Operating systems - summary

# 1 Introduction

## Keywords: referee, illusionist, glue

An operating system is, at its core, a layer of software that is responsible for managing the resources on a computer for its users and applications.

There are three important roles that an operating system must fill: being the referee, illusionist, and glue.

**Referee:** It can decide which application gets hardware access at any given time and must stop programs that are having bugs or malfunctions. In this way, the operating system is working as a referee in the game of running programs.

**Illusionist:** Programs running on the computer must be under the illusion that they have all the computer's resources to themselves. This includes the CPU and memory. We achieve this through virtualization (e.g. of memory addresses), such that each program need not worry about where they are in physical memory. This means that individual programs only worry about their own virtual memory space, while the OS mediates their physical memory placements. We achieve good fault isolation and avoid the case where one application's malfunction affects other applications.

**Glue:** The OS must facilitate common services across programs, such as copying and pasting text or images. The OS will work as a sort of glue through offering a common user interface that programs can use. This means that an application need not worry about how they will communicate with other processes and services on the computer, but rather write code that interacts with the interface already offered by the OS.

# 2 Kernel abstraction

## Keywords: booting, memory protection, reliability, security, privacy, dual-mode, privileged instructions, heap, stack, timer interrupts, user/kernel mode, exception, system call, process, upcall, vector table, registers, user/kernel stub, signals

**Reliability:** Proofing the OS from malfunctions caused by any application's malfunction. This will lead to the OS still being operational if a program suddenly stops working or crashes completely.

**Security:** The OS must protect the computer and itself by being selective of which applications get access to which resources. A malicious program with full access to the booting device might modify and corrupt the OS's source code. The OS must, in other words, prevent untrusted code/programs from modifying the system state.

**Privacy:** On a computer with multiple users, each user must only be allowed to view and modify their own data. The OS must prevent untrusted code from accessing unauthorized data.

**Fair resource allocation:** The OS must prevent applications from monopolizing the resources such as the CPU and RAM, to avoid starvation of other applications. It must implement a fair resource allocation approach, to avoid system instability.

**Dual-mode:** Refers to when the OS is based on having two CPU "modes": *user* and *kernel*. In user mode, the CPU must verify that the process requesting the instruction has *permissions to execute that instruction*, before executing. In kernel mode, the CPU executes with this *verification layer turned off*. A single bit in the *processor status register* represents the current mode of the processor.

**Privileged instruction:** Instructions that only can be handled when the CPU is in kernel mode. Instructions that only can be handled in kernel mode are referred to as *privileged instructions*.

**Process memory layout:** How a region of memory is structured per process. From lowest to highest memory address (in 32-bit):

Text segment/machine instructions, 0x00000000 and up

↓

Data segment/program data, 0x00010000 and up

↓

(Uninitialized data)

Heap, 0x10000000 and up towards the stack

↓

↑

Stack, 0x7FFFFFFF and down towards the heap

**Text segment:** Section where the program instructions are stored. The executable machine code instructions lay here.

**Data segment:** Area where static and global variables that have been initialized by the programmed are stored. For example "int j = 123;".

**Uninitialized data:** Data that have not been explicitly declared as anything or have been given the value "0" or "null/NULL". For example "int j;".

**Stack:** Area continuously managed by the OS used to store locally declared variables. Variables are put and removed using LIFO ordering (Last-In-First-Out).

**Heap:** Section of dynamically allocated variables. Data structures that require a longer lifecycle than attainable in the stack are stored here.

**User/kernel mode:** Decided by the bit in the processor status register. The value 0 represents kernel mode, while 1 represents user mode. User mode does not have direct access to connected devices and hardware and must access these through the kernel. Kernel mode has access to hardware and privileged instructions such as writing to persistent storage devices.

**Exception:** The response to, for example, a process attempting to access *unauthorized* *memory addresses* or a process attempting to *change its privilege level*. In this case: *processor exceptions*.

**System call:** The transfer of control from user mode to kernel mode due to the user requesting the kernel to perform a privileged instruction on its behalf. System calls serve as the communication platform between user-level applications and the OS kernel through a set interface. For example "exit()" in UNIX and "ExitProcess()" in Windows.

