Sean McShane

Operating Systems and Computer Architecture

9 April 2016

*Linux*

Customizable, open-source, and reliable are just a few adjectives that can be used to describe the Linux operating system. Basing the operating system as a derivation of UNIX, Linus Torvalds designed Linux as a college student. This operating system quickly became well-known in the tech community because of several features that made it different than the ordinary operating system. Some of the most recognizable features of Linux are that it is secure, programmer-friendly, and free. However, the most important feature that Linux has over its competitors: it is open-source. This means that after Linus publically released the operating system, developers were allowed to tweak it and release their own distributions built on top of the Linux kernel.

The Linux kernel boasts its modular structure over the less dynamic UNIX monolithic kernel. Monolithic structures basically have the whole operating system in one address space and have weaknesses when changes are made to the kernel. Linux’s modular structure eliminates this problem. As hinted by the name, modular structures are built using a collection of modules. A module in this sense can be anything that is directly apart of the kernel, for example a driver for a piece of hardware. “Related subroutines, data, and entry and exit points are grouped together in a single binary image, a loadable kernel object, called a module” (2). Each module has the ability to dynamically link. This means that the kernel does not have to stop and reload/relink other modules when a new one comes in. Rather, the new module can be loaded and linked while the kernel is already processing. This allows the kernel to continue running very efficiently.

In order for this whole modular structure to work, modular hierarchy is used in conjunction with a module table. The modular hierarchy reduces replication of code by creating dependences between similar modules which allows added efficiency by allowing the kernel to ensure the correct kernels are present before loading or unloading an unnecessary module. The module table is a type of linked list, with each record having multiple fields and pointers. Some fields include the size of the module, how many processes are using the module, and certain flags the module can set, among others. A record’s pointers include a next module pointer, a pointer to the module’s name, and a pointer to a list of modules that use the current module, among others. Each record also has a pointer to the current module’s symbol table. The module’s symbol table is used to list name/value pairs of symbols defined in the module. Overall, the modular architecture of the Linux kernel is very efficient over other kernel structures.

An important thing an operating system does is handling how it manages memory. Virtual memory allows an operating system to address more memory locations than it physically has access to. In Linux, this is done by taking advantage of paging. In order to address processes, a three-level page table is used. The first level in this page table is called the page directory. This level consists of one page and is used as multiple pointers to separate pages in the next level. The next level is called the page middle directory because it’s the median of communication between the first and last level of the table. This middle level consists of pointers to separate pages in the last level of the table. The last level is named the page table and is the final step in addressing virtual memory addresses. This table consists of pointers to virtual pages of the current process. Using each level of the overall page table as an index field in the virtual address, this allows easy addressing and tracing of addresses assigned to the process. A final, fourth field is added onto the virtual memory address to define the offset where the process is located in the page referenced to by the first three fields.

When dealing with paging, reading and writing to and from main memory and disk is very expensive and must be done efficiently. Without a good setup for this, there is a risk that the whole system can be slowed down. A big part in this process is utilizing the way page frames are used. Linux realizes the principle of locality and uses the buddy system to handle contiguous pages. These pages are organized into page frames of sizes of increasing powers of two up to thirty-two. This makes organizing pages easy when pages are added and deleted from the frame.

Another big part of ensuring efficient paging is making sure pages are replaced smartly. Linux uses a modified version of the LRU (Least Recently Used) algorithm in order to replace pages. When a page is inactive, it is placed on its “zone’s” inactive list. Similar to when a page is active, it is placed on its “zone’s” active list. These zones consist of other pages with similar virtual addresses. To execute the replacement algorithm, a background process is ran every once in a while. This process sweeps over the zones and reads the active and inactive lists. If the process, kswapd, reads a list, active or inactive, and the page read is being read by the process for the first time, a flag is set. This page must be read again before it switches lists. For example, an inactive page becomes active and vice-versa. However, this second access must be done within a certain period of time before the flag set expires. This allows the inactive pages to replace active ones in a simple, yet efficient manner.

