Chapter Summaries from the book

Operating System Concepts / Chapter 1

An operating system is software that manages the computer hardware, as well as providing an environment for application programs to run. Perhaps the most visible aspect of an operating system is the interface to the computer system it provides to the human user.

For a computer to do its job of executing programs, the programs must be in main memory. Main memory is the only large storage area that the processor can access directly. It is an array of bytes, ranging in size from millions to billions. Each byte in memory has its own address.

The main memory is usually a volatile storage device that loses its contents when power is turned off or lost. Most computer systems provide secondary storage as an extension of main memory. Secondary storage provides a form of nonvolatile storage that is capable of holding large quantities of data permanently.

The wide variety of storage systems in a computer system can be organized in a hierarchy according to speed and cost. The higher levels are expensive, but they are fast. As we move down the hierarchy, the cost per bit generally decreases, whereas the access time generally increases.

There are several different strategies for designing a computer system. Single-processor systems have only one processor, while multiprocessor systems contain two or more processors that share physical memory and peripheral devices.

The most common multiprocessor design is symmetric multiprocessing (or SMP), where all processors are considered peers and run independently of one another. Clustered systems are a specialized form of multiprocessor systems and consist of multiple computer systems connected by a local-area network.

To best utilize the CPU, modern operating systems employ multiprogram- ming, which allows several jobs to be in memory at the same time, thus ensuring that the CPU always has a job to execute. Time-sharing systems are an exten- sion of multiprogramming wherein CPU scheduling algorithms rapidly switch between jobs, thus providing the illusion that each job is running concurrently.

The operating system must ensure correct operation of the computer system. To prevent user programs from interfering with the proper operation of the system, the hardware has two modes:

user mode and kernel mode. Various instructions (such as I/O instructions and halt instructions) are privileged and can be executed only in kernel mode.

The memory in which the operating system resides must also be protected from modification by the user. A timer prevents infinite loops. These facilities (dual mode, privileged instructions, memory protection, and timer interrupt) are basic building blocks used by operating systems to achieve correct operation.

A process (or job) is the fundamental unit of work in an operating system. Process management includes creating and deleting processes and providing mechanisms for processes to communicate and synchronize with each other.

An operating system manages memory by keeping track of what parts of memory are being used and by whom. The operating system is also responsible for dynamically allocating and freeing memory space. Storage space is also managed by the operating system; this includes providing file systems for representing files and directories and managing space on mass-storage devices.

Operating systems must also be concerned with protecting and securing the operating system and users. Protection measures control the access of processes or users to the resources made available by the computer system. Security measures are responsible for defending a computer system from external or internal attacks.

Several data structures that are fundamental to computer science are widely used in operating systems, including lists, stacks, queues, trees, hash functions, maps, and bitmaps.

Operating System Structures / Chapter 2

Operating systems provide a number of services. At the lowest level, system calls allow a running program to make requests from the operating system directly. At a higher level, the command interpreter or shell provides a mechanism for a user to issue a request without writing a program. Commands may come from files during batch-mode execution or directly from a terminal or desktop GUI when in an interactive or time-shared mode. System programs are provided to satisfy many common user requests.

The types of requests vary according to level. The system-call level must provide the basic functions, such as process control and file and device manipulation. Higher-level requests, satisfied by the command interpreter or system programs, are translated into a sequence of system calls. System services can be classified into several categories: program control, status requests, and I/O requests. Program errors can be considered implicit requests for service.

The design of a new operating system is a major task. It is important that the goals of the system be well defined before the design begins. The type of system desired is the foundation for choices among various algorithms and strategies that will be needed.
Throughout the entire design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.
Once an operating system is designed, it must be implemented. Oper- ating systems today are almost always written in a systems-implementation language or in a higher-level language. This feature improves their implemen- tation, maintenance, and portability.
A system as large and complex as a modern operating system must be engineered carefully. Modularity is important. Designing a system as a sequence of layers or using a microkernel is considered a good technique. Many operating systems now support dynamically loaded modules, which allow adding functionality to an operating system while it is executing. Generally, operating systems adopt a hybrid approach that combines several different types of structures.
Debugging process and kernel failures can be accomplished through the use of debuggers and other tools that analyze core dumps. Tools such as DTrace analyze production systems to find bottlenecks and understand other system behavior.
To create an operating system for a particular machine configuration, we must perform system generation. For the computer system to begin running, the CPU must initialize and start executing the bootstrap program in firmware. The bootstrap can execute the operating system directly if the operating system is also in the firmware, or it can complete a sequence in which it loads progressively smarter programs from firmware and disk until the operating system itself is loaded into memory and executed.

