems.
* open() and close() / open\_connection() and close\_connection used with identifiers, with an accept\_process() syscalls to allow communication.
* Most processes receiving connections are special-purpose **daemons**, which are syscalls provided for that purpose.
* Source: **client**, receiving daemon: **server**, exchange messages via read\_message() and write\_message() syscalls.

#### Shared-Memory Model

* shared\_memory\_create() and shared\_memory\_write() used to create and gain access to regions of memory stored by other processes
* Processes ensure that neither one is writing to same location simultaneously.
* Allows maximum speed and convenience of communication, as it can be done at memory transfer speeds when done within a computer.
* can be problematic wtih protection and synchronisation between sharing processes.

### Protection

* Get and Set file permissions

Table

Description automatically generated

# System Services

* **System services**, **or System utilities**, provide convenient environment for program development and execution.
* Fall into categories:
  + **File management**: create, delete, copy, rename, print, list and access and manipulate files and directories
  + **Status Information**: Can range from simple stats to complex details of performance, logging and debugging. Some systems use **registry** to store and retrieve config information.
  + **File modification**
  + **Programming-Language Support**:Compilers, assemblers, debuggers, and interpreters**.**
  + **Program Loading and Execution**: Includes absolute, relocatable, linkage and overly loaders for loading into memory. Debugger systems needed for higher-level/machine languages.
  + **Communications**: Creating virtual connections between processes, users, and computer systems. Includes browsing webpages, sending emails, remote log-in, transferring files from one machine to another etc.
  + **Background Services**: Constantly running system-program processes aka services/subsystems/daemons
* As well as system programs, most OS’s come with application programs like:
  + Web browsers
  + Word processers
  + Text formatters
  + Spreadsheets
  + Database systems
  + Compilers
  + Plotting and statistical-analysis packages
  + Games

# Design Goals

* Requirements at highest level affected by choice of hardware and type of system
* Requirements beyond this split into two groups: **user goals** and **system goals**
* For a **user** the OS should be:
  + Convenient
  + Easy to learn and use
  + Reliable
  + Safe
  + Fast
* For a **developer** the OS should be:
  + Easy to design, implement and maintain
  + Flexible
  + Reliable
  + Error-free
  + Efficient
* All of these requirements can be interpreted in various ways.

## Mechanisms and Policies

* **Mechanism**: determine **how** to do something
* **Policy:** determine **what** will be done
* **E.**g. Timer construct is a mechanism for ensuring CPU protection, policy decides how long the timer is set for a specific user.
* A flexible mechanism that can work over a range of policies is desirable.

## Implementation

* Most OSs are now written in C or C++, with small amounts written in assembly.
* Using higher-level languages comes with following advantages (similar to developing application using HLL):
  + Can be written faster
  + More compact
  + Easier to understand and debug
  + Improvements in compiler tech improve generated code for entire OS via simple recompilation
  + **Important**: easier to port to other hardware, important for OSs running on several different hardware systems like small, embedded devices, Intel x86 systems and mobile/tablet ARM chips.
* Only disadvantage of using HLL is reduced speed and increased storage requirements. Compensated via modern compilers, able to perform complex analysis and apply sophisticated optimisation.
* Modern processors also have deep pipelining and multiple functional units, handling that can handle the complex dependencies.
* Major performance improvements in OSs will come from improving algorithms and data structures.
* Small amount of code critical to high performance:
  + Interrupt handlers
  + I/O manager
  + Memory manager
  + CPU scheduler

# Operating-System Structure

* Common approaches to structuring an OS involves breaking it down into smaller sections of functionality, as you would when designing a program

## Monolithic Structure

* Table

  Description automatically generatedAll functionality of the kernel in a single binary file, saved and run from a single address page, like a bucket.
* Used by UNIX (Linux)
* Has **modular** design, so it can be modified while running
* Advantages: very little overhead in system-call interface and inter-kernel communication is fast.
* Disadvantages: difficult to implement and extend
* **Tightly Coupled**: changes to one part of system have wide-ranging effects on other parts.

# Layered Approach

* **Diagram

  Description automatically generated**OS is broken into smaller components with limited and specific functionality, which is **loosely coupled** as shown.
* Each layer is an **abstract** object made of data and methods to manipulate the data.
* Details of lower levels are not known, just what the operations do.
* Higher levels invoke lower level operations.
* Debugging goes from lowest to highest level. At each level if there is a bug, it exists on that level as the level below has already been debugged.
* Using fewer layers each with more functionality provide most advantages of modularised code, while avoiding problems of layer definition and interaction
* Mainly networks (TCP/IP) and web apps.
* Advantages:
  + simple to construct and debug
  + System verification made easier.
* Disadvantages:
  + Appropriately defining functionality at each level is difficult, and
  + Overall performance can be poor due to overhead of requiring user programs to traverse through multiple layers to obtain OS service.

## Microkernels

* Involves removing all nonessential components from kernel and implementing them as user-level programs, residing in separate address space.
* Diagram

  Description automatically generatedProvide minimal process and memory management, with a communication facility.
* Main functionality: provide communication via **message passing** between client program and various services, also running in user space.
* E.g. if a client program needs access to a file, they exchange messages with the microkernel instead of directly interacting
* Advantages:
  + Extending OS is easier. All new services added to user space, and if kernel does need modified, changes are fewer as kernel is smaller
  + Portable: easier to port from one hardware to another
  + Provides more security and reliability as most services are running as user processes, not kernel. If a service fails, the rest of the OS is untouched.
* Disadvantages
  + Little consensus as to which services should remain in kernel.
  + Increased system-function overhead. When two user-level services must communicate, messages are copied between them, residing in separate address spaces. OS may also have to switch from one process to the next to exchange the message.

## Modules

* **Loadable Kernel Modules** (**LKMs**) are made up of core components and can link in additional services via loadable modules either at boot time or during run time.
* Idea: kernel provides core services while other services are implemented dynamically.
* Adding new features directly to kernel means it must be recompiled every time a change was made. Linking services dynamically helps solve this problem.
* Results in layered system, where each kernel system has defined, protected interfaces.
* Similar to kernel but does not need message passing to communicate.
* Linux uses LKM: allows modular and dynamic kernel, but maintains benefits of monolithic system.

## Hybrid Systems

* Linux: monolithic but modular, allowing dynamic addition of new functionality to kernel
* Windows: monolithic but retains some behaviours of microkernel systems, including providing support for:
  + Separate subsystems running as user processes.
  + Dynamically loadable kernel modules

### MacOS and iOS

* Various layers include:
  + **User Experience Layer**: defines software UI
  + **Application Frameworks Layer**: provide API for Objective-C and Swift programming languages.
  + **Core frameworks**: defines frameworks supporting graphics and media
  + Graphical user interface, diagram

    Description automatically generated**Kernel Environment** (**Darwin**): Mach microkernel and BSD UNIX Kernel.
* Rest of textbook discusses specific systems. Write notes on more if deemed necessary.