Operating Systems Notes Summaries

July 28, 2025

2.11 Summary

- An operating system provides an environment for the execution of programs by providing services to users and programs.
- The three primary approaches for interacting with an operating system are (1) command interpreters, (2) graphical user interfaces, and (3) touch-screen interfaces.
- System calls provide an interface to the services made available by an operating system. Programmers use a system call's application programming interface (API) for accessing system-call services.
- System calls can be divided into six major categories: (1) process control, (2) file management, (3) device management, (4) information maintenance, (5) communications, and (6) protection.
- The standard C library provides the system-call interface for UNIX and Linux systems.
- Operating systems also include a collection of system programs that provide utilities to users.
- A **linker** combines several relocatable object modules into a single binary executable file. A **loader** loads the executable file into memory, where it becomes eligible to run on an available CPU.
- There are several reasons why applications are operating-system specific. These include different binary formats for program executables, different instruction sets for different CPUs, and system calls that vary from one operating system to another.
- An operating system is designed with specific goals in mind. These goals ultimately determine the operating system's policies. An operating system implements these policies through specific mechanisms.
- A **monolithic** operating system has no structure; all functionality is provided in a single, static binary file that runs in a single address space. Although such systems are difficult to modify, their primary benefit is efficiency.
- A **layered** operating system is divided into a number of discrete layers, where the bottom layer is the hardware interface and the highest layer is the user interface. Although layered software systems have had some success, this approach is generally not ideal for designing operating systems due to performance problems.
- The **microkernel** approach for designing operating systems uses a minimal kernel; most services run as user-level applications. Communication takes place via message passing.
- A **modular** approach for designing operating systems provides operating-system services through modules that can be loaded and removed during run time. Many contemporary operating systems are constructed as **hybrid systems** using a combination of a monolithic kernel and modules.
- A **boot loader** loads an operating system into memory, performs initialization, and begins system execution.
- The performance of an operating system can be monitored using either **counters** or **tracing**. Counters are a collection of system-wide or per-process statistics, while tracing follows the execution of a program through the operating system.

1 Summary

- A process is a program in execution, and the status of the current activity of a process is represented by the program counter, as well as other registers.
- The layout of a process in memory is represented by four different sections: (1) text, (2) data, (3) heap, and (4) stack.
- As a process executes, it changes state. There are four general states of a process: (1) ready, (2) running, (3) waiting, and (4) terminated.
- A process control block (PCB) is the kernel data structure that represents a process in an operating system.
- The role of the process scheduler is to select an available process to run on a CPU.
- An operating system performs a context switch when it switches from running one process to running another.
- The `fork()` and `CreateProcess()` system calls are used to create processes on UNIX and Windows systems, respectively.
- When shared memory is used for communication between processes, two (or more) processes share the same region of memory. POSIX provides an API for shared memory.
- Two processes may communicate by exchanging messages with one another using message passing. The Mach operating system uses message passing as its primary form of interprocess communication. Windows provides a form of message passing as well.
- A pipe provides a conduit for two processes to communicate. There are two forms of pipes, ordinary and named. Ordinary pipes are designed for communication between processes that have a parent-child relationship. Named pipes are more general and allow several processes to communicate.
- UNIX systems provide ordinary pipes through the `pipe()` system call. Ordinary pipes have a read end and a write end. A parent process can, for example, send data to the pipe using its write end, and the child process can read it from its read end. Named pipes in UNIX are termed FIFOs.
- Windows systems also provide two forms of pipes—anonymous and named pipes. Anonymous pipes are similar to UNIX ordinary pipes. They are unidirectional and employ parent-child relationships between the communicating processes. Named pipes offer a richer form of interprocess communication than the UNIX counterpart, FIFOs.
- Two common forms of client-server communication are sockets and remote procedure calls (RPCs). Sockets allow two processes on different machines to communicate over a network. RPCs abstract the concept of function (procedure) calls in such a way that a function can be invoked on another process that may reside on a separate computer.
- The Android operating system uses RPCs as a form of interprocess communication using its binder framework.

