Chapter 3

OS Organization
Design of OS

- Factors influencing design of OS
  1. Performance
  2. Protection/Security
  3. Correctness
  4. Maintainability
  5. Commercial factors
  6. Standard & Open Systems
(1) Performance

- Functionality v/s Performance
  - More resource abstraction
  - Higher levels of resource abstraction

- Coding OS w.r.t. Performance
  - Assembly => Fast execution
  - BUT Assembly => Debugging ???

- Others?
(2) Protection & Security

- OS MUST NOT allow one process to interfere with the operations of another process
  - File access
  - Memory space
  - *Resources*

- Therefore, need to implement strategies that support *Isolation & Sharing*

- Challenge is:
  - If OS implements a policy, how to prevent *application* from changing it
(3) Maintainability & (4) Correctness

- Maintainability
  - Design and write systems to be maintainable
    => Sacrifice performance

- Correctness
  - Does the OS meet the requirements?
  - Can we write valid set of requirements?
(5) Commercial influence

- Commercial Influence
  - DOS => IBM-PC
  - UNIX => open platform

- Commercial influence
  => machine nuances that hinder portability
    - UNIX => portable
    - MAC ???
    - Windows ???
(6) Standards & Open Systems

- Early systems: User tied to ONE vendor

- Desire: User gets pieces from ANY set of vendors
  => Need for Standards and Open Systems

- Open Systems
  => Network of heterogeneous systems
  => Information flow [Big Endian v/s Little Endian]
(6) Standards & Open Systems

- Open systems achieved through
  - Application integration => common interface
  - Portability => more applications among hardware platforms
  - Interoperability
    - Standardize remote access facilities
      => All systems talk same language over the network

- POSIX = Open system
  - Standardize OS interfaces
Basic Functions of OS

1. Device Management
3. Memory Management
4. File Management
Device Management

- Isolation
- Allocation
- Share

- Need device drivers
  - Must be able to configure into OS without re-compiling OS (no Source Code)
Device Management

Device-Independent Part

Device-Dependent Part

Device-Dependent Part

Device-Dependent Part

Device

Device

Device

...
Process/Thread/Resource Management

- Process
  - Creating
  - Destroying
  - Blocking
  - Running

- Resource
  - Isolation
  - Sharing
Process/Thread/Resource Management

- Processor
- Multiprogramming
- Thread Abstraction
- Process Abstraction
- Generic Resource Manager
- Primary Memory
- Other
- Abstract Resources
Memory Management

- Allocation & use of main memory
  - Isolation & Protection
  - Sharing

- Virtual Memory
  - Main memory & storage devices
  - Reference ‘memory’ on storage devices

- Segmented VM – viable approach
  - Block & Offset
File Management

- Transfer from main memory to file
  - Code (VM)
  - Data (VM)
  - Editors

- Different file management strategies
  - Sequential
  - Indexed
  - Direct access
  - Networked
Basic OS Organization

Process, Thread & Resource Manager

File Manager

Memory Manager

Device Manager

Processor(s)

Main Memory

Devices
Implementation Considerations

- Process Modes
- Kernels
- Method of requesting system services
Processor Modes

- **Supervisor mode**
  - Can execute any instruction
  - Can reference all memory locations

- **User mode**
  - Subset of instructions
  - Can only reference a subset of memory locations

**In UNIX:**

What can root execute that application cannot?

- `renice` : OS call
- `chown` : OS call
- `IOCTL` (OS call) – if user interleaves output on printer
- Memory accesses
Kernel

- Trusted part of the OS
- Executes in Supervisor mode
- Generally, memory resident
- OS extension run in User mode
  - Example: Drivers

- Kernel functions are invoked by “trap”
Requesting Service from OS

- System call
  - Process traps to OS Interrupt Handler
  - Supervisor mode set
  - Desired function executed
  - User mode set
  - Returns to application
Requesting Svc: System Call

```
call(...);
trap

return;
```
System Call Using the trap Instruction

```c
fork();

fork() {
    ... trap N_SYS_FORK()
    ...
}
```

```
sys_fork() {
    /* system function */
    ...
    return;
}
```
Message Passing

- User process constructs message indicating function (service) needed
- Invokes send to pass message to OS
- Process blocks
  
- OS receives message
- OS initiates Function execution
- Upon Function completion, OS Returns (“OK”)

- Process un-blocks
  
*Send and Receive* analyze message for proper format, etc.
Requesting Svc: Message Passing
Message Passing...

- System call are more efficient

  BUT

  they also unduly tie the Application to specifics of the OS

- Tradeoffs ???