EDAF35: Operating Systems — Reference Sheet
Lund University

Karl Hallsby

Last Edited: March 21, 2020

Contents

List of Theorems

1 Operating System Introduction

A computer system can be roughly divided into 4 parts.

- The Hardware
- The Operating System
- The Application Programs
- The Users

Defn 1 (Hardware)**.** *Hardware* is the physical components of the system and provide the basic computing resources for the system.. Hardware includes the Central Processing Unit, Memory, and all I/O devices (monitor, keyboard, mouse, etc.).

Remark 1.1 (How to Differentiate). If you are finding it difficult to tell Hardware, Software, and Firmware apart, answer this simple question. Can you hit it with a hammer and break the thing?

Yes Then it is Hardware.
No Then it is Software.
Yes and No Then it is Firmware.

Defn 2 (Software)**.** *Software* is the code that is used to build the system and make it perform operations. Technically, it is the electrical signals that represent `0` or `1` and makes the Hardware act in a specific, desired fashion to produce some result.

On a higher level, this can be though of as computer code.

Remark 2.1 (How to Differentiate). If you are finding it difficult to tell Hardware, Software, and Firmware apart, answer this simple question. Can you hit it with a hammer and break the thing?

Yes Then it is Hardware.
No Then it is Software.
Yes and No Then it is Firmware.

Defn 3 (Operating System)**.** An *operating system* is a large piece of software that controls the Hardware and coordinates the many Application Programs various numbers of Users may use. It provides the means for proper use of these resources to allow the computer to run.

By itself, an operating system does nothing useful. It simply provides an **environment** within which other programs can perform useful work.

The fundamental goal of computer systems is to execute user programs and to make solving user problems easier. These programs require certain common operations, such as those controlling the I/O devices.

In addition, there is no universally accepted definition of what is part of the operating system. A simple definition is that it includes everything a vendor ships when you order "the operating system." The features included, however, vary greatly across systems. Some systems take up less than a megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are based entirely on graphical windowing systems. A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer—usually called the Kernel.

Remark 3.1 (Kernel-Level Non-Kernal Programs). Along with the Kernel, there are two other types of programs:

1. System Programs,
   - Associated with the Operating System but are not necessarily part of the Kernel.
2. Application Programs
   - Includes all programs not associated with the operation of the system

Defn 4 (Kernel)**.** The kernel is a computer program at the core of a computer's operating system with complete control over everything in the system. It is the "portion of the operating system code that is always resident in memory". It facilitates interactions between hardware and software components. On most systems, it is one of the first programs loaded on startup (after the bootloader). It handles input/output requests from software, translating them into data-processing instructions for the central processing unit. It handles memory and its mapping, peripherals like: keyboards, monitors, printers, and speakers. A kernel connects the application software to the hardware of a computer.

The critical code of the kernel is usually loaded into a separate area of memory, which is protected from access by application programs or other, less critical parts of the operating system. The kernel performs its tasks, such as running processes, managing hardware devices such as the hard disk, and handling interrupts, in this protected kernel space.

Defn 5 (Application Program)**.** An *application program* is a tool used by a User to solve some problem. This is the main thing a normal person will interact with. These pieces of software can include:

- Text editors

- Compilers
- Web browsers
- Word Processors
- Spreadsheets
- etc.

Defn 6 (User)**.** A *user* is the person and/or thing that is running some Application Programs.

Processes that the user starts run under the user-mode or user-level permissions. This are significantly reduced permissions compared to the Kernel-mode permissions the Operating System has.

Remark 6.1 (Thing Users). Not all Users are required to be people. The automated tasks a computer may do to provide a seamless experience for the person may be done by other users in the system.

1.1 User View

The user's view of the computer varies according to the interface they are using.

In modern times, most people are using computers with a monitor that provides a GUI, a keyboard, mouse, and the physical system itself. These are designed for one user to use the system at a time, allowing that user to monopolize the system's resources. The Operating System is designed for **ease of use** in this case, with relatively little attention paid to performance and resource utilization.

More old-school, but stil in use, a User sits at a terminal connected to a mainframe or a minicomputer. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization, to assure that all available CPU time, memory, and I/O are used efficiently and that no individual user takes more than their fair share.

In still other cases, Users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers, including file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

Lastly, there are Operating Systems that are designed to have little to no User view. These are typically embedded systems with very limited input/output.

1.2 System View

From the computer's point of view, the Operating System is the program that interacts the most with the hardware. A computer system has many resources that can be used to solve a problem:

- CPU time
- Memory space
- File-storage space
- I/O devices
- etc.

The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many Users access the same system.

Another, slightly different, view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.3 Computer Organization

The initial program, run ***RIGHT*** when the computer starts is typically kept onboard the computer Hardware, on ROMs or EEPROMs.

Defn 7 (Firmware)**.** *Firmware* is software that is written for a specific piece of hardware in mind. Its characteristics fall somewhere between those of Hardware and those of software. It is almost always stored in the Hardware's onboard storage. Typically it is stored in ROM (Read-Only Memory) or EEPROM (Electrically Erasable Programmable Read-Only Memory). It initializes all aspects of the system, from Central Processing Unit Registers to device controllers, to memory contents.

Remark 7.1 (How to Differentiate). If you are finding it difficult to tell Hardware, Software, and Firmware apart, answer this simple question. Can you hit it with a hammer and break the thing?