**Upcalls (signals and asynchronous events):** Virtualized interrupt and exceptions are referred to as upcalls. In *UNIX* they are called *signals*, and *asynchronous events* in *Windows*.The event where the kernel communicates to applications (kernel to user communication) using a set interface. For example "SIGTERM" in UNIX and "CTRL\_CLOSE\_EVENT" in Windows.

**Process:** An instance of a program created by the kernel through "fork()" followed by "exec()" in UNIX and "CreateProcess()" in Windows. A process is an active instance of a program being run, preferably, in RAM.

A diagram of a program

Description automatically generated**User/kernel stub:** A pair of stubs (user and kernel) mediate the communication between the user mode and kernel mode, for example when opening a file. The stubs work as middle-men/intermediaries between the user-level caller and the kernel's actual implementation of the system call. When attempting to open a file in a user-level process, the following steps are completed:

**1.** The user process makes a normal procedure call to the user stub linked to the process.

**2.** The stub executes the trap instruction, thus handing over control to the kernel trap handler (in this case: the kernel stub). The trap handler copies and verifies its arguments.

**3.** The trap handler calls a routine to do the actual file opening (now in kernel mode).

**4.** Once the actual operation is done, the code returns to the trap handler, which then copies the return value into user-level memory.

**5.** The user stub runs the rest of its code immediately after the trap has been completed.

**6.** The user stub returns to the user-level caller.

# 3 Programming interface

## Keywords: flexibility, safety, reliability, process management, Process Control Block (PCB), Windows/UNIX, CreateProcess() vs. fork() & exec(), I/O interfaces, pipes, IPC, monolithic kernel vs. microkernel

**Flexibility:**

**Safety:**

**Reliability:**

**Process Control Block (PCB):** Processes are kept track of using Process Control Blocks (PCB) that reside in kernel memory. The PCB stores information such as its location in memory, where its executable image on the disk is located, which user asked for its execution, its privileges etc.

**Windows/UNIX new process:** In Windows new processes are created using the *CreateProcess()* system call, which takes 10 arguments. In UNIX the system calls fork() and exec() are used in sequence to effectively create a new process. Fork() creates an exact duplicate of the process that uses the system call, while exec() is used to turn this duplicated process into another unique process (through reading another program's data from e.g. the disk).

**I/O interfaces:**

**Pipes:** Used to

**Inter-Process Communication:**

**Monilithic kernel vs. microkernel:**

# 4 Concurrency and threads

## Keywords: threads, TCB, multi-threading

**Threads:**

**Thread Control Block (TCB):**

**Multi-threading:**

# 5 Synchronizing access to shared objects

## Keywords: race conditions, atomic operations, "too much milk", shared object structures, locks, mutual exclusion locks (mutex), condition variables

**Race conditions:**

**Atomic operations:**

**"Too much milk":**

**Shared object structures:**

**Locks:**

**Mutex:**

**Condition Variable (CV):**

# 6 Multi-object synchronization

## Keywords: multiprocessor performance, deadlock vs. starvation, lock design patterns, lock contention, false sharing, MCS and RCU locks

**Deadlock:**

**Starvation:**

**Lock contention:**

**False sharing:**

**Mellor-Crummey and Scott (MCS) lock:**

**Read-Copy-Update (RCU) lock:**

# 7 Scheduling

## Keywords: uniprocessor scheduling, First-In-First-Out (FIFO), Shortest Job First (SJF), Round Robing(RR), Max-Min Fairness (MMF), multiprocessor scheduling

**Uniprocessor scheduling:**

**First-In-First-Out (FIFO):**

**Shortest Job First (SJF):**

**Round Robing (RR):**

**Max-Min Fairness (MMF):**

**Multiprocessor scheduling:**

# 8 Address translation

## Keywords: virtual to physical memory address, memory sharing, segmentation, paging/multi-level paging, translation lookaside buffer (TLB)