2 Summary

- A thread is a basic unit of CPU utilization; threads belonging to the same process share many process resources, including code and data.
- There are four primary benefits to multithreaded applications: (1) responsiveness, (2) resource sharing, (3) economy, and (4) scalability.
- **Concurrency** exists when multiple threads are making progress. **Parallelism** exists when multiple threads are making progress simultaneously. On a single-CPU system, only concurrency is possible; parallelism requires a multicore system with multiple CPUs.
- Designing multithreaded applications presents several challenges, including dividing and balancing work, dividing data between threads, identifying data dependencies, and the increased difficulty of testing and debugging.
- **Data parallelism** distributes subsets of the same data across different computing cores and performs the same operation on each core. **Task parallelism** distributes tasks (not data) across multiple cores, with each task running a unique operation.
- User applications create user-level threads, which must be mapped to kernel threads for execution on a CPU. Common mapping models include many-to-one, one-to-one, and many-to-many.
- A **thread library** provides an API for creating and managing threads. Key thread libraries include Windows, Pthreads, and Java threading. Windows is specific to Windows systems, Pthreads is for POSIX-compatible systems (UNIX, Linux, macOS), and Java threads run on any system supporting a Java Virtual Machine.
- **Implicit threading** involves identifying tasks (not threads) and allowing languages or API frameworks to create and manage threads. Approaches include thread pools, fork-join frameworks, and Grand Central Dispatch. Implicit threading is increasingly common for developing concurrent and parallel applications.
- Threads can be terminated using either **asynchronous cancellation** (immediate termination) or **deferred cancellation** (target thread periodically checks for termination, allowing orderly shutdown). Deferred cancellation is generally preferred due to issues with resource reclamation and data consistency in asynchronous cancellation.
- Unlike many other operating systems, Linux does not distinguish between processes and threads, referring to both as **tasks**. The Linux `clone()` system call can create tasks that behave more like processes or threads, depending on the flags passed for resource sharing.

3 Summary

- CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.
- Scheduling algorithms may be either preemptive (where the CPU can be taken away from a process) or nonpreemptive (where a process must voluntarily relinquish control of the CPU). Almost all modern operating systems are preemptive.
- Scheduling algorithms can be evaluated according to the following five criteria: (1) CPU utilization, (2) throughput, (3) turnaround time, (4) waiting time, and (5) response time.
- First-come, first-served (FCFS) scheduling is the simplest scheduling algorithm, but it can cause short processes to wait for very long processes.
- Shortest-job-first (SJF) scheduling is provably optimal, providing the shortest average waiting time. Implementing SJF scheduling is difficult, however, because predicting the length of the next CPU burst is difficult.
- Round-robin (RR) scheduling allocates the CPU to each process for a time quantum. If the process does not relinquish the CPU before its time quantum expires, the process is preempted, and another process is scheduled to run for a time quantum.
- Priority scheduling assigns each process a priority, and the CPU is allocated to the process with the highest priority. Processes with the same priority can be scheduled in FCFS order or using RR scheduling.
- Multilevel queue scheduling partitions processes into several separate queues arranged by priority, and the scheduler executes the processes in the highest-priority queue. Different scheduling algorithms may be used in each queue.
- Multilevel feedback queues are similar to multilevel queues, except that a process may migrate between different queues.
- Multicore processors place one or more CPUs on the same physical chip, and each CPU may have more than one hardware thread. From the perspective of the operating system, each hardware thread appears to be a logical CPU.
- Load balancing on multicore systems equalizes loads between CPU cores, although migrating threads between cores to balance loads may invalidate cache contents and therefore may increase memory access times.
- Soft real-time scheduling gives priority to real-time tasks over non-real-time tasks. Hard real-time scheduling provides timing guarantees for real-time tasks.
- Rate-monotonic real-time scheduling schedules periodic tasks using a static priority policy with preemption.
- Earliest-deadline-first (EDF) scheduling assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority.
- Proportional share scheduling allocates shares among all applications. If an application is allocated N shares of time, it is ensured of having N/T of the total processor time.
- Linux uses the completely fair scheduler (CFS), which assigns a proportion of CPU processing time to each task. The proportion is based on the `virtual runtime (vruntime)` value associated with each task.
- Windows scheduling uses a preemptive, 32-level priority scheme to determine the order of thread scheduling.
- Solaris identifies six unique scheduling classes that are mapped to a global priority. CPU-intensive threads are generally assigned lower priorities (and longer time quantums), and I/O-bound threads are usually assigned higher priorities (with shorter time quantums.)
- Modeling and simulations can be used to evaluate a CPU scheduling algorithm.