The memory dealt with so far has been mainly focused around virtual memory. However, the kernel also deals with pages frames in strictly physical memory. Its job is to “allocate and deallocate frames for particular uses” (1). For example, certain frames can be allocated towards processes, code, or used as cache memory. The kernel handles this by once again, using a buddy system. However, alongside with this system an algorithm called “slab allocation” is used in order to allocate more pages at one time. Slab allocation is similar to the buddy system and allows a varying size of sets of pages to be moved in between a series of linked lists, in order to allocate the physical memory addresses efficiently.

Just like any other operating system, Linux’s main job is handling processes and making sure scheduling of these processes is done correctly. To understand how Linux takes on this problem, one must first understand how processes are represented in this operating system. Firstly, in Linux, processes are represented as task\_structs. This data structure contains certain data about the current process and could relate to other processes as well. For example, the struct contains information about the state of the process which could be used to tell where the scheduler has placed the process. Other data concerning the process stored in task\_struct include the process’s identifier, links to parent/sibling processes, and space designated to the process. Another important field in the process’s struct is information regarding scheduling. In Linux, a process can be considered ‘real time’, which tells the scheduler to schedule this process before ‘normal’ ones. Along with this type of priority, priorities can be relatively defined to determine which process is executed when.

Linux uses an interesting solution when it comes to multithreading. In order to understand this solution, one must trace the process step by step. First, the process is broken into multiple threads in order for the process to execute and finish efficiently. Next, the threads are mapped as separate processes to the kernel. This part is what differs from other operating systems. “To the Linux kernel, there is no concept of a thread. Linux implements all threads as standard processes… Instead, a thread is merely a process that shares certain resources with other processes.” (2). For this to work, the processes, which are actually threads, contain the same group ID in order to maintain some type of knowledge that the processes are related.

There are multiple different process scheduling algorithms that have their own advantages. Linux takes advantage of this fact by using different scheduling algorithms for different scenarios. According to the textbook, there are three different scheduling classes in Linux, “SCHED\_FIFO…SCHED\_RR…SCHED\_OTHER” (1). The first (first-in-first-out) and second (round-robin) both handle situations regarding the Linux defined, ‘real-time’ threads. The last class deals with non-‘real time’ threads.

In each respective scheduling class, there are rules built on top of the actual algorithm to smooth the scheduling in Linux. One aspect that Linux uses in these classes is priorities. As seen before, real-time processes have priority over normal, non-real-time, processes. Built on top of these priorities are relative priorities inside of the classes determining which real-time process has the highest priority and the same for non-real-time class. To calculate these priorities, Linux takes into account the processes execution behavior. Since I/O processing is very slow, a process that relies on I/O would have a higher priority over a process relying solely on the processor. Another thing important in calculating the priority is making sure a process doesn’t starve for too long. This is why how much time a process sleeps correlates with a higher priority.

As CPU’s got more advanced and multiple processors were added, the O(1) scheduler was born in order to efficiently handle mapping processes to separate processors. Problems with the old scheduler revolve around inefficiently scheduling and handling processes. To instill the idea that this new scheduler was better, it was given its name because “the time to select the appropriate process and assign it to a processor is constant, regardless of the load on the system or the number of processors” (1). It achieves this feat by using priority queues and bitmap arrays for each priority level. Each level includes two priority queues and two bitmap arrays, one set for active processes and one set for expired processes. When a process is ready to be scheduled, it goes into the appropriate priority queue and assigned a time it’s allowed to run (unless there are no other processes ready where it will run in a FIFO-like manner). After running, it is placed in the appropriate expired queue and assigned another time-slice until completion. This is very efficient because each processor just runs the processes in the highest priority level, non-empty priority queue all the way until there are no many active processes to be run and then simply switches the active and expired queues.

Now that processes and threads have been discussed and defined in terms of the Linux operating system, the next step in understanding Linux is looking at how it handles concurrency. This topic consists of many different strategies Linux utilizes in order to handle different situations that can arise. The kernel must be able to prevent or at least recover from: mutual exclusion violations, deadlock, starvation, and possibly other scenarios. “Shared resources require protection from concurrent access because if multiple threads of execution access and manipulate the data at the same time, the threads may overwrite each other's changes or access data while it is in an inconsistent state” (2). In order for Linux to prevent this, it takes the multiple strategies devised and attempts to use them all for maximum efficiency. Linux takes from its predecessor, UNIX, many things when it comes to concurrency.

