EDAF35: Operating Systems — Reference Sheet
Lund University

Karl Hallsby

Last Edited: March 29, 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.

In modern systems, we tend to have much more than a single Central Processing Unit core. In these multicore/multiprocessor systems, there are 2 ways to organize the way jobs and cores are handled.

1. Symmetric Multiprocessor System

2. Asymmetric Multiprocessor System

Defn 5 (Symmetric Multiprocessor System)**.** In a *symmetric multiprocessor system*, there are multiple Central Processing Units working together. What makes this symmetric is that all CPUs are equal, **there is no single coordinating CPU**. This means that in a 4 CPU system, all 4 CPUs are peers and can work together.

Any CPU can do anything at any time, no matter what any other core is doing.

This is in contrast to an Asymmetric Multiprocessor System.

Defn 6 (Asymmetric Multiprocessor System)**.** An *asymmetric multiprocessor system* has multiple Central Processing Units working together. However, to coordinate all the calculations and operations, **a single CPU is designated the master CPU**. Then, all the other CPUs are slave/worker CPUs.

This is in contrast to an Symmetric Multiprocessor System.

Defn 7 (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.

Additionally, we can have multiple ways of working with system Memory. The 2 main ways are:

1. Uniform Memory Access
2. Non-Uniform Memory Access

Defn 8 (Uniform Memory Access)**.** In *Uniform Memory Access* (*UMA*), **ALL** system Memory is accessed the same way by **ALL** cores.

Defn 9 (Non-Uniform Memory Access)**.** In *Non-Uniform Memory Access* (*NUMA*), some cores have to behave differently to access some Memory.

Defn 10 (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 10.1 (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 11 (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 11.1 (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 12 (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 12.1 (Other Names)*.* On other, non-UNIX systems, Daemons are called other names. They can be called *services*, *subsystems*, or anything of that nature.

Defn 13 (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 14 (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 15 (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 16 (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 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.

1.6 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.

1.7 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.

1.7.1 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.

1.7.2 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.

1.7.2.1 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.

1.7.2.2 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.

1.7.2.3 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.

1.7.2.4 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.

1.7.3 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.

1.7.4 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.

1.7.5 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.

1.8 Operating System Debugging

1.8.1 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.

1.8.2 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.

1.9 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 C Programming

C is one of the lowest "high level" languages you can use today. It provides very minimal abstractions from hardware and assembly code, but allows you to relatively good typechecked code.

2.1 Memory Allocation

Because C is a language that does not provide many abstractions, it also requires the programmer to remember and manage their memory usage. So, **YOU** must be the one to manage the memory, there is **NO** built-in garbage collector for you to use.

Memory allocation is done on the heap of the program's execution space in memory. When you allocate memory in your program, you are actually requesting the operating system to give you the memory you want.

2.1.1 `malloc`

This is the simplest function of all possible memory allocation functions. `malloc`:

- Takes one argument:
  1. The number of bytes to allocate.
- Returns a **POINTER** to the front of the allocated memory.

`malloc` ***DOES NOT*** initialize memory, so it will be garbage.

2.1.2 `calloc`

This is quite similar to malloc. `calloc`:

- Takes 2 arguments:
  1. The number of spaces to allocate, for example the number of elements in an array.
  2. The number of bytes to allocate, for the type being stored.
- Returns a **POINTER** to the front of the allocated memory.

`calloc` ***ZEROS*** memory, so this does have a slight performance penalty.

2.1.3 `realloc`

`realloc` is used to **REALLOCATE** an existing memory location.

- Takes 2 arguments:
  1. The pointer to the memory location previously allocated with either `malloc` or `calloc`.
  2. The amount of memory to reallocate, in bytes.
- If the `NULL` pointer is passed to `realloc`, it will behave exactly like `malloc`.
- Returns a **POINTER** to the front of the reallocated memory

2.1.4 `free`

`free` is used to free memory that was previously allocated, removing from the programming space entirely.

- Takes 1 argument:
  1. A pointer to the memory to be deallocated.
- Returns `void`.

3 System Calls

Defn 28 (System Call)**.** Software may trigger an interrupt by executing a special operation called a *system call*. This can also be called a monitor call. A system call is a messaging interface between applications and the Kernel, with the applications issuing various requests and the Kernel fulfilling them or returning an error.

System calls provide an interface to the services made available by an Operating System. These services are a set of interfaces by which Processes running in User-space can interact with the system. These interfaces give User-level applications:

- Controlled access to hardware
- A mechanism with which to create new Processes
- A mechanism to communicate with existing ones
- The capability to request other Operating System resources

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 be written using assembly.

Remark 28.1 (Syscall)*.* In UNIX and UNIX-like systems, System Call is usually shortened to *syscall*.

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

System Calls provide a layer between the hardware and User-space Processes. This layer serves three primary purposes.

1. It provides an abstracted hardware interface for User-space programs. When reading or writing from a file, applications do not have to be concerned with the type of disk, media, or even the type of filesystem on which the file resides.
2. System Calls ensure system security and stability. With the Kernel acting as a middle-man between system resources and User-space, the Kernel can arbitrate access based on permissions, Users, and other criteria. This arbitration prevents applications from incorrectly using hardware, stealing other Processes' resources, or otherwise doing harm to the system.
3. There is a single common layer between User-space and the rest of the system allows for the virtualized system provided to Processes.

	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 3.1: System Calls in Unix and Windows

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

Defn 29 (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.

The system call interface in Linux, as with most UNIX systems, is provided in part by the C library. The C library implements the main Application Programming Interface on Unix systems, including the standard C library and the system

call interface. The C library is used by all C programs and is easily wrapped by other programming languages for use in their programs. POSIX is composed of a series of standards from the IEEE that aim to provide a portable Operating System standard roughly based on UNIX.

3.1 How to Use Syscalls

System Calls (often called Syscalls in Linux) are typically accessed via functionsdefined in the standard C library. These functions can define an arbitrary number of arguments and might[1] result in one or more side effects, for example writing to a file or copying some data into a provided pointer.

System Calls also provide a return value of type `long` that signifies success or error. Usually, though not always, a negative return value denotes an error. A return value of zero is usually (but again, not always) a sign of success.

The C library, when a System Call returns an error, writes a special error code into the global `errno` variable. This variable can be translated into human-readable errors via library functions such as `perror()`.

Finally, System Calls have well-defined behavior. For example, the System Call `getpid()` is defined to return an integer that is the current Process's PID. However, the definition of behavior says nothing of the implementation to achieve this behavior. The Kernel must provide the intended behavior of the System Call but is free to do so with whatever implementation it wants as long as the result is correct.

