Understanding Design Issues in Multiprocessor, Multicore, and Real-time Scheduling on Linux Machines

Hannah Cebulla k91c782
Bridget Wermers m89p335
Logan Shy s23j382
John Hultman z55b755

Introduction

The core concept of our research project consists of the various ways the operating system schedules processes for execution. This includes multicore (or multithreading), multiprocessor, and real time process scheduling. Some time is taken to introduce the reader to the basics of each scheduling concept, followed by a brief overview of the design analysis and algorithmic analysis when relevant. Further in the paper a handful of scheduling algorithms are compared and contrasted between each other, while also being observed at a more concrete level. We also take a close look at the Linux kernel and it's scheduling functionality, from the early days of 1.2 to the current optimized implementation, as well as the different ways the kernel provides process synchronization.

Through reading Chapter 10 in the Operating Systems textbook, and further research into the main topics that we found we were able to present a report that can provide readers with necessary information to increase their knowledge of scheduling algorithms. We specifically aim our focus towards algorithms that are used for multiprocessor, multicore, and real time designs including EDF, RM, Declustering and Priority Inversion. Each of the algorithms has advantages and disadvantages and the choices come down to what is most important to the system, from speed to priority or if there is preemption or not.

Background & Related Work

For a proper understanding of the content of this paper, the reader should have prior background knowledge on a computer's processor, and the ways that it performs execution of a user's given processes. For a comprehensive intro on this topic, the reader is encouraged to read one of the referenced articles, (Chovatiya 2). Understanding the complications that arise from scheduling requires knowledge about the desired outcomes of a task, the importance of hitting deadlines and how to recover from them. Throughout the report terms are defined when new topics are introduced, and we also provide references to articles where readers can go to brush up on things they may not be familiar with.

I. Multiprocessor Scheduling

Multiprocessors are extremely useful when performing CPU intensive tasks such as scientific computation, compiling large programs, and developing video games. However, a multiprocessor is only as efficient as its scheduling design. In this section, we explore the concept of multiprocessor scheduling and the varying optimization algorithms that attempt to approximate the best solution. A 2018 study done at the Rajasthan Technical University defines performance goals as “reduction of execution time and scheduling length, minimization of communication delay, and [maximization] of resource utilization" (Suman, Kumar, 2018). We will adopt these as our performance goals for the different scheduling algorithms explored.

A multiprocessor system is a system with one or more processors. Essentially, multiple CPUs are linked together in order to increase the execution speed of a system. There are three main ways this can be accomplished. First, a loosely coupled or distributed multiprocessor is composed of a collection of relatively autonomous systems. In this case, each processor has its own main memory and I/O channels. Second, a functionally specialized processor is essentially an I/O processor. This is composed of a general purpose processor that controls other specialized processors. Third, a tightly coupled multiprocessor is a set of processors that all share one common main memory and are all under the control of a single operating system. Each multiprocessor design requires different scheduling designs and presents a variety of trade-offs depending on the desired computational tasks. A tightly coupled multiprocessor has been shown to demonstrate the most issues. Therefore, this section will focus mainly on the issues regarding this system design.

One successful metric to compare multiprocessors to other architectures is granularity. This is a measure of the grain size of a specific task or the amount of computational work required to perform that task. Granularity is typically measured by the following formula.

$$G = \frac{T_{comp}}{T_{comm}}$$

Many different multiprocessor scheduling designs can increase or decrease this value. We will use this value throughout this section to provide a comparison of different scheduling designs.

Our comparison of different scheduling designs will be largely supported by the current literature in this field. As we approach an age of big data, many of the current experiments are concerned with computing power. Specifically, scientists experience the consequences of high granularity and the operating systems that are not necessarily equipped to handle these kinds of tasks. In the investigations described in our algorithm analysis section, we will explore these current issues a little deeper, as well as present some options for scheduling algorithms that support these endeavors.

II. Multicore Scheduling

Most current operating systems treat multicore systems as a multiprocessor system for the purposes of scheduling. Multicore systems are different from multiprocessor systems due to the fact there is only one chip and there are multiple cores on that chip to facilitate more efficient processing. The major issue related to multicore systems is that with an increase in cores per chip the need to reduce off-chip memory access becomes more important than maximizing processor utilization.

