Literature Study

J.L. van den Berg
jbg930, 2539148

2019-05-13

1 Introduction

Operating systems are programs or collections of programs that together form software that is the foundation of many computer systems. Their purpose is to handle the complex task of managing the various resources, such as memory, disks, and processors. They act as an intermediary for other applications to receive and access these resources and provide primitives and services for other applications to use, allowing them to focus less on the hardware specifics and more on providing actual functionality. While there isn't a clear notion of where the boundaries of the definition of the operating system lies, there is generally a single process on the computer system that is running at all times, the kernel.

1.1 Structure

There are several ways to structure operating systems. The most common way is by means of a Monolithic kernel. In a monolithic kernel, all functionality of the operating system is bundled into a single application. Inside monolithic kernels all functions can call all other functions, which is quite efficient and allows for a lot of freedom. They also usually support loading additional features and drivers through kernel modules. The biggest downside is that should a crash happen anywhere, for any reason, the entire kernel crashes with it, rendering the system inoperable. Such a crash becomes more likely when looking at drivers, as it only takes a single buggy driver to take the entire system down.

Another way to structure an operating system is by using a microkernel. Microkernels are the polar opposite of a monolithic kernel. By putting as little code inside the kernel as possible, they try to reduce the code surface that allows critical bugs and crashes. Microkernels attempt to encapsulate all functionality into small, interconnected modules. Only one of these modules would become the kernel, and all others become normal user processes. This includes drivers, ensuring that a buggy driver cannot bring the whole system down with it.

Next we have the Unikernel approach. Unikernels are constructed by combining several library operating systems, where the components of an operating system, such as processes and networking, are implemented as libraries. By mixing and matching these library routines, you can create a completely customized, fast, tiny operating system to fit your exact needs of your application. Unikernels only have a single address space, and thus kernel and userspace are the same.

Note

Normal vs Embedded vs Distributed?

1.2 System Components

1.2.1 Processor

Arguably the most important component of any computer system is its Central Processing Unit (CPU). Its job is to fetch an instruction from memory, decode it, execute it, and repeat the cycle until there is nothing left to do. This forms the way that programs are executed. Each processor has a set of available instructions it can execute, called the processors architecture, such as Intel x86.

Communication from the CPU to normal memory is relatively slow. To mitigate this, there are several memory slots called registers inside the CPU. Most of the instructions of the CPU will use these registers as arguments. There are several types of registers, where the most common is the General Purpose Registers (GPRs) which can store both data and addresses for instructions to use. Besides the GPRs there are also Special Purpose Registers (SPRs), such as the Program Counter, which keeps track of what instruction to execute next, the Stack Pointer, which keeps track of the current stack frame that is being operated on, a flag register, which keeps track of several flags such as results from comparisons or effects of instructions.

1.2.2 Memory

The second most important part of any computer system is its memory. Memory consists of an array of individually addressable bytes, the amount ranging from a few bytes up until multiple hundreds of gigabytes. It is the only storage the CPU has access to. Other storage is required to first copy data into memory before the CPU can use it. The notion of memory usually refers to main memory, usually called Random Access Memory (RAM) which contains the data and instructions that are needed in process execution by the CPU. RAM is normally implemented using volatile types of memory causing the information that is stored to be lost when powered down. Besides RAM, many systems also contain a small amount of non-volatile Read Only Memory (ROM) which can, for example, be used to load the bootstrapping code from at system boot.

While we ideally would have memory that is as fast and as big as possible, this is not possible due to the exorbitant costs this would entail. Instead, modern computer systems consists of a hierarchy of memory, where memory becomes more scarce the faster it becomes. At the pinnacle of this hierarchy sit the CPU registers, which are as fast as the CPU itself but are extremely limited in capacity, having just a handful in total. These registers are generally handled by the software itself.

Below registers we have cache memory. Caches remain small, but are much faster than main memory and are used to hold recently, or frequently used data, and is mostly controlled by hardware. Accessing these caches only takes a few clock cycles, whereas accessing RAM takes a handful of nanoseconds [1].

Note

I should probably expand on caches (design, workings). If we are going to talk about spectre/meltdown we will need it.

1.2.3 Processes and Threads

When a program is executed, the operating system will wrap it in a process, allowing it to keep track of its state during execution. Associated with each process is an address space for that process alone. In this address space the program text, data, and stack are loaded. Besides the address space, the process also contains all information related to the current state of the process, such as the program counter, stack pointer, other CPU registers, and open files and current positions in these files.

Should the OS need to suspend execution of a process at a given time, it can store the current state in the process. When the OS is ready to resume execution, it can reload the state from the process and restart the program, transparent to the running program itself. This save and reload cycle allows the OS to run many programs 'at the same time' on a single processor core by assigning each process a slice of the available processor time. The OS could also swap out a process if it has engaged some sort of IO. These operations take a, relatively speaking, long amount of time to complete, and as such the time waiting could be better spent working on something else.

Note

Maybe talk about process states? Might not be too useful given the goal of the lit study.