3.2 How are Syscalls Defined?

In this section, we will be analyzing the `getpid()` System Call. It is defined to return an **integer** (to the User-space) that represents the current Process's PID. The implementation of `getpid()` is shown below.

```
1 SYSCALL_DEFINE0(getpid) {
2         return task_tgid_vnr(current); // returns current->tgid
3 }
```

`SYSCALL_DEFINE0` is a macro that defines a system call with no parameters (hence the 0). The expanded code looks like this:

```
1 asmlinkage long sys_getpid(void)
```

System Calls have a strict definition.

1. The asmlinkage modifier on the function definition is a directive to tell the compiler to look only on the stack for this function's arguments. **This is a required modifier for all system calls.**
2. The function returns a `long`. For compatibility between 32- and 64-bit systems, system calls defined to return an `int` in User-space return a `long` in the Kernel.
3. The `getpid()` System Call is defined as `sys_getpid()` in the Kernel.
   - This is the naming convention taken with all System Calls in Linux.
   - System Call `bar()` is implemented in the Kernel as function `sys_bar()`.

3.2.1 Syscall Numbers

Defn 30 (Syscall Number)**.** Each system call is assigned a unique *syscall number*. This number is used to reference a specific System Call. When a User-space process executes a System Call, the syscall number identifies which syscall was executed; the Process does not refer to the syscall by name.

After a Syscall Number has been assigned, it cannot change, or already-compiled applications will break. Likewise, if a System Call is removed, its syscall number cannot be recycled, or previously compiled code would aim to invoke one System Call but would invoke another. Linux does provide a "not implemented" System Call, `sys_ni_syscall()`, which does nothing except `return -ENOSYS`, the error corresponding to an invalid System Call. This function is used in the rare event that a syscall is removed or otherwise made unavailable.

The Kernel keeps a mapping of all registered System Calls in the *system call table*, stored in `sys_call_table`. **This table is architecture-dependent.** This table assigns each valid syscall to a unique syscall number.

[1] Nearly all system calls have a side effect (they result in some change of the system's state). A few syscalls, such as `getpid()`, do not have side effects and just return data from the Kernel.

3.2.2 Syscall Performance

System calls in Linux are very fast. This is because of:

- Linux's fast context switch times; entering and exiting the kernel is a streamlined and simple affair
- The simplicity of the system call handler
- The simplicity of the individual system calls themselves

3.3 Syscall Handler

It is not possible for User-space applications to simply execute a Kernel-function call to a function existing in Kernel-space because the Kernel exists in a protected memory space. If applications could directly read and write to the Kernel's address space, system security and stability would be nonexistent.

Instead, User-space applications must somehow signal to the Kernel that they want to execute a System Call and have the system switch to Kernel mode, where the System Call can be executed in Kernel-space by the Kernel on behalf of the application. The mechanism to signal the Kernel is a Trap, a software interrupt. Incur a Trap, and the system will switch to Kernel-mode and execute the exception handler. However, in this case, the exception handler is actually the system call handler

The important thing to note is that, somehow, User-space causes an exception or trap to enter the Kernel.

3.3.1 Denoting Correct Syscall

Simply entering Kernel-space alone is not sufficient because multiple System Calls exist, all of which enter the Kernel in the same manner. Thus, the Syscall Number must be passed into the Kernel, usually through a Register.

3.3.2 Parameter Passing

In addition to the Syscall Number, most System Calls require that one or more parameters be passed to them. Somehow, User-space must relay the parameters to the Kernel during the trap. There are 2 main ways to do this.

1. The easiest way is to store the parameters in registers
2. If there are not enough registers, or the parameter will not fit in a single register, one is filled with a pointer to User-space memory where all the parameters are stored.

The return value is sent back to User-space also by a register.

3.4 Syscall Functions

3.4.1 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.

3.4.2 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 31 (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.

3.4.3 Device Manipulation

Defn 32 (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.

3.4.4 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.

3.4.5 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.

3.4.5.1 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, PID, 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.

3.4.5.2 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.

3.4.6 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.

3.5 Syscall Implementation

The actual implementation of a system call in Linux does not need to be concerned with the behavior of the system call handler. The hard work lies in designing and implementing the system call; registering it with the kernel is simple.

3.5.1 Implementing Syscalls

The first step in implementing a system call is defining its purpose.

- What will it do?
  - The syscall should have exactly one purpose.
  - Multiplexing syscalls (a single system call that does wildly different things depending on a flag argument) is discouraged in Linux.
- What are the new system call's arguments, return value, and error codes?
  - The system call should have a clean and simple interface with the smallest number of arguments possible.
  - The semantics and behavior of a system call are important; they must not change, because existing applications will come to rely on them.
- Be forward thinking; consider how the function might change over time.
- Can new functionality be added to your system call or will any change require an entirely new function?
- Can you easily fix bugs without breaking backward compatibility?
- Will you need to add a flag to address forward compatibility?
  - Many system calls provide a flag argument to address forward compatibility.
  - The flag is not used to multiplex different behavior across a single system call—as mentioned, but to **enable new** functionality and options without breaking backward compatibility or needing to add a new system call.
- Design the system call to be as general as possible.
  - Do not assume its use today will be the same as its use tomorrow.
  - The purpose of the system call will remain constant but its uses may change.
- Is the system call portable?
  - Do not make assumptions about an architecture's word size or endianness.

3.5.2 Parameter Verification

System calls must carefully verify all their parameters to ensure that they are valid and legal. The system call runs in kernel-space, and if the user can pass invalid input into the kernel without restraint, the system's security and stability can suffer.

For example, file I/O syscalls must check whether the file descriptor is valid. Process-related functions must check whether the provided PID is valid. Every parameter must be checked to ensure it is not just valid and legal, but correct. Processes must not ask the kernel to access resources to which the process does not have access.

One of the most important checks is the validity of any pointers that the user provides. Imagine if a process could pass any pointer into the kernel, unchecked, even passing a pointer to which the kernel-calling process did not have read access! Processes could then trick the kernel into copying data for which they did not have access permission, such as data belonging to another process or data mapped unreadable. Before following a pointer into user-space, the system must ensure that:

- The pointer points to a region of memory in user-space. Processes must not be able to trick the kernel into reading data in kernel-space on their behalf.
- The pointer points to a region of memory in the process's address space. The process must not be able to trick the kernel into reading someone else's data.
- The process must not be able to bypass memory access restrictions. If reading, the memory is marked readable. If writing, the memory is marked writable. If executing, the memory is marked executable.

Kernel code must never blindly follow a pointer into user-space!

The kernel provides two methods for performing the requisite checks and the desired copy to and from user-space. One of these two methods must always be used.

1. For writing **to** user-space, the function `copy_to_user()` is provided. It takes three parameters.
   (a) The first is a pointer to the destination memory address in the process's address space.
   (b) The second is a pointer to the source pointer in kernel-space.
   (c) The third is the size, in bytes, of the data to copy.
2. For reading **from** user-space, the method `copy_from_user()` is analogous to `copy_to_user()`. It also takes 3 parameters.
   (a) The first is a pointer to the destination memory address in Kernel-space.
   (b) The second is a pointer to the source memory address in the Process's address space.
   (c) The third is the size, in bytes, of the data to read.

