EDAF35: Operating Systems — Reference Sheet
Lund University

Karl Hallsby

Last Edited: April 22, 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-Kernel 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

The operating system runs in Kernel-mode, meaning it is allowed to interact with every part of the system, without protection. This allows the OS: to load User programs into User-memory, to dump for errors, to perform I/O, to Context Switch and store the information required for such a thing, and in general, function.

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.

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.

[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.

	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

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.

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()` .

4.1.1 Process States

Every Process exists in a state that describes how the process can and will behave.

- **New**: The process is in the stage of being created.
- **Ready**: The process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions.
- **Running**: The CPU is working on this process's instructions.
- **Waiting**: The process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur.
- **Terminated**: The process has completed.

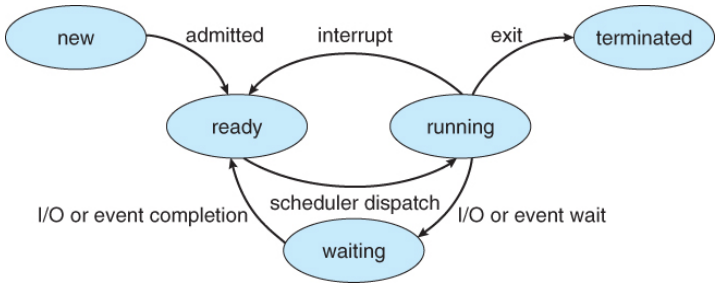


Figure 4.1: Process Life Cycle

4.1.2 Process Control Blocks and Context Switching

The information that is saved during a Context Switch (An example is shown in Figure 4.2) is saved in the Process Control Block.

Defn 36 (Context Switch)**.** A *context switch* is performed when a computer is performing several different Processes in sequence. A context switch involves saving the state of the currently running Process into a Process Control Block, and then loading a waiting process from its Process Control Block. After saving the information, every register in the CPU has its values changes to the new control blocks values, and the CPU continues execution.

A visualization of a context switch is shown in Figure 4.2.

Defn 37 (Process Control Block)**.** The *process control block* (*PCB*) contains **ALL** the state information (the processor's context) needed for a CPU to perform a context switch, either because of Preemption or because the process terminated. The information that the PCB contains includes:

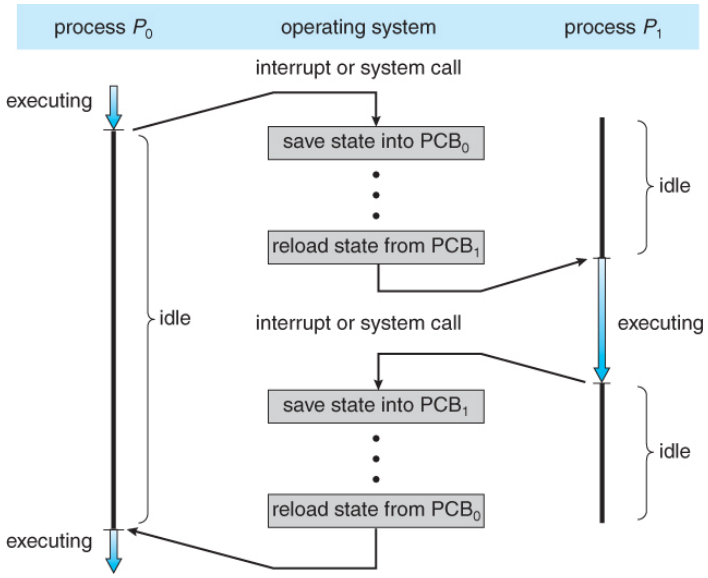


Figure 4.2: Diagram of a Context Switch

- Process State: Running, waiting, etc., as discussed in Section 4.1.1
- Process ID (PID), and parent process ID (PPID).
- CPU registers and Program Counter: Saved and restored when swapping processes in and out of the CPU.
- CPU-Scheduling information: Priority information and pointers to scheduling queues.
- Memory-Management information: Page tables or segment tables.
- Accounting information: User and Kernel CPU time consumed, account numbers, limits, etc.
- I/O Status information: Devices allocated, open file tables, etc.

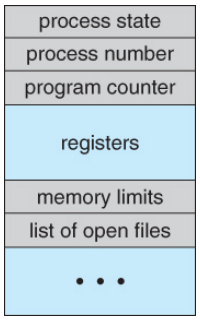


Figure 4.3: Process Control Block

Remark. In Figure 4.3, the Process Number field is the PID of the Process.

The time spent performing a context-switch is pure overhead, because the system does no useful work while switching. Switching speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (a single instruction to load or store all registers). A typical speed is a few milliseconds. Context-switch times are also highly dependent on hardware support.

The more complex the operating system, the greater the amount of work that must be done during a context switch. In addition, advanced memory-management techniques may require that extra data be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for use.

4.2 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.

4.2.1 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.

4.2.2 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.

`fork()` returns `0` in the newly created child process, and the PID of the child in the parent. Then, `exec()` **COMPLETELY REPLACES THE PROCESS' MEMORY**, meaning that after an `exec()`, the child is not executing the same program anymore.

4.3 Process Scheduling

With multiprogramming, the objective is to have **some** Process running at all times. Note that we are discussing a computer with a single CPU, and likely a single Thread here. Time sharing makes the CPU Context Switch so frequently that users can interact with each program and it seems like they are all running concurrently.

Defn 38 (Process Scheduler)**.** The *process scheduler* is responsible for selecting an available process (one in the New, Ready, or Running states discussed in Section 4.1.1) from one of possibly many queues to run next.

4.3.1 Scheduling Queues

When a Process enters the system, it is put in the **Job Queue**, which consists of **EVERY** process on the system.

The processes that are in main memory, are ready, and waiting to execute are in the **Ready Queue**. The Process Control Blocks are kept in a doubly linked list, with the first and last PCBs explicitly marked by the list itself and a pointer to the next PCB in the ready queue in each PCB.

There are other queues as well:

- **Device Queue**: Completion of I/O Request(s)
  - Each device has its own queue.

These various queues can be represented by Figure 4.4.

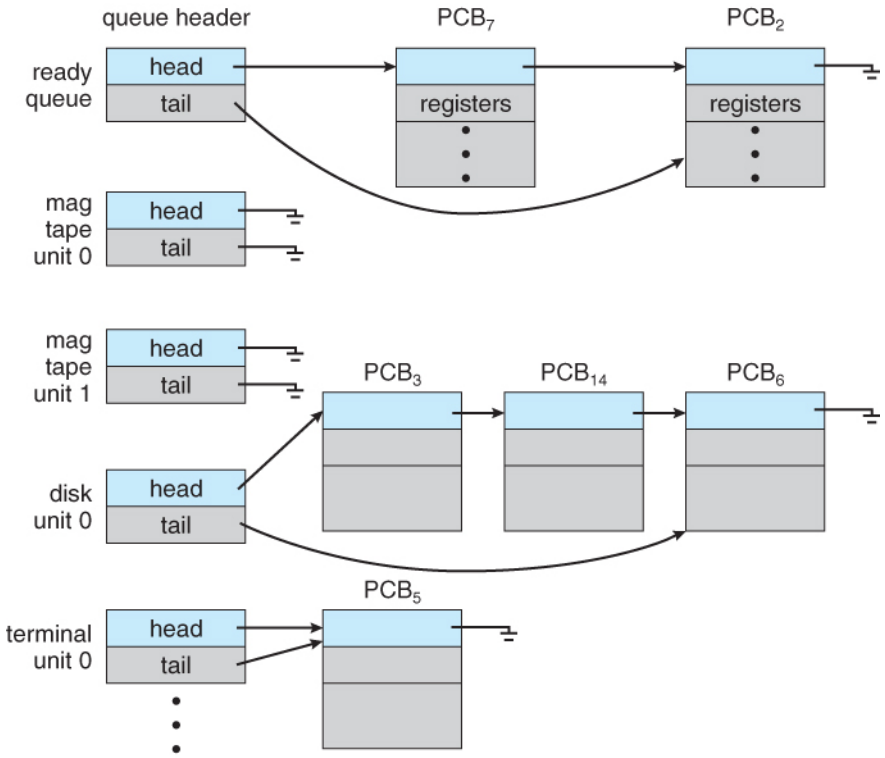


Figure 4.4: Various System Queues

A good visualization of how Processes and their queues work is shown in Figure 4.5.

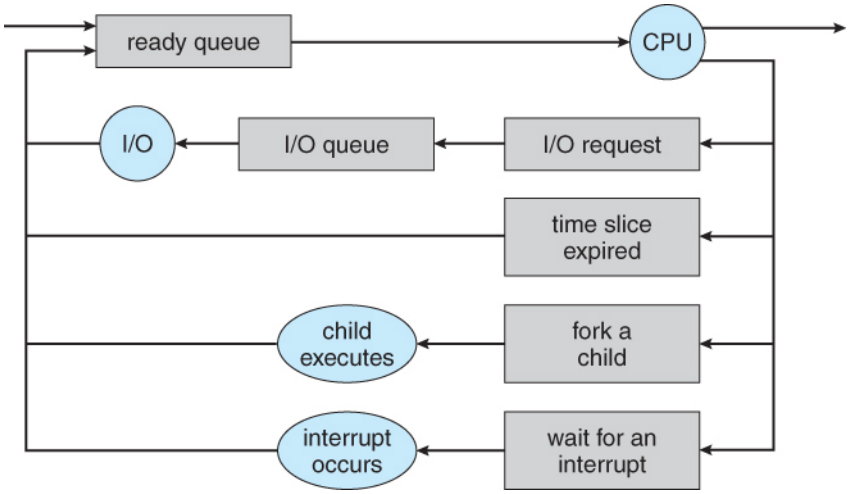


Figure 4.5: Queuing Diagram for Process Scheduling

4.3.2 Schedulers

Defn 39 (Scheduler)**.** A *scheduler* is responsible for selecting the Process from one of the queues to execute next.

Typically, many more Processes are submitted at once than can be handled immediately. So, some are sent to a mass-storage device where they are keyp for later scheduling by the *long-term scheduler* or *job scheduler*. The *short-term scheduler*, or the *CPU scheduler* selects from the Processes that are ready to execute and allocates the CPU to them. The CPU scheduler is run every hundred milliseconds, whereas the job scheduler may be run every few minutes. This allows the job scheduler to run for longer periods of time, because it is active for less time overall.

The job scheduler determines the *degree of multiprogramming*, by determining how many Processes can live in memory at any given time. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Most desktop Operating Systems do not use the long-term scheduler. Instead they put all tasks into the short-term queue, and rely on human behavior to help control the business of the systems.

Defn 40 (I/O Bound)**.** An *I/O Bound* Process is one where most of the process's time is spent performing I/O rather than computations.

Defn 41 (CPU-Bound)**.** A *CPU-Bound* Process spends most of its time performing computations.

It is important to have a good mix of I/O Bound and CPU-Bound processes, so that no single queue is ever too full, improving overall system throughput.

There are also *medium-term schedulers* that perform *swapping*. This scheduler removes a Process from memory, store it somewhere, then reintroduce the process to memory again later.

4.4 The Process Descriptor and Task `struct`

Defn 42 (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 43 (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>`.

4.4.1 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 44 (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.

4.4.2 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.

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

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`.

4.4.3 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.

4.4.4 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.

4.4.5 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.

4.4.6 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`.

4.5 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 45 (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 46 (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 46.1 (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.

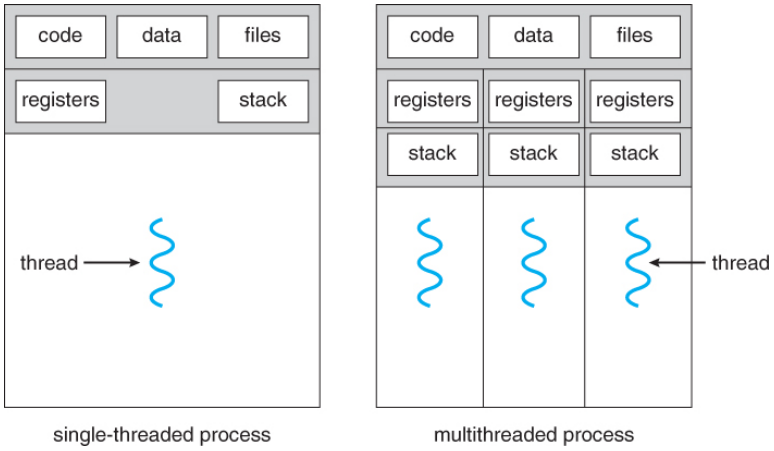


Figure 5.1: Single- vs. Multithread Diagram

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

Defn 48 (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 49 (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 50 (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 51 (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 52 (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 the Kernel-space and do not context switch into User-space. Kernel threads are schedulable and preemptable, the same as normal Processes.

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 53 (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 54 (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 55 (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 56 (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 57 (Thread Cancellation)**.** *Thread cancellation* means terminating a Thread before it has completely finished its execution.

Remark 57.1 (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 58 (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.

5.5 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.