Yes Then it is Hardware.
No Then it is Software.

Yes and No Then it is Firmware.

A Central Processing Unit will continue its boot process, until it reaches the `init` phase, where many other system processes or Daemons start. Once the computer finishes going through all its `init` phases, it is ready for use, waiting for some event to occur. These events can be a Hardware Interrupt or a software System Call.

Defn 8 (Daemon)**.** In UNIX and UNIX-like Operating Systems, a *daemon* is a System Program process that runs in the "background", is started, stopped, and handled by the system, rather than the User. Daemons run constantly, from the time they are started (potentially the computer's boot) to the time they are killed (potentially when the computer shuts down). Typical systems are running dozens, possibly hundreds, of daemons constantly.

Some examples of daemons are:

- Network daemons to listen for network connections to connect those requests to the correct processes.
- Process schedulers that start processes according to a specified schedule
- System error monitoring services
- Print servers

Remark 8.1 (Other Names)*.* On other, non-UNIX systems, Daemons are called other names. They can be called *services*, *subsystems*, or anything of that nature.

Defn 9 (Interrupt)**.** An *interrupt* is a special event that the Central Processing Unit **MUST** handle. These could be system errors, or just a button on the keyboard was pressed. Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.

When a CPU receives an interrupt, it immediately stops what it is doing and transfers execution to some fixed address. To ensure that this happens as quickly as possible, a Interrupt Vector is created.

Defn 10 (Trap)**.** A *trap* or *exception* is a software-generated Interrupt caused by:

- A program execution error (Division-by-zero or Invalid Memory Access).
- A specific request from a user program that an operating-system service be performed (Print to screen).

Defn 11 (Interrupt Vector)**.** The *interrupt vector* is a table/list of addresses that redirect the Central Processing Unit to the location of the instructions for how to handle that particular Interrupt. Since only a predefined number of interrupts is possible, a table of pointers to interrupt routines is used to provide the necessary speed. These locations hold the addresses of the interrupt service routines for the various devices. This array, or interrupt vector, of addresses is then indexed by a unique device number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. The interrupt routine is called indirectly through the table, with no intermediate routine needed. Generally, this is stored in low memory (the first hundred or so locations).

1.4 Storage Management

Defn 12 (File)**.** The Operating System abstracts from the physical properties of its storage devices to define a logical storage unit, the *file*. The operating system maps files onto physical media and accesses these files via the storage devices.

1.5 System Calls

Defn 13 (System Call)**.** Software may trigger an interrupt by executing a special operation called a *system call*. This can also be called a monitor call.

System calls provide an interface to the services made available by an Operating System. These calls are generally available as routines written in C and C++. Some of the lowest-level tasks (for example, tasks where hardware must be accessed directly) may have to be written using assembly instructions.

There are roughly 6 different types of system calls:

1. Process Control
    - End, Abort
    - Load, Execute
    - Create process, Terminate process
    - Get process attributes, Set process attributes
    - Wait for time
    - Wait event, Signal event
    - Allocate and Free memory
2. File Manipulation
    - Create file, Delete file

- Open, Close
- Read, Write, Reposition
- Get file attributes, Set file attributes

3. Device Manipulation
    - Request device, Release device
    - Read, Write, Reposition
    - Get device attributes, Set device attributes
    - Logically attach or detach devices
4. Information Maintenance
    - Get time or date, Set time or date
    - Get system data, Set system data
    - Get Process, File, or Device attributes
    - Set Process, File, or Device attributes
5. Communications
    - Create, Delete communication connection
    - Send, Receive messages
    - Transfer status information
    - Attach or Detach remote devices
6. Protection

	Windows	**Unix**
Process Control	`CreateProcess()`	`fork()`
	`ExitProcess()`	`exit()`
	`WaitForSingleObject()`	`wait()`
File Manipulation	`CreateFile()`	`open()`
	`ReadFile()`	`read()`
	`WriteFile()`	`write()`
	`CloseHandle()`	`close()`
Device Manipulation	`SetConsoleMode()`	`ioctl()`
	`ReadConsole()`	`read()`
	`WriteConsole()`	`write()`
Information Maintenance	`GetCurrentProcessID()`	`getpid()`
	`SetTimer()`	`alarm()`
	`Sleep()`	`sleep()`
Communications	`CreatePipe()`	`pipe()`
	`CreateFileMapping()`	`shm_open()`
	`MapViewOfFile()`	`mmap()`
Protection	`SetFileSecurity()`	`chmod()`
	`InitializeSecurityDescriptor()`	`umask()`
	`SetSecurityDescriptorGroup()`	`chown()`

Table 1.1: System Calls in Unix and Windows

System Calls are exposed to the programmer by an Application Programming Interface.

Defn 14 (Application Programming Interface)**.** An *Application Programming Interface* (*API*) specifies a set of functions that are available to an application programmer. They specify the parameters that are passed to each function and the return values the programmer can expect.

Typically, API calls perform System Calls in the background, without the programmer knowing about them.

1.5.1 Process Control

A running program needs to be able to halt its own execution, either normally or abnormally. If a system call is made to terminate the currently running program abnormally, or if the program runs into a problem and causes an error Trap, a dump of memory is sometimes taken and an error message generated. The dump is written to disk and may be examined by a debugger—a system program designed to aid the programmer in finding and correcting errors, or bugs—to determine the cause of the problem.