Both of these functions return the number of bytes they failed to copy on error. On success, they return zero. It is standard for the syscall to return `-EFAULT` in the case of such an error.

Both `copy_to_user()` and `copy_from_user()` may block. This occurs if the page containing the user data is not in physical memory but is swapped to disk. In that case, the process sleeps until the page fault handler can bring the page from the swap file on disk into physical memory.

A final possible check is for valid permission. In older versions of Linux, it only `root` could perform these actions. Now, a finer-grained "capabilities" system is in place.

The new system enables specific access checks on specific resources. A call to `capable()` with a valid capabilities flag returns nonzero if the caller holds the specified capability and zero otherwise. See `<linux/capability.h>` for a list of all capabilities and what rights they entail. For example, `capable(CAP_SYS_NICE)` checks whether the **caller** has the ability to modify nice values of other processes.

By default, the superuser possesses all capabilities and nonroot possesses none.

3.5.3 Final Steps in Binding a Syscall

After the System Call is written, it is trivial to register it as an official System Call:

1. 1. Add an entry to the end of the system call table. This needs to be done for each architecture that supports the System Call (which, for most calls, is all the architectures). The position of the syscall in the table, starting at zero, is its Syscall Number. For example, the tenth entry in the list is assigned syscall number nine.
2. For each supported architecture, define the syscall number in `<asm/unistd.h>`.
3. Compile the syscall into the Kernel image (as opposed to compiling as a module). This can be as simple as putting the System Call in a relevant file in `kernel/`.

Look at these steps in more detail with a fictional System Call, `foo()`. First, we want to append `sys_foo()` to the system call table, and record its Syscall Number. This number is the zero-indexed location of the new `sys_foo()` function. For most architectures, the table is located in `entry.S`. Although it is not explicitly specified, the System Call is implicitly given the next subsequent syscall number.

For each architecture you want to support, the System Call must be added to the architecture's system call table. Usually you would want to make the System Call available to each architecture, so it must be placed in each architecture's system call table. The System Call does not need to receive the same syscall number under each architecture. Then, the Syscall Number is added to `<asm/unistd.h>`.

Because the System Call must be compiled into the core Kernel image in all configurations, so the `sys_foo()` function must be placed somewhere in `kernel/*.c`. You should put it wherever the function is most relevant. For example, if the function is related to scheduling, you could define it in `kernel/sched.c`.

3.5.4 Why NOT Implement a Syscall

The previous sections have shown that it is easy to implement a new System Call, but that in no way should encourage you to do so. Often, much more viable alternatives to providing a new System Call are available.

Let's look at the pros, cons, and alternatives. The pros of implementing a new interface as a syscall are:

- They are simple to implement and easy to use.
- Their performance on Linux is fast.

The cons:

- You need a syscall number, which needs to be officially assigned to you.
- After the System Call is in a stable series Kernel, it is written in stone. The interface cannot change without breaking user-space applications, which Linux explicitly disallows.
- Each architecture needs to separately register the System Call and support it.
- System Calls are not easily used from scripts and cannot be accessed directly from the filesystem.
- Because you need an assigned syscall number, it is hard to maintain and use a System Call outside of the master Kernel tree.
- For simple exchanges of information, a System Call is overkill.

The alternatives:

- Implement a device node and `read()` and `write()` to it. Use `ioctl()` to manipulate specific settings or retrieve specific information.
- Certain interfaces, such as semaphores, can be represented as file descriptors and manipulated as such.
- Add the information as a file to the appropriate location in `sysfs`.

3.6 Syscall Context

The Kernel is in Process-context during the execution of a System Call. The current pointer points to the current task, which is the Process that issued the System Call.

In Process-context, the Kernel is capable of sleeping (for example, if the system call blocks on a call or explicitly calls `schedule()`) and is fully preemptible. The capability to sleep means that system calls can make use of the majority of the Kernel's functionality to simplify its own programming. The fact that Process context is preemptible implies that, like User-space, the current task may be preempted by another task. Because the new task may then execute the same System Call, care must be exercised to ensure that the calls are reentrant. Of course, this is the same concern that Symmetric Multiprocessor Systems introduce.

When the System Call returns, control continues in `system_call()` , which ultimately switches to User-space and continues the execution of the User process.

4 Process Management

Defn 33 (Process)**.** A *process* is a Program in the midst of execution, and all its related resources. In fact, two or more processes can exist that are executing the same program. Processes are, however, more than just the executing program code (often called the text section in Unix). They also include a set of resources such as open files and pending signals, internal Kernel data, processor state, a memory address space with one or more memory mappings, one or more Threads of execution, and a data section containing global variables.

Processes, in effect, are the living result of running program code.

Defn 34 (Program)**.** A *program* is object code stored on some media, typically as a File. These contain the instructions that the processor will execute when the program is running as a Process. These instructions are stored in what is called the *text section* of the program. It also contains statically allocated information, such as `static` variables.

Occassionally, Threads can be subject to Preemption.

Defn 35 (Preemption)**.** *Preemption* is the act of temporarily interrupting a task being carried out by a computer system, without requiring its cooperation, and with the intention of resuming the task at a later time. Such changes of the executed task are known as context switches. It is normally carried out by a privileged task or part of the system known as a preemptive scheduler, which has the power to preempt, or interrupt, and later resume, other tasks in the system.

In any given system design, some operations performed by the system may not be preemptible. This usually applies to Kernel functions and service interrupts which, if not permitted to run to completion, would tend to produce race conditions resulting in deadlock.

On modern Operating Systems, Processes provide two virtualizations:

1. a virtualized processor
   - The virtual processor gives *this* Process the illusion that it alone monopolizes the system, despite possibly sharing the processor among hundreds of other processes.
2. Virtual memory lets the process allocate and manage memory as if it alone owned all the memory in the system.
   - Threads share the virtual memory abstraction, whereas each receives its own virtualized processor.

4.1 The Process Life Cycle

A Process begins its life when, the `fork()` System Call is called. This creates a new Process by duplicating an existing one. The Process that calls `fork()` is the **parent**, whereas the new Process is the **child**. The `fork()` System Call returns from the kernel twice: once in the **parent** and once in the newborn **child**. The parent resumes execution and the child starts execution at the same place, where the call to `fork()` returns.

Often, immediately after a `fork` it is desirable to execute a new, different Program. The `exec()` family of function calls creates a new address space and loads a new program into it.

Finally, a program exits via the `exit()` System Call. This function terminates the Process and frees all its resources. A parent Process can inquire about the status of a terminated child via the `wait4()` System Call, which enables a Process to wait for the termination of a specific Process. When a Process exits, it is placed into a special zombie state that represents terminated Processes until the parent calls `wait()` or `waitpid()` .