5.5.1 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.

```
1 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 5.2 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 5.2: `clone()` Flags

5.5.2 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 52 (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 the 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()`.

6 CPU Scheduling and Synchronization

The growing importance of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads, which may be sharing data, are running in parallel on different processing cores. These Processes are called Cooperating Processes.

Defn 59 (Cooperating Process)**.** A *cooperating process* is one that can affect or be affected by other Processes executing on a system. They can share a logical address space (code and data), **Threads**, or can share data through files and/or messages, **Communications**.

Given the way that multiple Threads can be scheduled, namely in any order (relatively speaking), as programmers, we cannot be certain about which thread will be scheduled first. This leads to all sorts of problems because of sharing information between multiple users. The largest, and likely the most common, error in a multiThreaded program is the Race Condition.

Defn 60 (Race Condition)**.** A *race condition* is when several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place. The only way to prevent a race condition is to ensure that **only one Thread can change the value at a time**.

6.1 Process/Thread Synchronization

The main problem that occurs in multiThreaded programs is that there is a small portion of code that is a Critical Section. This leads to the development of the Critical Section Problem.

Defn 61 (Critical Section)**.** The *critical section* of a Process is a portion where the Thread and/or Process is changing common variables, updating a table, writing a file, or other global state changes.

6.1.1 Critical Section Problem

The *Critical Section Problem* is the issue of coordinating multiple Threads about a Critical Section of the code. The problem is to design a protocol that the Processes/Threads can use to cooperate. Each Process must request permission to enter its critical section. The section of code implementing this request is the Entry Section. The critical section may be followed by an Exit Section. The remaining code is the Remainder Section.

Defn 62 (Entry Section)**.** The *entry section* of a Process is the portion where the request to execute the Critical Section occurs. In the case of a Mutex, this is the process of aquiring the it. For a Semaphore, it is the process of manipulating the value it currently contains.

Defn 63 (Exit Section)**.** In the *exit section*, the constructs used to ensure coordination in the Critical Section are freed. In the case of a Mutex, this is the process of releasing the it. For a Semaphore, it is the process of manipulating the value it currently contains in the opposite direction it was initially manipulated by.

Defn 64 (Remainder Section)**.** The *remainder section* is the rest of the code, after this Critical Section. This code may be parallelized, or not. It could contain further Critical Sections.

Any solution to this problem **MUST** satisfy one of the following 3 requirements:

1. **Mutual Exclusion**. If Process P_i is executing its Critical Section, then **no other** processes can execute their critical sections.
2. **Progress**. If no Process is executing its Critical Section, and some processes wish to enter their critical sections, then only those processes that **are not executing** in their Remainder Sections can decide which will enter the Critical Section next. Essentially, the only way a process gets a voice in the choice is by not having executed the critical section yet.

3. **Bounded Waiting**. There exists a bound on the number of times that other Processes are allowed to enter their Critical Sections after a process has made a request to enter its critical section and before that request is granted.

To handle the Critical Section Problem, there are 2 main types of Kernels that present solutions.

1. Nonpreemptive Kernels. Not used frequently today.
2. Preemptive Kernels. The most common type today.

Defn 65 (Nonpreemptive Kernel)**.** A *nonpreemptive kernel* is a Kernel that does **NOT** use Preemption on Processes or Threads running in kernel-mode.

Defn 66 (Preemptive Kernel)**.** A *preemptive kernel* is a Kernel that uses Preemption on Processes or Threads running in kernel-mode. This means that we cannot say anything definitive about the state of the Kernel's data structures at a given time, because we cannot say which process/thread is running at that time.

6.1.2 Hardware Support for Synchronization

Software-based solutions to handling multithreading and multiprocessing tends to be better than hardware-based solutions, as they are more flexible. Many of the solutions that will be presented here are based on the idea of **Locking**.

Defn 67 (Lock)**.** A *lock* allows **only one** Thread to enter the portion of code that is locked. While a thread holds this lock no other Thread can execute on this code portion.

Remark 67.1 (Binary Semaphore). Locks can be represented as *binary Semaphore*s.

In a single-processor system, we can solve the Critical Section Problem by preventing interrupts from being handled. This would prevent the currently running instruction from being preempted in any way, and allow it to finish. However, this does not really work on a multiprocessor system, because disabling interrupts and their handling on all processors is time consuming.

However, the idea of certain instructions being Atomic is an elegant solution to the Critical Section Problem. So, most computer systems provide special hardware-level instructions that allow us to test and modify the contents of a word, or swap the contents of 2 words Atomically.

Defn 68 (Atomic)**.** An *atomic* operation is one that cannot be interrupted, preempted, or altered in any way. As soon as an atomic operation begins, the system **MUST** finish handling it before it may do anything else.

Some operations on data are possible to do at any given point in time, without affecting the potential outcome. One example of this is **reading** from a location in memory. However, if this location can also be written to, we need to limit the number of writers. Additionally, if someone is waiting to write, they should get some priority over anything waiting to read. Thus, the Read/Write Lock was created.

Defn 69 (Read/Write Lock)**.** *Read/Write Lock*s allow either an unlimited number of readers **OR** 1 writer at any given time. Writers will be scheduled to use the lock sooner than readers, so the value is updated first, before anyone reads it again. But, the writer will have to wait until everyone currently reading the value is done reading, otherwise the value in memory will change underneath the readers.

6.1.3 Mutex Locks

The hardware-based solutions presented in Section 6.1.2 are typically not available to application programmers. Instead, operating system designers build software tools to handle the Critical Section Problem. The simplest tool is that of a Mutex.

Defn 70 (Mutex)**.** A *mutex* (short for ***mut****ual* ***ex****clusion*) is the same as a Lock, **but it can be system wide (shared by multiple processes)**. A mutex lock protects critical regions and prevents race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section.

The `acquire()` function acquires the lock, preventing any other Thread and/or Process from using the thing the lock protects. Likewise, the `release()` function releases the lock, allowing another Thread and/or Process take acquire the lock and use the resource it protects. To perform its function correctly, the lock's `acquire()` and `release()` functions must be Atomic.

When a Thread and/or Process attempts `acquire()` the lock, while it is already owned by someone else, it is put in a `WAITING` state.

There are a variety of Mutexes, depending on the way the way they are used is implemented. In one variety, the Spinlock, the Process/Thread that is attempting to `acquire` the lock will continuously call the `acquire()` function until it gets the lock.

Defn 71 (Spinlock)**.** A *spinlock* is one way of having a system use a Mutex lock. In this implementation, the Process/Thread that wants to `acquire` the lock loops continuously, calling `acquire()` until it gets it. This means that there is no Context Switch to another task.

This means that while a Process/Thread is not executing anything useful and may be hogging the CPU's cycles, if the lock is freed soon, then no Context Switch back is required.

6.1.4 Semaphores

Semaphores are a more general way of approaching resource allocation and mutual exclusion within a system. Because of this generality, there are 2 kinds of Semaphores:

1. Counting Semaphore
2. Binary Semaphore

Defn 72 (Semaphore)**.** A *semaphore* regulates the number of things (Processes or Threads) performing operations on something (Shared Resource). Functionally, this is the same as a Mutex but allows x Processes/Threads to enter or use the resource at a time. This allows for limits on the number of CPU, I/O or RAM intensive tasks running at the same time.

A semaphore can only be interacted with through 2 operations `wait()` and `signal()`. `wait()` is similar to a Mutex's `acquire()` function and the `signal()` function is similar to the Mutex's `release()` function. Here, if a Process or Thread `wait`s, if there is more of the resource, then the requester gets the resource, and the internal count of the semaphore is decremented. If a Process or Thread `signal`s, then it is done with the resource, and the requester loses access to the resources, and the internal count of the semaphore is incremented. Again, these manipulations **MUST** be Atomic.

Remark 72.1 (Confusion with Mutexes)*.* Technically, you can create a Semaphore that acts like a Mutex by giving it a binary value. However, this is typically in poor programming taste, because while both function similarly, the Semaphore is for signalling the amount of a resource available and the Mutex is for signalling if code is capable of execution.

Remark 72.2 (Correct Use of Semaphores)*.* The correct use of a Semaphore is for signaling from one task to another. A Mutex is meant to be taken and released, always in that order, by each task that uses the shared resource it protects. By contrast, tasks that use Semaphores either signal or wait, not both.

A Semaphore contains not only its integer value, but also a pointer to the list of waiting Threads/Processes. This way the things waiting on the Semaphore's values can be reached. However, we now have issues with Starvation and Deadlock.

Defn 73 (Starvation)**.** *Starvation* is when a thing is waiting for a resource indefinitely. For example, if a Process/Thread is waiting for a Semaphore to become available, and these requesters are services in a Last-In First-Out order, its possible the first requester to enter the queue never leaves.

Semaphores have some additional issues to go with them:

- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

6.1.4.1 Counting Semaphore A *counting Semaphore* allows for the value contained within the Semaphore to range over a domain of integer values. For example, the integers from 0 to 10, allowing for 10 different users to access the same resource at once.

Once the Semaphore count reaches 0, nothing else can use the resource until one of the current users relinquishes control. So, the first Process/Thread that uses `wait()` when the Semaphore's internal value is 0 will be put in either a Spinlock-like state, or will be put into a waiting queue.

Typically, the Processes/Threads that are using the Semaphore will be qusing it for quite some time, new requesters are put into a waiting queue/state. Then, the Scheduler is called, performing a Context Switch to a Thread/Process that is ready to run, and Context Switches.

Thus, whenever another user of the Semaphore `signal`s, one of the Processes/Threads should be awoken from this waiting state, moved to the ready state, and be made schedule-able. Which waiting thing is chosen depends on the way the queue is structured, and how things get scheduled, and many other factors.

> In this system, we don't guarantee immediate execution of the thing requesting the Semaphore, only that it can be scheduled.

How this queue is structured is different between every Kernel, but it is typically constructed out of the Process Control Block or the Thread control block.

Negative Semaphore Values In some implementations of a Semaphore, the internal value can become negative. In the strict, classical definition of a Semaphore, this internal value **CANNOT** be negative. However, if we allow the negative value, and the Semaphore reaches it, it represents the number of things that are waiting on the Semaphore to be freed up.

6.1.4.2 Binary Semaphore

A *binary Semaphore* only allows the internal value within the system to be zero or one. This effectively acts as a Mutex.

6.1.5 Priority Inversion

Priority Inversion is a scheduling challenge where a higher-priority process needs to read or modify kernel data that is currently being used by a lower-priority process. Since the kernel data is typically protected with a Lock, the higher-priority process must wait for a lower-priority one to finish with that resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

It occurs only in systems with more than two priorities. Typically these systems solve the problem by implementing a Priority-Inheritance Protocol.

Defn 74 (Priority-Inheritance Protocol)**.** A *priority-inheritance protocol* is one where a lower-priority Process that is using resourced needed by a higher-priority process will inherit the higher priority until it is done with the resource. Then, the lower-priority process is returned to its original priority level.

6.1.6 Monitors

Using Semaphores can be difficult, with some common errors being:

- Using `signal()` before the Critical Section and `wait()` afterwards. This will may allow multiple requesters to execute their critical sections at times when they shouldn't.
- Using `wait()` twice, once before the Critical Section and once after. This will cause a Deadlock to occur.
- Forgetting either the `wait()` before the Critical Section or the `signal()` after it. Forgetting the `wait()` will violate mutual exclusion. Forgetting the `signal()` will cause a Deadlock.

To handle these kinds of issues, the Monitor type was created.

Defn 75 (Monitor)**.** A *monitor* is an abstract data type that includes:

- Programmer-defined operations that must be executed with mutual exclusion.
- The variables whose values define the state of an instance of that type.
- The bodies of functions that operate on those varables.

A monitor ensures that only one Process or Thread can execute any actions in the monitor at any given time. Thus, the programmer **does not** need to code the synchronization explicitly.

The general syntax of a monitor type is shown in Listing 1.

The use of `conditions` in the Monitor allow for operations that require the use of those conditions to `wait()` and `signal()` them. This prevents Processes and/or Threads from performing other actions that require those conditions until they are freed. The difference with `signal()` in a Monitor and a Semaphore is that if that condition is already active, **nothing happens**.

In Monitors, just like with Semaphores, when something `wait`s a condition that is already completely used,the requester is put into a queue. How the requester is pulled from the queue depends on the setup of the Monitor

Monitors have some additional issues to go with them:

- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

6.2 Scheduling

CPU scheduling is the basis of multiprogrammed operating systems. In a single-processor system, only one process can run at a time. Others must wait until the CPU is free and can be rescheduled. By switching the CPU among processes, the operating system can maximize CPU utilization.

For example, a Process is executed until it must wait. Typically the process waits for the completion of some I/O request. In a simple computer system, the CPU then just sits idle. All this waiting time is wasted; no useful work is accomplished.

Scheduling of this kind is a fundamental operating-system function. Almost all computer resources are scheduled before use.

