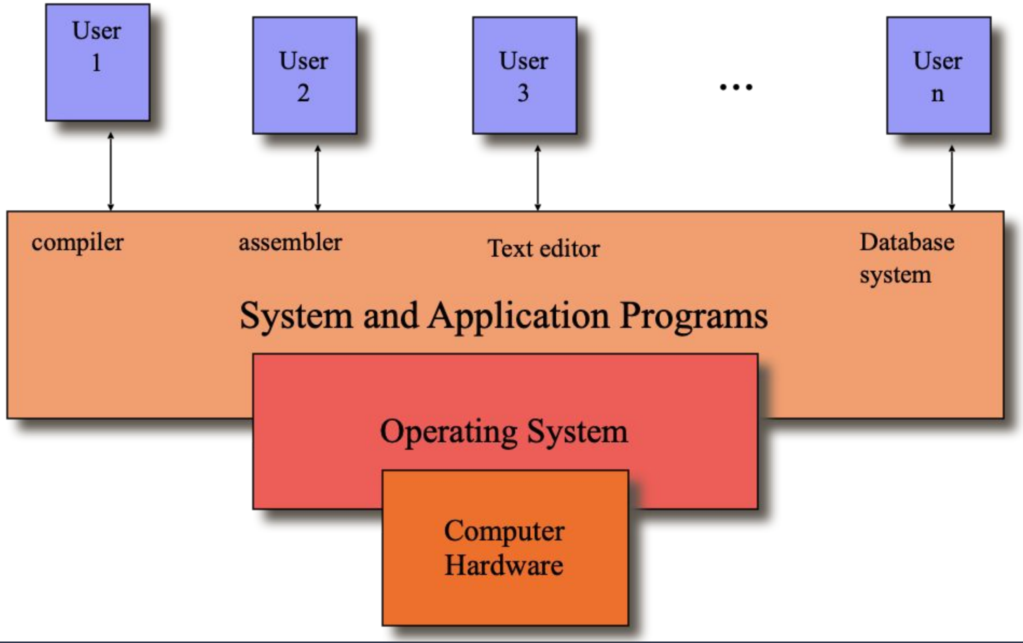
Operating Systems Security

Abstract View of a Computer

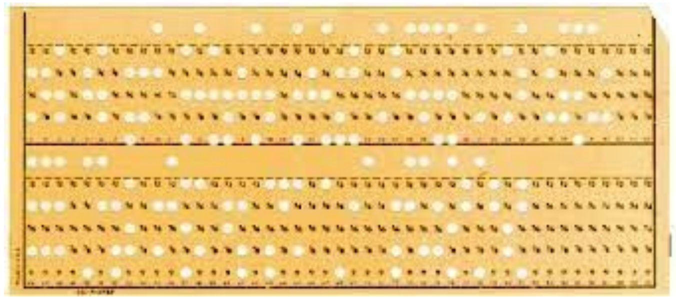


Operating System

- An OS is a program that acts an intermediary between the users and computer hardware.
- It is responsible for:
    - Resource Management and Mediation
        - Resource allocation among users, applications
        - Isolation of different users, applications from each other
        - Communication between users, applications
    - Virtualization
        - Each application appears to have the entire machine to itself
        - For each process, all processors, entire memory and a reliable storage is at its disposal

Evolution of Operating Systems : Early Systems

- Large machines that were run using consoles
- Single user systems
- Programs were written on punch cards/paper tape
- Inefficient resource utilization
- Required constant presence of an operator to load programs



Batch Systems

- Similar jobs (which used similar resources) were grouped together to form batches
- This made resource allocation faster
- However, loading of batch programs took longer
- If one job fails, all the later processes will have to wait indefinitely
- CPU remains idle whenever a process is doing I/O operations

Waiting for I/O

How do we know that I/O is complete?

- **Polling:**
  - Device sets a flag when it is busy.
  - Program tests the flag in a loop waiting for completion of I/O.
- **Interrupts:**
  - On completion of I/O, device sends an interrupt signal to the CPU
  - OS handles the interrupt and returns to the code it was executing prior to servicing the interrupt.