Under either normal or abnormal circumstances, the operating system must transfer control to the invoking command interpreter. The command interpreter then reads the next command.

To determine how bad the execution halt was, when the program ceases execution, it will return an exit code. By convention, and for no other reason, an exit code of `0` is considered to be the program completed execution successfully. Otherwise, the greater the return value, the greater the severity of the error.

1.5.2 File Manipulation

We first need to be able to `create()` and `delete()` files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to `open()` it and to use it. We may then `read()`, `write()`, or perform any other Application Programming Interface-defined action(s). Finally, we need to `close()` the file, indicating that we are no longer using it.

We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary.

Defn 15 (File Attribute)**.** A *file attribute* contains metadata about the file. This includes the file's name, type, protection codes, accounting information, and so on.

Remark. If the system programs are callable by other programs, then each can be considered an Application Programming Interface by other system programs.

1.5.3 Device Manipulation

Defn 16 (Device)**.** A *device* in an Operating System is a resource that must be controlled. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).

A system with multiple users may require us to first `request()` a device, to ensure exclusive use of it. After we are finished with the device, we `release()` it. These functions are similar to the `open()` and `close()` system calls for files. Other operating systems allow unmanaged access to devices. The hazard then is the potential for device contention and perhaps Deadlock.

Once the device has been requested (and allocated to us), we can `read()`, `write()`, just as we can with files. In fact, the similarity between I/O devices and files is so great that many operating systems, including UNIX, merge the two into a combined file–device structure. In this case, a set of system calls can be shared between both files and Devices. Sometimes, I/O devices are identified by special file names, directory placement, or file attributes.

1.5.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the User program and the Operating System. For example, most systems have a system call to return the current `time()` and `date()`. Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to `dump()` memory. A program `trace` lists each system call as it is executed. In addition, the Operating System keeps information about all its processes, and System Calls are used to access this information.

1.5.5 Communications

Both of the models discussed are common in operating systems, and most systems implement both. Message Passing is useful for exchanging smaller amounts of data, because no conflicts need to be avoided. It is also easier to implement than is shared memory for intercomputer communication. Shared-Memory allows maximum speed and convenience of communication, since it can be done at memory transfer speeds when it takes place within a computer. Problems exist, however, in the areas of protection and synchronization between the processes sharing memory.

1.5.5.1 Message Passing Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known. Each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The `get_processid()` system call does this translation. The identifiers are then passed to general-purpose `open()` and `close()` calls provided by the file system or to specific `open_connection()` and `close_connection()` system calls, depending on the model of communication. The recipient process usually must give its permission for communication to take place with an `accept_connection()` call.

Most processes that will be receiving connections are special-purpose Daemons. They execute a `wait_for_connection()` call and are awakened when a connection is made. The source of the communication, known as the client, and the receiving Daemon, known as a server, then exchange messages by using `read_message()` and `write_message()` system calls. The `close_connection()` call terminates the communication.

1.5.5.2 Shared-Memory

In the shared-memory model, `shared_memory_create()` and `shared_memory_attach()` system calls are used by processes to create and gain access to regions of memory owned by other processes. The operating system tries to prevent one process from accessing another process's memory, so shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data is determined by the processes and is not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

1.5.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multiprogrammed computer systems with several Users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection.

1.6 System Programs

Another aspect of a modern system is its collection of system programs.

Defn 17 (System Program)**.** *System programs*, also known as *system utilities*, provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls. Others are considerably more complex. They can be divided into these categories:

File Management These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

Status Information Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a registry, which is used to store and retrieve configuration information.

File Modification Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.

Programming-Language Support Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.

Program Loading and Execution Once a program is assembled or compiled, it must be loaded into memory to be executed. Debugging systems for either higher-level languages or machine language are needed as well.

Communications These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.

Background Services All general-purpose systems have methods for launching certain System Program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. These are typically called Daemons, and systems have dozens of them. In addition, operating systems that run important activities in user context rather than in kernel context may use Daemons to run these activities.

1.7 Operating System Design and Implementation

One important principle is the separation of Policy from Mechanism.

Defn 18 (Mechanism)**.** A *mechanism* determines how to do something.

Defn 19 (Policy)**.** A *policy* determines **what** will be done given the Mechanism works correctly.

The separation of Policy and Mechanism is important for system flexibility. Policies are likely to change across places or over time. In the worst case, each change in policy would require a change in the underlying mechanism. A general mechanism insensitive to changes in policy would be more desirable. A change in policy would then require redefinition of only certain parameters of the system. For instance, consider a mechanism for giving priority to certain types of programs over others. If the mechanism is properly separated from policy, it can be used either to support a policy decision that I/O-intensive programs should have priority over CPU-intensive ones or to support the opposite policy.

Most Operating Systems were built with assembly. However, in recent times (since the invention of C), they have been built with higher-level languages. The advantages of using a higher-level language, or at least a systems-implementation language, for implementing operating systems are the same as those gained when the language is used for application programs:

- The code can be written faster
- Is more compact
- Is easier to understand and debug

In addition, improvements in compiler technology will improve the generated code for the entire operating system by simple recompilation. Finally, an Operating System is far easier to port—to move to some other hardware — if it is written in a higher-level language.

Defn 20 (Port)**.** A *port* is the process of moving a piece of software that was written for one piece of Hardware to another. In some cases, this only requires a recompilation of the higher-level software. In others, it may require completely rewriting the program.