```
monitor name {
        condition x;
        condition y;
        /* Shared Variable Declarations */
        function P1 (...) {
                ...
        }

        function P2 (...) {
                ...
        }

        function Pn (...) {
                ...
        }

        initialization_code (...) {
                ...
        }
}
```

Listing 1: Syntax of a Monitor Type

6.2.1 CPU and I/O Bursts

To properly schedule a Process, its CPU Bursts and I/O Bursts need to observed.

Defn 76 (CPU Burst)**.** A *CPU burst* is one of the states of execution for a Process. This is the state when the process is actively using the CPU to perform computations. In this state, the CPU is performing activity for **this** Process, and **IS NOT** waiting for an I/O device to perform some action or return information.

Defn 77 (I/O Burst)**.** An *I/O burst* is one of the states of execution for a Process. This is the state when the process is waiting on the I/O device to return the requested information or perform the desired action. In this state, the CPU is doing no activity for **this** Process.

A Process alternates between these two bursts, with the final CPU Burst terminating this Process's execution. The distribution of length of CPU bursts is an exponential or hyperexponential graph. This means:

- There is a large number of short duration CPU bursts.
- There is a small number of long duration CPU bursts.

We can categorize these into either CPU-Bound programs or I/O Bound programs.

- I/O-bound programs have a small number of CPU bursts which have a relatively short duration relative to the I/O operations. The I/O operations take up a majority of the time the Process executes.
- CPU-bound programs have a large number of CPU bursts, which have a relatively long duration relative to the I/O operations. The CPU operatiosn take up a majority of the time the Process executes.

6.2.2 CPU Scheduler

Whenever the CPU becomes idle, i.e. it has finished the current CPU burst early, or there is an I/O operation, the Operating System must select the next Process and/or Thread to schedule. This is handled by the Short-Term Scheduler.

Defn 78 (Short-Term Scheduler)**.** The *short-term scheduler* is responsible for scheduling either the next Process or Thread for execution on the **CPU** from all the possible ones in memory. This is run quite frequently, every couple hundred milliseconds, usually.

Remark 78.1 (CPU Scheduler). Because the Short-Term Scheduler only schedules tasks for the CPU, it is also called the *CPU Scheduler*.

6.2.2.1 Preemption and Scheduling

There are 4 times when CPU scheduling occurs:

1. When a process switches from the `RUNNING` state to the `WAITING` state.
2. When a process switches from the `RUNNING` state to the `READY` state (for example, when an interrupt occurs).
3. When a process switches from the `WAITING` state to the `READY` state (for example, at completion of I/O).
4. When a process terminates.

In the case of Times 1 and 4, there are no options in terms of scheduling. In the first situation, a Process is being made unschedule-able by the Operating System. So, another process must be switched in for the one that was made to wait. Similarly, when a Process terminates, there is no option for how to schedule it, because it's done executing.

A Nonpreemptive Kernel only allows scheduling during Times 1 and 4. In this system, once a Process gets allocated to the CPU, it keeps the CPU until it terminates, it switches to the `WAITING` state, or it voluntarily yields control of the CPU.

A Preemptive Kernel allows scheduling during all Times (1–4). This can only be done on certain hardware platforms (all major ones today), because a timer is needed, among other special hardware. In this system, Race Conditions appear. We also need to design the Kernel to allow for Preemption, such as when the kernel is busy performing a System Call for a Process. If we don't have to worry about real-time computing, then we can wait for the System Call to finish before moving onto another task.

6.2.2.2 Interrupt Handling

Interrupts can happen **AT ANY TIME**, which must be accepted at almost all times. On multiprocessor systems, it is costly to turn off interrupt handling on all cores, but sometimes it is necessary. If we don't, input might be lost or output overwritten. To ensure that these code sections are not accessed concurrently by several processes, they disable interrupts at entry and reenable interrupts at exit. The sections of code that disable interrupts do not occur often and typically contain few instructions.

6.2.2.3 Dispatcher

The Dispatcher **MUST** be as fast as possible to minimize the amount of time spent working on switching betweeen Processes/Threads. This is measured as the Dipatch Latency.

Defn 79 (Dispatcher)**.** The *dispatcher* is a Short-Term Scheduler function. It is the code responsible for giving control of the CPU to the Process that the scheduler selected. This involves:

1. Performing a Context Switch.
2. Switching to the appropriate mode, User-mode or Kernel-mode.
3. Jumping to the proper location in the Process to continue its execution.

Defn 80 (Dipatch Latency)**.** The *dispatch latency* is the amount of time required by the Dispatcher to stop one Process and switch to another.

6.2.3 Scheduling Criteria

The choice of how to schedule tasks for the CPU is completely dependent on the desired outcome of the scheduling algorithm and the desired performance of the system. The criteria for most algorithms are:

- **CPU utilization**. We want to keep the CPU as busy as possible. In a real system, it utilization should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).
- **Throughput**. If the CPU is busy executing processes, then work is being done. The number of processes that are completed per time unit is called throughput.
- **Turnaround time**. From the point of view of a process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. This includes time spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time**. The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O, only the amount of time that a process spends in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time**. In an interactive system, turnaround time may not be the best criterion. A process can produce some output early and then continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until its response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time. Usually the average measure is optimized. However, sometimes it is better to optimize the minimum or maximum values rather than the average.

6.2.4 Scheduling Algorithms

Here, we will discuss the different kinds of Scheduling Algorithms used by the Short-Term Scheduler.

Defn 81 (Scheduling Algorithm)**.** The *scheduling algorithm* is responsible for choosing the next Process/Thread to run from the set of all `READY` processes.

6.2.4.1 First-Come First-Served Scheduling In this algorithm, the first task that becomes `READY` is the first to execute. This is easily handled with a FIFO queue. The most recent task to appear in the task queue is appended as the tail of the FIFO queue's list, and moves forward until it gets to execute. Once it has its chance to execute, this Process will execute for its entire period, making this a nonpreemptive scheduling method.

The problem with FCFS scheduling is the average waiting time for Processes, especially if there is a large difference in the amount of CPU time required by some of the processes. This is illustrated visually in Figure 6.1.

Example 6.1: FCFS Calculations.

Suppose 3 processes arrive at the same time and have these CPU bursts required.

Process	Time Required
P_1	24
P_2	3
P_3	3

In a FCFS system, the average waiting time is

$$\frac{0+24+27}{3} = 17\,\text{ms}$$

However, if we were to schedule the short processes first, then the average waiting time would be

$$\frac{0+3+6}{3} = 3\,\text{ms}$$

Defn 82 (Gantt Chart)**.** A *Gantt chart* is a way to visualize the order and amount of time a processor spends executing a task. An example of one is shown in Figure 6.1. The start and end times of a particular task are also shown on the chart.

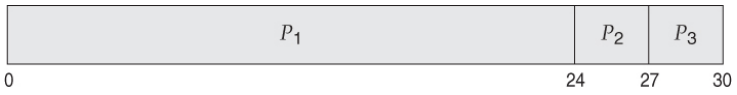


Figure 6.1: Gantt Chart

Additionally, if there are a lot of small I/O tasks and just a few large CPU ones, then the Convoy Effect starts.

Defn 83 (Convoy Effect)**.** The *convoy effect* is a result of having few very CPU-intensive tasks that use the CPU fro their entire duration and a lot of small I/O-intensive tasks. There, the CPU-Bound task will block the I/O Bound tasks. This lowers overall throughput and device utilization.

6.2.4.2 Shortest-Job-First Scheduling Here, the job with the shortest amount of time required for the next CPU burst is scheduled first. If 2 Processes have the same amount of time required, then the tie is broken with FCFS among those tasks. This can be either preemptive or nonpreemptive. The question comes up when a newly arrived Process has a shorter CPU burst than the one currently running. If this new process preempts the currently running one, then it is a preemptive SJF scheduler, and is sometimes called *Shortest-Remaining-Time-First scheduling.*

Note that this is the length of the **next CPU burst**, not the overall length of CPU execution.

Example 6.2: SJF Calculations.

Suppose 4 processes arrive at the same time with their CPU bursts as shown.

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

In this case, these processes would be scheduled in this order.

1. P_4
2. P_1
3. P_3
4. P_2

With an average delay of

$$\frac{0+3+9+16}{4} = 7\,\text{ms}$$

In a FCFS system, the average delay would be 10.25 ms.

This is provably optimal, theoretically making it the best algorithm. However, it is hard for a Process to know the length of its next CPU request. There is also no way to know the length of the next CPU burst.

However, SJF scheduling is used in the long-term scheduler, because time limits can be set on the execution of various tasks, essentially forcibly giving a value to the next CPU burst.

To calculate the length of the next CPU burst, we tend to approximate it as an exponential average of the previous CPU bursts. This is shown in Equation (6.1).

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n \tag{6.1}$$

τ_{n+1}: The predicted value of the next CPU burst.
t_n: The length of the nth CPU burst.
α: A value to determine how much we take the past into account. $0 \leq \alpha \leq 1$.

- If $\alpha = 0$, then the most recent CPU burst has no effect, and only the longer past is considered.
- If $\alpha = 1$, then **ONLY** the most recent CPU has any effect, and the older past is ignored.

6.2.4.3 Priority Scheduling Priority scheduling is the idea that each Process has a priority associated with it that determines which process should be scheduled next. The process with the highest priority is scheduled next. In fact, the Shortest-Job-First Scheduling is one case of the priority scheduling system.

We discuss scheduling in terms of **high priority** and **low priority**, which is represented by a fixed range of integers. The definition of high and low vary from system to system with 0 being the lowest on some and highest on another.

In this document, we assume the lower the priority number, the higher the priority. This makes a Process or Thread with a priority of 0 have the highest priority.

What makes a Process have a certain priority depends on if the priority is set internally or externally.

- Internally-Defined Priorities use a measurable quantity or quantities to compute the priority of a process.
- Externally-Defined Priorities are set based on criteria outside the Operating System.

Priority scheduling can be nonpreemptive or preemptive. Like in SJF, if a process arrives with a higher priority than the one currently running, the scheduler is preemptive if the higher priority process is allows to Context Switch in. A nonpreemptive scheduler will allow the current Process to finish execution, while the new, higher priority process is put at the front of the job queue.

Example 6.3: Priority Scheduling Calculations.

Suppose 5 processes arrive at the same time with their CPU bursts and priority levels as shown.

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

In this case, these processes would be scheduled in this order.

1. P_2
2. P_5
3. P_1
4. P_3
5. P_4

With an average delay of

$$\frac{0+1+6+16+18}{5} = 13.4\,\text{ms}$$

In a FCFS system, the average delay would be 13.4 ms.

A major problem with priority scheduling is the concept of Starvation. A process that is ready to run, but waiting for the CPU is considered blocked. If there is a system where enough high priority Processes are created to the point where a lower priority process never gets a chance to execute, the low priority process is said to be starved.

To solve this problem, Process Aging is used.

Defn 84 (Aging)**.** *Aging* is a process in a priority-based system that forces a CPU to eventually execute the lowest priority Processes by gradually increasing the priority of the Process. Eventually, the process would reach a high enough priority to be executed.

6.2.4.4 Round-Robin Scheduling This algorithm is designed for time-sharing systems. It is similar to First-Come First-Served Scheduling, but preemption is allowed and the process list is circular. New tasks are added to the tail of the circular list. A small amount of time, called a Time Slice is used, and each Process in this circular list is allowed to execute for one of these slices.

Defn 85 (Time Slice)**.** A *time slice* is a small, indivisible amount of processor time, typically in the range of 10 ms to 100 ms. Inside this time slice, a single Process/Thread is run. When this time slice ends, a new task is executed.

Remark 85.1 (Time Quantum). A Time Slice is also called a *Time Quantum*.

In this scheduling algorithm, no process is allocated the CPU for more than 1 Time Slice in a row (unless it is the only runnable process). If a task requires more CPU time than a single Time Slice can provide, then that task is interrupted and is allowed to execute again the next time it is given a Time Slice. If a task requires less CPU time than a single Time Slice, then the next time slice begins when the process terminates. This makes the round-robin algorithm preemptive by definition.

To find the amount of time a Process will have the CPU, use Equation (6.2). To find how long the Process will have to wait to get the CPU again, use Equation (6.3).

$$PT = \frac{1}{n} \tag{6.2}$$

$$W = (n-1)q \tag{6.3}$$

PT: The amount of processor time a task will get.
n: The number of Processes in the `RUNNING` queue.
W: The amount of time a Process will have to wait to get the CPU allocated to it again.
q: The duration of a Time Slice.

The duration of a Time Slice is a major factor in the relative efficiency of a round-robin algorithm. If the Time Slice is:

- too large, then this becomes a FCFS algorithm.
- too small, then there will be a large number of Context Switches.

The duration of a Time Slice should also take the time required for a Context Switch into account. A time slice should be large relative to the time required for a context switch.

6.2.4.5 Multilevel Queue Scheduling Depending on the situation, there are times when Processes/Threads are easily grouped. For example, foreground and background processes (interactive and batch). These groups have different requirements on them, thus having different scheduling needs. Additionally, since foreground processes are in direct interaction with the user, they have a ghigher priority than background ones. This forms the basis for a Multilevel Queue scheduling algorithm.

Defn 86 (Multilevel Queue)**.** A *multilevel queue* is a system that uses multiple queues for different groups of tasks that have different parameters they must adhere to. Processes/Threads assigned to one queue are assigned there permanently, based on some property about the process.

Each queue can have its own scheduling algorithm, ensuring the different demands from the queues are met. Additionally, there is scheduling for which queue will be chosen next.

An example Multilevel Queue is shown below. In order from the highest priority to lowest, are 5 queues, each containing different processes.

1. System Processes
2. Interactive Processes
3. Interactive Editing Processes
4. Batch Processes
5. Student Processes

In this system, each queue has absolute priority over another, meaning Interactive Processes can execute **ONLY IF** there are no System Processes left to handle. If an Interactive Process is currently executing, and a System Process comes into its queue, the Interactive Process is switched out as soon as possible, and the System Process is switched in.

Another possibility is to give a certain amount of processor time to each queue and allow the queue to fill up that time as it sees fit.

6.2.4.6 Multilevel Feedback Queue Scheduling A Multilevel Queue does not allow a Process/Thread to move between one of the queue groups. However, in a *Multilevel Feedback Queue Scheduling* (*MLFQS*) system, they can. The separate queues still maintain their different scheduling algorithms, and each queue must be scheduled according to some master controller. However, there is also an algorithm to move tasks between the different queues.

In this system, a Process is moved between queues based on characteristics of their CPU Bursts. If a process uses too much CPU time, it is moved down in priority. This allows I/O Bound processes get quick access to the CPU and execute quickly. Then, when these higher priority queues are empty, a low priority task can execute longer.

Since this is effectively a Priority Scheduling system, Aging must be used to ensure a Process does not sit in a lower-level queue for too long.

In general, this scheduling algorithm is defined by the following parameters:

- The number of queues.
- The scheduling algorithm for each queue.
- The method used to determine when to upgrade a process to a higher-priority queue.
- The method used to determine when to demote a process to a lower-priority queue.
- The method used to determine which queue a process will enter when that process needs service.

6.3 Thread Scheduling

On operating systems that support them, Kernel Threads, not Processes are scheduled by the operating system. However, the terms "process scheduling" and "thread scheduling" are often used interchangeably. Process scheduling is used when discussing general scheduling concepts and thread scheduling to refer to thread-specific ideas.

6.3.1 User- vs. Kernel-Level Thread Scheduling

On systems implementing the Many-To-One Model (Section 5.1.1) and Many-To-Many Model (Section 5.1.3), the thread library schedules user-level threads to run on an available LWP.

Defn 87 (Process-Contention Scope)**.** *Process-Contention Scope* is where each of the Threads **WITHIN A SINGLE Process** compete with each other for execution.

In Process-Contention Scope, the Thread Library schedules the User Threads onto available Lightweight Processes, or directly onto Kernel Threads. However, this does not mean that the User Thread are being scheduled for execution. For that to happen, the Kernel must be involved.

To decide what Threads can execute on the CPU, we broaden our view of contention to System-Contention Scope.