4 Summary

- **Race Condition**:
  - Occurs when processes concurrently access shared data.
  - Final result depends on the specific order of concurrent accesses.
  - Can lead to corrupted values of shared data.
- **Critical Section**:
  - A code segment where shared data may be manipulated.
  - A possible race condition may occur here.
  - **Critical-section problem**: Design a protocol for processes to synchronize activity and cooperatively share data.
- **Requirements for a Critical-Section Solution**:
  1. **Mutual exclusion**: Only one process active in its critical section at a time.
  2. **Progress**: Processes cooperatively determine which process enters its critical section next.
  3. **Bounded waiting**: Limits the time a program waits before entering its critical section.
- **Software Solutions**:
  - Peterson's solution is an example.
  - Do not work well on modern computer architectures due to instruction reordering.
- **Hardware Support for Critical Section**:
  - Memory barriers.
  - Hardware instructions (e.g., 'compare-and-swap' instruction).
  - Atomic variables.
- **Mutex Locks**:
  - Provide mutual exclusion.
  - Process must acquire a lock before entering a critical section.
  - Process must release the lock on exiting the critical section.
- **Semaphores**:
  - Can provide mutual exclusion, similar to mutex locks.
  - Unlike mutex locks (binary value), semaphores have an integer value.
  - Can solve a wider variety of synchronization problems.
- **Monitors**:
  - An abstract data type (ADT).
  - Provide a high-level form of process synchronization.
  - Use condition variables:
    - Allow processes to wait for specific conditions to become true.
    - Allow processes to signal one another when conditions are met.
- **Liveness Problems**:
  - Solutions to the critical-section problem may suffer from liveness issues.
  - Includes problems like deadlock and starvation.
- **Tool Evaluation**:
  - Various synchronization tools can be evaluated under different contention levels.
  - Some tools perform better than others under specific contention loads (e.g., uncontended, moderate, high contention).

5 Summary

- Classic process synchronization problems:
  - Bounded-buffer problem.
  - Readers-writers problem.
  - Dining-philosophers problem.
- Solutions use tools from "Synchronization Tools" chapter:
  - Mutex locks.
  - Semaphores.
  - Monitors.
  - Condition variables.
- **Windows synchronization**:
  - Uses dispatcher objects.
  - Uses events to implement synchronization tools.
- **Linux synchronization**:
  - Uses various approaches to protect against race conditions.
  - Includes atomic variables.
  - Includes spinlocks.
  - Includes mutex locks.
- **POSIX API synchronization**:
  - Provides mutex locks.
  - Provides semaphores.
  - Provides condition variables.
  - Two forms of semaphores:
    - Named semaphores: easily accessed by unrelated processes by name.
    - Unnamed semaphores: cannot be shared as easily; require placement in shared memory.
- **Java synchronization**:
  - Rich library and API for synchronization.
  - Available tools:
    - Monitors (provided at language level).
    - Reentrant locks (supported by API).
    - Semaphores (supported by API).
    - Condition variables (supported by API).
- **Alternative approaches to critical-section problem**:
  - Transactional memory.
  - OpenMP.
  - Functional languages.
- **Functional languages**:
  - Intriguing alternative programming paradigm to procedural languages.
  - Unlike procedural languages, do not maintain state.
  - Generally immune from race conditions and critical sections.

6 Summary

- Deadlock: set of processes, each waiting for event caused by another process in set.
- Four necessary conditions for deadlock:
  - Mutual exclusion.
  - Hold and wait.
  - No preemption.
  - Circular wait.
- Deadlock only possible if all four conditions present.
- Deadlocks modeled with resource-allocation graphs; cycle indicates deadlock.
- Deadlock prevention: ensure one of four conditions cannot occur.
- Practical prevention: eliminate circular wait.
- Deadlock avoidance: use banker's algorithm; don't grant resources if leads to unsafe state.
- Deadlock detection: algorithm evaluates processes/resources on running system to find deadlocked state.
- Deadlock recovery:
  - Abort one process in circular wait.
  - Preempt resources assigned to deadlocked process.