Remark 20.1 (Port Confusion)*.* It is important to note that the Port is ***NOT*** the same thing as a Port.

1.8 Operating System Structure

A system as large and complex as a modern operating system must be engineered carefully if it is to function properly and be modified easily.

1.8.1 Monolithic Approach

Many operating systems do not have well-defined structures. Frequently, such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was originally designed and implemented by a few people who had no idea that it would become so popular. It was written to provide the most functionality in the least space, so it was not carefully divided into modules.

Defn 21 (Monolithic Kernel)**.** A *monolithic kernel* is an Operating System architecture where the entire operating system is working in Kernel space, and typically uses only its own memory space to run. The monolithic model differs from other operating system architectures (such as the Microkernel) in that it alone defines a high-level virtual interface over computer hardware. A set of System Calls implement all Operating System services such as process management, concurrency, and memory management.

Device drivers can be added to the Kernel as Kernel Modules.

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail.

However, this was partly because MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

1.8.2 Layered Approach

With proper hardware support, Operating Systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS and UNIX systems. The Operating System can then retain much greater control over the computer and over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular Operating Systems. Under a top-down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

1.8.2.1 How to Make Modular Kernels A system can be made modular in many ways. One method is the layered approach, in which the Operating System is broken into a number of layers. The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface.

1.8.2.2 How to Use Modular Kernels

A typical operating-system layer, layer M consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can ***ONLY*** invoke operations on lower-level layers and itself.

1.8.2.3 Advantages of Modular Kernels

The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so that each uses functions and services of only lower-level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system. Once the first layer is debugged, its correct functioning can be assumed while the second layer is debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified. Each layer is implemented only with operations provided by lower-level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do.

1.8.2.4 Disadvantages of Modular Kernels

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. Even with planning, there can be circular dependencies created between layers. For example, the backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types.

1.8.3 Microkernels

This method structures the Operating System by removing all nonessential components from the Kernel and implementing them as system and user-level programs, resulting in a smaller Kernel. There is little consensus regarding which services should remain in the kernel and which should be implemented in user space. Typically, however, microkernels provide minimal process and memory management, in addition to a communication facility.

Defn 22 (Microkernel)**.** A *microkernel* (often abbreviated as μ-kernel) is the near-minimum amount of software that can provide the mechanisms needed to implement an Operating System. These mechanisms include:

- Low-level address space management
- Thread management
- Inter-Process Communication (IPC)

If the hardware provides multiple rings or CPU modes, the microkernel may be the only software executing at the most privileged level, which is generally referred to as supervisor or kernel mode. Traditional Operating System functions, such as device drivers, protocol stacks and file systems, are typically removed from the microkernel itself and are instead run in user space.

In terms of the source code size, microkernels are often smaller than monolithic kernels.

The main function of the Microkernel is to provide communication between the client program and the various services that are also running in user space. Communication is provided through Message Passing.

One benefit of the Microkernel approach is that it makes extending the Operating System easier. All new services are added to user space and consequently do not require modification of the Kernel. When the Kernel does have to be modified, the changes tend to be fewer, because the Microkernel is smaller. The resulting Operating System is easier to port from one hardware design to another. The Microkernel also provides more security and reliability, since most services are running as User— rather than Kernel—processes. If a service fails, the rest of the Operating System remains untouched.

Unfortunately, the performance of Microkernels can suffer due to increased system-function overhead.

1.8.4 Kernel Modules

In this architecture, the Kernel has a set of core components and links in additional services via Kernel Modules, either at boot time or during runtime.

Defn 23 (Kernel Module)**.** A *kernel module* is code that can be loaded into the Kernel image at will, without requiring users to rebuild the kernel or reboot their computer. The modular design ensures that you do not have to make and/or compile a complete Monolithic Kernel that contains all code necessary for hardware and situations.

The idea of the design is for the Kernel to provide core services while other services are implemented dynamically, as the Kernel is running. Linking services dynamically is preferable to adding new features directly to the Kernel, which would require recompiling the Kernel every time a change was made. Thus, for example, we might build CPU scheduling and memory management algorithms directly into the Kernel and then add support for different file systems by way of loadable Kernel Modules.

The overall result resembles a Layered Approach in that each Kernel section has defined, protected interfaces. However, it is more flexible than a Layered Approach, because any Kernel Module can call any other Kernel Module. The approach is also similar to the Microkernel approach in that the primary module has only core functions and knowledge of how to load and communicate with other modules. But it is more efficient, because Kernel Modules do not need to invoke Message Passing to communicate.

1.8.5 Hybrid Systems

In practice, very few Operating Systems adopt a single, strictly defined structure. Instead, they combine different structures, resulting in hybrid systems that address performance, security, and usability issues. For example, both Linux and Solaris use Monolithic Kernels, because having the Operating System in a single address space provides very efficient performance. However, they are also modular, so that new functionality can be dynamically added to the Kernel.

Windows uses a Monolithic Kernel as well (again primarily for performance reasons), but it retains some behavior typical of Microkernel systems. It does this by providing support for separate subsystems (known as operating-system personalities) that run as User-mode processes. Windows also provide support for dynamically loadable Kernel Modules.

1.9 Operating System Debugging

1.9.1 Failure Analysis

If a process fails, most Operating Systems write the error information to a log file to alert Users that the problem occurred. The operating system can also take a Core Dump—— and store it in a file for later analysis.