Part Two

Process Management

A process can be thought of as a program in execution. A process will need certain resources — such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.
A process is the unit of work in most systems. Systems consist of a collection of processes: operating-system processes execute system code, and user processes execute user code. All these processes may execute concurrently.
Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads.
The operating system is responsible for several important aspects of process and thread management: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

Processes / Chapter 3

A process is a program in execution. As a process executes, it changes state. The state of a process is defined by that process's current activity. Each process may be in one of the following states: new, ready, running, waiting, or terminated. Each process is represented in the operating system by its own process control block (PCB).

A process, when it is not executing, is placed in some waiting queue. There are two major classes of queues in an operating system: I/O request queues and the ready queue. The ready queue contains all the processes that are ready to execute and are waiting for the CPU. Each process is represented by a PCB.

The operating system must select processes from various scheduling queues. Long-term (job) scheduling is the selection of processes that will be allowed to contend for the CPU. Normally, long-term scheduling is heavily influenced by resource-allocation considerations, especially memory manage- ment. Short-term (CPU) scheduling is the selection of one process from the ready queue.

Operating systems must provide a mechanism for parent processes to create new child processes. The parent may wait for its children to terminate before proceeding, or the parent and children may execute concurrently. There are several reasons for allowing concurrent execution: information sharing, computation speedup, modularity, and convenience.
The processes executing in the operating system may be either independent processes or cooperating processes. Cooperating processes require an interpro- cess communication mechanism to communicate with each other. Principally, communication is achieved through two schemes: shared memory and mes- sage passing. The shared-memory method requires communicating processes to share some variables. The processes are expected to exchange information through the use of these shared variables. In a shared-memory system, the responsibility for providing communication rests with the application pro- grammers; the operating system needs to provide only the shared memory. The message-passing method allows the processes to exchange messages. The responsibility for providing communication may rest with the operating system itself. These two schemes are not mutually exclusive and can be used simultaneously within a single operating system.

Communication in client–server systems may use (1) sockets, (2) remote procedure calls (RPCs), or (3) pipes. A socket is defined as an endpoint for communication. A connection between a pair of applications consists of a pair of sockets, one at each end of the communication channel. RPCs are another form of distributed communication. An RPC occurs when a process (or thread) calls a procedure on a remote application. Pipes provide a relatively simple ways for processes to communicate with one another. Ordinary pipes allow

communication between parent and child processes, while named pipes permit unrelated processes to communicate.

Threads / Chapter 4

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and scalability factors, such as more efficient use of multiple processing cores.
User-level threads are threads that are visible to the programmer and are unknown to the kernel. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads, because no intervention from the kernel is required.
Three different types of models relate user and kernel threads. The many- to-one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to- many model multiplexes many user threads to a smaller or equal number of kernel threads.
Most modern operating systems provide kernel support for threads. These include Windows, Mac OS X, Linux, and Solaris.
Thread libraries provide the application programmer with an API for creating and managing threads. Three primary thread libraries are in common use: POSIX Pthreads, Windows threads, and Java threads.
In addition to explicitly creating threads using the API provided by a library, we can use implicit threading, in which the creation and management of threading is transferred to compilers and run-time libraries. Strategies for implicit threading include thread pools, OpenMP, and Grand Central Dispatch.
Multithreaded programs introduce many challenges for programmers, including the semantics of the fork() and exec() system calls. Other issues include signal handling, thread cancellation, thread-local storage, and scheduler activations.

Process Synchronization / Chapter 5

Given a collection of cooperating sequential processes that share data, mutual exclusion must be provided to ensure that a critical section of code is used by only one process or thread at a time. Typically, computer hardware provides several operations that ensure mutual exclusion. However, such hardware- based solutions are too complicated for most developers to use. Mutex locks and semaphores overcome this obstacle. Both tools can be used to solve various synchronization problems and can be implemented efficiently, especially if hardware support for atomic operations is available.
Various synchronization problems (such as the bounded-buffer problem, the readers – writers problem, and the dining-philosophers problem) are impor- tant mainly because they are

examples of a large class of concurrency-control problems. These problems are used to test nearly every newly proposed synchronization scheme.

The operating system must provide the means to guard against timing errors, and several language constructs have been proposed to deal with these problems. Monitors provide a synchronization mechanism for sharing abstract data types. A condition variable provides a method by which a monitor function can block its execution until it is signaled to continue.

Operating systems also provide support for synchronization. For example, Windows, Linux, and Solaris provide mechanisms such as semaphores, mutex locks, spinlocks, and condition variables to control access to shared data. The Pthreads API provides support for mutex locks and semaphores, as well as condition variables.

Several alternative approaches focus on synchronization for multicore systems. One approach uses transactional memory, which may address syn- chronization issues using either software or hardware techniques. Another approach uses the compiler extensions offered by OpenMP. Finally, func- tional programming languages address synchronization issues by disallowing mutability.