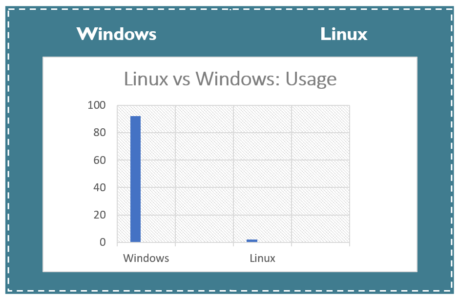
**COP 4610 Operating Systems Project**

**By: Dov Cattan, Hashir Dagra, Giovanni Michel**

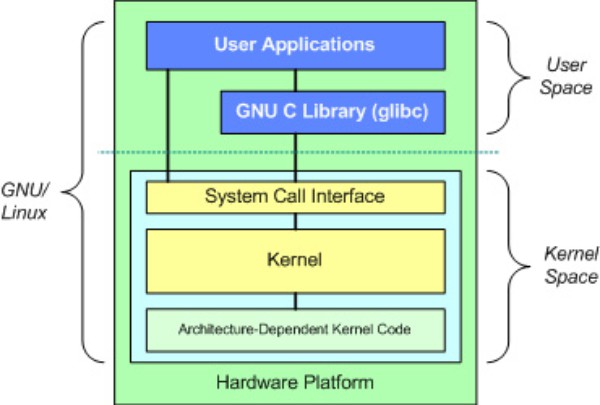
**Introduction**

One of the most common definitions for an operating system is software that manages and controls a computer’s hardware. With the operating system, programs and applications can be executed between the user and hardware with the operating system being the intermediary between the two. When building an operating system there are many things to consider such as who will be using this operating system on a user point of view and the type of computer hardware that will be needed to support this operating system. An efficient operating system makes it easy to provide an environment so that user can be comfortable to navigate through UI to perform task and run applications. Little does the average person know, but operating systems are everywhere from in your security system running through the house or the operating system in your car, or maybe even the operating system that is initiated when NASA sends a rocket into space. Since so many operating systems have been created in these past fifty years it’s hard to just pick one, in this case we will be comparing two of one of the most common operating systems Windows and Linux. Windows and Linux are both populator operating system’s when asking anyone no matter the job title or not considering the cost of the computer. In this research document we will be reviewing Processes, comparing the two types of implementations of these processes in Windows and Linux based operating systems. This research experiment will cover process states, program counters, the different attributes of a process and different ways the OS schedules these processes.

To begin with, the differences between Windows and Linux range from the types of kernels to the usage rate of the CPU, for example Linux has 3 types of users that can be accessed on the operating system such as (Regular, Administrative(root) and Service users), while windows have 4 types of users that can be accessed (Administrator, Standard, Child and Guest). Some might prefer Windows because the user-friendly user-interface, which is understandable when market data suggests that most people have Windows running on there PC’s, while Linux would be almost extinct when comparing usage.



OS Architecture Windows vs Linux

Linux and Windows both have similarities and differences in the way their Operating System architecture works. To start off, there are similarities and differences between the ways Linux and Windows have their user and kernel spaces set up. For kernel space, both Linux and Windows have hardware, Operating System Kernels themselves, and kernel drivers or code set up in their respective kernel spaces. However, for additional components, while Linux just has the system call interface in their kernel space, Windows has their Hardware Abstraction Layer (HAL), which is in between the kernel and the hardware, and has their file system drivers in the kernel as well. For user space, both Linux and Windows have of course user applications because that how the user can operate their computer. The differences between the user spaces of Linux and Windows are that while Windows needs drivers for their user mode to truly work, Linux has it all built in within their kernel. Another notable difference is while Linux has their GNU C library (glibc) written in the user space, Windows replaces that with their APIs. Diagram

Description automatically generated

With the user space and kernel space structures mentioned, it is time we speak about which kernel structures themselves are Linux and Windows implementing. While the Linux Kernel is a microkernel, the Windows kernel is rather monolithic. The reason why mentioning this is important, is because unlike Linux’s microkernel, Windows’s monolithic kernel has user services and kernel services implemented under the same address space which makes it faster, while a microkernel would have them run on different addresses which causes it to be extendible and makes it slower. A big advantage for having a microkernel is that when the kernel service crashes, only that working kernel gets fixed as it should, and the system is fine. On the other hand, when the monolithic kernel’s service crashes, the entire system i.e., computer crashes along with service failure as a security measure. It is worth noting that it is also possibly an effect of monolithic kernels using up more space than microkernels, despite that a microkernel has a lot more code than a monolithic kernel.