One, among many things Linux takes from UNIX, is the concept of piping. Pipes are modeled after the producer-consumer relationship. These pipes are provided by using a first-in-first-out circular queue, where one process write and the other reads. In this concept of piping, there are named and unnamed pipes that dictate which processes can share a given pipe. “Only related processes can share named pipes, while either related or unrelated processes can share named pipes” (1). This relationship ensures mutual exclusion by only allowing one process to access the pipe at a time.

Another mechanism shared between Linux and UNIX is the idea of messages. These typed messages are sent and received by processes. The messages are stored in a given process’s message queue, which is basically like a mailbox. They are mainly used to provide information between processes. Messages of this type help schedule the receiving process in order to not conflict with the message sending process. These messages sent and read between processes really helps ensure no violations will occur.

Finally, the concept of shared memory is taken from UNIX in order to aid in concurrency in Linux. This concept is fairly straight forward as multiple processes read and write from the same blocks of virtual memory. According to the textbook, this is “the fastest form of interprocess communication” (1). The only problem with this is that this concept does not account for mutual exclusion, starvation, or deadlock violations. This means that the shared memory must allow the processes acting upon it to ensure no violations will occur.

Along with these strategies provided by UNIX, Linux also creates its own strategies in order to keep a smooth running system. In normal concurrency mechanisms, operating systems, including UNIX, use signals. “As the name suggests, interrupt signals provide a way to divert the processor to code outside the normal flow of control. When an interrupt signal arrives, the CPU must stop what it's currently doing and switch to a new activity” (3). Most signals work on a first come, first serve basis. Linux is similar to other operating systems in this regard, however it builds a special type of signaling on top of the normal signaling mechanism. Just like processes can be considered real-time, signals can be considered real-time as well. These real-time signals have very similar properties to real-time processes. For example, real-time processes have priority over normal processes. In signaling, real-time signals take priority and can actually be put in a priority queue. This differs from normal signals as these normal signals can only come one at a time. Another feature of real-time signals is that these signals can carry values or messages in order to bring a detailed report to the targeted process. Overall, signals are a very important feature when it comes to concurrency because it allows simple and primitive communication between the kernel and processes.

A special feature that Linux brings to the table is atomic operations. These operations at their base do the same thing as normal operations. There are operations for reading, adding, subtracting, and so forth. However, there is a huge difference with these atomic operations that make them unique. First of all, these operations can only be performed on a thread that is executing in kernel mode. This kernel mode just means a heightened privilege mode where the kernel can do special things that a process running in user mode cannot. These atomic operations completely throw race conditions out the window. Race conditions are when two or more processes are fighting for resources. It accomplishes this feat by not allowing the operation to be interrupted after it is executed. This might seem counterintuitive at first, but the benefits of these operations prove otherwise and will be discussed shortly. Finally, not only are these operations strictly performed on integers, but they can also be performed on bitmaps. This allows these operations to be very dynamic and have many uses throughout the Linux system.

So how exactly do these atomic operations avoid race conditions and what exactly are the benefits? “The easiest way to prevent race conditions due to "read-modify-write" instructions is by ensuring that such operations are atomic at the chip level. Every such operation must be executed in a single instruction without being interrupted in the middle and avoiding accesses to the same memory location by other CPUs” (3). These operations can be considered slightly different on older, single processor computers. Since there is only one processor, there is no worry of issues including executing a process on more than one processor. So these computers simply just do not allow the operation to be interrupted. The problem that arises with multiprocessor computers is the one described above: a process can be threaded throughout multiple processes. For example, if a variable is being operated upon within a certain thread and then this thread is interrupted by another thread using this same variable before the first thread is done executing, problems will arise. To account for this in multiprocessor computers, the process is not only kept from being interrupted, but it also locks the variable that is being operated upon which keeps it from being changed in threads that contain this variable among other processors.