**Virtual to physical memory addresses:**

**Memory sharing:**

**Segmentation:**

**Paging:**

**Multi-level paging:**

**Translation Lookaside Buffer (TLB):**

# 9 Caching and virtual memory

## Keywords: memory cache, latency and cache hit/miss, 1st and 2nd level TLB, main/data center memory, local disk/non-volatile memory, Zipf model, memory cache lookup, replacements: random, FIFO, Optimal Cache Replacement (MIN), Least Recently Used (LRU), Least Frequently Used (LFU), Belady's Anamoly, Memory-Mapped Files

**Memory cache:**

**Latency:**

**Cache hit:**

**Cache miss:**

**1st and 2nd level TLB:**

**Non-volatile memory:**

**Zipf mode:**

**Memory cache lookup:**

**Optimal Cache Replacement (MIN):**

**Least Recently Used (LRU):**

**Least Frequently Used (LFU):**

**Belady's Anamoly:**

**Memory-Mapped Files (MMF):**

# 11 File systems

## Keywords: performance, naming, controlled sharing, reliability, non-volatile storage, persistent/stable storage, large capacity and low cost, file, file system, file metadata, file data, file directory, root directory, hard links, soft links, symbolic links, shortcuts, aliases, volume, creating and deleting files, opening and closing files, file descriptor/handle/stream, memory-mapped I/O, direct memory access (DMA), interrupts and polling

**Performance:**

**Reliability:**

**Naming:**

**Controlled sharing:**

**Non-volatile storage:**

**Persistent/stable storage:**

**Large capacity and low cost:**

**File:**

**File system:**

**File metadata:**

**File data:**

**File directory:**

**Root directory:**

**Hard links:**

**Soft/symbolic links:**

**Volume:**

**Creating and deleting files:**

**Opening and closing files:**

**File descriptor/handle/stream:**

**Memory-mapped I/O:**

**Direct Memory Access (DMA):**

**Interrupts and polling:**

# 13 Files and directories

## Keywords: performance, flexibility, persistence, reliability, directories, index structures, free space maps, locality heuristics, defragmentation, file name -> file number, hard and soft/symbolic links, (Microsoft) File Allocation Table (FAT), (Microsoft) New Technology File System (NTFS), (Unix) Fast File System (FFS),

**Performance:**

**Flexibility:**

**Persistence:**

**Reliability:**

**Directories:**

**Index structures:**

**Free Space Maps:**

**Locality heuristics:**

**Defragmentation:**

**Filename to file number:**

**Hard and soft/symbolic links:**

**File Allocation Table (FAT):**

**New Technology File System (NTFS):**

**Fast File System (FFS):**

# Security

## Keywords: hashing, unique salt, setuid, fgets() vs. gets(), monolithic kernel vs. microkernel, principle of least privilege, permissions, authorization, authentication, safe password storage, ciphertext, symmetric encryption, asymmetric encryption, Tenex password bug and attack, ASLR, return-oriented-programming, return-to-libc, access control matrix, symmetric key: DES and IDEA, public key: RSA and PGP, two-factor authentication, session key, worm, virus, Internet worm (Morris), Ping of Death, stack-overflow, buffer-overflow, shoulder-surfing, keylogger, sniffer, spoofing, phishing, security assurance, security monitoring,

**Hashing and salt:**

**Setuid:**

**fgets() and gets():**

**Monolithic kernel vs. microkernel:**

**Principle of least privilege:**

**Permissions:**

**Authorization:**

**Authentication:**

**Two-factor authentication:**

**Safe password storage:**

**Ciphertext:**

**Symmetric and asymmetric encryption (DES, IDEA):**

**Tenex password bug and attack:**

**ASLR:**

**Return-oriented-programming:**

**Return-to-libc:**

**Access control matrix:**

**Public key (RSA, PGP):**

**Session key:**

**Worms and viruses:**

**Stack-overflow:**

**Buffer-overflow:**

**Shoulder-surfing:**

**Keylogger:**

**Sniffer:**

**Spoofing:  
Phishing:**

**Security assurance:**

**Security monitoring:**

*Hvis tid*

**Internet Worm (Morris)**

**Ping of Death:**