Table

Description automatically generated

Linux has both production and development kernels running with the developments being the odd number version while the even number is the production version which is the kernel the user uses. Windows on the other hand, has the kernel running accordingly to which windows version it uses. To conclude, while Windows and Linux share similar features in their architectures for their respective Operating Systems to work, they also have important differences in their architecture to show their unique ways of implementing certain functions within them.

**Process states of Windows and Linux**

A Program doesn't do anything except if its instructions are executed by a CPU. A program in execution is known as a cycle. To achieve its undertaking, process needs the PC assets.

There may exist more than one cycle in the framework which might require a similar asset simultaneously. In this manner, the working framework needs to deal with every one of the cycles and the assets in a helpful and productive manner.

A few assets might should be executed by one cycle at one time to maintain the consistency if not the framework can become conflicting and stop might happen.

The working framework is answerable for the accompanying exercises regarding Process Management

**Attributes of a process**

The Attributes of the process cycle are utilized by the Operating System to make the process control block (PCB) for every one of them. This is additionally called (context) or the setting of the interaction. Attributes which are put away in the PCB are portrayed underneath.

**Process ID**

When a process is created, a unique id is assigned to the process which is used for unique identification of the process in the system.

**Program counter**

A program counter stores the address of the last instruction of the process on which the process was suspended. The CPU uses this address when the execution of this process is resumed.

**Process State**

The Process, from its creation to the completion, goes through various states which are new, ready, running and waiting.

**Priority**

Every process has its own priority. The process with the highest priority among the processes gets the CPU first. This is also stored on the process control block.

**General Purpose Registers**

Every process has its own set of registers which are used to hold the data which is generated during the execution of the process.

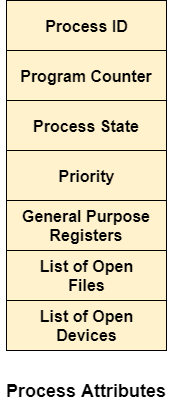
**List of open files**

During the Execution, every process uses some files which need to be present in the main memory. OS also maintains a list of open files in the PCB.

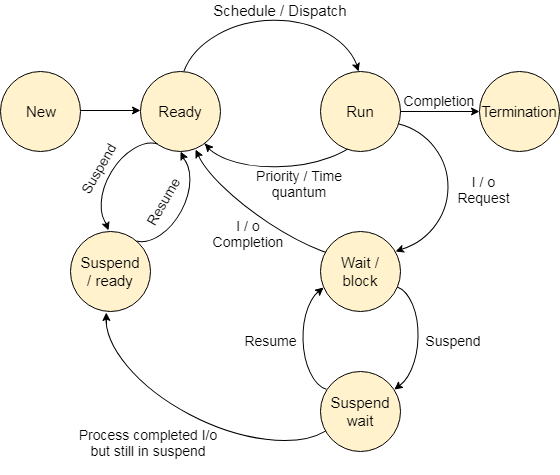
**List of open devices**

OS also maintain the list of all open devices which are used during the execution of the process.

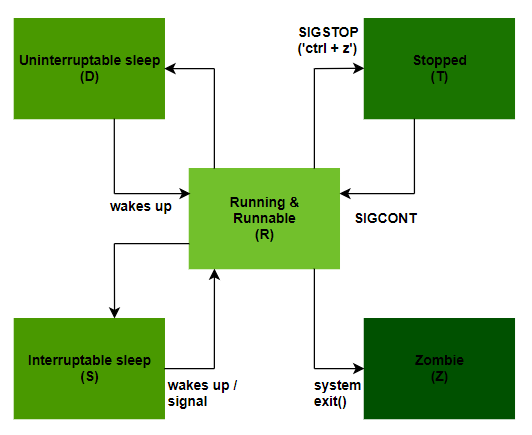
**Diagram**



**Process states: windows**



**Process state Linux**



**New state**

A new state is in which all the programs arrive at the operating system

**Ready state**

Ready state is referred to when the new programs are ready to be executed. Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory.

The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state

**Running state/ running or runnable state**

One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system, then we can have n processes running simultaneously.

**Waiting state/ stopped state**

From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process.