This is where multicore systems have specific issues with scheduling, because most current architectures use multiple levels of caches and not every core shares the same caches. Ideally for two threads to share memory resources the cores they are assigned to should be adjacent to improve locality but if they do not share memory resources they can be assigned to nonadjacent cores to balance the load on the cpu.

There are two main considerations with cache sharing that cause issues in scheduling specific to multicore systems, cooperative resource sharing and resource contention. Cooperative resource sharing is when multiple threads access the same main memory locations, in which it is ideal to assign threads to adjacent cores. This will allow the threads to access a shared cache of memory. In resource contention if the threads are on adjacent cores and are competing for memory locations there will be performance degradation. This is because if one thread has more of the cache dynamically allocated to that thread competing threads will have less cache space to utilize. They therefore should be scheduled in a way that will maximize the effectiveness of cache memory to minimize off chip memory access.

III. Real-Time Scheduling

Real-time computing is when the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced. There are hard and soft real time tasks where hard means that the deadline must be met otherwise there will be system damages or failure and soft deadlines are desired but not vital. Tasks can also be classified as aperiodic that has a deadline by which it must finish and/or start, or in the case of a periodic task it is scheduled "once per period T" or "exactly T units apart." Real time operating systems have unique requirements in 5 main areas; Determinism, Responsiveness, User control, Reliability and Fail-soft operations.

With multiple processes competing for time and resources, no system will be fully deterministic, only to the extent that it performs operations at fixed times or at fixed time intervals. To measure the ability of an OS to function deterministically calculate the maximum delay from the arrival of the interrupt to when servicing of it begins. In real time OS' process requests are triggered by external events and timings. Responsiveness considers how long after an interrupt is acknowledged it takes an OS to service it. Aspects of responsiveness include the time to handle the interrupt and begin executing the ISR, the time to perform the ISR and any effects of interrupt nesting. If a process switch is required then it will take longer to begin the ISR and if an ISR can be interrupted the service will also be delayed.Together determinism and responsiveness make up response time to external events which is critical for systems to meet timing requirements.

In a real time system it is essential to give the user control over task priority such as distinguishing between hard and soft tasks, give relative priorities and allow characteristic specifications for mapping and more. It is also generally more important that a system is reliable since a real time system is responding to and controlling events in real time there is not time to reconfigure a process failure or reboot like there would be with non-real world systems.

Fail-soft operation refers to the ability of a system to fail and preserve as much capability and data as possible. Real-time systems will attempt to solve the issue or minimize the effects of the issue and continue to run. While the system may slow down, it still works unless a full shutdown becomes necessary in which case the system will work to maintain files and data. Stability is an important aspect of fail soft operations and a real time system is stable if it meets the deadlines of the most high priority tasks even when it is not possible to hit all task deadlines.

In general, the following features are common to most real-time operating systems when compared to ordinary operating systems.

- A stricter use of priorities with preemptive scheduling to meet real-time requirements
- Interrupt latency is bounded and relatively short
- More precise and predictable timing characteristics

Real Time Scheduling Algorithms

Real time scheduling algorithms can be broken down into 3 main classes based on a static or dynamic schedulability analysis, and whether the results create a schedule or plan which tasks to dispatch at runtime.

Static table-driven approaches perform static analysis of feasible schedules to dispatch where the result is a schedule determining at run-time when a task must begin. This algorithm applies to periodic tasks where input to the analysis includes arrival time, execution time, deadline and priority for the task. An example of a static table-driven approach is earliest deadline first (discussed further below) along with other periodic deadline techniques.

Static priority-driven preemptive approaches are also a static analysis but no schedule is generated and tasks are assigned a priority and a priority-driven preemptive scheduler is used. In a real-time system, priority assignment relates to the time constraints each task holds. Rate monotonic algorithm (discussed further below), which assigns static priorities to tasks based on the length of their periods is one example of this approach in use.

Dynamic planning-based approaches determine feasibility at run time, making scheduling dynamic and resulting in a schedule used to dispatch tasks. This approach addresses arriving tasks by scheduling them in a way that deadlines are met and the current schedule is not disrupted, such as forcing other tasks to miss deadlines. In dynamic best effort approaches, no analysis is done, and the system attempts to meet all deadlines. This results in tasks whose deadlines have been missed being aborted. As a task arrives it is given a priority based on it’s characteristics which prevents the system from knowing if a timing constraint will be met until either the task finishes or the deadline is reached and the task is not complete.