There are multiple other benefits from using these atomic types, but first one must understand how the operations work. For integer operations, the operations do not simply add two integers of type int together. However, a special data type is created for these operations. This data type was given the name “atomic\_t” and acts as a special type of integer. At a high level, it acts as a normal integer and can be added to or subtracted from. There are many benefits that prove why creating this special type was a good idea. Firstly, all atomic operations operate on variables that are sure to be protected from race conditions. This means that when using this atomic\_t type, variables being used are static will not be changed throughout the execution of the operation. Secondly, atomic\_t types are only allowed to be used in atomic operations which adds a security layer of not corrupting other, normal operations with this type. Thirdly, in order to provide complete and rigid structure of these operations, when the operation is compiled, the compiler must address the values of atomic operations directly by their specific memory address. The final benefit of using a new data type is that when these operations are used, the atomic operation structure does not have to change when implemented in different systems. The atomic\_t type stays the same and the atomic operations know how to operate on these special types which allow no confusion when transitioning into a new environment.

Atomic operations are as simple as it gets when dealing with concurrency in Linux. Most of the other strategies for providing concurrency are complex and very dynamic. One of the most important things the kernel must do is make sure two processes are not in their critical sections at the same time. Mutual exclusions violations can render both processes useless, as they are both acting upon the wrong data. In order to ensure these violations will not occur, Linux takes use of spinlocks, semaphores, and barriers.

“The most common technique used for protecting a critical section in Linux is the spinlock” (1). At a very high level, spinlocks are merely integer variables used by processes to know if they are allowed to enter their critical section or not. As simple an idea this sounds, it is quite powerful, but has its downsides. Let’s say a process is ready to enter its critical section. Before doing so, it must go to the memory location where the spinlock is located and check the value of the location. If the value is 0, the process is allowed to enter its critical section. However, before leaving the spinlock, the process must set the value to 1 to let other processes that are checking this spinlock know that they cannot enter their critical sections. Once the process completes and finishes executing its critical section, it unlocks the spinlock by resetting the value to 0. Now, say when the process initially when to check the spinlock, the value was 1 instead of 0. This means that the process cannot enter its critical section and must wait and keep checking the spinlock’s value until it is 0 once again. What if the process currently in its critical section will be executing for quite some time? This means that the process waiting to go into its critical section will be stuck waiting for the spinlock to be reset for quite some time. The waiting process can do nothing productive in this situation except wait. This is why “spinlocks are most effective in situations where the wait time for acquiring a lock is expected to be very short” (1).

Although spinlocks are a very simple concept, there are varying versions of spinlocks that help keep the system as dynamic as possible. Most of the time, normal spinlocks will be used. These spinlocks account for situations where interrupts are disabled during the execution of a process’s critical section. However, there are times where this is not the case. In some cases, interrupts are always enabled no matter if a process is in its critical section or not. In this case, “\_irq” spinlocks will be used. On another note, sometimes it will not be known whether the interrupts will be disabled or not when entering a critical section. In this case, “\_irqsave” spinlocks will be used. This is called \_irqsave because since the spinlock is unsure about the nature of interrupts when a process enters its critical section, the spinlock saves the state of the interrupts of the system before the critical section is encountered. The final non-normal type of spinlock is known as “\_bh”. This spinlock enables and disables something called the bottom half. An interrupt handler only does so much work when an interrupt is encountered. The other work is done by the bottom half. “In an ideal world, this is nearly all the work because you want the interrupt handler to perform as little work (and in turn be as fast) as possible” (2). This spinlock ensures the bottom half does not mess up the process in its critical section.

As seen before, different concepts must be implemented slightly differently in single processor systems vs multiprocessor systems in order to account for the changes multiprocessors make. When it comes to spinlocks, this is once again true. On a multiprocessor system, it is possible for multiple processors to change the same spinlock value. For this reason, spinlocks are compiled into the actual code and are used to check the values to ensure no mutual exclusion violations are made. This is different in single processor systems, as only one processor has access to the spinlocks. These systems must account for both kernel preemption and non-preemption, meaning a process in kernel mode can be interrupted or not. If kernel preemption is on, the spinlocks compile into the code. Not to check the values, but to simply lock and unlock the spinlock. When non-preemption is on, this is not needed as the spinlocks are not needed because no interrupts can occur. Preemption also puts extra stress on the system as the entire context of the process changes. In order for the system to handle this, it uses a “schedule()” function. “The kernel, however, must know when to call schedule(). If it called schedule() only when code explicitly did so, user-space programs could run indefinitely. Instead, the kernel provides the need\_resched flag to signify whether a reschedule should be performed” (2). The scheduler will check this flag in order to know when and when not to use the schedule method.