When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.

**Termination state/ zombie state**

When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

**Suspend ready/ sleeping state**

A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.

If the main memory is full and a higher priority process comes for the execution, then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.

**Suspend wait**

Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.

**Schedulers**

A process scheduling is a process managing activity that handles removing a running process from the CPU and selecting another process based on a particular strategy. Process scheduling is an important part of a multiprogramming operating system. In such operating systems, multiple processes can be loaded into executable memory at the same time, and the loaded processes use time division multiplexing to share the CPU.

The operating system keeps all PCBs in the process scheduling queue. The OS maintains a separate queue for each process state, and the PCBs of all processes in the same running state are placed in the same queue. When the status of a process changes, the PCB is disconnected from the current queue and moved to the new status queue. The operating system maintains the following critical process scheduling queues- Job Queue-This queue holds all processes in the system. Ready Queue-This queue keeps a set of all processes in main memory ready and waiting to run. New processes are always placed in this queue. Device Queues-These queues are processes that were blocked because I / O devices were unavailable.



The scheduler is special system software that processes process plans in different ways. The main task is to select the jobs to send to the system and determine the process to run. There are three types of schedulers:

* Long-term scheduler
* Short-term scheduler
* Medium-term scheduler

**Long-term scheduler**

Also known as job scheduler. The long-term scheduler determines which programs are included in the system for processing. Select a process from the queue and load it into memory for execution. The process is loaded into memory for CPU scheduling. The main goal of Job scheduler is to provide a balanced combination of jobs. B. I / O bound and processor bound. It also controls the degree of multiprogramming. If the level of multiprogramming is stable, the average speed of process creation should be equal to the average speed of processes leaving the system. On some systems, the long-term scheduler may not be available or may be minimal. Timeshare operating systems do not have a long-term scheduler. When the process status changes from new to ready, the long-term scheduler is used.

**Short-term scheduler**

Also known as the CPU scheduler. Its main goal is to improve system performance according to selected criteria. This is a change from the ready state to the running state of the process. The CPU scheduler selects a process from among the processes that are ready to run and assigns a CPU to one of them. The short-term scheduler, also known as the dispatcher, determines the next process to run. Short-term planners are faster than long-term planners.

**Mid-term scheduler**

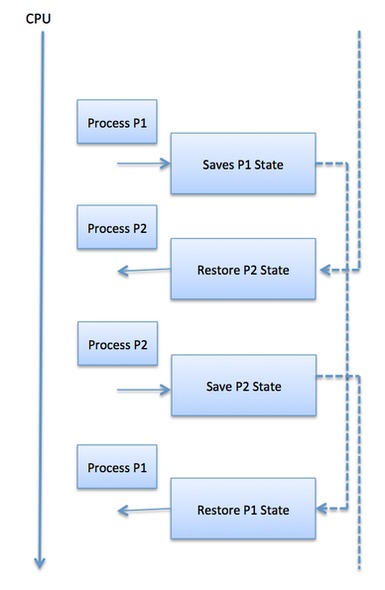
Mid-term scheduler is part of the exchange. Delete the process from memory. Reduce the degree of multiprogramming. The medium-term scheduler takes over the processing of the outsourced-out process. Making a I / O request can disrupt a running process. Paused processes cannot proceed to completion. In this state, the suspended process is moved to secondary storage, the process is removed from storage, and space is reserved for other processes. This process is called swapping, and this process is called outsourcing or rollout. It may need to be replaced to improve the process mix.

## **Comparison among Schedulers**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Long-Term Scheduler** | **Short-Term Scheduler** | **Medium-Term Scheduler** |
| 1 | It is a job scheduler | It is a CPU scheduler | It is a process swapping scheduler. |
| 2 | Speed is lesser than short term scheduler | Speed is fastest among other two | Speed is in between both short- and long-term scheduler. |
| 3 | It controls the degree of multiprogramming | It provides lesser control over degree of multiprogramming | It reduces the degree of multiprogramming. |
| 4 | It is almost absent or minimal in time sharing system | It is also minimal in time sharing system | It is a part of Time-sharing systems. |
| 5 | It selects processes from pool and loads them into memory for execution | It selects those processes which are ready to execute | It can re-introduce the process into memory and execution can be continued. |