Approaches to real-time task scheduling are based on gathering additional information about tasks and aim to start and complete tasks at the correct time rather than focusing on speed alone. Ready time, starting deadline, processing time, resource requirements, priority and subtask structure are all pieces of information that may be needed to schedule real time functions with deadlines. The system has to consider which task to schedule next and what preemption is allowed. Preemption is a good strategy for a system with completion deadlines, while non preemptive schedulers make the most sense when starting deadlines need to be met.

IV. Linux Kernel Scheduling Implementation

Linux has long attempted to provide a process scheduling solution that is efficient under a wide array of computing scenarios, as well as one that is robust enough to endure the constant changes the Linux kernel undergoes. Generally, the scheduler is responsible for creating and managing all of the different processes running on a system. Things like creating process threads and prioritizing processes all fall under the responsibility of the scheduler. Linux provides all of the process management functionality found in Unix systems i.e. fork, exec and wait, as well as adding a few of its own API system calls such as clone. As the kernel source code for sched/core.c is well over 6000 lines of code long, it's easy to see that a lot of system functionality is provided within this one file. Thus, the scope will be narrowed to focus mostly on the concepts of scheduling and synchronization, and a closer observation of the kernel's source code (Linux 9).

Historically, the Linux scheduler has undergone some dramatic functional changes. The original 1.2 scheduler was a simple circular linked list, that acted on a round-robin management basis. This was simple enough and efficient, however it did not take into account complex system architectures or any kind of multicore processing. As the scope and size of modern architectures became apparent, it was obvious a change was necessary.

This change came in the Linux 2.4 update, with the addition of scheduling priorities and multiple data structures. An O(n) scheduler was implemented that utilized over a hundred different run queues, with each number representing a different priority. After running the program of highest priority, it was placed from the ready queue to a matching expired queue. After the ready queue is empty, the expired queue becomes the ready queue and vice versa. The solution was simple, however it was inefficient and relatively unscalable. Because the scheduler is forced to iterate over every process to find the highest priority, the O(n) time complexity hurts the performance as the number of tasks increases. This also presents the possibility of a process starving if a higher priority process keeps being added to the set of queues.

With Linux 2.6, an O(1) scheduler was implemented to solve many of the problems inherent to 2.4. The scheduling data structure gets an overhaul, and now each CPU is given a queue with an array for ready and expired processes. Each array has 140 pointers to doubly linked lists representing a priority level, with different processes inside of these lists. When a process is run it is placed into the matching expired queue, and the ready queue and expired queue swap once all programs have run. To further optimize the O(1) scheduler, Ingo Molnar replaced it with the *Completely Fair Scheduler*, or CFS. It mimics operation of the O(1), except it utilizes red-black trees for storing processes and their priorities, which leads to all processes having a fair go while maintaining the O(1) time complexity (Jones 6).

The Completely Fair Scheduler differs from many of its predecessors. Unlike Unix or earlier Linux implementations, the CFS does away with the static time quantum a thread can spend on the processor. Instead of each process being given the same time slice, a *target latency* is defined, with each process needing to run at least once inside of this timespan. So with each task or thread vying for a slice of the allotted time, the actual amount of execution time a process can have becomes less when there are more processes. The time slice is measured by 1/N, with N being the amount of processes currently trying to run. For example if the target latency is 50 ms and there are 10 processes, each process will get 5 ms of execution time.

At the same time, the CFS changes how a thread's priority is structured. Instead of an arbitrary priority, how long a process has had to run (known as *vruntime*) on the processor acts as it's priority, with lower vruntimes acting as higher priority. When a thread is preempted, it is placed on the right side of the red-black scheduling tree and allowed to self-balance. When a new process is chosen for execution, the left-most node is selected. This will ensure the process chosen next has had the least time on the processor, thus giving the scheduler it's 'Completely Fair' namesake (Kalin 7).

How Linux synchronizes different processes and allows mutual exclusion is similar to an onion or an ogre, in that it has layers. Each of these layers is used as an abstraction to provide different functionality depending on the current situation at hand, with most of the core functionality sometimes being several wrappers deep. The main tools that the kernel uses is composed of a set of low level atomic operations, a spinlock, and a mutex, with parts of the latter two being built from the first..