7 Summary

- Memory: central to modern computer systems; large array of bytes, each with own address.
- Address space allocation: using **base and limit registers**.
- **Base register**: smallest legal physical memory address.
- **Limit**: specifies size of address range.
- Binding symbolic address references to physical addresses:
  - Compile time
  - Load time
  - Execution time
- **Logical address**: generated by CPU.
- **Memory Management Unit (MMU)**: translates logical address to **physical address**.
- Memory allocation approach: contiguous memory partitions of varying sizes.
- Partition allocation strategies:
  - **First fit**
  - **Best fit**
  - **Worst fit**
- Modern OS: use **paging** to manage memory.
- **Physical memory**: divided into fixed-sized blocks called **frames**.
- **Logical memory**: divided into blocks of same size called **pages**.
- Paging: logical address divided into **page number** and **page offset**.
- **Page number**: index into per-process **page table**.
- **Page table**: contains frame in physical memory holding the page.
- **Offset**: specific location in the frame.
- **Translation Look-aside Buffer (TLB)**: hardware cache of page table.
- Each TLB entry: page number and corresponding frame.
- TLB in address translation:
  - Get page number from logical address.
  - Check if frame for page is in TLB.
  - If in TLB: frame obtained from TLB.
  - If not in TLB: retrieve from page table.
- **Hierarchical paging**: logical address divided into multiple parts for different page table levels.
- Problem with expanding addresses (beyond 32 bits): large number of hierarchical levels.
- Strategies to address this: **hashed page tables** and **inverted page tables**.
- **Swapping**: moves pages to disk to increase degree of multiprogramming.
- Intel 32-bit architecture: two levels of page tables; supports 4-KB or 4-MB page sizes.
- **Page-address extension**: allows 32-bit processors to access physical address space ¿ 4 GB.
- x86-64 and ARM v8 architectures: 64-bit architectures using hierarchical paging.

8 Summary

- Virtual memory: abstracts physical memory into extremely large uniform array of storage.
- Benefits of virtual memory:
  - Program can be larger than physical memory.
  - Program does not need to be entirely in memory.
  - Processes can share memory.
  - Processes can be created more efficiently.
- **Demand paging**: pages loaded only when demanded during program execution.
- Pages never demanded are never loaded.
- **Page fault**: occurs when page not in memory is accessed.
- Page must be brought from backing store into available page frame.
- **Copy-on-write**: child process shares same address space as parent.
- If child or parent modifies page, copy of page is made.
- When available memory low: page-replacement algorithm selects existing page to replace.
- Page-replacement algorithms: FIFO, optimal, LRU.
- Pure LRU: impractical to implement; most systems use LRU-approximation algorithms.
- **Global page-replacement algorithms**: select page from any process for replacement.
- **Local page-replacement algorithms**: select page from faulting process.
- **Thrashing**: system spends more time paging than executing.
- **Locality**: set of pages actively used together.
- Process execution: moves from locality to locality.
- **Working set**: based on locality, set of pages currently in use by a process.
- **Memory compression**: compresses number of pages into single page.
- Alternative to paging, used on mobile systems without paging support.
- **Kernel memory**: allocated differently than user-mode processes.
- Allocated in contiguous chunks of varying sizes.
- Two common techniques for kernel memory allocation:
  - Buddy system.
  - Slab allocation.
- **TLB reach**: amount of memory accessible from TLB.
- Equal to number of entries in TLB $\times$ page size.
- Technique to increase TLB reach: increase page size.
- Linux, Windows, Solaris: manage virtual memory similarly.
- Use demand paging, copy-on-write, and variations of LRU approximation (clock algorithm).