A context switch is a mechanism for saving and restoring a CPU state or context to a process control block so that process execution can be resumed at the same point in time. This technique allows the context switcher to allow multiple processes to share a single CPU. Context switching is an important part of the functionality of a multitasking operating system. When the scheduler switches the CPU from running one process to running another, the status of the currently running process is saved in the process control block. The status of the next process is then loaded from its own circuit board and used to configure PCs, registers, and so on. At this point, you can start running the second process. Changing the context is computationally expensive because the status of the registers and memory must be saved and restored. To avoid long context switch times, some hardware systems use a set of two or more processor registers. When the process switches, the following information is saved for later reference:

* Program counter
* scheduling information
* Base value and limit register value
* Currently used register
* Changed state
* I / O State information
* Accounting information



There are six popular process scheduling algorithms

First-Come, First-Served (FCFS) Scheduling

* Shortest-Job-Next (SJN) Scheduling
* Priority Scheduling
* Shortest Remaining Time
* Round Robin (RR) Scheduling
* Multiple-Level Queues Scheduling

**Scheduling in Linux**

Linux is a preemptive operating system. The preemptive operating system determines when to stop a process and when to start a new process. The time a process runs is usually determined before it is planned. This is called a time slice and is effectively part of the processor time.

In collaborative operating systems, the scheduler is process-dependent and explicitly informs the scheduler that it is ready to stop (this is often referred to as yielding). The problem with collaborative operating systems is that unyielding tasks can crash the entire operating system. The last mainstream collaborative operating systems were Mac OS 9 and Windows 3.1.

The Linux scheduler has gone through some iterations. The latest scheduler, CFS (Completely Fair Scheduler), uses the concept of fair scheduling from queuing theory. The Schedule Guidelines are the rules that planners follow to determine what to do and when. An effective scheduling policy needs to consider both kinds of processes: I/O bound processes and CPU bound processes. I/O bound processes spend most of their time waiting for I/O operations, like a network request or keyboard operation, to complete. GUI applications are usually I/O bound because they spend most of their time waiting on user input. I/O bound processes often run for a short time because they block while waiting for I/O operations to complete. CPU bound processes spend most of their time executing code. CPU bound processes are often preempted because they don`t block on I/O requests very often. An example of a CPU bound task would be one that performs a lot of Math calculation, like MATLAB.

Some processes are I/O bound and CPU bound at different times. For example, a word processor is normally waiting for user input, but there might be regular CPU intensive operations like spellchecking. One type of scheduling algorithm is priority scheduling, which gives different tasks a priority based on their need to be processed. High-priority tasks run before low-priority tasks, and processes with the same priority are scheduled in a round-robin fashion.

The kernel uses two separate priority values. Great value and real-time priority value. The nice value is a number between 20 and +19, with a default value of 0. The higher the nice value, the lower the priority (the process becomes nice by running another process instead). Processes with a low nice value receive a large share of system processor time, and processes with a high nice value receive a small share. Nice values ​​are the standard priority range for Unix systems, but values ​​are used differently by operating system. On OS X, the nice value controls the absolute time slice assigned to a process. On Linux, the nice value controls the ratio of time slices. Real-time priority values ​​range from 0 to 99, but the values ​​are configurable. Real-time values ​​are the opposite of nice values. The higher the value, the higher the priority. "All real-time processes have a higher priority than regular processes, which means that real-time and nice values ​​are in a relatively prime space."

**Linux Scheduling Algorithms**

The Linux scheduler supports different scheduling algorithms for scheduling different types of processes. These are known as scheduler classes. Each scheduler class has a different priority. The scheduler iterates over each scheduler class in order of priority. The highest priority scheduler class in the runnable process takes precedence, and the winning scheduler class selects the next process to run. The CFS scheduler class is a class registered for normal processes (SCHED\_NORMAL). CFS is defined in kernel / sched\_fair.c.

**Process Scheduling on Unix Systems**

There are some issues with traditional Unix scheduling, each process is assigned an absolute time slice, and the nice value affects the absolute time slice: to the absolute time slice.

A standard time slice value assignment for the nice value assignment. This can lead to suboptimal behavior. Leveling up or down the process has different effects depending on the starting value of Nice.

Absolute time slices must be multiples of timer ticks and can cause problems. Prioritizing newly awakened tasks can lead to situations where processes are improperly allocated a lot of time. These issues were solved by the CFS approach, which eliminates time slices and allocates a share of the processor to each process. "Therefore, CFS provides some fairness, but exchange rates fluctuate.