4.2 The Process Descriptor and Task `struct`

Defn 36 (Task List)**.** The Kernel stores the list of Processes in a circular doubly-linked list called the *task list*. Each element in the task list is a Process Descriptor of the type `struct task_struct` .

Defn 37 (Process Descriptor)**.** The *process descriptor* contains all the information about a specific Process, including:

- Open files
- The Process's address space,
- Pending signals,
- The Process's state,
- The Process's priority
- The Process's policy
- The Process's parent
- The Process's id (PID)

In Linux, the process descriptor is of type `struct task_struct` , which is defined in `<linux/sched.h>`.

4.2.1 Allocating the Process Descriptor

Like in any other programming language, the `task_struct` record must be initialized somehow. This is done with the *slab allocator* to provide object reuse and Cache Coloring.

Defn 38 (Cache Coloring)**.** *Cache coloring* (also known as page coloring) is the process of attempting to allocate free pages that are contiguous from the CPU cache's point of view, in order to maximize the total number of pages cached by the processor. Cache coloring is typically employed by low-level dynamic memory allocation code in the operating system, when mapping virtual memory to physical memory. A virtual memory subsystem that lacks cache coloring is less deterministic with regards to cache performance, as differences in page allocation from one program run to the next can lead to large differences in program performance.

4.2.2 Storing the Process Descriptor

The system identifies Processes by a unique Process Identification Value or PID. The PID is a numerical value represented by the opaque type[2] `pid_t` , which is typically an `int` . Because of backward compatibility with earlier UNIX and Linux versions, the default maximum value is only 32,768 (that of a `(short int)`), although the value can be increased as high as four million (this is controlled in `<linux/threads.h>`). The Kernel stores this value as PID inside each Process Descriptor. This maximum value is important because it is essentially the maximum number of Processes that may exist concurrently on the system.

Inside the Kernel, tasks are typically referenced directly by a pointer to their `task_struct` structure. In fact, most Kernel code that deals with Processes works directly with `struct task_struct` . Consequently, it is useful to be able to quickly look up the Process Descriptor of the currently executing task, which is done via the current macro. This macro must be independently implemented by each architecture. Some architectures save a pointer to the `task_struct` structure of the currently running Process in a register, enabling for efficient access. Other architectures make use of the fact that `struct thread_info` is stored on the Kernel stack to calculate the location of `thread_info` and subsequently the `task_struct` .

[2] "An opaque type is a data type whose physical representation is unknown or irrelevant" Love 2010, pg. 26.

4.2.3 Process State

The state field of the process descriptor describes the current condition of the process. Each process on the system is in exactly one of five different states. This value is represented by one of five flags:

1. `TASK_RUNNING` : The process is runnable; it is either currently running or on a runqueue waiting to run. This is the only possible state for a Process executing in User-space; it can also apply to a process in Kernel-space that is actively running.
2. `TASK_INTERRUPTIBLE` : The Process is sleeping (that is, it is blocked), waiting for some condition to exist. When this condition exists, the Kernel sets the Process's state to `TASK_RUNNING` . The Process also awakes prematurely and becomes runnable if it receives a signal.
3. `TASK_UNINTERRUPTIBLE` : This state is identical to `TASK_INTERRUPTIBLE` **except that it does not wake up and become runnable if it receives a signal**. This is used in situations where the Process must wait without interruption or when the event is expected to occur quite quickly. Because the task does not respond to signals in this state, `TASK_UNINTERRUPTIBLE` is less often used than `TASK_INTERRUPTIBLE` .
4. `__TASK_TRACED` : The Process is being traced by another Process, such as a debugger, via `ptrace`.
5. `__TASK_STOPPED` : Process execution has stopped; the task is not running nor is it eligible to run. This occurs if the task receives the `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, or `SIGTTOU` signal or if it receives any signal while it is being debugged.

4.2.4 Manipulating the Current Process's State

Kernel code often needs to change a process's state. The preferred mechanism is using

```
1 set_task_state(task, state); /* set task 'task' to state 'state' */
```

This function sets the given `task` to the given `state`. If applicable, it also provides a memory barrier to force ordering on other processors. This is only needed on SMP systems.

4.2.5 Process Context

Normal program execution occurs in User-space. When a program executes a system call or triggers an exception, it enters Kernel-space. At this point, the Kernel is said to be "executing on behalf of the process" and is in Process-context. When in process context, the `current` macro is valid.

Remark. Other than process context there is Interrupt-context. In interrupt context, the system is not running on behalf of a process but is executing an interrupt handler. No Process is tied to interrupt handlers.

Upon exiting the kernel, the Process resumes execution in User-space, unless a higher-priority process has become runnable in the interim. If that happens, the scheduler is invoked to select the higher priority process. System Calls and exception handlers are well-defined interfaces into the kernel. A Process can begin executing in kernel-space only through one of these interfaces. All access to the Kernel is through these interfaces.

4.2.6 Process Family Tree

All processes are descendants of the init Process, whose PID is one. The kernel starts `init` in the last step of the boot process. The `init` process, in turn, reads the system initscripts and executes more programs, eventually completing the boot process.

Every process on the system has exactly one parent. Likewise, every process has zero or more children. Processes that are all direct children of the same parent are called siblings. The relationship between processes is stored in the process descriptor. Each `task_struct` has a pointer to the parent's `task_struct` , named `parent`, and a list of children, named `children`.

4.3 Process Creation

Most operating systems implement a `spawn` mechanism to create a new process in a new address space, read in an executable, and begin executing it. UNIX takes the unusual approach of separating these steps into two distinct functions: `fork()` and `exec()` . The first, `fork()` , creates a child process that is a copy of the current task. It differs from the parent only in:

- Its PID (which is unique)
- Its PPID (parent's PID, which is set to the original process)
- Certain resources and statistics, such as pending signals, which are not inherited

The second function, `exec()` , loads a new executable into the address space and begins executing it.

4.3.1 Copy-on-Write

In Linux, `fork()` is implemented through the use of copy-on-write pages. Copy-on-write (or CoW) is a technique to delay or altogether prevent copying of the data. Rather than duplicate the process address space, the parent and the child can share a single read-only copy.

The data, however, is marked in such a way that if it is written to, a duplicate is made and each Process receives their own unique copy. Consequently, the duplication of resources occurs only when they are written; until then, they are shared read-only. This technique delays the copying of each page in the address space until it is actually written to. In the case that the pages are never written—for example, if `exec()` is called immediately after `fork()` —they never need to be copied.

The only overhead incurred by `fork()` is the duplication of the parent's page tables and the creation of a unique Process Descriptor for the child. In the common case that a Process executes a new executable image immediately after forking, this optimization prevents the wasted copying of large amounts of data.

4.3.2 Forking

The bulk of the work in forking is handled by `do_fork()` , which is defined in `kernel/fork.c` , by calling `copy_process()` and then starting the process. The interesting work is done by `copy_process()` :

1. It calls `dup_task_struct()` , which creates a new kernel stack, `thread_info` structure, and `task_struct` for the new process. The new values are identical to those of the current task. At this point, the child and parent Process Descriptors are identical.