Defn 88 (System-Contention Scope)**.** *System-Contention Scope* is where **ALL** of the Threads on the system, from all Processes compete with each other for execution.

If a Kernel uses the One-To-One Model, then **ALL** threads are scheduled using System-Contention Scope.

6.3.2 Multiprocessor Scheduling

If multiple processors are available for use in a system, then Load Sharing is possible.

Defn 89 (Load Sharing)**.** *Load sharing* involves splitting the load of running Processes and their Threads between multiple processors. This does not limit the possibility of multithreading, rather it complements it.

In this course, we consider each processor to be equivalent, or homogenous. This means that each one will have the same basic set of functionality and can reach all resources present in the system. This does not mean that all processors can reach all the resources with the same cost/delay. There may be an I/O device attached to the private bus of a core, or there may be Non-Uniform Memory Access.

6.3.2.1 Approaches to Multiprocessor Scheduling There are 2 main approaches to multiprocessor scheduling, without taking the idea of non-uniform resource availability.

1. Asymmetric Multiprocessor System
2. Symmetric Multiprocessor System

In an Asymmetric Multiprocessor System, one of the processors acts as the master server. It runs the scheduler, allocating jobs to each of the slave worker processors. This reduces the need for data sharing, making coding and debugging such a thing much easier.

The other approach, Symmetric Multiprocessor System, allows each processor to schedule itself. Tasks to execute may be in 2 types of queues:

1. A common queue for all processors.
2. A private queue, one for each processor.

Each processor must select a job from the ready queue and execute it. We must ensure that no two processors schedule and execute the same task at the same time, and that processes are not lost from the queue.

> Throughout this section, and most of this document, we discuss Symmetric Multiprocessor System (SMP) systems. These are more commonly in use in the desktop space and are more interesting to discuss.

6.3.3 Processor Affinity

Processes/Threads can migrate between processors, no matter the type of multiprocessor system used. However, if such a migration happens, the processor's cache must be invalidated from the origin and the destination cache must be populated. This is a very high cost to system productivity, so we attempt to keep a Process/Thread running on the same processor for as long as possible, this is known as Processor Affinity.

Defn 90 (Processor Affinity)**.** *Processor affinity* is the idea of keeping one Process/Thread running on the **same** processor for as long as possible. This prevents relatively costly process migrations from becoming common.

There are 2 kinds of processor affinity:

1. Soft Affinity
2. Hard Affinity

Defn 91 (Soft Affinity)**.** *Soft affinity* is where Processor Affinity is used, but no guarantees about the migration of a Process can be made. The Operating System will attempt to keep a process running on the same processor for as long as possible, but the process can still migrate between processors.

Defn 92 (Hard Affinity)**.** *Hard affinity* is where once a Process/Thread is assigned to a set of processors, it will **ONLY EXECUTE ON THOSE**.

The organization of Memory in a system will affect the performance of the Processor Affinity choices made. In a Non-Uniform Memory Access system, not all memory is **DIRECTLY** available to every processor. In this type of system, a Process's affinity should be set according to the memory location it inhabits. Otherwise, there will be long delays during memory acccesses. This is illustrated in Figure 6.2.

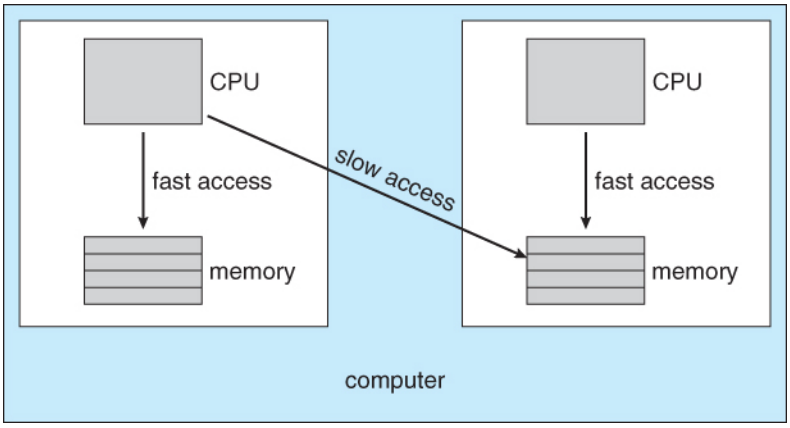


Figure 6.2: Non-Uniform Memory Access, Processor Affinity, and CPU Scheduler Choices

6.3.4 Load Balancing

Because we have multiple processors available to us, we need to make sure they are all doing a fair amount of work. We do not want to have one processor sitting idle while others have very high loads with long queues. This forms the basis of Load Balancing.

Defn 93 (Load Balancing)**.** *Load balancing* is the process of making sure that every processor in a system is assigned an even amount of work. This ensures that **ALL** processors are active at one time, preventing the issue of some processors sitting idle while others have very high loads and long queues.

There are 2 main ways to provide load balancing:

1. Push Migration
2. Pull Migration

Remark 93.1 (Common vs. Private Queues). Like stated earlier in this section, some multiprocessor designs use a common task queue for all processors, and others use private queues for each individual processor. In the case of a common queue, then Load Balancing is unnecessary, because each processor will extract a job from the common queue. Load Balancing is only necessary on systems that use private queues, which most multiprocessors today use to some extent.

Push Migration and Pull Migration **ARE NOT** mutually exclusive. In fact, most systems support both.

Defn 94 (Push Migration)**.** In *push migration*, there is a task inserted into the queue that checks the load on each processor. If it finds a large imabalance, then that processor's tasks are evenly redistributed among all other processors.

Defn 95 (Pull Migration)**.** *Pull migration* is where each processor, once it becomes idle, goes to other processors and attempts to pull a waiting task from the other processor's queue.

Load Balancing counteracts the benefits of Processor Affinity. The benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory. Either pulling or pushing a process from one processor to another removes this benefit.

6.3.5 Multicore Processors

In more recent times, multiple processors are being built onto the same physical chip, making a *multicore processor*. To the Operating System, they appear as separate CPUs, because each core maintains its own architectural state. This allows for multicore processor systems that are faster and consume less power than traditional multiprocessor systems.

The rise of multicore processors has also lead to the rise of hardware threads. This is because a Central Processing Unit spends much of its execution time waiting for Memory and its contents to become available. When a CPU waits for memory to returns its contents, it is called a Memory Stall.

Defn 96 (Memory Stall)**.** A *memory stall* is when a CPU must go to memory to retrieve a value. This could happen because of a cache miss, for instance.

By having multiple hardware threads, when one of the threads has a Memory Stall, the other thread can take over execute while the first waits for the memory to be returned. These are called *logical processors*.

Each of these logical processors appears to the Operating System as a separate physical CPU.

The number of processors that the Operating System "sees" can be calculated by Equation (6.4).

$$C = Pt \tag{6.4}$$

C: The number of cores the Operating System sees.
P: The number of processors present in the system.
t: The number of hardware threads present in each processor.

There are 2 ways to multithread a processor core:

1. Coarse-Grained Multithreading
2. Fine-Grained Multithreading

Defn 97 (Coarse-Grained Multithreading)**.** In *coarse-grained multithreading*, a hardware thread will execute until it reaches a long-latency event, such as a Memory Stall. The processor then switches to the other thread and continues execution while waiting for the long-latency event to return.

This has a high cost, because the entire instruction pipeline must have the first thread's instructions emptied and the new thread's instructions fill the pipeline. This is quite costly.

Defn 98 (Fine-Grained Multithreading)**.** *Fine-grained multithreading* switches between the hardware threads much more frequently and quickly, typically between every other instruction. To ensure this does not take too much time, hardware logic is built-in for thread switching.

The use of hardware threads means there is another layer of scheduling required.

1. The highest layer is scheduling User Threads onto Kernel Threads by the Thread Library.
2. The next layer is the Operating System scheduling Kernel Threads for execution.
3. Next is the hardware scheduler for which **hardware thread** to execute next.

Each one of these layers can use different scheduling algorithms, to meet performance criteria in each section.

6.4 Real-Time Scheduling

Real-time operating systems have their own class of scheduling issues. This depends on whether the Operating System is a Soft Real-Time System or a Hard Real-Time System.

Defn 99 (Soft Real-Time System)**.** *Soft real-time system*s do not provide a guarantee about the scheduling of a critical real-time process.

Defn 100 (Hard Real-Time System)**.** *Hard real-time system*s guarantee the execution time of a real-time process. These tasks will be serviced by its deadline, otherwise the process will not be executed at all.

POSIX also provides support for real-time scheduling through 2 functions with 2 scheduling types.

1. `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
2. `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`

1. `SCHED_FIFO`
2. `SCHED_RR`

6.4.1 Minimizing Latency

The key aspect here is the amount of time it takes for a system to respond to an event. This is called Event Latency.

Defn 101 (Event Latency)**.** *Event latency* is the amount of time that elapses from when an event occurs to when it is serviced. Different events can have different event latency requirements.

There are 2 factors that affect Event Latency.

1. Interrupt Latency. The amount of time from the arrival of an Interrupt to the start of the Interrupt Service Routine (ISR). This includes the amount of time needed to get the currently running instruction to a point where it can be switched. Also included is the amount of time needed to perform the switch.
2. Dispatch Latency the amount of time the scheduler needs to stop one process and start another. There are 2 parts that affect the value of the dispatch latency:
   (a) Preemption of **ANY** process running tin the kernel.
   (b) Release of resources used by low-priority process for higher-priority processes.

6.4.2 Scheduling

In this case, there are not as many choices of Scheduling Algorithm for real-time systems as other systems. All algorithms must be roughly based on a priority-based system all of which must support Preemption.

Most modern Operating Systems offer support for Soft Real-Time Systems with their scheduling priorities. Note however, that pure priority-based algorithms only guarantee soft real-time functionality, not hard.

Processes are considered periodic; they require the CPU at constant intervals. Once a periodic process has acquired the CPU, it has a fixed processing time t, a deadline d by which it must be serviced by the CPU, and a period p. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The rate of a periodic task is $\frac{1}{p}$. Schedulers can take advantage of these characteristics and assign priorities according to a process's deadline or rate requirements. What is different about this algorithm is a process may have to announce its deadline to the scheduler. Using an admission-control algorithm, the scheduler either admits the process, guaranteeing that the process will complete on time, or rejects the request if it cannot guarantee that the task will be serviced by its deadline.

6.4.2.1 Rate-Monotonic Scheduling

Defn 102 (Rate-Monotonic Scheduling)**.** *Rate-monotonic scheduling* is a Scheduling Algorithm for periodic tasks that uses a static priority policy with preemption. If a higher-priority process arrives, and a lower priority one is running, it is immediately preempted. The priority is statically calculated based on the inverse of the period of the task. Less frequent (longer period) tasks have a lower priority, and more frequent ones have higher priority. Additionally, the size of the CPU burst is assumed to be constant during every period.

Rate-Monotonic Scheduling is considered optimal because given a set of processes that cannot be scheduled by this algorithm, no other static-priority algorithm can schedule them either.

The worst-case CPU utilization for scheduling N processes is shown in Equation (6.5)

$$N\left(2^{\frac{1}{N}} - 1\right) \tag{6.5}$$

6.4.2.2 Earliest-Deadline-First Scheduling

Defn 103 (Earliest-Deadline-First Scheduling)**.** *Earliest-Deadline-First scheduling* dynamically assigns priorities based on the deadlines of tasks. The earlier/sooner the deadline, the higher the priority. To ensure this works, when a process becomes runnable, it must announce its deadline requirement.

Unlike Rate-Monotonic Scheduling, Earliest-Deadline-First Scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. EDF is theoretically optimal; it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. However, it is impossible to achieve this due to the cost of Context Switching between Processes and Interrupt handling.

6.4.2.3 Proportional Share Scheduling

Defn 104 (Proportional Share Scheduling)**.** In *proportional share scheduling*, the CPU execution time is split into T shares of time. Each application receives N shares of that T, $\frac{N}{T}$ shares of total processing time. In addition to this, there needs to be an admission-control policy to guarantee that an application is run only if there are enough time slots for the process to execute.

6.5 Algorithm Evaluation

Now that we have selected a Scheduling Algorithm to use, how do we know that it was the right choice? First, we need to know what our criteria were. Some systems might have multiple criteria at a time, such as:

- Maximum CPU response time is 1 second.
- Turnaround time is (on average) linearly proportional to total execution time.

To do this, there are 4 main ways to do this:

1. Deterministic Modeling
2. Queuing Models
3. Simulations
4. Implementation

6.5.0.1 Deterministic Modeling

Deterministic modeling is simple and fast. It gives us exact numbers, allowing us to compare the algorithms. However, it requires **exact numbers for input**, and its answers apply *only* to those cases.

This method takes a particular predetermined workload and defines the performance of each algorithm for that workload.

The main uses of deterministic modeling are in describing Scheduling Algorithms and providing examples. In cases where we are running the same program repeatedly, we can measure the program's processing requirements exactly, allowing us to select a Scheduling Algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately.

6.5.0.2 Queuing Models

On many systems, the processes that are run vary from day to day, so deterministic modeling is impossible. Instead, we can find the the distribution of CPU Bursts and I/O Bursts.

Commonly, this distribution is exponential and is described by its mean. Similarly, we can describe the distribution of times when processes arrive in the system (the arrival-time distribution). From these two distributions, it is possible to compute the average throughput, utilization, waiting time, and so on for most algorithms.

The CPU is described as a server with a queue of `READY` processes. The I/O system with its device queues is another instance of these servers. Knowing arrival rates and service rates, we can compute utilization, average queue length, average wait time, among other parameters. For a more detailed discussion of these topics, refer to ETSN10: Network Architecture and Performance.

Little's Formula, Equation (6.6), shown below, is useful because we can calculate a large variety of information from just a few parameters.

$$n = \lambda \times W \tag{6.6}$$

n: The average queue length, excluding the process currently executing on the CPU.
λ: The average arrival rate of new processes.
W: The average amount of time a process waits in the queue.

Queueing analysis can be useful in comparing scheduling algorithms, but the classes of algorithms and distributions that can be handled is limited. The mathematics of complicated algorithms and distributions are difficult to work with, and arrival and service distributions are often defined in mathematically tractable (but unrealistic) ways. Generally, it is necessary to make a number of independent assumptions, which may not be accurate. This means queuing models are only approximations of real systems, and the accuracy of the computed results are questionable.

6.5.0.3 Simulations

To get a more accurate evaluation of scheduling algorithms, simulations can be used. Running simulations involves programming a model of the computer system. Software is used to represent major components of the system. As "time" progresses, the simulator modifies the system state to reflect the activities of the devices, the processes, and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered. Typically, input is generated from a random-number generated that is programmed according to probability distributions.

However, simulations can be expensive, often requiring hours of processor time. A more detailed simulation provides more accurate results, but takes more computer time.

6.5.0.4 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a Scheduling Algorithm is to code it, put it in the Operating System, and see how it works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The major difficulty with this approach is the high cost.

One of the biggest difficulties is the expense incurred in coding the algorithm and modifying the operating system to support it (along with its required data structures). The environment around the algorithm used will change not only in the usual way (New programs written and the types of problems change) but also as a result of the performance of the scheduler. If short processes are given priority, then users may break larger processes into sets of smaller processes. If interactive processes are given priority over noninteractive processes, then users may switch to interactive use.

6.6 Deadlocks

Deadlock is a serious issue in CPU Schedulers because Processes lock resources for themselves. A good example of a deadlock is "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Defn 105 (Deadlock)**.** *Deadlock* is when 2 processes require information or resources 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.

There are only 2 options for handling Deadlocks:

1. Prevent them from happening in the first place.

2. Identify them and fix the problem that is causing them.
3. Hope they don't happen and consider them as unlikely events to occur.
    - This is what most desktop Operating Systems do.

Most Operating Systems do **NOT** provide functionality to identify Deadlocks and correct them.

A system consists of a finite number of resources to be distributed among a number of competing processes. The resources may be partitioned into several types (or classes), each consisting of some number of identical instances. CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types. If a system has two CPUs, then the resource type CPU has two instances. Similarly, the resource type printer may have five instances. If a process requests an instance of a resource type, the allocation of **any** instance of the type should satisfy the request. If it does not, then the instances are not identical, and the resource type classes have not been defined properly. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. Request. The process requests the resource. If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