" Complete Faire Scheduling "CFS is based on a simple concept. Model the process plan as if the system had an ideal complete multitasking processor. In such a system, each process has processor time. Receives 1 / n, where n is the number of runnable processes and schedules them in an infinitely short time so that all n processes are present in a measurable time. do". CFS runs each process for a period and then selects the next process with the least number of runs. CFS calculates the time it takes to run a process as a function of the total number of runnable processes.

The nice value is used to weight the percentage of processor time that a process receives. This solves the problem that the effect of increasing or decreasing the value depends on the first nice value, because the nice value makes a geometric difference rather than an additive difference. Each process runs in a time slice in proportion to its weight divided by the total weight of all executable processes.

CFS sets a target wait time, which is the total time all processes need to run. For example, if the target latency is 10 ms, then two evenly weighted tasks will run for 5 ms each, and five tasks will run for 2 ms each. To avoid excessive context switching costs, there is a minimum amount of time (called minimum particle size) that a process can run. This is set to 1 millisecond by default.

**Scheduling in windows**

Windows implements a priority-controlled preemptive scheduling system. The thread selected for execution is always executed with the highest priority runnable (ready) thread, with the limitation that it may be restricted by the processors that the thread is allowed to execute.

When a thread to run is selected, it runs for a period called Quantum. Quantum is the length of time a thread is allowed to run before another thread of the same priority level (or higher level that can occur on a multiprocessor system) runs. Quantum values ​​can vary from system to system and from process to process for three reasons: system configuration settings (long or short quantum), process foreground / background status, or using job objects to change quantities. However, threads may not be able to complete the Quantum.

Windows implements a preemptive scheduler, so if another high-priority thread is ready to run, the currently running thread may end prematurely before the time slice ends. In fact, you can select a thread to run next and end prematurely before the quantum begins. Windows scheduling code is implemented in the kernel. However, there is no single "scheduler" module or routine. The code is distributed throughout the kernel where scheduling-related events occur. The routines that perform these tasks are collectively called the kernel dispatcher. The following events may require thread dispatch.

* A thread becomes ready to execute—for example, a thread has been newly created or has just been released from the wait state.
* A thread leaves the running state because its time quantum ends, it terminates, it yields execution, or it enters a wait state.
* A thread’s priority changes, either because of a system service call or because Windows itself changes the priority value.
* A thread’s processor affinity changes so that it will no longer run on the processor on which it was running.

At each of these junctions, Windows needs to decide which thread to run next. When you select a new thread that Windows runs, the context switches to that thread. A context switch is a procedure for saving the state of a volatile machine associated with a running thread, loading the volatile state of another thread, and starting the execution of a new thread. As mentioned earlier, Windows plans with thread granularity. This approach makes sense given that the process is not running, but it only provides the resource and the context in which the thread runs. Planning decisions are made strictly on a thread-by-thread basis, so it does not consider which process the thread belongs to. For example, if process A has 10 runnable threads, process B has 2 runnable threads, and all 12 threads have the same priority, then each thread is theoretically of CPU time. Get 1/12-Windows doesn't provide 50% of CPU Process A and 50% Process B to it.

**Dead Locks**

A deadlock is a situation that occurs in the operating system when a process enters a wait state because another waiting process is holding the requested resource. Deadlocks are a common problem in multiprocessing, where multiple processes share certain types of mutually exclusive resources, called soft locks or software.

**Example of dead lock**

* A real-world example would be traffic, which is going only in one direction.
* Here, a bridge is considered a resource.
* So, when Deadlock happens, it can be easily resolved if one car backs up (Preempt resources and rollback).
* Several cars may have to be backed up if a deadlock situation occurs.
* So starvation is possible.

Diagram, shape, rectangle

Description automatically generated

**What is circular wait**

One process is waiting for resources held by the second process, and the second process is also waiting for resources from the third process. This continues until the last process waits for the resource held by the first process. This will create a circular chain. For example, process A is assigned resource B when requesting resource A. Similarly, process B is assigned resource A and requests resource B. This creates a circular wait loop.

**Example of circulation waiting time**

For example, a computer has three USB drives and three processes. Each of the three processes can accommodate one of the USB drives. Therefore, if each process requests a different drive, the three processes will be deadlocked because each process is waiting for the USB drive currently in use to be released. This will create a circular chain.