2. It then checks that the new child will not exceed the resource limits on the number of processes for the current user.

3. The child needs to differentiate itself from its parent. Various members of the Process Descriptor are cleared or set to initial values. Members of the Process Descriptor not inherited are primarily statistical information. The bulk of the values in `task_struct` remain unchanged.

4. The child's state is set to `TASK_UNINTERRUPTIBLE` to ensure that it does not yet run.

5. `copy_process()` calls `copy_flags()` to update the flags member of the `task_struct` . The `PF_SUPERPRIV` flag, which denotes whether a task used superuser privileges, is cleared. The `PF_FORKNOEXEC` flag, which denotes a process that has not called `exec()` , is set.

6. It calls `alloc_pid()` to assign an available PID to the new task.

7. Depending on the flags passed to `clone()` , `copy_process()` either duplicates or shares open Files, filesystem information, signal handlers, process address space, and namespace. These resources are typically shared between Threads in a given Process; otherwise they are unique and copied here.

8. Finally, `copy_process()` cleans up and returns to the caller a pointer to the new child.

Back in `do_fork()` , if `copy_process()` returns successfully, the new child is woken up and run. Deliberately, the kernel runs the child process first.

4.4 Linux Implementation of Threads

Threads are a popular modern programming abstraction. They provide multiple executors within the same program in a shared memory address space. They can also share open files and other resources. Threads enable concurrent programming and, on multiple processor systems, true parallelism.

The Linux kernel is unique in that there is no concept of a Thread. Instead, Linux implements all Threads as standard Processes. The Linux kernel does not provide any special scheduling semantics or data structures to represent Threads. Instead, a Thread is merely a Process that shares certain resources with other Processes. Each Thread has a unique `task_struct` and appears to the kernel as a normal Process which just happen to share resources, such as an address space, with other Processes.

For example, assume you have a Process that consists of four Threads. In Linux, there are simply four Processes and thus four normal `task_struct` structures. The four Processes are set up to share certain resources. The result is quite elegant. However, on systems with explicit Thread support, one Process Descriptor might exist that points to the four different Threads. The Process Descriptor describes the shared resources, such as an address space or open files. The Threads then describe the resources they alone possess.