2. Use. The process can operate on the resource (for example, if the resource is a printer, the process can print on that printer).
3. Release. The process releases the resource.

For each use of a Kernel-managed resource by a Process or Thread, the Operating System checks to make sure that the process has requested and has been allocated the resource. A system table records whether each resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

> A set of Processes is in a Deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set.

6.6.1 Conditions for Deadlocks

A deadlock situation can arise if the following **four conditions hold simultaneously** in a system:

1. Mutual exclusion. At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. Hold and wait. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. No preemption. Resources cannot be preempted, so, a resource can be released voluntarily only by the process holding it, after that process has completed its task.
4. Circular wait. A set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by P_2, ..., P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

All four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

6.6.2 Resource-Allocation Graph

Deadlocks can be described by a directed graph, called a *system resource allocation graph*. In Figure 6.3, there is a representation of the set of processes P_i and resources R_i. If there is an arrow pointing $P_i \rightarrow R_j$ then Process i is requesting Resource j, forming a *Request Edge*. If there is an arrow pointing $R_i \rightarrow P_j$ then Resource j is allocated to Process j, forming a *Assignment Edge*.

In Figure 6.3, there are:

- The sets P, R, and E:
    - $P = \{P_1, P_2, P_3\}$
- Resource instances:
    - One instance of resource type R_1
    - Two instances of resource type R_2
    - One instance of resource type R_3
    - Three instances of resource type R_4

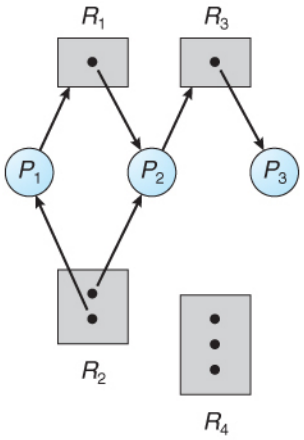
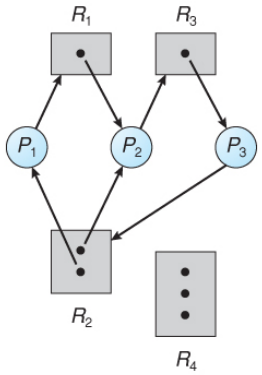


Figure 6.3: Resource Allocation Graph

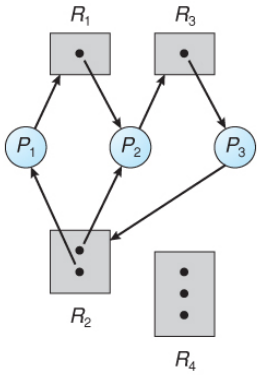
- Process states:
  - Process P_1 is holding an instance of resource type R_2 and is waiting for an instance of resource type R_1.
  - Process P_2 is holding an instance of R_1 and an instance of R_2 and is waiting for an instance of R_3.
  - Process P_3 is holding an instance of R_3.

Given the definition of a resource-allocation graph, if the graph contains no cycles, then no process in the system is Deadlocked. If the graph does contain a cycle, then a Deadlock may exist.

- If each resource type has exactly one instance, then a cycle implies that a Deadlock has occurred.
  - In this case, a cycle in the graph is both a necessary and a sufficient condition for the existence of Deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a Deadlock has occurred.
  - In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of Deadlock.
  - This is illustrated by Figures 6.4a to 6.4b.



(a) RAG with Cycle and Deadlock



(b) RAG with Cycle but no Deadlock

Figure 6.4: Resource Allocation Graph with Cycle

6.7 Handling Deadlocks

There are 3 main ways to handle a deadlock:

1. We can use a protocol to prevent (Section 6.7.1) or avoid (Section 6.7.2) deadlocks, ensuring that the system will never enter a deadlocked state.
2. We can allow the system to enter a deadlocked state, detect it, and recover (Section 6.7.3).
3. We can ignore the problem altogether and pretend that deadlocks never occur in the system.

The third method of handling a Deadlock may not make sense, but from a cost perspective, it does. Ignoring the possibility of deadlocks is cheaper. In many systems, deadlocks occur infrequently (once per year), the extra expense of the methods

may not seem worthwhile. In addition, methods used to recover from other conditions may be put to use to recover from deadlock. In some circumstances, a system is in a frozen state but not in a deadlocked state.

6.7.1 Deadlock Prevention

Defn 106 (Deadlock Prevention)**.** *Deadlock prevention* is done by providing a set of methods that ensure any one of the Conditions for Deadlocks cannot occur.

6.7.1.1 Mutual Exclusion That is, at least one resource must be nonsharable. Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock. A process never needs to wait for a sharable resource.

Read-only files are a good example of a sharable resource. If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.

6.7.1.2 Hold and Wait To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.

We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol allows a process to request resources only when it has none.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, Starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

6.7.1.3 No Preemption To ensure that no Preemption of resources that have already been allocated occurs, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted, implicitly releasing them. The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as mutex locks and semaphores.

6.7.1.4 Circular Wait One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration. Possible side effects of preventing deadlocks are low device utilization and reduced system throughput.

Formally, we define a one-to-one function

$$F : R \to \mathbb{N}$$

where R is the set of resources and $\mathbb{N}$ is the set of natural numbers.

Each process can request resources only in an increasing order of enumeration. Meaning, a process can initially request any number of instances of a resource type, R_i. After that, the process can request instances of resource type R_j if and only if $F(R_j) > F(R_i)$. This could be similarly defined using a more generous comparison; a process can request instances of resource type R_j if and only if $F(R_j) \geq F(R_i)$.

This property can be proven, Paragraph 6.7.1.4.

Proof by Contradiction of Circular Waiting. Let the set of processes involved in the circular wait be $\{P_0, P_1, \ldots, P_n\}$, where P_i is waiting for a resource R_i, which is held by process[3] P_{i+1}.

Then, since process P_{i+1} is holding resource R_i while requesting resource R_{i+1}, we must have

$$\forall i. F(R_i) < F(R_{i+1})$$

But this condition means

$$F(R_0) < F(R_1) < \cdots < F(R_n) < F(R_0)$$

By transitivity, $F(R_0) < F(R_0)$, which is impossible.
$\therefore$ there can be no circular wait. ■

> Keep in mind that developing an ordering, or hierarchy, does not in itself prevent deadlock. It is up to application developers to write programs that follow the ordering.

[3] Modulo arithmetic is used on the indexes.

6.7.2 Deadlock Avoidance

Defn 107 (Deadlock Avoidance)**.** *Deadlock avoidance* requires the Operating System be gined additional information in advnace about what resources a process will request and possible use within its lifetime.

To avoid Deadlocks, the Operating System can require additional information about how resources are to be requested. With knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock. Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

There are 2 major ways to achieve this:

1. Safe State
2. Resource Allocation Graph Algorithm

6.7.2.1 Safe State A system is in a safe state only if there exists a safe sequence. A sequence of processes $\langle P1, P2, \ldots, Pn \rangle$ is a safe sequence for the current allocation state if, for all resource requests that P_i makes, they can be satisfied by the currently available resources plus the resources held by all P_j, where $j < i$. If the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When P_j has finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources, and terminate. When P_i terminates, P_{i+1} can obtain its needed resources, and so on. If no such sequence exists, then the system state is said to be *unsafe*.

> A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state. Not all unsafe states are deadlocks, however.

6.7.2.2 Resource Allocation Graph Algorithm If we have a resource-allocation system **with only one** instance of each resource type, we can use a variant of the resource-allocation graph. The resources must be claimed a priori in the system.

We introduce a new type of edge, called a claim edge. A claim edge $P_i \to R_j$ indicates that process P_i may request resource R_j at some time in the future. Before process P_i starts executing, all its claim edges must already appear in the resource-allocation graph.

6.7.3 Deadlock Detection

Defn 108 (Deadlock Detection)**.** *Deadlock detection* is an algorithm that determines if the current state of a system indicates a Deadlock has occurred.

In this system, either of these options may be provided.

- An algorithm that examines the state of the system to determine whether a deadlock has occurred.
- An algorithm to recover from the deadlock.

6.7.3.1 Single Instance of a Resource Type If all resources have only a single instance, then we can define a deadlock-detection algorithm that uses a variant of the resource-allocation graph, called a wait-for graph. We obtain this graph from the resource-allocation graph by removing the resource nodes.

An edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. More formally,

$$P_i \to P_j = P_i \to R_q, R_q \to P_j$$

6.7.3.2 Multiple Instances of a Resource Type The algorithm employs several time-varying data structures that are similar to those used in the banker's algorithm. I do not heavily invest in working with this variant of this problem.

6.7.3.3 Usage of Detection Algorithms There are 2 main questions about when we should invoke these detection algorithms.

1. How **often** is a deadlock likely to occur?
2. How **many** processes will be affected by deadlock when it happens?

Resources allocated to deadlocked processes will be idle until the deadlock can be broken. If deadlocks occur frequently, then the detection algorithm should be invoked frequently. In addition, the number of processes involved in the deadlock cycle may grow.

6.7.4 Deadlock Recovery

When a detection algorithm determines that a deadlock exists, several alternatives are available.

1. One possibility is to inform the operator that a Deadlock has occurred and to let the operator deal with the Deadlock manually.
2. Another possibility is to let the system recover from the Deadlock automatically. There are two options for breaking a Deadlock.
   (a) Simply abort one or more processes to break the circular wait.
   (b) Preempt some resources from one or more of the Deadlocked processes.

6.7.4.1 Process Termination To eliminate deadlocks by aborting a process, there are two methods. In both, the system reclaims all resources previously allocated.

1. Abort all deadlocked processes. This method clearly will break the deadlock cycle, but at the cost of lost work among all terminated processes.
2. Abort one process at a time until the deadlock cycle is eliminated. This method incurs considerable overhead, since after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.

Which one to choose depends on a variety of factors.

- What the priority of the process is.
- How long the process has computed.
- How much longer the process will compute before completing its designated task.
- How many and what types of resources the process has used.
- How many more resources the process needs in order to complete.
- How many processes will need to be terminated.
- Whether the process is interactive or batch.

6.7.4.2 Resource Preemption In this system, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken. If preemption is required to deal with deadlocks, then three issues need to be addressed:

1. **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include parameters such as:
   - Number of resources a deadlocked process is holding
   - Amount of time the process has thus far consumed
2. **Rollback**. If we preempt a resource from a process, what should be done with that process? It cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. It is more effective to roll back the process only as far as necessary to break the deadlock, however, this method requires the system to keep more information about the state of all running processes. So the process is rolled back to the beginning usually.
3. **Starvation**. How do we ensure that starvation will not occur?

7 Main Memory

To truly benefit from CPU Scheduling and Synchronization, discussed in Section 6, we must be able to keep multiple Processes in memory at the same time.

In a Central Processing Unit, the only things that can be accessed directly are Registers and main memory. The CPU can reach the registers within one clock cycle, but accessing main memory takes several clock cycles, because access is done through the memory bus. To prevent Memory Stalls, we can use multiple Threads, and we can add additional Caches to the CPU itself. This Hardware automatically speeds up memory access, without Operating System intervention.

Defn 109 (Cache)**.** A *cache* is a very small amount of memory, built onto the Central Processing Unit itself, and acts as a buffer between the CPU and main memory. A cache will have a copy of a small portion of what is in main memory, and the CPU goes to find the next thing, whatever it may be, from the cache first. Because the cache sits on the CPU itself, and does not have to interact with the memory bus, it is significantly faster than accessing main memory, but still slightly slower than accessing a register.

Not only do we want our memory and its access to be fast, we also want to make sure that the memory and its contents are used correctly. For proper system operation we must protect the operating system from access by user processes. On multiuser systems, we must additionally protect user processes from one another. This protection must be provided by the hardware because the operating system doesn't usually intervene between the CPU and its memory accesses (because of the resulting performance penalty). A Memory Management Unit implements this production, in hardware, in several different possible ways.

Defn 110 (Memory Management Unit)**.** The *Memory Management Unit* (*MMU*) is a piece of hardware built onto a Central Processing Unit (See Remark 110.1). It is the main controller for all main memory accesses made by a CPU.

It handles the run-time mapping from Virtual Addresses to Physical Addresses.

Remark 110.1 (MMU Location). The location of the Memory Management Unit is **typically** on the CPU. This holds true for modern CPUs, however, older CPUs like the Motorola 68000 series have physically separate MMUs, which can be attached to the motherboard that the CPU sits on.

To ensure that everything works, the first thing that needs to be done is to separate each Process into its own memory space. This protects the processes from each other and is fundamental to having multiple processes loaded in memory for concurrent execution. To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.

The simplest version of this protection can be realized by using two registers, usually a `base` and a `limit`. The base register holds the smallest legal physical memory address; the limit register specifies the size of the range. Thus, to find the largest legal physical memory address, we must add the limit register's value to the base register's. The only way these registers can be loaded is by using a special, privileged, instruction. Since privileged instructions can only be executed in Kernel-mode, only the Operating System can load the base and limit registers. This allows only the OS to change the value of these registers, and not any User-Processes.

7.1 Address Binding

Just as in programming languages, there are different possible times when a Program stored on a disk can have Memory addresses bound to it, this is called Address Binding.

Defn 111 (Address Binding)**.** *Address binding* is the act of putting "something" (a variable, a function, a value, anything) at a location in Memory. Because these memory locations have unique addresses assigned to them, the "something" that was put there is bound to that memory address.

For a Program to be executed, it must be brought into memory and placed within a Process. The processes that are waiting for their binary images to be brought to memory from the disk before beginning execution form the Input Queue.

Defn 112 (Input Queue)**.** The *input queue* is the queue in which Processes are placed while they wait for their Program binary image to arrive from the disk.

There are 3 major times when Address Binding can occur:

1. Compile-Time Address Binding
2. Load-Time Address Binding
3. Execution-Time Address Binding

7.1.1 Compile-Time Address Binding

If the addresses that the Process will use are known at compile-time, then Absolute Code can be generated.

Defn 113 (Absolute Code)**.** In *absolute code*, the memory location of the Process is fixed during every execution, and because the Program is always the same size, anything that requires a memory can be referenced by this unchanging value.

If, at some later time, the starting location changes, then the whole program must be recompiled. In the case of assembly languages, this might mean recalculating the absolute memory locations of everything in the program.

These reasons led to the developement of relative addressing in assembly languages and the use of Load-Time Address Binding.

7.1.2 Load-Time Address Binding

If it is not known at compile time where the process will reside in memory, then the compiler must generate Relocatable Code.

Defn 114 (Relocatable Code)**.** *Relocatable code* uses relative addressing from certain known points. For example, a compiler will know that to access some variable `x`, the location where `x` is stored in memory is 14 bytes past the beginning of the function's start of the frame. Thus, if the starting address of the function changes, the code only needs to be reloaded to find a newly changed value.

In this case, the final binding of "stuff" to memory addresses is delayed until load time. If the starting address changes, the user code only needs to be reloaded to incorporate this changed value. This is how most programs are written and executed. If you change something in a C program, you must recompile and reload the program to have the changes take effect.

7.1.3 Execution-Time Address Binding

If the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this scheme to work. Most general-purpose operating systems use this method.

There are 4 major items that affect how this works:

1. Dynamic Loading
2. Dynamic Linking
3. Swapping
4. Paging

7.2 Logical, Physical, Virtual Address Spaces

The compile-time and load-time address-binding methods generate identical logical and physical addresses.

Defn 115 (Logical Address)**.** The CPU generates virtual/*logical address*es. How these logical addresses look, behave, and correspond to Physical Addresses depend on the memory management technique used by the Memory Management Unit.

The user program deals with logical addresses. The memory-mapping hardware converts logical addresses into Physical Addresses.