The most basic operation that Linux has at its disposal is the atomic_t type and it's set of related functions. This is a struct that holds one integer value, often used for counting, that can be atomically incremented and decremented. As C cannot guarantee atomic behaviour, the underlying architecture is relied on to provide the implementation. Most of the time this is written directly in assembly code, but can vary from system to system (Torvalds 3).

A spinlock is a way to ensure mutual exclusive actions on critical regions of code. Once a thread acquires the lock, any thread that wants to act on the critical section gets taken and put in a special wait queue, where it busy-waits until the lock is available. Having the thread be forced to busy-wait does not allow for another thread to have the chance on the processor, and usually is more efficient than trying to preempt the process. A special variation is a Read/Write spinlock, where multiple threads can have the lock for reading purposes up to a limit, while the lock must be empty for a thread to perform a write.

A mutex is built off a spinlock as well as an atomic_t type, and functions very similarly to the spinlock in regards to their mutually exclusive natures. In essence, Linux provides the kernel with semaphore functionality. The mutex lock, after having failed to acquire the lock, will not be busy-waiting trying to reacquire the lock. Instead it will preempt, and allow another task to have a turn on the processor. This can be a costly operation, and knowing when to use a mutex over a can greatly affect performance (Jones 5).

V. Algorithm Comparison

A. Stochastic Search Scheduling Method based on the Genetic Algorithm

Multiprocessor process scheduling can be quite a tough problem to handle. So tough in fact that it is considered to be NP-hard, an intractable problem. To get a close enough approximation, optimizations and heuristics are used to deal with the variables that the processor faces, such as the number of processes running, how uniform the processing time is for each task, and the architecture of the multicore system (Hou 4).

One way to solve this problem is based on Charles Darwin's theory of evolution, with a stochastic method of genetic variation among offspring used to provide a random yet structured search optimization. We start by defining a variable, often represented by binary, that we call a gene. A string of these genes make up a chromosome, and the set of all chromosomes is our population. We consider this to be the set of all solutions. We then define a fitness function, or some attribute that we want to select for (or search for). The closer the chromosome is to our defined fitness function, the higher it's fitness score, and the more likely it will be selected to reproduce. Two fit chromosomes will be selected as parents and their genes will be shared at a random point within the string, known as the crossover point. Randomly, a small genetic mutation will change a number of an individual's genes, flipping a bit from 1 to 0 or 0 to 1. This is to ensure we don't have a premature convergence, which will give us a poor approximation. After an unknown amount of generations, eventually all individuals within a population will converge, giving us our solution (Mallawaarachchi 10).

The actual implementation of a genetic algorithm in regards to multiprocessor scheduling can vary greatly, with different steps being taken depending on system structure and whether or not preemption on tasks is available. Usually the implementation will involve a relations graph between tasks, which is an acyclic graph that represents constraints between tasks and usually some other data, such as a tasks processor time. It becomes a problem of optimizing an objective function to promote efficient execution of the given applications (Hou 4).

B. Dynamic Critical-Path Scheduling Algorithm

This paper proposes a static scheduling algorithm that allocates tasks to fully connected multiprocessors. The proposed algorithm is demonstrated to outperform six other similar algorithms. The algorithm makes improvements from the tested six in the following ways. First, the critical path of the task graph is determined and the next node is selected to be scheduled dynamically. Second, the schedule is rearranged on each processor dynamically. This ensures that schedules are not fixed until all nodes have been considered. Third, the algorithm selects a processor that is suitable for a node using a lookahead method. These scientists conclude that "despite having a number of new features, the DCP algorithm has admissible time complexity, is economical in terms of the number of processors used and is suitable for a wide range of graph structures" (Kwok 1).

C. Earliest Deadline First (EDF)

With the earliest deadline first approach a task is assigned priority based on the absolute deadline, giving the most priority to the task whose deadline is the closest and allows that task to run until completion. The highest priority task may not be ready, and this may result in the processor waiting, disregarding other ready tasks. In the end all scheduling requirements are met, without having maximum efficiency of the processor. Earliest deadline first scheduling can be used in static or dynamic real time scheduling. The schedulability test for EDF is

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Where the {Ci} are the worst-case computation-times of the n processes and the {Ti} are their respective inter-arrival periods (assumed to be equal to the relative deadlines).