4.4.1 Creating Threads

Threads are created the same as normal Processes, i.e. `fork()` is used. The difference is that the `clone()` System Call is passed flags corresponding to the specific resources to be shared.

```
clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

The code above results in behavior identical to a normal `fork()`, except that the address space, filesystem resources, file descriptors, and signal handlers are shared. In other words, the new task and its parent are what are popularly called Threads.

The flags provided to `clone()` help specify the behavior of the new process and detail what resources the parent and child will share. Table 4.1 lists the `clone()` flags, which are defined in `<linux/sched.h>`, and their effect.

Flag	Meaning
CLONE_FILES	Parent and child share open files.
CLONE_FS	Parent and child share filesystem information.
CLONE_IDLETASK	Set PID to zero (used only by the idle tasks).
CLONE_NEWNS	Create a new namespace for the child.
CLONE_PARENT	Child is to have same parent as its parent.
CLONE_PTRACE	Continue tracing child.
CLONE_SETTID	Write the TID back to user-space.
CLONE_SETTLS	Create a new TLS for the child.
CLONE_SIGHAND	Parent and child share signal handlers and blocked signals.
CLONE_SYSVSEM	Parent and child share SystemV SEM_UNDO semantics.
CLONE_THREAD	Parent and child are in the same thread group.
CLONE_VFORK	`vfork()` was used and the parent will sleep until the child wakes it.
CLONE_UNTRACED	Do not let the tracing process force `CLONE_PTRACE` on the child.
CLONE_STOP	Start process in the `TASK_STOPPED` state.
CLONE_SETTLS	Create a new TLS (thread-local storage) for the child.
CLONE_CHILD_CLEARTID	Clear the TID in the child.
CLONE_CHILD_SETTID	Set the TID in the child.
CLONE_PARENT_SETTID	Set the TID in the parent.
CLONE_VM	Parent and child share address space.

Table 4.1: `clone()` Flags

4.4.2 Kernel Threads

It is often useful for the kernel to perform some operations in the background. The kernel accomplishes this via kernel threads, standard processes that exist solely in kernel-space.

Defn 39 (Kernel Thread)**.** *Kernel thread*s are like regular Threads, except that they can only be started by the Kernel and its previous kernel threads. Additionally, they do not have an address space (Their `mm` pointer, which points at their address space, is `NULL`.). They operate only in Kernel-space and do not context switch into User-space. Kernel threads are schedulable and preemptable, the same as normal Processes.

Linux delegates several tasks to kernel threads, most notably the `flush` tasks and the `ksoftirqd` task. Kernel threads are created on system boot by other kernel threads. The kernel handles this automatically by forking all new kernel threads off of the `kthreadd` Kernel process. The interface for Kernel Threads is declared in `<linux/kthread.h>`.

When started, a Kernel Thread continues to exist until it calls `do_exit()` or another part of the kernel calls `kthread_stop()`, passing in the address of the `task_struct` structure returned by `kthread_create()`.

4.5 Process Termination

When a process terminates, the Kernel releases the resources owned by the process and notifies the child's parent of its demise. Usually, process destruction is self-induced. It occurs when the process calls the `exit()` System Call. This can be

done either explicitly when it is ready to terminate or implicitly on return from the main subroutine of any program (The C compiler places a call to `exit()` after `main()` returns).

A process can also terminate involuntarily. This occurs when the process receives a signal or exception it cannot handle or ignore.

Regardless of how a process terminates, the bulk of the work is handled by `do_exit()`, defined in `kernel/exit.c`, which completes a number of chores:

1. It sets the `PF_EXITING` flag in the flags member of the `task_struct`.
2. It calls `del_timer_sync()` to remove any kernel timers. Upon return, it is guaranteed that no timer is queued and that no timer handler is running.
3. If BSD process accounting is enabled, `do_exit()` calls `acct_update_integrals()` to write out accounting information.
4. It calls `exit_mm()` to release the `mm_struct` held by this process. If no other process is using this address space, i.e. the address space is not shared, the Kernel then destroys it.
5. It calls `exit_sem()`. If the process is queued waiting for an IPC semaphore, it is dequeued here.
6. It then calls `exit_files()` and `exit_fs()` to decrement the usage count of objects related to file descriptors and filesystem data, respectively. If either usage counts reach zero, the object is no longer in use by any process, and it is destroyed.
7. It sets the Process's exit code, stored in the `exit_code` member of the `task_struct`, to the code provided by `exit()` or whatever Kernel mechanism forced the termination. The exit code is stored here for optional retrieval by the parent.
8. It calls `exit_notify()` to send signals to the Process's parent, reparents any of the Process's children to another thread in their thread group or the init process, and sets the Process's exit state, stored in `exit_state` in the `task_struct` structure, to `EXIT_ZOMBIE`.
9. `do_exit()` calls `schedule()` to switch to a new process. Because the process is now not schedulable, this is the last code the Process will ever execute. `do_exit()` never returns.

At this point, we have a Zombie Process.

Defn 40 (Zombie Process)**.** A *zombie process* in Linux is a process that has been completed, but its entry still remains in the process table due to lack of correspondence between the parent and child Processes. This happens when the Process has been terminated, but the Process Descriptor has **not** be deallocated yet.

- All objects associated with the Process are freed.
  - This assumes that this Process was the only one using these objects, i.e. no other Threads/Processes were using them.
- The Process is not runnable and no longer has an address space in which to run.
- The process is in the `EXIT_ZOMBIE` exit state.
- The only memory it occupies is its Kernel stack, the `thread_info` structure, and the `task_struct` structure.
- The Process exists solely to provide information to its parent. After the parent retrieves the information, or notifies the Kernel that it is uninterested, the remaining memory held by the process is freed and returned to the system for use.

4.5.1 Removing a Process Descriptor

After `do_exit()` completes, the Process Descriptor for the terminated Process still exists, but the Process is a Zombie Process and is unable to run. By remaining a Process, albeit a Zombie Process, this enables the system to obtain information about a child Process after it has terminated.

Consequently, the acts of cleaning up after a Process and removing its Process Descriptor are separate.

After the parent has obtained information on its terminated child, or signified to the Kernel that it does not care, the child's `task_struct` is deallocated. The `wait()` family of functions are implemented via a single System Call, `wait4()`. The standard behavior is to suspend execution of the calling task until one of its children exits, at which time the function returns with the PID of the exited child. Additionally, a pointer is provided to the function that on return holds the exit code of the terminated child.

When it is time to finally deallocate the Process Descriptor, `release_task()` is invoked. It does the following:

1. Calls `__exit_signal()`, which calls `__unhash_process()`, which in turns calls `detach_pid()` to remove the process from the PIDhash and remove the process from the task list.

2. `__exit_signal()` releases any remaining resources used by the now dead process and finalizes statistics and bookkeeping.

3. If the task was the last member of a thread group, and the leader is a zombie, then `release_task()` notifies the zombie leader's parent.

4. `release_task()` calls `put_task_struct()` to free the pages containing the Process's Kernel stack and `thread_info` structure and deallocate the slab cache containing the `task_struct`.

At this point, the Process Descriptor and all resources belonging solely to the process have been freed.

4.5.2 Parentless Tasks

If a parent exits before its children, some mechanism must exist to reparent any child tasks to a new process, or else parentless terminated processes would forever remain Zombie Processes, wasting system memory. The solution is to reparent a task's children on exit to either another Process in the current Thread group or, if that fails, the `init` process.

When a suitable parent for the child(ren) has been found, each child needs to be located and reparented to this `reaper` parent Process.

With the Process(es) successfully reparented, there is no risk of stray Zombie Processes. The `init` process routinely calls `wait()` on its children, cleaning up any zombies assigned to it.

5 Threads

Defn 41 (Thread)**.** *Threads of execution*, often shortened to *threads*, are the objects of activity within the process. Each thread includes:

- Thread ID
- A unique program counter
- Process stack
- Set of processor Registers

The Kernel schedules the individual threads, not Processes.

In traditional UNIX systems, each Process consists of one thread. In modern systems, however, multithreaded programs/processes are common. In this case, this Process's threads share:

- The Code Section
- The Data Section
- Operating System resources, such as files and signals.

Remark 41.1 (Threads in Linux)*.* Linux has a unique implementation of threads; it does not differentiate between Threads and Processes. To Linux, a thread is just a special kind of process.

Threads are very useful in modern programming whenever a Process has multiple tasks to perform independently of the others. The use of Threads is even more aparent when the single process/program must perform many similar tasks. This is particularly true when one of the threads may block, and it is desired to allow the other threads to proceed without blocking.