Defn 116 (Physical Address)**.** The *physical address* is the actual value of a particular byte of memory's address. How the physical address is calculated from the Logical Address is handled by the Memory Management Unit.

The user program deals with Logical Addresses. The memory-mapping hardware converts logical addresses into physical addresses.

However, the execution-time address-binding scheme results in differing logical and physical addresses, so we call the Logical Address a Virtual Address instead.

Defn 117 (Virtual Address)**.** The CPU generates logical/*virtual address*es. How these virtual addresses look, behave, and correspond to Physical Addresses depend on the memory management technique used by the Memory Management Unit.

The user program deals with logical/virtual addresses. The memory-mapping hardware converts logical/virtual addresses into Physical Addresses.

Remark 117.1 (Use of Logical and Virtual Address). In this document, I use Logical Address and Virtual Address interchangeably.

The set of all logical addresses/Virtual Addresses generated by a program is a Logical Address Space/Virtual Address Space.

Defn 118 (Logical Address Space)**.** The *logical address space* consists of all the Logical Addresses generated by a Program.

Remark 118.1. The Logical Address Space is only calculated by one Program/Process at a time. To find the total logical address space used, all Processes must have their logical address spaces aggregated.

The set of all Physical Addresses corresponding to these logical addresses is the Physical Address Space.

Defn 119 (Physical Address Space)**.** The *physical address space* consists of all the Physical Addresses that the Logical Addresses/Virtual Addresses map to.

Remark 119.1. The Physical Address Space is only calculated by one Program/Process at a time. To find the total physical address space used, all Processes must have their physical address spaces aggregated.

Thus, in the execution-time address-binding scheme, the logical and physical address spaces differ. The Memory Management Unit handles the mapping of the Virtual Address Space/Logical Address Space to the Physical Address Space. The Program generates only Logical Addresses and thinks that the Process runs in memory locations from 0 to max.

For example, assume a Process's base register, for the lowest valid memory address is R. If we wanted to access the 14th byte from the beginning of the Process, the CPU would attempt to retrieve 14, the Logical Address. However, on the way to memory, the Memory Management Unit would intercept this and redirect the access to $R + 14$, the Physical Address.

7.3 Dynamic Loading

Because of the way we have set up our memory and Processes before, a Process could only be as big as our physical memory. However, we can skirt these issues by using Dynamic Loading.

Defn 120 (Dynamic Loading)**.** *Dynamic Loading* is used to load routines into memory only when needed. The routines that can by dynamically loaded must be stored on disk in a Relocatable Code format. If the main program that was loaded in and began execution needs a routine, it checks to see if the routine has been loaded. If it has not, the loader loads the desired routine into memory, then updates the Process's address tables to point to the newly retrieved routine.

The advantage of Dynamic Loading is that a routine is loaded only when it is needed. This is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. This allows the total program size to be large, but the portion used (and hence loaded) be much smaller.

Dynamic Loading does not require special support from the operating system. It is the responsibility of the users to design their programs to take advantage of such a method. Operating Systems may help the programmer, however, by providing library routines to implement Dynamic Loading.

7.4 Dynamic Linking

Operating Systems provide many useful libraries to allow programmers and their work to use higher-level abstractions to make their programming life easier. An example of this is a language library for localization, allowing a programmer to write their input and output statements in one language and it be translated to another automatically.

To achieve this, Dynamic Linking is used, and Stubs are generated during the compilation process to inform the running Process where to find the necessary information.

Defn 121 (Dynamic Linking)**.** A Program that makes use of *dynamic linking* is compiled like normal. However, the library routines that would be provided by a *dynamically linked library* (*Shared Library*) are left as Stubs. When the program begins execution, the Process executes like normal, potentially using Dynamic Loading in the meantime. However, once the Process reaches a Stub, it will: proceed like normal (if the stub is replaced by the actual code), or will fetch and load the routine, replace the stub, and then execute the routine.

Remark 121.1 (Dynamic Linking vs. Dynamic Loading). In Dynamic Linking, a Program is compiled down to the machine code, with the Stubs in place, which are replaced **DURING** program execution. This allows us to share a binary library file **BETWEEN DIFFERENT** Programs. Whereas, in Dynamic Loading, the code is brought in from memory from the **same** Program binary image.

This means that if an Operating System did not support Dynamic Linking, each Program that required a certain library must include it. This leads to space inefficiencies on both the disk and in main memory.

Defn 122 (Stub)**.** A *stub* is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. A stub is included in the Program's binary image **FOR EACH** library-routine reference.

When the stub is executed, it checks to see whether the needed routine is already in memory. If it is not, the program loads the routine into memory and replaces the stub with the address of the routine and executes the routine. Then, the next time that routine is reached, the routine is executed directly, incurring no cost for Dynamic Linking.

This feature can be extended to handle library updates (such as bug fixes). If a library is replaced by a new version, all programs that use that library will automatically use the new version. Without Dynamic Linking, all such programs would need to be relinked to gain access to the new library. Version information is included in the Program and the library so that programs will not accidentally execute new, incompatible versions of libraries. Multiple versions of the same library may be loaded into memory, and each program uses its version information to decide which library to use.

Unlike Dynamic Loading, Dynamic Linking and dynamically linked libraries require help from the Operating System. If the Processes in memory are protected from one another, then the operating system is the only entity that can check all the Processes. Only the OS can:

- See if the needed routine is in another process's memory space.
- Allow multiple processes to access the same memory addresses.

7.4.1 Static vs. Dynamic Linking

As programmers, we are more familiar with Static Linking; where a library is compiled **INTO** and assembled **WITH** our code **INTO** our executable binary image.

Defn 123 (Static Linking)**.** *Static linking* is the act of running a linker after the compiler has generated Stubs for library routines. By running the linker, the library's code is brought into our program before it is assembled. This means that the library (and all the code supporting the library) is compiled **INTO** our binary image as well.

7.5 Swapping

A Process can only be executed if it is in memory. However, while it is **NOT** being executed, it is not required for it to be in memory. This idea forms the basis of Swapping.

Defn 124 (Swapping)**.** *Swapping* is the action of moving a Process out of memory to a Backing Store, or vice versa.

Defn 125 (Backing Store)**.** A *backing store* is another form of storage media. There are typically very few requirements on the type of storage media used in a backing store, but it **MUST** be readily assessible, and be quick (not as fast as RAM, but faster than traditional file system storage). Typically, a fast disk is used as the backing store.

Remark 125.1 (Swap)*.* Sometimes the Backing Store is called *the swap*, *swap-area*, *swap partition*, *swapfile*, etc. They all mean the same thing[4].

Swapping allows for the total Physical Address Space of **ALL** Processes to exceed the real physical memory of the system. However, the Context Switch time in a swapping system is fairly high.

The system maintains the ready queue consisting of all processes whose memory images are on the backing store or in memory and are ready to run. Whenever the CPU scheduler decides to execute a process, it calls the Dispatcher. The dispatcher checks to see where the next Process is.

- If the process is in memory, the dispatcher will reload registers and transfer control, like normal.
- If the process is in the swap, the dispatcher will swap the selected process into memory, reload registers, and transfer control.
- If the process is in the swap, **and** if there is no free memory region, the dispatcher swaps out a process currently in memory and swaps in the desired process, reloads registers, and transfers control.

Clearly, it would be useful to know exactly how much memory a User Process **is** using, not simply how much it **might** be using. This way, we only need to swap what is actually used, thereby reducing swap time. For this method to be effective, the user **must** keep the system informed of any changes in memory requirements. Thus, a process with dynamic memory requirements will need to issue system calls (`request_memory()` and `release_memory()`) informing the Operating System of its changing memory needs.

Swapping is constrained by other factors as well. **If we want to swap a process, we must be sure that it is COMPLETELY idle.** Of particular concern is any pending I/O. A process may be waiting for an I/O operation when we want to swap that process to the Backing Store. However, if the I/O is asynchronously accessing user memory for I/O buffers, then the process cannot be swapped.

There are two main solutions to this problem:

1. Never swap a process with pending I/O.
2. Execute I/O operations only into Operating System buffers.
    - Transfers between operating-system buffers and process memory only occur when the process is **swapped in**.
    - Double buffering adds overhead.
    - Now need to copy the data twice, once into kernel memory, then from kernel memory to user memory, before the user process can access it.

Standard Swapping as described above is not used in modern operating systems. However, modified versions of swapping, are used on many systems. One common variation, is to disable swapping until the amount of free memory (unused memory available for the operating system or processes to use) falls below a threshold amount. Swapping is then halted when the amount of free memory increases again. Another variation involves swapping portions of processes, rather than the entire process, to decrease swap time. Typically, these modified forms of swapping work in conjunction with Virtual Memory.

7.6 Contiguous Memory Allocation

Main memory must contain everything for the system to run, including both the Operating System and User Processes. It is our jobs, as operating system engineers to make the allocation of memory for the Operating System as efficient as possible. The earliest and simplest method is that of Contiguous Memory Allocation.

Memory is usually divided into two partitions: one for the Operating System and one for the user processes. We can place the operating system in either low memory or high memory, depending on the location of the Interrupt Vector. Typically, the interrupt vector is in low addresses, so the OS is usually put there too. Throughout this document, we assume that the OS inhabits the lowest addresses.

Defn 126 (Contiguous Memory Allocation)**.** In *contiguous memory allocation*, each Process is contained in a single contiguous section of memory that is also contiguous with the next process.

In short, this means that each Process sits in its own contiguous block and is right next to the next process.

[4]Swap Partition and Swapfile have specify the type of the Backing Store

7.6.1 Memory Protection

In Contiguous Memory Allocation, we can provide memory protection by using concepts from earlier. If the `base` register is considered as the `relocation` register (because of Logical Address conversion to Physical Address), and use the `limit` register as before, then we can protect this Process. Since the Operating System is the only entity that can change the values of the `relocation` and `limit` registers, this (currently executing) process and others cannot interfere with each other.

When the CPU Scheduler selects a Process for execution, the Dispatcher loads the `relocation` and `limit` registers with the correct values as part of the Context Switch. Because every address generated by a CPU is checked against these registers, we can protect the operating system and other users' programs and data from being modified by this running process. This scheme is an effective way to allow the Operating System to change size dynamically, which is highly desireable.

7.6.2 Memory Allocation

Within the User memory, there are 2 main methods of Process-memory allocation:

1. Multiple-Partition Scheme
2. Variable-Partition Scheme

Both methods use the same idea of dividing all free memory into separate partitions, however the size of these partitions and how they are divided is the differentiating factor.

7.6.2.1 Multiple-Partition Scheme In the *multiple-partition scheme*, all free memory is statically divided into equal sized partitions. Thus, the size of a partition is fixed throughout the execution of the Operating System. When a partition is free, a Process is selected from the input queue and placed into the free partition. When the process terminates, the partition is returned to the pool of available partitions.

7.6.2.2 Variable-Partition Scheme In the *variable-partition scheme*, all free memory is pooled together. The free memory is called a *hole*.

> In many ways, this mirrors the use of the heap in programs. Many of the principles from heap and the allocation of stuff to the heap also apply in this case as well.

As Processes are selected from the input queue to run, their memory requirements are considered. The system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. The process is then given an appropriate amount of space from memory, taken from the pool of free memory; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

At this point, the system may need to check whether there are processes waiting for memory and whether this newly freed and recombined memory could satisfy the demands of any of these waiting processes. The next process to run has its memory requirements considered, **AND** the free memory available. What happens next depending on the Operating System:

- Wait until a large enough block of memory is available for this process.
- Skip through the input queue to find a process that can use the memory (because of smaller requirements).

Just like in heap allocation and garbage collection, this is a particular instance of the general dynamic storage-allocation problem. In Operating Systems, there are 3 common strategies used to allocate this memory to Processes:

1. First-Fit
2. Best-Fit
3. Worst-Fit

First-Fit Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. Stop searching as soon as we find a free hole that is large enough.

Best-Fit Allocate the smallest hole that is big enough, the best fitting hole. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst-Fit Allocate the largest hole, the worst fitting hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

7.6.3 Fragmentation

When allocating space to Processes, we may run into the issue of Fragmentation.

Defn 127 (Fragmentation)**.** *Fragmentation* is when memory that was allocated to a Process is **not** used. Technically, this is an inefficiency, not a problem.

There are 2 kinds of fragmentation possible:

1. External Fragmentation
2. Internal Fragmentation

7.6.3.1 External Fragmentation Statistical analysis of First-Fit reveals that, given N allocated blocks, another $0.5N$ blocks will be lost to External Fragmentation. Meaning, one-third of memory may be unusable. This property is known as the *50-percent rule*.

Defn 128 (External Fragmentation)**.** *External fragmentation* is when there is enough total unused memory to satisfy a request, but the unused memory is **not** contiguous. If the memory is fragmented into a large number of small holes, then contiguous memory requests cannot be satisfied. As processes are loaded and removed from memory, the free memory space is broken into little pieces.

This is not necessarily a problem by itself, as much as it is an inefficiency. However, this inefficiency will lead to problems if it is not handled.

One solution to the problem of External Fragmentation is *compaction* (like garbage collection for the heap). The goal is to shuffle the memory contents so as to place all free memory together in one large block, however it cannot always be used. When compaction is possible, we must determine its cost, because it can be quite expensive.

- If memory locations are static and done at assembly or load time, compaction cannot be done.
- If memory locations are dynamic and done at execution time, compaction can be done by relocation.
  - Relocation moves the program and data, then changes the `base/relocation` register to reflect the new base address.

The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory.

Another solution to the External Fragmentation problem is to permit the logical address space of the processes to be noncontiguous. This allows a process to be allocated physical memory **wherever** memory is available. Two complementary techniques (which can also be combined) can solve this problem:

1. Segmentation (Section 7.7)
2. Paging (Section 7.8)

7.6.3.2 Internal Fragmentation Memory fragmentation can be internal as well as external. Consider a multiple-partition allocation scheme with a hole of 18464 bytes. Suppose that the next process requests 18462 bytes. If we allocate exactly the requested block, we are left with a hole of $18464 - 18462 = 2$ bytes. The difference between these two numbers is Internal Fragmentation.

Defn 129 (Internal Fragmentation)**.** *Internal Fragmentation* is the issue of unused memory that is internal to a partition. This means that a Process is given more memory than it actually needs, and will not be using all of it.

Like External Fragmentation, this is not necessarily a problem as much as it is an inefficiency that can lead to problems.

7.7 Segmentation

Dealing with memory in terms of its physical properties is inconvenient to both the Operating System and the programmer. If the hardware could provide a memory mechanism that mapped the programmer's view to the actual physical memory, the system would have more freedom to manage memory, while the programmer would have a more natural programming environment. Segmentation provides such a mechanism.