Multiprogramming

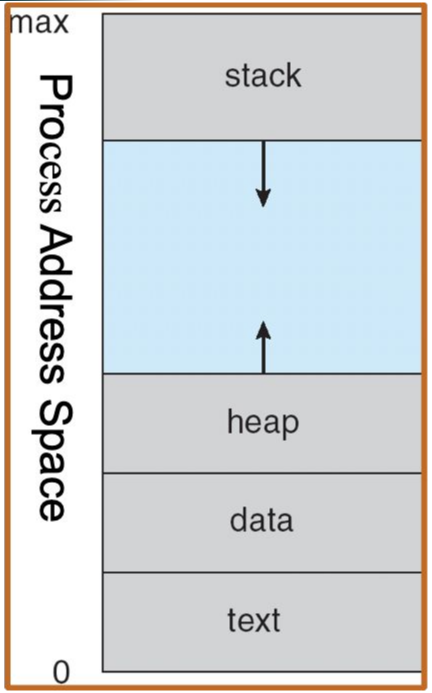
- Multiple processes are run simultaneously.
- Whenever a process does I/O operations, instead of waiting for it to complete the operation, CPU will be allotted to some other process
- When the I/O operation is completed, an interrupt will be sent
- This increases the CPU utilization
- Increases throughput

Time Sharing Systems

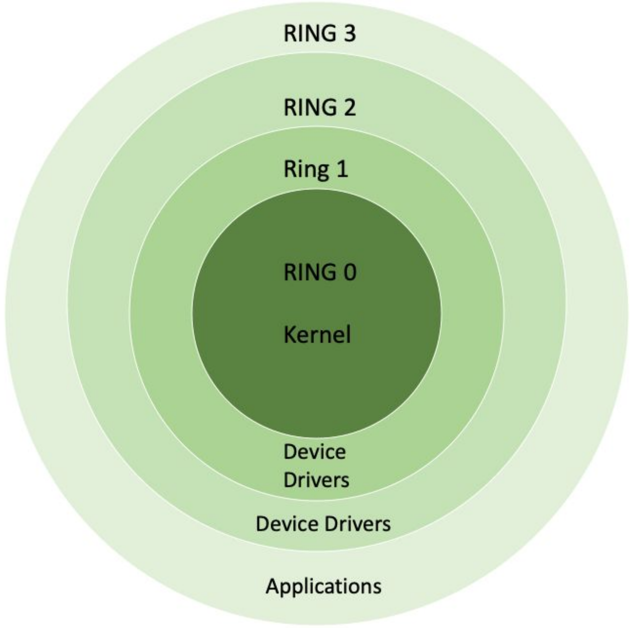
- Each user is given a fixed time slot after which the processor will switch to another user
- This increases the response time
- Reduces CPU idle time
- May have to spend more time in switching between the users

Process

- Process can be defined as a program under execution OR an instance of a program
- Its an active entity
- We can create multiple processes corresponding to a single program
- Each process has a unique identifier called PID
- A process also has an address space which is made of addresses that the process can access



Protection Rings



User Mode Vs Kernel Mode

User Mode

- Less privilege
- User applications run in this mode
- Cannot access the system resources directly

Kernel Mode

- Higher privilege
- Kernel runs in this mode
- Can access system resources directly

System Calls

- System call is a function call into OS code that runs at a higher privilege level of the CPU
- Provides an interface between user-space and the kernel-space
- These interfaces give applications **controlled access to hardware**, a mechanism with which they can create new processes and communicate with existing ones, and the capability to request other operating system resources
- The interfaces act as the messenger between applications and the kernel, with applications issuing requests and the kernel fulfilling them
- The existence of these interfaces and the fact that applications are not free to directly do whatever they want, is key to providing a stable system

Some Example System Calls

- Open () : Opens a given file descriptor
- Read (): Reads content from a given file descriptor
- Write(): write to a file descriptor
- fork(): Creates a child process
- exec(): Makes a process execute a given executable