There is one final version of spinlocks that builds on-top of the idea of spinlocks in order to ensure a safer environment where it is close to impossible for any violations to occur. This version also allows more concurrency to be going on at once instead of only one process. Spinlocks that do this are known as reader-writer spinlocks. This takes off the idea that since reading does not change any data, more than one reader should be allowed to process at once. However, since writers do change the data they are operating on, only one writer can be active at once. This is implemented as a spinlock by adding a counter along with a flag on top of the normal spinlock. The counter counts the readers and the flag is set when a thread has acquired use of the spinlock. There is one big problem with this spinlock however. The idea of starvation is not accounted for. Readers are given precedence and will continue to flow in and keep the spinlock locked for as long as there is a reader. This means that if there is a writer waiting, it could be waiting for a very long time. All the versions of spinlocks work to a certain degree and are a very good introduction to the complex side of concurrency.

Semaphores, meaning signal carrier, are another approach Linux uses to take the concurrency problem head on. Semaphores are similar to spinlocks in that their main goal is to only allow one process to be in its critical section at one time. This concept also does its best to ensure that other violations are safe as well. In Linux, this is done through a set of functions. The purpose of these functions is to tell other processes when and when not they can enter their critical sections. Only the kernel can call these functions and cannot be called by users through series of calls to the system. Like spinlocks, semaphores can be implemented in different ways.

The first way a semaphore can be implemented is through either binary or counting semaphores. Binary and counting semaphores are similar except for the way they are initialized. Initially, they are both initialized by a call to their respective function. Binary semaphores are initialized by calls to “init\_MUTEX” and “init\_MUTEX\_LOCKED”, which represent either initializing the semaphore to 0 or 1. They are known as “mutexes”, short for mutual exclusion. Counting semaphores, on the other hand, are initialized through calls to “sema\_init” and can be initialized and changed to any value, per the counting name. These semaphores are passed as parameters with functions like “up” and “down” which release the semaphore passed to the function or attempt to use the semaphore, respectively.

Traditionally, the down function first checks to see if the given semaphore is available. If the semaphore is available, it will lock it and do whatever it needs to. However, if the semaphore is not available, the calling thread will go to “sleep”. The thread will not be awaken until the thread using this semaphore calls the up function in order to notify the sleeping thread that the semaphore is now available. Two other versions of this down function exist. The first of the two versions is called “down\_interruptible”. Normally, when a thread is asleep and waiting on a given semaphore, it cannot be interrupted by anything except the semaphore’s up function. In this particular version, when a thread is waiting on a semaphore, it is possible for the kernel to override this sleep and notify the thread. This is useful if a thread is waiting a significant amount of time for a semaphore and is needed elsewhere. The second version of the down function is called “down\_tryblock”. This version is useful as it doesn’t put threads to sleep if it is unavailable. If a thread tries to access a semaphore that is unavailable, it merely just returns a value and doesn’t block the thread.

The reader-writer approach is once again taken here with semaphores. Like before, it divides threads into readers and writers and allows multiple readers to read at the same time but only one writer to write at a given time when no readers are reading. This type of semaphore uses different function calls in order to implement this idea. There is both a down and up function for both the readers and the writers. When implemented, a counting semaphore is used for writers as there can be multiple at one time. However, since there can only be one writer at any given time, a binary semaphore is used to keep track of writers. This is an efficient approach to the reading-writing problem, however it can still cause starvation of writers.

There is one final mechanism that Linux uses in order to ensure no deadlocks or starvations will occur, barriers. “When using optimizing compilers, you should never take for granted that instructions will be performed in the exact order in which they appear in the source code. For example, a compiler might reorder the assembly language instructions in such a way to optimize how registers are used” (3). These barriers make sure that the order a program is executed in a way that no problems will occur with the system. However, it is possible that if two accesses were to get swapped, the whole program would be off. Barriers set the guidelines on what can get swapped and what cannot. To do this, barriers use functions that create barriers. For example, the rmb() function keeps load operations from being reordered past where the function was called. Similarly for store operations, there is a wmb() function that keeps store operations from being reordered past where the function was called. To be sure of no mistakes, there is also a mb() function which keeps both loads and stores from being reordered past the barrier.