Defn 24 (Core Dump)**.** A *core dump* captures the memory of the process right as it fails and writes it to a disk.

Remark 24.1 (Why Core?). The reason a Core Dump is named the way it is is because memory was referred to as the "core" in the early days of computing.

Running programs and Core Dumps can be probed by a debugger, which allows a programmer to explore the code and memory of a process. Operating-system kernel debugging is more complex than usual because of:

- The size of the Kernel
- The complexity of the Kernel
- The Kernel's control of the hardware
- The lack of user-level debugging tools.

Defn 25 (Crash)**.** A failure in the Kernel is called a *crash*.

When a Crash occurs, error information is saved to a log file, and the memory state is saved to a Crash Dump.

Defn 26 (Crash Dump)**.** When a Crash occurs in the Kernel, a *crash dump* is generated. This is like a Core Dump, in that the entire contents of that process's Memory is written to disk, except the Crashed Kernel process is written, instead of a User program.

Operating-system debugging and process debugging frequently use different tools and techniques due to the very different nature of these two tasks.

1.9.2 Performance Tuning

Performance tuning seeks to improve performance by removing processing bottlenecks. To identify bottlenecks, we must be able to monitor system performance. Thus, the Operating System must have some means of computing and displaying measures of system behavior. In a number of systems, the operating system does this by producing trace listings of system behavior. All interesting events are logged with their time and important parameters and are written to a file. Later, an analysis program can process the log file to determine system performance and to identify bottlenecks and inefficiencies. Traces also can help people to find errors in operating-system behavior.

Another approach to performance tuning uses single-purpose, interactive tools that allow users and administrators to question the state of various system components to look for bottlenecks. One such tool employs the UNIX command `top` to display the resources used on the system, as well as a sorted list of the "top" resource-using processes.

1.10 System Boot

The procedure of starting a computer by loading the Kernel is known as booting the system. On most computer systems, a small piece of code known as the Bootloader is the first thing that runs.

Defn 27 (Bootloader)**.** The *bootloader* (or bootstrap loader) is a bootstrap program that:

1. Locates the Kernel
2. Loads the Kernel into main memory
3. Starts the Kernel's execution

Some computer systems, such as PCs, use a two-step process in which a simple Bootloader fetches a more complex boot program from disk, which in turn loads the Kernel.

When a CPU receives a reset event, the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial Bootloader program. This program is in the form of read-only memory (ROM), because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

Remark. A reset event on the CPU can be the computer having just booted, or it has been restarted, or the reset switched was flipped.

The Bootloader can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It also initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the Operating System. Cellular phones, tablets, and game consoles store the entire operating system in ROM. Storing the operating system in ROM is suitable only for:

- Small operating systems
- Simple supporting hardware
- Ensuring rugged operation

A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips.

All forms of ROM are also known as Firmware. A problem with firmware in general is that executing code there is slower than executing code in RAM. Some systems store the Operating System in firmware and copy it to RAM for fast execution.

A final issue with Firmware is that it is relatively expensive, so usually only small amounts are available. For large operating systems, or for systems that change frequently, the Bootloader is stored in Firmware, and the Operating System is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that boot block. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution.

More typically, it is simple code (as it fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program. GRUB is an example of an open-source Bootloader program for Linux systems. All of the disk-bound bootstrap, and the Operating System that is loaded, can be easily changed by writing new versions to disk. A disk that has a boot partition is called a boot disk or system disk. Now that the full bootstrap program has been loaded, it can traverse the file system to find the Operating System's Kernel, load it into Memory, and start its execution. It is only at this point that the system is said to be running.

2 CPU Scheduling and Synchronization

Defn 28 (Deadlock)**.** *Deadlock* is when 2 processes require information from each other to continue running. If this happens, neither process will provide the other with its required information, so they will both wait for each other, forever.

3 Network

Defn 29 (Port)**.** A *port* is a single instance of a data flow. There are many different flows of data. These may be from different applications or different instances of the same application. In both TCP and UDP, flows are given unique *port numbers.*

Some of these are standard for particular applications, e.g. port 80 for HTTP (web), port 25 for SMTP (email). The transport protocol uses the port number to deliver data to the correct application.

Remark 29.1 (Port Confusion). It is important to note that the Port is ***NOT*** the same thing as a Port.

A Computer Components

A.1 Central Processing Unit

Defn A.1.1 (Central Processing Unit)**.** The *Central Processing Unit*, *CPU*, is a chip that performs all actions in the computer. It calculates mathematical and logical values and acts based on them. It has several components built onto it, and can be thought of as the "brain" of the computer.

The design of a CPU determines some of the functionality it has. Therefore, more specialized processors can be made for special tasks, and more general processors can be built to handle a wide variety of calculations.

A.1.1 Registers

Defn A.1.2 (Register)**.** A *register* is a data storage mechanism built directly onto the Central Processing Unit. It is several hundred times faster than the system Memory. Registers are generally used when the currently running program is performing calculations. Since they are so fast, they are used as both source and destination operands in instructions.

Remark A.1.2.1. Depending on the Central Processing Unit architecture, there may be cases when Registers behave slightly differently between processors. This is something that can only be found by checking the Central Processing Unit manufacturer's documentation.