Figure 5.1 illustrates what a multithreaded application roughly looks like, visually.

The biggest benefits of using multiple Threads are:

1. **Responsiveness**. Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces.

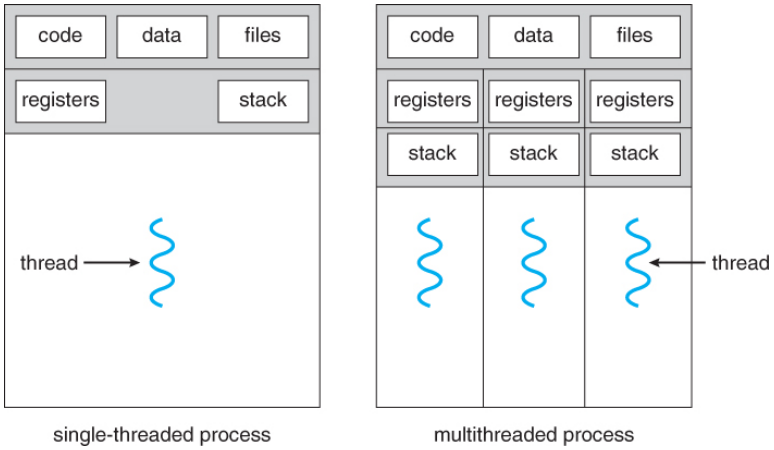

Figure 5.1: Single- vs. Multithread Diagram

2. **Resource sharing**. Processes can only share resources through techniques such as Message Passing and Message Passing. Using these techniques requires explicit arrangement by the programmer. However, threads share the memory and the resources of the process to which they belong, allowing an application to have several different threads of activity within the same address space.
3. **Economy**. Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.
4. **Scalability**. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

Defn 42 (Parallelism)**.** *Parallelism* is where a system can perform more than one task simultaneously.

Defn 43 (Concurrency)**.** *Concurrency* supports more than one task executing simulataneiously, and allows all the tasks to make progress. Thus, it is possible to have concurrency without Parallelism.

To handle the increase in Process Thread counts, many CPUs support more than one thread per core. This means multiple threads can be loaded into the CPU for faster switching. On desktop Intel CPUs, this is called **hyperthreading**.

The biggest difficulties in using multiple Threads are:

1. **Identifying tasks**. Examine applications to find areas that can be divided into independent tasks that can be run concurrently on individual cores.
2. **Balance**. While identifying tasks that can run in parallel, programmers must also ensure that the tasks perform equal work of equal value. In some instances, a certain task may not contribute as much value to the overall process as other tasks.
3. **Data splitting**. Just as applications are divided into separate tasks, the data accessed and manipulated by the tasks must be divided to run on separate cores.
4. **Data dependency**. The data accessed by the tasks must be examined for dependencies between two or more tasks. When one task depends on data from another, the tasks must be synchronized to accommodate the data dependency.
5. **Testing and debugging**. When a program is running in parallel on multiple cores, many different execution paths are possible, making testing and debugging much harder.

There are 2 distinct types of Parallelism, though many applications use both of them.

1. Data Parallelism
2. Task Parallelism

Defn 44 (Data Parallelism)**.** *Data parallelism* focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core. For example, summing the contents of an array of size N. On a single-core system, one thread would simply sum the elements $[0] \dots [N-1]$. On a dual-core system, however, thread A, running on core 0, could sum the elements $[0] \dots [\frac{N}{2} - 1]$ while thread B, running on core 1, could sum the rest of the elements $[\frac{N}{2}] \dots [N-1]$.

Defn 45 (Task Parallelism)**.** *Task parallelism* involves distributing tasks/Threads across multiple computing cores. Each thread is performing a unique operation. Different threads may be operating on the same data, or they may be operating on different data.

5.1 User and Kernel Threads

A relationship must exist between User Threads and Kernel Threads. There are three common ways of establishing such a relationship:

1. The Many-To-One Model
2. The One-To-One Model
3. The Many-To-Many Model

Defn 46 (User Thread)**.** *User thread*s are Threads created by a User Process. They are supported above the kernel and are managed without kernel support.

Defn 47 (Kernel Thread)**.** *Kernel thread*s are Threads created by the Kernel. They are supported and managed directly by the operating system.

5.1.1 Many-To-One Model

This model maps many user-level threads to a single Kernel Thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

5.1.2 One-To-One Model

This model maps each user thread to an individual kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors.

The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread, which is quite expensive. The overhead of creating kernel threads can hurt the performance of an application. To combat this, most implementations of this model restrict the number of threads supported by the system. Most desktop operating systems use this model.

5.1.3 Many-To-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).

Whereas the Many-To-One Model allows the developer to create as many User Threads as they wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time. The One-To-One Model allows greater concurrency, but the developer must be mindful of the number of threads within an application.

The many-to-many model suffers from neither of these shortcomings:

- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
- When a thread performs a blocking system call, the kernel can schedule another thread for execution.

5.2 Thread Libraries

Defn 48 (Thread Library)**.** A *thread library* is an Application Programming Interface that allows a programmer to create and manage Threads.

There are 2 main ways to implement a Thread Library:

1. Provide a library entirely in user space with no kernel support. All code and data structures for the library exist in **user-space**. This means that invoking a library function results in a function call in user space and not a system call.
2. Implement a kernel-level library supported directly by the operating system. All code and data structures for the library exist in **kernel-space**. Invoking a library function results in a System Call to the Kernel.

There are 3 different libraries that are used frequently.

1. POSIX Pthreads.
    - This provides a standard interface for **how the threads should behave**.
    - The actual implementation details are left for the implementor to decide.
    - Global variables are shared between threads.
    - Brought in by the standard library's `pthread.h` header.

2. Windows' Threads.
    - Global variables are shared between threads.
    - Must include the `windows.h` header.
3. Java's Threads.

Although there are these different libraries, all of them provide similar functionality. So, throughout this section (unless otherwise specified), all information will be general.

5.2.1 Synchronous/Asynchronous Threading

There are 2 main ways to create multiple threads.

1. Asynchronous
    - Once the parent thread creates a child thread, the parent resumes execution.
    - The parent and child execute concurrently.
    - Each thread is independent.
    - The parent does not need to know if a child terminates.
    - Little data sharing between threads.
2. Synchronous
    - The parent creates one or more children, and then waits for all of them to finish executing.
    - Called the ***fork-join*** strategy.
    - The child threads perform their work concurrently, but the parent **MUST** wait for all children to finish.
    - Typically a lot of data sharing between threads.
    - The program must provide a " `run` "-like function that is the first thing a new thread executes.
        - The actual name of this function varies depending on the Thread Library in use.
    - The programmer must provide a point where the main program waits by having each child thread `join` .
        - In the Pthreads and Java libraries, to wait for multiple threads, a `for` loop is constructed and iterates over all threads, making them `join` .
        - The actual name of this function varies depending on the Thread Library in use.

5.2.2 Thread Attributes

When creating Threads, some information can be passed to the constructing function regarding the attributes for the thread. These attributes include:

- Security Information
- Stack Size
- Flag to indicate if thread starts in a suspended state (Can/cannot be scheduled).

5.3 Implicit Threading

Because programs are starting to use so many Threads that it is becoming hard to manage, the creation and management of threads is moving from developers to compilers/run-time libaries. This way, the computer manages threads rather than the programmer.

Defn 49 (Implicit Threading)**.** *Implicit Threading* is the transfer of Thread creation and management away from the programmer, and to the compiler and/or run-time libaries. This frees the programmer from having to think/worry about the issues that arise because of multithreading, while still allowing programs to take advantage of the benefits of multithreading.

There exist 3 main methods for implementing implicit threading:

1. Thread Pools
2. OpenMP
3. Grand Central Dispatch

5.3.1 Thread Pools

In a Thread Pool system, all (or at least most) of the Threads available for use by a Process are created during startup. They are then placed in a pool, and wait for work to arrive.

Defn 50 (Thread Pool)**.** A *thread pool* system is one where all Threads that can be used by any Process on the system is in a pool, hence the name. When a job comes in that would use one (or more) of these threads, they are pulled out of the pool and allowed to execute. When they finish execution, they return to the pool.

If there are jobs ready, but there are no threads available in the pool, they wait until one is available.

A Thread Pool offers these benefits:

1. Servicing a request with an **existing thread** is faster than waiting **to create a thread**.
2. A thread pool limits the number of threads that exist at any one point, preventing performance degradation. This is particularly important on systems that cannot support a large number of concurrent threads.
3. Separating the task to be performed from the mechanics of creating the task allows us to use different strategies for running the task.

Additionally, the number of Threads available in the Thread Pool can be set dynamically. Some factors that can affect the number of threads in the pool are:

1. Number of CPUs in the system
2. Amount of physical memory