Diagram, schematic

Description automatically generated

**Dead lock detection**

The occurrence of a deadlock can be detected by the resource scheduler. The resource scheduler helps the operating system keep track of all resources assigned to different processes. Therefore, if a deadlock is detected, you can resolve the deadlock using the following methods:

**Dead lock prevention**

It is important to prevent deadlocks before they occur. The system checks all transactions before execution to ensure that no deadlock conditions occur. As a result, even small changes can lead to deadlocks, and operations that could lead to deadlocks in the future did not allow the process to run. There are several methods used to prevent at least one condition from being met.

**No preemptive action**

No Preemption – A resource can be released only voluntarily by the process holding it after that process has finished its task

* If a process which is holding some resources request another resource that can’t be immediately allocated to it, in that situation, all resources will be released.
* Preempted resources require the list of resources for a process that is waiting.
* The process will be restarted only if it can regain its old resource and a new one that it is requesting.
* If the process is requesting some other resource, when it is available, then it was given to the requesting process.
* If it is held by another process that is waiting for another resource, we release it and give it to the requesting process.

**Mutual exclusion**

Mutual exclusion is the complete form of mutexes. This is a special type of binary semaphore used to control access to shared resources. It includes a priority inheritance mechanism to avoid the problem of high priority inversion. This allows you to keep your current high priority task blocked for the shortest amount of time. Shared resources such as read-only files do not lead to deadlocks, but resources such as printers and tape drives require exclusive access by a single process.

**Hold and wait**

In this state, you need to prevent a process from holding one or more resources while the process is waiting for one or more other resources at the same time.

**Circular wait:**

This enforces the overall order of all resource types. Circular wait also requires each process to request resources in ascending order.

**Evaluation**

At the end of the day, these two operating systems have their own pros and cons, so the best way to figure out which one is the most efficient and on costly point of view. The Linux kernel uses a micro-kernel which requires more space compared to the Windows monolithic kernel but when comparing the system running efficiency Windows is more superior. Windows takes with letting their users of Windows be certain that almost any software is compatible of there PC because of Windows great legacy support. Windows also has an edge when mentioning what is used most by non-technical users, with the UI being user-friendly for example, if you are a gamer most would say that Windows is the best operating system to run on your gaming PC. On a programming, reverse engineer or cybersecurity level, others will say that Linux is the best for programming because it supports all the major programming languages and IDE’s. Also, the Linux terminal is a lot more powerful compared to the Window’s command line for developers. Lastly, Linux is Open-Source as opposed to Windows which is Proprietary anybody can edit and make tweaks whenever they wish.

To conclude, after reading this article the reader will understand the different Processes being used on the two OS (Linux and Windows), learning everything from process states, program counters, how processes support the difference kernels and which OS is used more by most customers. Even though we did not discuss every criterion needed to build a OS the reader will understand the concepts mentioned in these experiments to use on their own.

**Works cited**

“Os Process States - Javatpoint.” *Www.javatpoint.com*, <https://www.javatpoint.com/os-process-states>.

“OS Attributes of a Process - Javatpoint.” *Www.javatpoint.com*, https://www.javatpoint.com/os-attributes-of-a-process.

Lakshmanan, Ram, et al. “What Are the Process States in Unix/Linux?” *JAXenter*, 19 Feb. 2021, https://jaxenter.com/linux-process-states-173858.html.

“States of a Process in Operating Systems.” *GeeksforGeeks*, 21 Sept. 2021, https://www.geeksforgeeks.org/states-of-a-process-in-operating-systems/.

“Linux vs Windows.” Edureka, Nov 25,2020, https://www.edureka.co/blog/linux-vs-windows/#linuxvswindows

*Components of Linux System ?* Linux Delhi. (2016, January 27). Retrieved November 11, 2021, from https://linux-delhi.org/components-of-linux-system/.

Viviano, A. (2021, August 18). *User mode and kernel mode - windows drivers*. Windows drivers | Microsoft Docs. Retrieved November 11, 2021, from https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode.

“Difference between Linux and Windows.” *GeeksforGeeks*, 13 May 2020, https://www.geeksforgeeks.org/difference-between-linux-and-windows/.

“Monolithic Kernel and Key Differences from Microkernel.” *GeeksforGeeks*, 5 Aug. 2021, https://www.geeksforgeeks.org/monolithic-kernel-and-key-differences-from-microkernel/.