Should a single process want to execute several procedures at once, the process could spawn a secondary process. These new child processes will then receive their own address space, program counter, and register set separate from the parent process. However, it is sometimes convenient to have several interactions happen at the same time while explicitly sharing the address space, allowing the interactions to work together without inter-process communication.

Note

Maybe elaborate on IPC?

We call this multithreading, and each individual flow of control a thread. These threads share an address space, program text, and data, but each have their own set of registers and stack.

There are several advantages to threads over processes, such as being more response by allowing the entire process to continue running while a single thread is blocked, easier communication between the threads as compared to processes, and being far cheaper to spawn than regular processes.

Note

Do I need to extend with this section?

1.2.4 I/O and drivers

Computer Systems generally have some kind of in or outputs attached to them, think disks, mice, keyboards, networks, etc. It is one of the operating system's jobs to manage all these devices and provide ways to normal applications to access these devices. It does so by providing generic interfaces to deal with categories of devices, as well as low level primitives for interacting with things such as ports or busses. The problem with

providing generic interfaces is that there can be massive differences between the various devices under that interface, that need specific code to handle its quirks. To deal with these, operating systems generally provide a way for hardware vendors to provide drivers, special programs that can convert the generic IO calls into the relevant calls to lower level primitives. How these drivers are implemented is wholly dependent on the OS. As such, decisions such as if it is loaded statically or dynamically, provided with the OS or separately, and whether or not its ran in kernelspace or userspace are freely decided by the OS itself.

1.2.5 Filesystems

Not all workload's storage needs can be met by solely relying on RAM. Some workloads can require large amounts of storage, larger than the amount of RAM equipped on the system. Other workloads might require the stored data to be accessible across system restarts. One solution for these problems lies in using external storage devices, such as hard disk drives (HDDs) or solid state disks (SSDs).

While writing data directly to the disks is a valid solution to this problem, Operating Systems generally provide more user friendly abstractions built upon the disks. Partitions allow the operating system to subdivide the disk into several regions that can each be managed separately.

Note

Expand on partitions

Filesystems provide a user friendly abstraction to actually storing data on disks, and generally consist of two parts: files and the directory structure. Files are the operating systems structure for the storage of a single piece of data on the file system, and are typically named. Its name is supposed to be a unique, usually human readable, identifier for storing and retrieving the data from the disk, and is detached from the actual position the file is stored at on the disk. Besides uniqueness, rules for naming files differ from filesystem to filesystem, such as case sensitivity and length. However, most file names have an extension, a short identifier after the name that specifies the type of data stored within the file. Depending on the filesystem or operating system, this could trigger different behaviours regarding the file, but it could also just serve as a convention reminding the user what type of data is contained inside the file. Besides the name, file systems tend to store more OS metadata such as the size and timestamps such as when the file was created.

Besides regular files, which contain actual data, there are also several special types of files, such as directories, device special files. Device special files are generally used as file-based interfaces to device drivers, such as input devices, or have special meanings, such as `/dev/null` on Unix-like systems.

Directories form the basis of the file structure of the file system. While old filesystems might have had just a very simple structure, having just a single directory called the root directory, more modern filesystems have a hierarchical structure, where you still have the root directory, but in each directory one can have many subdirectories and files. This allows the user to arbitrarily group files in a way that makes sense for the workload, and for providing access controls as we will see later in section 3.

2 Threat Model

Outline

- Threat Model
- Information Leaks
- Privilege Escalation
- Interference

3 Protecting

3.1 Principles

Outline

- Security Principles of Saltzer & Schroeder
  - Economy of Mechanism
  - Fail-safe defaults
  - Complete Mediation
  - Open Design
  - Seperation of Privilege
  - Least Privilege / Least authority
  - Least common mechanism
  - Psychological Acceptability
- Authorization & Authentication
  Maybe limit scope?
- (H) How these relate to the different OS designs

3.2 Access Control

Outline

- Bell-LaPadula and Biba
- Capabilities
- Access Control List
- Discretionary Access Control
- Mandatory Access Control
- Role Based Access Control

3.3 Memory

Outline

- Segmentation
- Virtual Memory
- Paging
- Capability Based Addressing
- (H?) What if no MMU
- (H) IOMMU
- (?) Protection Keys (Intel MPK, Intel MPX)
- (?) Tiny bit of taint tracking

3.4 File system

Outline

- Permissions
- Secure Deletion

3.5 Kernel

Outline

- Rings / Privilege Levels
  ring -1, ME, SMM
- (k)CFI
- (k)ASLR
- W^X

3.6 Isolation

Outline

- Virtualization
- Containers (LXC, Docker)
- SMEP / SMAP
- (?) Effects of Cloud
- (?) Enclaves (SGX / TrustZone)

3.7 Hardware Security

Outline

- Hardware Trust
- Trusted Compute Base
- Secure Boot
- Driver Signing

3.8 Unknown

I'm not yet quite sure where to put these

- (?) Issues caused by architectures & solutions

References

[1] D. Levinthal, "Performance analysis guide for intel® core™ i7 processor and intel® xeon™ 5500 processors," 2008.