# **Annotated Bibliography**

# **Brinch Hansen, P. 1973. Operating System Principles. Prentice-Hall, Englewood Cliffs, NJ, (July)**

The main theme of the book is that operating systems are not radically different from other programs. The difficulties encountered in the design of efficient, reliable operating systems are the same as those one encounters in the design of other large programs, such as compilers or payroll programs.

The historical importance of operating systems is that they led to the discovery of new principles of resource sharing, multiprogramming, and program construction. These principles have a general validity beyond operating systems.

The purpose of an operating system is to share computational resources among competing users. To do this efficiently a designer must respect the technological limitations of these resources.

Present computers consist of a small number of components (processors, store modules, and peripherals) which operate strictly sequentially. It is possible to multiplex a single processor and a small internal store (supported by a large backing store) among several computations to create the illusion that they are executed concurrently and have access to a large, homogeneous store. But these abstractions are not supported by the underlying technology, and if they are carried too far, the result is a total collapse of computational service known as thrashing.

One of the difficulties of operating systems is the highly unpredictable nature of the demands made upon them. Independent users submit jobs with varying resource requirements at irregular intervals. An operating system is expected to schedule this unpredictable mixture of jobs in such a manner that the resources are utilized efficiently and the users can expect response within reasonably predictable times.

The only way to satisfy these expectations is probably to put restrictions on the characteristics of jobs so the designer can take advantage of the expected usage of resources. This is certainly the main reason for the success of small, specialized operating systems. It also gives a plausible explanation of the failure of recent ‘general-purpose’ operating systems which try to handle a much greater variety of jobs.

**R. Love, Linux Kernel Development, Third Edition, Developer’s Library (2010)**

This book targets Linux developers and users who are interested in understanding the Linux kernel. The goal of this book is to provide enough information on the design and implementation of the Linux kernel that a sufficiently accomplished programmer can begin developing code in the kernel. This book covers both the usage of core kernel systems and their design and implementation. It helps the developer understand the inner workings of the kernel and how the interfaces are actually used. At the same time, it gave a good understanding of the implementation behind the interface. This is akin to learning some library’s API versus studying the actual implementation of the library. Consequently, the author discusses both the design and usage of kernel subsystems

The author also discusses about the multiprogramming systems. Although most components of present computers are sequential in nature, they can work simultaneously to some extent. The main difficulty of multiprogramming is that concurrent activities can interact in a time-dependent manner which makes it practically impossible to locate programming errors by systematic testing. Perhaps, more than anything else, this explains the difficulty of making operating systems reliable.

If we wish to succeed in designing large, reliable multiprogramming systems, we must use programming tools which are so well-structured that most time-dependent errors can be caught at compile time. However, it seems hopeless to try to solve this problem at the machine level of programming since we can’t expect to improve the situation by means of so-called "implementation languages," which retain the traditional right of systems programmers to manipulate addresses freely.

[**James H. Hill , Aniruddha S. Gokhale, Visual OS: design and implementation of a visual framework for learning operating system concepts, Proceedings of the 43rd annual Southeast regional conference, March 18-20, 2005, Kennesaw, Georgia**](https://dl.acm.org/citation.cfm?id=1167448)

An operating system can be described as software composed of numerous components providing a distinct functionality, such as CPU scheduling, disk scheduling, virtual memory and paging, while working together to efficiently manage the hardware and resources of a computer system. Understanding their features and interplay can be a non-trivial task. To aid in their understanding of the OS dynamics, a number of aids including textbooks, journals and simulators exist. Although these aids suffice to understand simple OS concepts, some algorithms, such as paging, synchronization and process control, and their interactions are too complex to understand without a way to visualize these interactions and operations.

This paper provides three contributions to the R&D on visualizing the dynamics of operating systems. First, they describe the design architecture of Visual OS, which is the OS visualization engine. Second, they describe how they used software design patterns to make their framework extensible to accommodate a variety of OS features that cater to different domains, such as generic computing, real-time systems and embedded systems.

Finally, they described how they have used Visual OS for a programming assignment in a senior level OS class at Vanderbilt University.