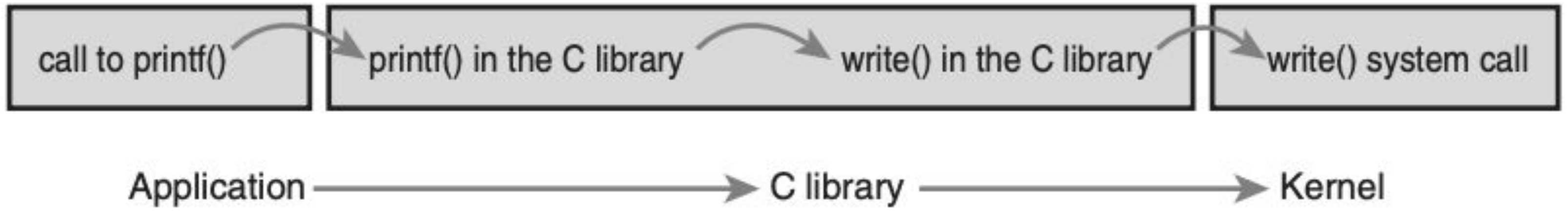
Application Programming Interface

- Defines a set of programming interfaces used by the application programs
- Typically, applications are programmed against the APIs and do not directly use system calls
- As a result there is no need for direct correlation between APIs and the system calls
- There can be multiple implementations of an API
- Internal implementations of them may differ

POSIX

- A standard set of system calls that an OS must implement
- POSIX is one of the most common APIs in the Unix World
- POSIX is composed of series of standards that aim to provide a portable operating system standard
- Programs written to the POSIX API can run on any POSIX compliant OS
- Most modern OSes are POSIX compliant
- Ensures program portability

System Calls



Secure Operating Systems

- A secure operating system provides security mechanisms that ensure that the system’s security goals are enforces despite the threats faced by the system
- A security goal defines a security requirement that the system design can satisfy and that a correct implementation must fulfill
- Security goals can be defined in terms of
  - Security requirements - Example :a confidentiality requirement
  - Functional requirements - Example:Principle of least privilege
- Operating System Security can be of two forms
  - Constrained systems that can enforce security goals with high degree of assurance
  - General-purpose systems that can enforce limited security goals with low to medium degree of assurance

Trust Model

- A systems trust model defines the set of softwares and data upon which the system depends for correct enforcement of system security goals
- For an OS, its trust model is synonymous with the system’s Trusted Computing Base (TCB)
- The OS provides resource protection through access control
- A request always has two main components:
    - the entity making the request
    - the target resource being requested
- OS authenticates the subjects (Process/user) and takes authorization decision based on access control policy

Operating System and Resource Protection

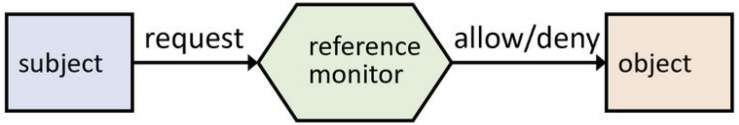
- Ideally, a system TCB should consists of the minimal amount of software necessary to enforce the security goals correctly
- As OS controls access to resources, it is important that it functions as a trusted computing base (TCB) for a computer system.
- Fundamentally, the enforcement mechanism is run within the OS and there are no protection boundaries between the OS functions

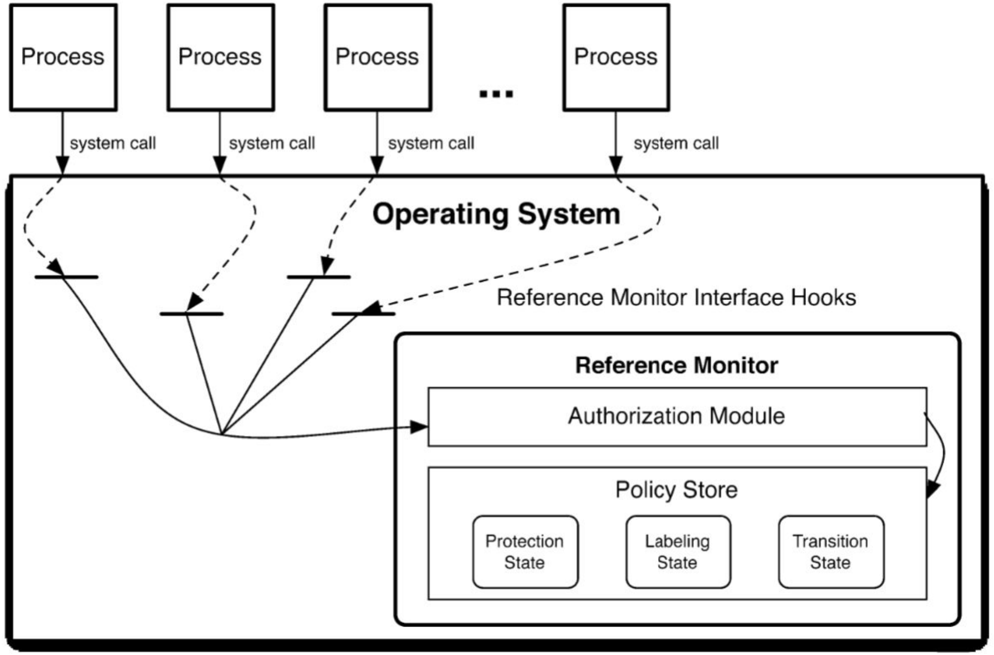
Operating System and Resource Protection

- The following requirements should be met by a trusted computing base.
    - Complete Mediation
    - Tamper-proof
    - Correctness/Verifiability
- It provides complete mediation through indirections such as system calls, file descriptors, virtual addresses, etc.