Defn 130 (Segmentation)**.** *Segmentation* is a memory-management scheme that supports a programmer's view of memory of segments of a program. Each segment has a name and a length. Segments vary in length, and the length of each is intrinsically defined by its purpose in the program. Elements within a segment are identified by their offset from the beginning of the segment Therefore, when a programmer specifies something in the program, it is reached by an address by two quantities: a segment name and an offset.

This makes a Logical Address Space is a collection of segments. These addresses specify both the segment name and the offset within the segment.

Remark 130.1 (Memory Locations when using Segmentation). Segmentation is a method of breaking up a program into segments that behave as logical units in a program and are each referenced, and thus located, separately. This allows the Physical Address Space of a process to be noncontiguous.

However, it makes no attempt to avoid External Fragmentation.

For example, a C compiler might create separate segments for:

1. Code
2. Global variables
3. The heap, from which memory is allocated
4. The stacks used by each thread
5. The standard C library

7.7.1 Hardware Support for Segmentation

The values that the programmer specifies are now 2-dimensional objects, and a memory address is one-dimensional, so the Memory Management Unit must handle the mapping of these 2D Logical Addresses to the 1D Physical Addresses.

A logical address consists of two parts: a segment number, s, and an offset (displacement) into that segment, d. To map the logical address pair to the physical address, the *segment table* is used. The segment table is essentially an array of `base`–`limit` register pairs. Each entry in the segment table has a `segment base` and a `segment limit`. The segment `base` contains the starting physical address where the segment resides in memory, and the segment `limit` specifies the length of the segment.

The segment number is used as an **INDEX** to the segment table. The offset d of the logical address must be between 0 and the `segment limit`. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment). When an offset is legal, it is added to the `segment base` to produce the address in Physical Memory of the desired byte.

7.8 Paging

Most memory-management schemes used before the introduction of Paging suffered from the problem of fitting memory chunks of varying sizes on the Backing Store This arises because space must be found on the backing store, when main memory needs to be swapped out. The backing store has the same Fragmentation problems discussed in connection with main memory, but access is much slower, so compaction is impossible.

Defn 131 (Paging)**.** *Paging* allows the Physical Address Space of a Process to be noncontiguous. This is achieved breaking up Physical Memory and Logical Address Space into equally, fixed-size blocks.

- Physical Memory is broken up into *frames*.
- Logical Address Space is broken up into *pages*.
- The Backing Store is also broken up, as a multiple (1 through n) of the frame size.

The list of pages and their mapping to frames for this Process is stored in the Page Table. There is a page table in each process, which is used when this process Context Switches into the CPU. Therefore, paging also increases context switch time.

Paging avoids the issue of External Fragmentation and solves the problem of fitting memory chunks of varying sizes on the Backing Store. However, Internal Fragmentation is still an issue.

Paging is implemented through cooperation between the operating system and the computer hardware. Similar to how Segmentation has a segment table, **EACH Process** in a Paging system has a Page Table.

Defn 132 (Page Table)**.** A *page table* is a table that maintains the mapping of logical pages to physical frames in memory. The page table is a simple lookup table, where the current Process's current page number is also the value of the index. The value contained at that element is the location of the frame in memory. This is combined with the use of Logical Addresses that are generated by the CPU.

To reach an address, the CPU generates a Logical Address that has 2 parts, a page number p and a page offset d. Like before, the page number is used as an **INDEX** in the Page Table. The page table contains the base address of each page in physical memory. This base address is combined with the page offset to define the physical memory address that is sent to the memory unit.

The frame size, and thus the page size, are defined by the hardware. The size of a page is a power of 2, varying between 512 bytes and 1 GiB[5] per page, depending on the computer architecture. A power of 2 as a page size makes the translation of a Logical Address into a page number and page offset particularly easy.

[5]GiB is a gibibyte, or 2^{30} bytes

If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the high-order $m - n$ **bits** of a Logical Address designate the page number, and the n low-order **bits** designate the page offset.

$$\begin{aligned} p &= m - n \\ d &= n \end{aligned} \tag{7.1}$$

Thus, to translate the Logical Address to a Physical Address, Equation (7.2) is used.

$$f = \big(i(p) \times s\big) + d \tag{7.2}$$

f: Resulting Physical Address.
$i(p)$: Mapped frame number of the given page number p from the Page Table.
s: Size of the page/frame.

If Process size is independent of page size, we expect Internal Fragmentation to average one-half page per process. This consideration suggests that small page sizes are desirable. However, overhead is involved in maintaining the Page Table itself, with each page-table entry increasing the overhead. Also, disk I/O is more efficient when the amount data being transferred is larger.

Generally, page sizes have grown over time as processes, data sets, and main memory have become larger. Today, pages typically are between 4 KiB and 8 KiB in size, and some systems support even larger page sizes. Some CPUs and kernels now support multiple page sizes. Variable on-the-fly page size is still being developed.

A 32-bit CPU uses 32-bit addresses, meaning that a given Process's memory space can only be 2^{32} bytes (4 GiB). However, a single 32-bit entry can point to one of 2^{32} different physical frames. If frame size is 4 KB (2^{12}), then a system with 4-byte (32-bit) entries can address 2^{44} bytes (or 16 TiB) of physical memory. Therefore, paging lets us use physical memory that is larger than what can be addressed by the CPU's address pointer length. This means that the size of physical memory in a paged memory system is different from the maximum logical size of a process.

When a Process arrives in the system to be executed, its size, in pages, is examined. Because pages and frames are the same size, if the process requires n pages, at least n frames must be available in memory. If the required n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the that frame's number is put in the Page Table for this process for this page. The next page is loaded into another frame, its frame number is put into the page table, and so on.

Paging offers a clear separation between the programmer's view of memory and the actual hardware. The logical addresses that the programmer uses are translated into physical addresses that the hardware uses. This mapping is hidden from the programmer and is controlled by the operating system. This allows the programmer to view memory as one contiguous space, containing only this one program (which helps create our definition of Virtual Memory). However, the user program may be scattered throughout Physical Memory, which also holds other programs. The difference between the programmer's view of memory and the actual physical memory is reconciled by the address-translation hardware in the Memory Management Unit.

Because the Operating System is managing Physical Memory, it must be aware of the allocation details of physical memory, including:

- Which frames are allocated.
- Which frames are available.
- The total number of frames.
- etc.

This information is generally kept in a data structure called a Frame Table.

Defn 133 (Frame Table)**.** The *frame table* has one entry for each physical frame, indicating whether it is free or allocated and, if it is allocated, to which page of which process or processes.

This behaves like an ownership or state table, rather than a lookup table. It says whether this frame is in use, and if it is, who is using it.

7.8.1 Hardware Support for Paging

Every access to memory must go through the paging map, so efficiency is a major consideration. Each Operating System has its own methods for storing Page Tables. Some allocate a page table for each Process, then a pointer to the page table is stored with the other register values in the Process Control Block. Other operating systems provide only a few page tables, which decreases the overhead involved when processes are Context Switched.

The hardware implementation of the Page Table can be done in several ways. In the simplest case, the page table is implemented as a set of high-speed, dedicated registers, making paging-address translation efficient. The CPU Dispatcher reloads these registers, along with all the other registers. Instructions to load or modify the page-table registers are privileged so that only the Operating System can change the memory map.

Registers can be used for the Page Table only if the page table is reasonably small. However, modern computers, allow the page table to be quite large, making the use of registers to implement the page table not feasible. Instead, the page table is kept in main memory, and a `page-table base register` (`PTBR`) points to the page table. Thus, changing the currently active page table requires changing only this one register, substantially reducing Context Switch time. However, now this requires 2 memory access:

1. Memory Access 1 is needed to enter the Page Table and find the frame entry.
2. Memory Access 2 is needed to travel to Physical Memory and retrieve the value.

7.8.1.1 Translation Look-Aside Buffers To accomodate the Page Table existing in memory, a hardware cache is usually provided on the CPU, called a *translation look-aside buffer* (*TLB*). The TLB is associative, high-speed memory that contains two part entries: a key and a value. When the associative memory is presented an item, the item is compared with all keys **simultaneously**. If the item is found, the corresponding value is returned. This search is very fast. In modern hardware, a TLB lookup is part of the instruction pipeline, essentially adding no performance penalty. To be able to execute the search within a pipeline step, the TLB is small, typically between 32 and 1,024 Page Table entries in size.

1. When a Logical Address is generated by the CPU, its page number is presented to the TLB.
2. These steps are executed as part of the instruction pipeline within the CPU, adding no performance penalty compared with a system that does not implement paging.
3. If the page number is found, its frame number is immediately available and is used to access memory.
4. If the page number is not in the TLB (a *TLB miss*), a memory reference to the Page Table must be made.
    - Depending on the CPU, this may be done automatically in hardware or via an Interrupt to the Operating System.
    - When the frame number is obtained, we can use it to access memory.
    - In addition, the page number and frame number are added to the TLB, so that they will be found quickly on the next reference.
    - If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies vary.
        - Least Recently Used (LRU)
        - Round-robin
        - Random
    - Some CPUs allow the operating system to participate in LRU entry replacement, while others handle the matter themselves.

Some TLBs allow certain entries to be wired down, meaning that they cannot be removed from the TLB. Typically, TLB entries for key kernel code are wired down.

TLBs are a hardware feature and the Operating System designer needs to understand the function and features of **that** hardware platform's TLBs. For optimal operation, Paging must be implemented according to the platform's TLB design. Likewise, a change in the TLB design may necessitate a change in the paging implementation of the operating systems that use it.

7.8.1.2 Address-Space Identifiers Some TLBs store *address-space identifier*s (*ASID*s) in **each** TLB entry. An ASID uniquely identifies each process and provides address-space protection for that process. When the TLB attempts to resolve virtual page numbers, it ensures that the ASID for the currently running Process matches the ASID associated with the virtual page. **If the ASIDs do not match**, the attempt is treated as a **TLB miss**. ASIDs also allow the TLB to contain entries for several **different** processes simultaneously. If the TLB does not support separate ASIDs, every time a new page table is selected (context switch), the TLB must be flushed (erased) to ensure that the next executing process does not use the wrong translation information. Otherwise, the TLB could include old entries that contain valid virtual addresses but have incorrect or invalid physical addresses left over from the previous process.

7.8.2 Memory Protection

The User Process, by definition, is unable to access memory it does not own. The user process has no way of addressing memory outside of its Page Table, since the page table includes **only** those pages that this process owns. In addition, the operating system must be aware that user processes operate in **user space**, and all Logical Addresses must be mapped to the proper user memory to produce Physical Addresses. If a user makes a system call and provides an address as a parameter (a buffer, for instance), that address must be mapped to produce the correct physical address.

On top of the ability for **only** the Operating System to manipulate the currently active memory map, there are explicit bits that are used to protect the current memory map and its pages. Normally, these bits are kept in the page table and are associated with each page.

7.8.2.1 Page Permission Bits Some bits, the *permission bits*, can define a page's permissions (read–write, read-only, execute-only, etc.) or any combination of these. Every memory reference goes through the page table to find the frame number. At the same time the physical address is being computed, the permission bits can be checked. A violation of the given permissions will cause a hardware Trap to the operating system (for memory-protection violation).

7.8.2.2 Page Valid-Invalid Bit One additional bit is generally attached to each entry in the page table: a *valid–invalid bit*.

- When this bit is set to valid, the associated page is in the Process's Logical Address Space and is a legal (or valid) page.
- When the bit is set to invalid, the page is not in the process's logical address space.

Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

This runs into issues when Internal Fragmentation occurs. A Process can request a page, but not need all of it, and the permission bit applies to the whole page, it is possible to access a memory address in the page, but is not used by the Process.

7.8.2.3 Page-Table Length Register Because a Process may be given a much larger Logical Address Space than it will use, it would be wasteful to create a Page Table with entries for every possible page. Some systems provide hardware, in the form of a *page-table length register* ($PTLR$), to indicate the size of the page table. This value is checked against every Logical Address to verify that the address is in the valid range for the process. Failure of this test causes an error Trap to the Operating System.

7.8.3 Shared Pages

An advantage of paging is the possibility of sharing common code. This consideration is particularly important in a time-sharing environment. If the code is Reentrant code (or pure code), the code pages can be shared. Each Process will then get its own data page. Thus, two or more processes can execute the same code at the same time, while maintaining its own data.

Defn 134 (Reentrant)**.** *Reentrant* means that something can be reentered at any time, and it will behave the same as if it were entered as soon as it were reached. Reentrant code then, is non-self-modifying code: it never changes during execution.

Heavily used programs can be shared, such as:

- Compilers
- Window systems
- Run-time libraries (And through Dynamic Linking)
- Database systems
- etc.

To be sharable, the code must be Reentrant. The read-only nature of shared code **should not** be left to the correctness of the code; the **Operating System should enforce this property**.

The sharing of memory among Processes on a system is similar to the sharing of the address space of a process by multiple Threads. Furthermore, Shared-Memory can be used as a method of interprocess communication (Paragraph 3.4.5.2). Some Operating Systems implement shared memory using shared pages. Organizing memory according to pages provides numerous benefits in addition to allowing several processes to share the same physical pages.

7.9 Page Table Structure

Because of the ever-increasing size of Page Tables, we need to start organizing them more efficiently. For example, consider a system with a 32-bit logical address space. If the page size in such a system is 4 KiB[6] (2^{12}), then a page table may consist of up to 1 million entries ($\frac{2^{32}}{2^{12}}$). Assuming that each entry consists of 4 bytes, each process may need up to 4 MiB of physical address space for the page table alone. Allocating this page table contiguously is not what we want to do, because if not all the pages are in-use, then we want to be able to use the space that would otherwise be taken up by the page table.

There are 3 techniques for handling this discussed in the textbook:

1. Hierarchical Paging
2. Hashed Page Tables
3. Inverted Page Tables

[6] KiB is a kibibyte, or 2^{10} bytes

7.9.1 Hierarchical Paging

One simple solution to this problem is to divide the page table into smaller pieces. We can accomplish this division in several ways. One way is to use a two-level paging algorithm, in which the page table itself is also paged.

For example, consider the system with a 32-bit Logical Address Space and 4 KiB again. Fromt he perspective of the CPU, theLogical Address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. However, because we page the page table, the page number is further divided into a 10-bit outer page number and a 10-bit page offset.

If the memory capacity of the computer increased, and therefore the page count did as well, then this would start becoming problematic again. We could divide the outer page table and page it again, giving us a three-level Paging scheme. If we continued to do this, a machine with a 64-bit Logical Address Space would require 7 level of paging. Remember that each time a Page Table is used, we need another memory lookup and access. This makes multiple levels of paging prohibitively slow.

7.9.2 Hashed Page Tables

A hashed page table, is a Page Table where the page number is the hash value. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields:

1. Hashed page number.
2. Value of the mapped frame.
3. Pointer to the next element in the linked list (or `NULL`).

The algorithm works as follows:

Algorithm 7.1: Hashed Page Table Usage