9 Summary

- Hard disk drives (HDDs) and nonvolatile memory (NVM) devices: major secondary storage I/O units.
- Modern secondary storage: structured as large one-dimensional arrays of logical blocks.
- Drives attached to computer:
  1. Through local I/O ports on host.
  2. Directly connected to motherboards.
  3. Through communications network or storage network connection.
- Requests for secondary storage I/O: generated by file system and virtual memory system.
- Each request: specifies device address as logical block number.
- Disk-scheduling algorithms: improve HDD effective bandwidth, average response time, variance in response time.
- Algorithms (SCAN, C-SCAN): improve via disk-queue ordering strategies.
- HDD performance: varies greatly with scheduling algorithms.
- Solid-state disks (SSDs): no moving parts, performance varies little among algorithms.
- SSDs: often use simple FCFS strategy.
- Data storage/transmission: complex, frequently result in errors.
- **Error detection**: attempts to spot problems, alert system for corrective action, avoid error propagation.
- **Error correction**: detects and repairs problems (depends on correction data, corruption amount).
- Storage devices: partitioned into one or more chunks of space.
- Each partition: can hold a volume or be part of a multidevice volume.
- File systems: created in volumes.
- OS manages storage device's blocks.
- New devices: typically pre-formatted.
- Device partitioned, file systems created.
- Boot blocks: allocated to store system's bootstrap program (if device contains OS).
- Block/page corrupted: system must lock out or logically replace with spare.
- Efficient swap space: key to good performance in some systems.
- Some systems: dedicate raw partition to swap space.
- Others: use file within file system.
- Still others: provide both options (user/admin decision).
- Large systems storage: secondary storage devices frequently made redundant via RAID algorithms.
- RAID algorithms: allow more than one drive for operation, allow continued operation/automatic recovery from drive failure.
- RAID algorithms: organized into different levels (each provides reliability/high transfer rates combination).
- **Object storage**: used for big data problems (e.g., Internet indexing, cloud photo storage).
- Objects: self-defining collections of data, addressed by object ID (not file name).
- Typically uses replication for data protection.
- Computes based on data: on systems where copy of data exists.
- Horizontally scalable: for vast capacity and easy expansion.

10 Summary

10.1 Key Points

- Basic I/O hardware elements: buses, device controllers, devices.
- Data movement: CPU (programmed I/O) or DMA controller.
- Device driver: kernel module controlling a device.
- System-call interface handles basic hardware categories: block devices, character-stream devices, memory-mapped files, network sockets, programmed interval timers.
- System calls usually block processes, but nonblocking/asynchronous calls used by kernel/applications that must not sleep.
- Kernel's I/O subsystem provides services: I/O scheduling, buffering, caching, spooling, device reservation, error handling.
- Name translation: connects hardware devices to symbolic file names.
- Involves multiple mapping levels: character-string names $\rightarrow$ device drivers/addresses $\rightarrow$ physical addresses (I/O ports/bus controllers).
- Mapping can be within file-system name space (UNIX) or separate device name space (MS-DOS).
- STREAMS: UNIX mechanism for dynamic assembly of driver code pipelines.
- Drivers can be stacked, data passes sequentially and bidirectionally.
- I/O system calls are costly:
  - Context switching (kernel protection boundary).
  - Signal/interrupt handling.
  - CPU/memory load for data copying (kernel buffers $\leftrightarrow$ application space).

11 Summary

- File: abstract data type, sequence of logical records (byte, line, complex data). OS may support record types or leave to application.
- OS task: map logical file concept to physical storage (hard disk, NVM). May order logical records into physical records.
- Directories: organize files.
  - Single-level directory: naming problems in multiuser systems (unique names required).
  - Two-level directory: separate directory for each user, solves naming problems. Lists file name, location, length, type, owner, times.
  - Tree-structured directory: generalization of two-level, allows subdirectories for organization.
  - Acyclic-graph directory: allows sharing of subdirectories/files, but complicates searching/deletion.
  - General graph structure: complete flexibility in sharing, but may require garbage collection for unused space.
- Remote file systems: challenges in reliability, performance, security. Distributed information systems manage user, host, access info for shared state.
- File protection: needed on multiuser systems.
  - Access controlled by type: read, write, execute, append, delete, list directory.
  - Protection via access lists, passwords, other techniques.