**Samy Pessé: How to Make a Computer Operating System**

This text by Samy Pessé is a guide on how to write, test and run, a very simple operating system in C++ and Assembly from scratch. The book gives step by step instructions to build a very simple UNIX-based operating system in C++, not just a "proof-of-concept". The resulting OS would be able to boot, start a user land shell, and be extensible. The OS will be built for the x86 architecture, running on 32 bits, and compatible with IBM PCs.

The first step is to setup a good and viable development environment. The writer has suggested using Vagrant and Virtual box in order to compile and test the resulting OS from all the OSs (Linux, Windows or Mac). The writer has explained the instruction provided in make file and how ‘Make file’ defines some basics rules for building the kernel, the user library and some user land programs.

A kernel can be written in C++ just as it can be in C, with the exception of a few pitfalls that come with using C++ (runtime support, constructors, etc.). The compiler will assume that all the necessary C++ runtime support is available by default, also, we need to add some basic functions that can be found in the C++ file.

Compiling a kernel is not the same thing as compiling a Linux executable, we can't use a standard library and should have no dependencies to the system. Our ‘Make file’ should define the process to compile and link our kernel.

A large number of instructions are available in Assembly but there is not equivalent in C (like cli, sti, in and out), so we need an interface to these instructions. In C, we can include Assembly using the directive ‘ASM’, while ‘GCC use gas to compile the assembly.

**Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau :** [**Operating Systems: Three Easy Pieces**](http://pages.cs.wisc.edu/~remzi/OSTEP/)

The three easy pieces refer to the three major thematic elements the book is organized around: virtualization, concurrency, and persistence. They have explained that the steps to solve a problem is to first find what the most important issue is and solve it via the techniques, algorithms, and ideas. They have also explained how a system works by showing its behavior over time. It has been emphasized that how these timelines are at the essence of understanding.

Virtual memory is an abstraction of physical memory. The purpose of virtual memory is generally to simplify application development and to let processes address more memory than what is actually physically present in the machine.

Some use of segmentation is still necessary to allow for code to execute under different privilege levels. Managing memory is a big part of what an operating system does. Paging and page frame allocation deals with that.

A file system that is slightly more advanced than just the bits of one file is a file with metadata. The metadata can describe the type of the file, the size of the file and so on. A utility program can be created that runs at build time, adding this metadata to a file. This way, a ‘file system in a file’ can be constructed by concatenating several files with metadata into one large file. The result of this technique is a read-only file system that resides in memory.

A virtual file system (VFS) creates an abstraction on top of the concrete file systems. A VFS mainly supplies the path system and file hierarchy, it delegates operations on files to the underlying file systems.

[**Erik Helin, Adam Renberg**](http://littleosbook.github.io/)**: The little book about OS development, 2015-01-19**

This text is a practical guide to writing your own x86 operating system. Developing an operating system is no easy task, and the question “How do I even begin to solve this problem?” is likely to come up several times during the course of the project for different problems. This text helps in setting up the development environment and booting of a very small and primitive operating system.

When developing an OS, it is very convenient to be able to run your code in a virtual machine instead of on a physical computer, since starting your OS in a virtual machine is much faster than getting your OS onto a physical medium and then running it on a physical machine.

Booting an operating system consists of transferring control along a chain of small programs, each one more powerful than the previous one, where the operating system is the last program.

When the PC is turned on, the computer will start a small program that adheres to the Basic Input Output System (BIOS) standard. This program is usually stored on a read only memory chip on the motherboard of the PC. The BIOS program will transfer control of the PC to a program called a bootloader. The bootloader’s task is to transfer control to us, the operating system developers, and our code.

GRUB will transfer control to the operating system by jumping to a position in memory. Before the jump, GRUB will look for a magic number to ensure that it is actually jumping to an OS and not some random code. Once GRUB has made the jump, the OS has full control of the computer.

The code must now be linked to produce an executable file, which requires some extra thought compared to when linking most programs.

The executable must be placed on a media that can be loaded by a virtual or physical machine. the kernel ISO image can be created with a program. A folder must first be created that contains the files that will be on the ISO image. A configuration file for GRUB must be created. This file tells GRUB where the kernel is located.