```
1 The page number in the Logical Address is hashed into the hash table.
2 The virtual page number is compared with field 1 in the first element in the linked list. if Fie
    then There is a match
3   Corresponding page frame (Field 2()) is used to form the desired physical address.
4 else There is no match
5   subsequent entries in the linked list are searched for a matching virtual page number.
```

7.9.2.1 Clustered Page Tables A variation of Hashed Page Tables that is useful for 64-bit address spaces, called *clustered page tables*, has been proposed. They are similar to their origin, except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single clustered Page Table entry can store the mappings for multiple frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous **AND** scattered throughout the address space.

7.9.3 Inverted Page Tables

Usually, each Process has an associated page table with one entry for each page that is in use. However, each Page Table may consist of millions of entries, and there may be many page tables. These tables may consume large amounts of physical memory just to keep track of how memory is being used.

To solve this problem, we can use an inverted page table. An inverted page table has **one entry for each frame** of memory. Each entry consists of

- The Logical Address of the page stored in that physical frame
- Address-Space Identifiers, information about the Process that owns the page.
  - The table usually contains several different process address spaces that are mapped to physical memory.
  - Ths way the logical page for a particular process is mapped to the correct corresponding physical frame.

Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

To use the table, a tuple (Equation (7.3)) is created by the CPU as the Logical Address.

$$\langle \texttt{PID}, p, d \rangle \tag{7.3}$$

`PID`: The Process ID.
p: The page number.
d: The offset of the desired location within the frame, i.e. in physical memory.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the table because it is sorted by physical address, and lookups occur on virtual addresses, meaning the whole table might need to be searched before a match is found. To skirt this issue, a hash table is used, just like in Hashed Page Tables to limit the search. However, each access to the hashed inverted page table adds another memory reference. So one virtual memory reference requires at least two real memory reads:

1. One for the hashed inverted page table entry.
2. One for the resulting page table.

One downside of inverted page tables is that Shared-Memory is difficult to implement. Shared memory is usually implemented as multiple Logical Addresses (one for each Process sharing the memory) that are mapped to the same Physical Address. However, the usual method does not work here, because there is only one entry for every frame, one frame cannot have multiple shared Logical Addresses mapped to it.

8 Virtual Memory

Defn 135 (Virtual Memory)**.** Virtual memory involves the separation of logical memory as perceived by users from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming much easier, because the programmer no longer needs to worry about the amount of physical memory available; she can concentrate instead on the problem to be programmed.

Defn 136 (Physical Memory)**.** *Physical memory* is the memory that is physically installed in the computer. There is a finite amount of this, determined by how much is installed by the system designer.

Defn 137 (Virtual Address Space)**.** The *virtual address space* of a Process consists of all the Virtual Addresses generated by a Program. It refers to the logical (virtual) view of **how a Process is stored in memory**. Typically, this view is one of perfectly continguous memory locations.

Remark 137.1. The Virtual Address Space is only calculated by one Program/Process at a time. To find the total virtual address space used, all Processes must have their virtual address spaces aggregated.

9 Network

Defn 138 (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 138.1 (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.