A.1.2 Program Counter

A.1.3 Arithmetic Logic Unit

A.1.4 Cache

A.2 Memory

Defn A.2.1 (Memory)**.** *Memory*, or *RAM* (*Random Access Memory*), is a Volatile data storage mechanism. It is directly connected to the Central Processing Unit. This is the location that the Central Processing Unit writes to when it cannot or should not store something in the Central Processing Unit's Registers.

Remark A.2.1.1 (Volatility). Memory is volatile because each of the cells is a small capacitor. In between the clock cycles on the Central Processing Unit and Memory, the capacitors discharge. On the clock cycle, the capacitors are refreshed with electrical power, which does one of 2 things:

1. Keep the data bits the same, 1 to 1.
2. Update the data bits from 0 to 1.

Defn A.2.2 (Volatile)**.** If a data storage mechanism is called *volatile*, it means that once the storage mechanism loses power, the data is lost. This is in contrast to Non-Volatile data storage mechanisms.

A.2.1 Stack

Defn A.2.3 (Call Stack)**.** The *call stack* is an imaginary construct that resembles the traditional stack data structure. It is a way to visualize and organize the way memory is used during the execution of a program and its function/methods. It is filled with Stack Frames. This **does not** hold the code that is used in the function, rather it is everything that is needed for the function to be able to run.

Defn A.2.4 (Stack Frame)**.** A *stack frame*, *activation frame*, or *activation record* are objects that represent necessary portions of a function. These include:

- A Dynamic Link
- Local Variables
- Temporary Variables
- Static Links
- Function Arguments
- Return Address

Additionally, there are 2 registers used as pointers to move around and interact with the stack frame.

1. `FP` is in register `%rbp`. It is the Frame Pointer.
2. `SP` is in register `%rsp`. It is the Stack Pointer.

Defn A.2.5 (Dynamic Link)**.** The *dynamic link* or *dynlink* is a memory address pointer and sits at the bottom of a Stack Frame. It is a pointer back to the previous function's dynamic link. This ensures that any function can find its parent/calling function.

The dynamic link also serves as a means to access any variable that might be needed by this function. To access any variable in *this* function, you can subtract a byte multiple that you need to access the proper value. To access any variable in the calling function, you can add a byte multiple that would correspond to the proper variable.

In the `x86_ 64` architecture, instruction set, and convention that we used, the register `%rbp` was the dynamic link.

Remark A.2.5.1. Note: Due to the conventions we used, when accessing arguments passed to the function, we treated them as local variables, just further down in the call stack. This also means that we need to skip over the Return Address block in memory.

Remark A.2.5.2. Because we used the `%rbp` register to store our current Dynamic Link's address, the dynamic link might also be called the *base pointer*.

Defn A.2.6 (Local Variable)**.** *Local variables* are handled very simply. They get an appropriate amount of memory allocated to them on the stack, and that is it.

There is no way to give a variable a name in assembly. (Usually. Depends on the architecture and instruction set). However, there is no way to name something in memory. But, because the size of all the objects is known at compile-time, allocating the proper amount of memory required by each variable is possible.

Remark A.2.6.1. This holds true for strongly-typed, static, compiled languages, like Java, C, C++, etc. However, Python is slightly different in this regard, and handles it differently.

Defn A.2.7 (Temporary Variable)**.** A *temporary variable* is one that is allocated on this function's Stack Frame while calculating values. Once the calculations are completed, these values are deallocated. These temporary variables can also point to objects on the Heap. When the function has finished running, then these values are deallocated, along with all other Local Variables in use.

For example, since assembly-level addition only allows for 2 operands, but in general, addition can have more than 2 operands in use, there needs to be a way to store the value used in the addition. While we can accumulate and use that value in the addition, the values being added together are *not* modified.

Defn A.2.8 (Static Link)**.** The *static link* is an implicit argument, meaning it ***ALWAYS*** gets pushed onto the stack as a Function Argument, when appropriate.

Appropriate in this context could mean several things:

- When an object is alive, and when it is being acted on by a function.
  - In this case, the static link points to an instance of a class on the Heap.
- When a language allows for nested function declarations.
  - Then the static link points to the Dynamic Link of the outer function
  - This allows access to the outer function's Local Variables like normal, and allows us to go back later.

Defn A.2.9 (Function Argument)**.** *Function argument*s are handled very simply. If a function call takes an argument, then the argument is calculated, and then that argument is pushed onto the stack in *this* Stack Frame.

Remark A.2.9.1. If more than one argument is passed to a function, there are 2 ways to push values onto the Stack Frame:

1. In the order they are passed to the function
   - Say a function with 3 arguments is called, then the stack would have arguments in this order
     (a) argument0 (Lowest memory address)
     (b) argument1
     (c) argument2 (Highest memory address)
2. In reverse order
   - Say the same function is called with the same 3 arguments, then the stack would have arguments in this order
     (a) argument2 (Lowest memory address)
     (b) argument1
     (c) argument0 (Highest memory address)

When the values that were passed need to be accessed, and if the memory sizes of things are known at compile time, then we can calculate how far down we need to go in the stack to find the value. This is done by adding a positive value to the `%rbp`

Defn A.2.10 (Return Address)**.** The *return address* is used by the `%rip` register. It is calculated and pushed onto the Stack Frame stack when the `CALL` macro is used. It is the thing that allows us to jump around in the code from the `text` area of our program.

Remark A.2.10.1. The `%rip` register is the *register instruction pointer*. It holds the value of the *next* instruction to execute. Technically, it is the Program Counter's value.