3. Expected number of concurrent job requests

There are more sophisticated architectures that offer varying benefits. Some even allow for the shrinking of the pool as needed, to reduce the footprint of the running Process.

5.3.2 OpenMP

OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared-memory environments. OpenMP allows for parallel programming by identifying parallel regions as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions they know can be executed in parallel, and these directives instruct the OpenMP runtime library to execute the region in parallel.

It can create as many threads as are available in a system with the `#pragma omp parallel` directive. There are also directives for parallelizing loops, automatically dividing the work among the spawned threads.

This system also allows developers to choose the way the OpenMP library behaves, allowing for several different levels of parallelism. These include:

- Setting the number of threads manually.
- Allowing developers to identify whether data is shared between threads or are private to a thread.

5.3.3 Grand Central Dispatch

Grand Central Dispatch (*GCD*) is a Thread Pool system developed by Apple for OSX and iOS. If is a combination of C extensions, an Application Programming Interface, and a runtime library. This allows GCD to use POSIX threads and manage the creation/destruction of threads as needed. This allows developers to identify sections of code that may be run in parallel.

Like in OpenMP, blocks of code that may be parallelized must be identified by the programmer with the `^{ code; }` syntax. When executing, GCD will schedule blocks for execution by placing them on a dispatch queue. When an item from the dispatch queue is removed, it is assigned to a Thread from the queue's Thread Pool.

There are 2 types of dispatch queues:

1. Serial (**Cannot** be executed in parallel).
   - Items are removed in a FIFO order.
   - One block **MUST** finish executing before another can be drawn from the dispatch queue.
   - Each process gets its own serial queue, the *main queue*.
   - Additional serial dipatch queues can be made for local for execution.
2. Concurrent (**Can** be executed in parallel).
   - Items in this dispatch queue are also removed in a FIFO order, but multiple may be removed at once.
   - This allows multiple blocks to execute in parallel.
   - There are 3 system-wide dispatch queues, according to priority.
     (a) Low
     (b) Default
     (c) High
   - Priority is a relative importance of the blocks.

5.4 Threading Issues

In this section, we will discuss some common issues that arise because of multithreading.

5.4.1 The `fork()` and `exec()` System Calls

Since `fork()` creates a separate, but duplicate, child Process from its parent, what are the semantics when creating a Thread on a UNIX system, since Threads are just another kind of Process?

If one thread in a program calls `fork()`, does the new process duplicate all threads, or is the new process single-threaded? To answer this, there are two versions of `fork()`, one that duplicates all threads and another that duplicates only the thread that invoked the `fork()` system call.

The `exec()` system call is relatively unchanged; if a thread invokes the `exec()` system call, the program specified in the parameter to `exec()` will **replace the entire process**, including all threads.

Which version of `fork()` to use depends on the application. If `exec()` will be called immediately after forking, then duplicating **all** threads is unnecessary, as the program specified in the parameters to `exec()` will replace the process anyways, thus duplicating only the calling thread is appropriate. If, however, the separate process does not call `exec()` after forking, the separate process should duplicate all threads.

5.4.2 Signal Handling

Defn 51 (Signal)**.** A UNIX *signal* is used to notify a Process that an event has occurred. The reception of the signal can be synchronous or asynchronous. Synchronous in this context means that the signal is **delivered to the same proces that caused the signal**. Asynchronous means the signal is generated by an event **external to the running process**, such as keyboard presses or timer expiration.

Which one depends on the source of and the reason for the event to be signaled. However, all signals have the same general pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

When a Signal is delivered, it must be handled by either:

1. The default signal handler
2. The user-defined signal handler

Every Signal has a default handler that the Kernel runs to handle the Signal. However, these actions can be overridden by a user-defined signal handler. How a Process responds to a signal depends on the process and the type of signal that is received.

Signal handling in single-threaded processes is simple, because the only Thread is also the Process. However, multithreaded programs have some complications because there are multiple Threads for the single Process.

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

How a Signal is delivered depends on the type of signal generated. Synchronous signals need to be delivered **to the Thread causing the Signal**. Some asynchronous signals need to be delievered to all threads in a Process.

On UNIX and UNIX-like systems, a Thread can specify what Signals it will accept, and which it will block. Thus, the asynchronous signal may be delievered to only the threads that are not blocking that signal. In addition, since signals need to be handled **only once**, they are typically only delivered to the first non-blocking thread found.

5.4.3 Thread Cancellation

Defn 52 (Thread Cancellation)**.** *Thread cancellation* means terminating a Thread before it has completely finished its execution.

Remark 52.1 (Target Thread). The Thread that is to be cancelled is often called the *target thread*.

There are 2 main ways to cancel a Thread:

1. **Asynchronous cancellation**

- One thread immediately terminates the Target Thread.
- Difficult when resources have been allocated to a cancelled Thread.
- Difficult when a thread is updating data that is shared with other threads.
- Operating System will reclaim some, but not all resources from a canceled thread, so a necessary system-wide resource may not be freed.

2. **Deferred cancellation**.
   - Target Thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

To change the way Pthreads handle potential cancellation signals, they can enable and disable various ways to cancel themselves. POSIX Pthreads support 3 different types of modes/states for how a Target Thread will handle a request.

Mode	State	Type
Off	Disabled	—
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

Table 5.1: Pthread Cancellation Modes

The default cancellation type is deferred cancellation, when a Thread has reaced a cancellation point. At this point, a *cleanup handler* is invoked, which frees any resources the Thread may have had allocated to it.

5.4.4 Thread-Local Storage

Threads that belong to a Process all share their data. However, sometimes a thread will need its own copy of data, called *thread-local storage*.

> It is easy to confuse TLS with local variables; however, TLS data is visible throughout a Thread's execution. TLS is similar to `static` data, except each piece of TLS data is unique to each thread.

5.4.5 Scheduler Activations

When using the Many-To-Many Model of Kernel-Thread Library communications, there may be some issues. To handle these issues, many systems implement Lightweight Processes as an intermediary between the User and Kernel.

Defn 53 (Lightweight Process)**.** A *lightweight process* (*LWP*) is a virtual processor onto which the application can schedule a User Thread to run. Then, each lightweight process is attached to a Kernel Thread, and those kernel threads are scheduled by the Operating System to run on real, physical processors.

These Lightweight Processes are used for each of the potentially Kernel-block tasks that may occur. When this happens, the Kernel Thread blocks, which blocks the Lightweight Process, which then blocks the User Thread. An application can use as many Lightweight Processes as it wants, but if the Process only has a limited amount, some of the LWPs may need to be queued.

One way to handle these communications is by using *scheduler activation*. The steps for this to work are shown below.

1. The Kernel provides a Process with a set of Lightweight Processes, LWPs, virtual processors.
2. When a User Thread is about to block, the Kernel makes an *upcall* ot the LWP informing it that it is about to block, and identifies the Thread that will block.
3. The Kernel will then allocate a new LWP to the Process to run the *upcall handler* on. This saves the state of the blockeing thread, schedules another thread, and context switches to the other thread.
4. When the blocking event that the Thread was waiting on, the Kernel makes another *upcall* to the LWP informing the blocked thread that it may run and unblocks it.
5. The unblocked User Thread can now be scheduled. It eventually is scheduled to execute and does.

6 CPU Scheduling and Synchronization

Defn 54 (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.

7 Network

Defn 55 (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 55.1 (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.

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

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

$$\underset{x}{\operatorname{argmax}}$$

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

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

References

[CRK05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux 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.

[Lov10] Robert Love. *Linux Kernel Development. A thorough guide to the design and implementation of the Linux kernel*. English. 3rd ed. Addison-Wesley, Mar. 2010. 467 pp. ISBN: 9788131758182.

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