One final major part that any operating system must take care of is handling I/O. Generally speaking, input and output is very slow when compared to processes handled within the CPU. “One of the slowest operations in a modern computer is disk seeks. Each seekpositioning the hard disk's head at the location of a specific blocktakes many milliseconds. Minimizing seeks is absolutely crucial to the system's performance” (2). Overall, the Linux I/O facility is very similar to that of the UNIX I/O facility. Both associate device drivers with files kept in memory. This seems like a rather simple approach, but Linux adds on special ways of handling this drivers in order to have a most efficient operating system.

Just as seen before when scheduling processes, inputs and outputs must also be scheduled in a smart manner. This is called disk scheduling as the processes read and write to and from the disk. Linux takes on this type of scheduling several different ways. The first way uses one single queue for all I/O processes and is therefore called the elevator scheduler. To make even more sense of the name, the algorithm acts as a sort of elevator. First, the disk starts at the lowest block memory address in the queue. The disk then continues in one direction, satisfying each request in the queue. As new I/O requests come in, the queue is then sorted to ensure the disk does not skip past a particular block address. It is possible, however, for a new request to come into the queue after its memory address has already been passed. In this situation, the request will be added to the tail end of the queue to be processed the next time around.

The elevator scheduler technique is quite simple and there are possible downsides which is why the deadline scheduler is often used. This scheduler attempts to build on top of the elevator scheduler in order to eliminate possible problems. Firstly, it is very possible for a request to be starved for a very long time in the elevator scheme. If a request is requesting access to a very high memory block, since the queue of processes is constantly sorted, this high request could take a while before it is satisfied. Another possible problems arises when read requests are issued. These reads must wait until any write requests are done that are acting upon the same data. This waiting could cause the scheduler to slow down significantly. The deadline scheduler attempts to fix these problems by adding two more queues on top of the elevator queue. One of these queues is strictly used for read requests and the other used for write requests. These two queues both act in a first in first out fashion. When a request comes in, it is added both to the elevator and its respective queue. The queues help keep track of the time the requests come in and also tag an expiration date on the requests. Normally, the scheduler will take requests from the elevator queue, but in order to keep a process from being starved, if the process at the head of one of the other queues is older than its expiration time, it is granted access. When a request is satisfied, it is removed from both queues.

“Simply sending out requests to the block devices in the order that the kernel issues them, as soon as it issues them, results in awful performance” (2). For one final time, the previous two scheduling algorithms are once again improved to make this disk scheduling as efficient as possible. The final algorithm is named anticipatory scheduling, for an obvious reason: it anticipates requests. Elevator and deadline scheduling both attempt to satisfy requests as quickly as possible by satisfying new requests as soon as old ones are finished. This can be counterproductive at times because of the principle of locality. If one request attempts to read from a memory address, it is very likely that the process will send another read request from the same address. In the previous algorithms, the scheduler would simply continue on and go past this memory address. In anticipatory scheduling, the scheduler will wait for a predefined time before moving on to the next request.

Using UNIX as a base, Linux added to and modified its predecessor and produced a beautiful operating system. 20 years later, these modifications are still coming, making the system better and more secure very often. These modifications can be credited to many developers, as Linux continues to be open-source, unlike some of its competitors. Linux really benefits by being open-source, as more people can make suggestions and ensure the system is as efficient as possible. Linux distributions come in many shapes and size, but are all built on top of the same Linux kernel. This, along with the way the kernel operates, makes the Linux operating system the most secure, customizable, and fun to use operating system available.

Works Cited

1. Stallings, William. *Operating Systems: Internals and Design Principles*. Print.
2. Love, Robert. *Linux Kernel Development Second Edition*. Novell, 2005. Web. 10 Mar. 2016. <http://www.staroceans.org/kernel-and- driver/Linux\_Kernel\_Development\_2nd\_Ed-Robert\_Love-Sams-2005.pdf>.
3. Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel*. Beijing: O'Reilly, 2006. Web. 01 Apr. 2016. <http://gauss.ececs.uc.edu/Courses/c4022/code/memory/understanding.pdf>.