Reference Monitor

- A Reference Monitor is the classical access enforcement mechanism
- It has three components
  - Reference Monitor Interface
  - Authorization Module
  - Policy Store





Complete Mediation

- It requires that all program paths that lead to a security-sensitive operation be mediated by the reference monitor interface
- The common approach is to mediate through system calls
- Sometimes it may be insufficient as some system calls may implement multiple operations
- Example: Open system call involves opening a set of directory objects before reaching the target file
- If not properly handled, it may lead to Time-of-Check-to-Time-of-Use (TOCTOU) attack

How Can We Trust an OS?

- The CPU executes in different execution modes
- System calls are used to transfer control between user and system code.
- These calls must enter the operating system in a controlled fashion. They come through **call gates**, which are accompanied by a processor privilege ring change on system entry and exit.
- Crossing this boundary means that the operating system will have to change some data structures that keep track of memory mappings, because we will be able to access memory now that we couldn't access before.

How Can We Trust an OS?

- In addition, certain registers will have to be saved, while others have to be loaded.
- In x86 architectures, we have explicit instructions to cross the system boundary: sysenter to enter the operating system during the system call and sysexit to return from the system call.
- The extra work of adjusting privilege rings, changing memory mappings, and loading/saving additional registers makes these calls more costly.

Untrusted User Code Isolation

- Protecting processes from each other and protecting the operating system from untrusted process code follow the same mechanism.
- From a process's point of view, it has the entire computer to itself. It isn't aware that it shares physical memory with other processes.
- This is made possible by the address space abstraction that the operating system creates for a process upon creating the process itself.
- The address space serves as the unit of isolation between processes sharing the same physical hardware resources.
- Page tables translate virtual addresses into physical addresses
- Two page tables, each for a different process, will not contain a mapping to the same physical page at the same time.

Process Protection through Memory Management

- Page table lookups prevent a process from accessing any physical pages that belong to the operating system, since the operating system has not explicitly mapped those locations into the process's address space.
- This translation process is so important that we typically have a piece of hardware called a memory management unit (MMU), that helps the OS perform this translation efficiently, using memory caches such as translation lookaside buffers.
- The hardware will block an attempt to generate an address and complete a read or write in a memory location that belongs to the operating system.
- In addition to resolving a physical address from a virtual address, a page table entry can provide information about different types of memory access available for that page.

Preventing Malicious Code Execution

- One of the strategies that we can use for protecting against a stack buffer overflow attack is to use a non-executable stack.
- While a malicious user might still be able to inject malicious instructions onto the stack, those instructions will not actually be executed.
- The operating system can achieve this by not writing an executable bit for any page table entry associated with the virtual addresses within the process's stack.
- Windows, OS X and Linux all make the stack non-executable.

OS Isolation from Application Code

- The operating system (kernel) resides in a portion of each process's address space.
- The address space in which a process executes now has two sections: the user code/data and the kernel code/data. This partitioning exists for every process that we have in the system.
- Whenever a process wants to access a portion of the address space that contains kernel data or code, the process must make a system call to traverse that boundary.
- Page tables, protection bits, and execution rings all play a role in establishing and enforcing this segregation within the address space.
- For 32-bit Linux systems, the address space is 4GB. Of that 4GB, the lower 3GB is used for user code/data, and the top 1GB is used for the kernel.
- The 1GB points to the same kernel code in each process (i.e. we aren't copying code for each process that is created), but the 3GB section is unique for each process.

References

- Silberschatz, Galvin, Gagne, Operating System Principles, Wiley
- https://www.cs.columbia.edu/~suman/security_arch/s00126ed1v01y200808spt001.pdf
- https://www.omscs-notes.com/information-security/operating-system-security
- https://www.usna.edu/Users/cs/choi/it430/lec/l11/lec.html
-