Defn A.2.11 (Garbage Collection)**.** *Garbage collection* is the act of deallocating objects that may still be on the heap and organizing the heap. Since the heap allocates continuous "blocks" of memory required by an object, the heap may have the necessary memory to allocate an object, but in discontinuous locations.

Defn A.2.12 (Frame Pointer)**.** The *frame pointer* is a pointer that ***ALWAYS*** points to the current function's Dynamic Link. The frame pointer is commonly abbreviated as *FP*. This is the thing that allows us to access Local Variables, Function Arguments, and everything else inside of the Stack Frame. The value is held in the `%rbp` register.

Defn A.2.13 (Stack Pointer)**.** The *stack pointer* is a pointer that ***ALWAYS*** points to the top of the current Stack Frame. The stack pointer is commonly abbreviated as *SP*. This value is held in the `%rsp` register. This is the pointer that allows us to push and pop onto this function's Stack Frame.

Defn A.2.14 (Class Descriptor)**.** The *class descriptor* is a portion of memory set aside for the methods that are in an object.

A.2.2 Heap

Defn A.2.15 (Heap)**.** The *heap* is a memory organization construct that helps visualize the way memory is used during the execution of a program. It is ***SIGNIFICANTLY*** larger than the Call Stack.

In an object-oriented language, this is where instances of classes exist. These objects are allocated on a first-come first-serve basis. When the object is allocated, the amount of memory that the object requires is allocated ***in a continuous block***. The problem is that if the heap has been extensively used, then there might be enough total memory required to allocate an object, but because it is not in a continuous block, the object cannot be allocated.

This makes deallocation more complex, because when they are deallocated, the program might still view the memory as in-use. Additionally, because of the continuous block allocation nature of the heap, it must be organized every once in a while. This is called Garbage Collection.

Remark A.2.15.1. In an object-oriented language, like Java, when objects are stored on the heap, **ONLY** their class fields are stored there. The class methods are stored elsewhere, in the Class Descriptor.

A.3 Disk

Defn A.3.1 (Non-Volatile)**.** If a data storage mechanism is called *non-volatile*, it means that once the storage device loses power, the data is still safely stored. This is in contrast to Volatile data storage mechanisms.

A.4 Fetch-Execute Cycle

B Complex Numbers

Complex numbers are numbers that have both a real part and an imaginary part.

$$z = a \pm bi \tag{B.1}$$

where

$$i = \sqrt{-1} \tag{B.2}$$

Remark (i vs. j for Imaginary Numbers). Complex numbers are generally denoted with either i or j. Since this is an appendix section, I will denote complex numbers with i, to make it more general. However, electrical engineering regularly makes use of j as the imaginary value. This is because alternating current i is already taken, so j is used as the imaginary value instad.

$$Ae^{-ix} = A\left[\cos(x) + i\sin(x)\right] \tag{B.3}$$

B.1 Complex Conjugates

If we have a complex number as shown below,

$$z = a \pm bi$$

then, the conjugate is denoted and calculated as shown below.

$$\overline{z} = a \mp bi \tag{B.4}$$

Defn B.1.1 (Complex Conjugate)**.** The conjugate of a complex number is called its *complex conjugate*. The complex conjugate of a complex number is the number with an equal real part and an imaginary part equal in magnitude but opposite in sign.

The complex conjugate can also be denoted with an asterisk ($*$). This is generally done for complex functions, rather than single variables.

$$z^* = \overline{z} \tag{B.5}$$

B.1.1 Complex Conjugates of Exponentials

$$\overline{e^z} = e^{\overline{z}} \tag{B.6}$$

$$\overline{\log(z)} = \log(\overline{z}) \tag{B.7}$$

B.1.2 Complex Conjugates of Sinusoids

Since sinusoids can be represented by complex exponentials, as shown in Appendix C.2, we could calculate their complex conjugate.

$$\begin{aligned}\overline{\cos(x)} &= \cos(x) \\ &= \frac{1}{2}\left(e^{ix} + e^{-ix}\right)\end{aligned} \tag{B.8}$$

$$\begin{aligned}\overline{\sin(x)} &= \sin(x) \\ &= \frac{1}{2i}\left(e^{ix} - e^{-ix}\right)\end{aligned} \tag{B.9}$$

C Trigonometry

C.1 Trigonometric Formulas

$$\sin(\alpha) + \sin(\beta) = 2\sin\left(\frac{\alpha+\beta}{2}\right)\cos\left(\frac{\alpha-\beta}{2}\right) \tag{C.1}$$

$$\cos(\theta)\sin(\theta) = \frac{1}{2}\sin(2\theta) \tag{C.2}$$

C.2 Euler Equivalents of Trigonometric Functions

$$e^{\pm j\alpha} = \cos(\alpha) \pm j\sin(\alpha) \tag{C.3}$$

$$\cos(x) = \frac{e^{jx} + e^{-jx}}{2} \tag{C.4}$$

$$\sin(x) = \frac{e^{jx} - e^{-jx}}{2j} \tag{C.5}$$

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2} \tag{C.6}$$

$$\cosh(x) = \frac{e^{x} + e^{-x}}{2} \tag{C.7}$$

C.3 Angle Sum and Difference Identities

$$\sin(\alpha \pm \beta) = \sin(\alpha)\cos(\beta) \pm \cos(\alpha)\sin(\beta) \tag{C.8}$$

$$\cos(\alpha \pm \beta) = \cos(\alpha)\cos(\beta) \mp \sin(\alpha)\sin(\beta) \tag{C.9}$$

C.4 Double-Angle Formulae

$$\sin(2\alpha) = 2\sin(\alpha)\cos(\alpha) \tag{C.10}$$

$$\cos(2\alpha) = \cos^2(\alpha) - \sin^2(\alpha) \tag{C.11}$$

C.5 Half-Angle Formulae

$$\sin\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1-\cos(\alpha)}{2}} \tag{C.12}$$

$$\cos\left(\frac{\alpha}{2}\right) = \sqrt{\frac{1+\cos(\alpha)}{2}} \tag{C.13}$$

C.6 Exponent Reduction Formulae

$$\sin^2(\alpha) = \frac{1-\cos(2\alpha)}{2} \tag{C.14}$$

$$\cos^2(\alpha) = \frac{1+\cos(2\alpha)}{2} \tag{C.15}$$

C.7 Product-to-Sum Identities

$$2\cos(\alpha)\cos(\beta) = \cos(\alpha-\beta) + \cos(\alpha+\beta) \tag{C.16}$$

$$2\sin(\alpha)\sin(\beta) = \cos(\alpha-\beta) - \cos(\alpha+\beta) \tag{C.17}$$

$$2\sin(\alpha)\cos(\beta) = \sin(\alpha+\beta) + \sin(\alpha-\beta) \tag{C.18}$$

$$2\cos(\alpha)\sin(\beta) = \sin(\alpha+\beta) - \sin(\alpha-\beta) \tag{C.19}$$

C.8 Sum-to-Product Identities

$$\sin(\alpha) \pm \sin(\beta) = 2\sin\left(\frac{\alpha \pm \beta}{2}\right)\cos\left(\frac{\alpha \mp \beta}{2}\right) \tag{C.20}$$

$$\cos(\alpha) + \cos(\beta) = 2\cos\left(\frac{\alpha + \beta}{2}\right)\cos\left(\frac{\alpha - \beta}{2}\right) \tag{C.21}$$

$$\cos(\alpha) - \cos(\beta) = -2\sin\left(\frac{\alpha + \beta}{2}\right)\sin\left(\frac{\alpha - \beta}{2}\right) \tag{C.22}$$

C.9 Pythagorean Theorem for Trig

$$\cos^2(\alpha) + \sin^2(\alpha) = 1^2 \tag{C.23}$$

C.10 Rectangular to Polar

$$a + jb = \sqrt{a^2 + b^2}e^{j\theta} = re^{j\theta} \tag{C.24}$$

$$\theta = \begin{cases} \arctan\left(\frac{b}{a}\right) & a > 0 \\ \pi - \arctan\left(\frac{b}{a}\right) & a < 0 \end{cases} \tag{C.25}$$

C.11 Polar to Rectangular

$$re^{j\theta} = r\cos(\theta) + jr\sin(\theta) \tag{C.26}$$

D Calculus

D.1 Fundamental Theorems of Calculus

Defn D.1.1 (First Fundamental Theorem of Calculus)**.** The *first fundamental theorem of calculus* states that, if f is continuous on the closed interval $[a, b]$ and F is the indefinite integral of f on $[a, b]$, then

$$\int_a^b f(x)\,dx = F(b) - F(a) \tag{D.1}$$

Defn D.1.2 (Second Fundamental Theorem of Calculus)**.** The *second fundamental theorem of calculus* holds for f a continuous function on an open interval I and a any point in I, and states that if F is defined by

$$F(x) = \int_a^x f(t)\,dt,$$

then

$$\begin{aligned} \frac{d}{dx}\int_a^x f(t)\,dt &= f(x) \\ F'(x) &= f(x) \end{aligned} \tag{D.2}$$

Defn D.1.3 (argmax)**.** The arguments to the *argmax* function are to be maximized by using their derivatives. You must take the derivative of the function, find critical points, then determine if that critical point is a global maxima. This is denoted as

$$\operatorname*{argmax}_x$$

D.2 Rules of Calculus

D.2.1 Chain Rule

Defn D.2.1 (Chain Rule)**.** The *chain rule* is a way to differentiate a function that has 2 functions multiplied together.

If

$$f(x) = g(x) \cdot h(x)$$

then,

$$\begin{aligned} f'(x) &= g'(x) \cdot h(x) + g(x) \cdot h'(x) \\ \frac{df(x)}{dx} &= \frac{dg(x)}{dx} \cdot g(x) + g(x) \cdot \frac{dh(x)}{dx} \end{aligned} \tag{D.3}$$

E Laplace Transform

Defn E.0.1 (Laplace Transform)**.** The *Laplace transformation* operation is denoted as $\mathcal{L}\{x(t)\}$ and is defined as

$$X(s) = \int\limits_{-\infty}^{\infty} x(t)e^{-st}dt \tag{E.1}$$

References

[CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linus Device Drivers. Where the Kernel Meets the Hardware*. English. 3rd ed. O'Reilly Media, Feb. 2005. 633 pp. ISBN: 9780596005900.

[KR88] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language*. English. 2nd ed. Prentic Hall Software Series, Mar. 1988. 288 pp. ISBN: 9780133086218.

[SGP13] Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts*. 9th ed. Wiley Publishing, Jan. 2013. 944 pp. ISBN: 9781118129388.