Advantages of EDF over other algorithms including rate monotonic include the ability to maximize processor utilization, it is preemptive, priorities do not have to be defined offline and there is less context switching than there is with RM. (Earliest 8) EDF is optimal because it is dynamic, priority driven, preemptive and works on a uniprocessor or multiprocessor making it a qualified candidate to schedule real time processes. The two main disadvantages of EDF are high task migration cost due to changes in the system when new tasks arrive with earlier deadlines or when a task is completed and that there is unpredictable behavior in overloaded condition. (Lindh 8) Additionally, earliest deadline first algorithms are less predictable and have less control over execution when compared to rate monotonic algorithms. (Earliest 8)

The Earliest Deadline First (EDF) algorithm is analyzed further in a paper that explores "the best-known real-time scheduling techniques executing on multiple processors" (Kwok 1).

D. Rate Monotonic (RM)

This scheduling algorithm will schedule the highest priority task with the shortest period, then the next highest priority with the next shortest period, and so on. When more than one task is available the one with the shortest period is serviced first. The task's period is the amount of time between the arrival of one of the task's instances and the next instance of that task. Though utilization will be lower than Earliest Deadline First, the performance difference is small, but stability is easier to attain. With partitioned and global algorithms an RM scheduler tends to improve performance with higher maximum possible utilization of periodic tasks, however EDF is more efficient for lower maximum possible utilization systems. (Zapata, 13)

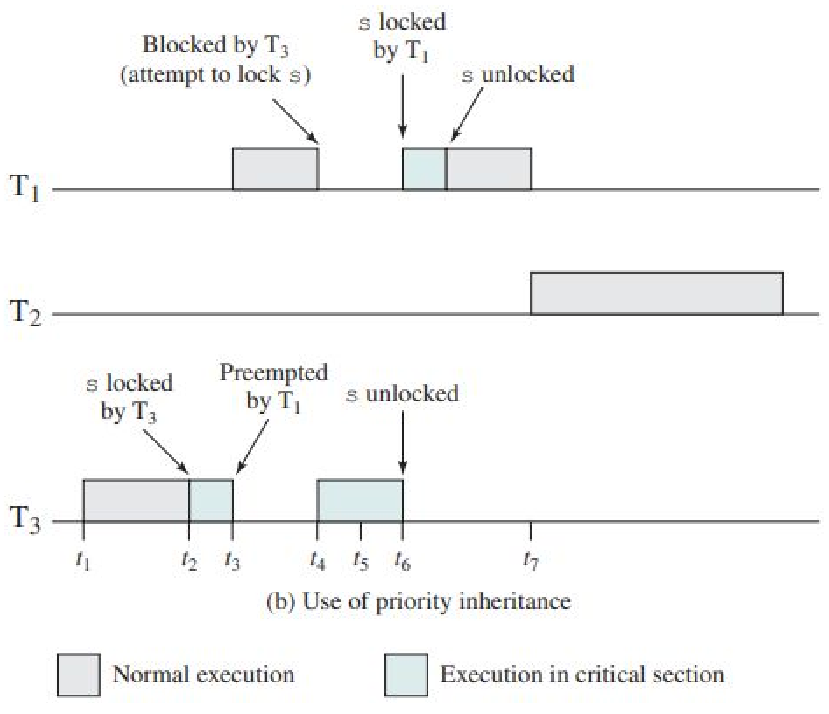
E. Declustering

Declustering is a scheduling algorithm that maps precedence graphs onto multiprocessor architecture. This algorithm has four main stages, the first divides the graph into elementary clusters. The second combines the clusters in a hierarchy using intercluster interprocessor communication and parallelism relationships. The third stage decomposes the hierarchy systematically by breaking the higher level clusters into subclusters and assigning the subclusters onto different processors. The fourth stage analyzes the best schedule that has been obtained and breaks down the clusters if the load on the processor needs to be balanced more efficiently. Declustering does outperform traditional clustering algorithms in the first stage of this algorithm. It also gains additional performance due to its methodical approach to attacking the schedule limiting progression as opposed to the critical path. (Sih, 11)

F. Priority Inversion

In any priority driven scheduling algorithm a system always should be executing the highest priority task, but priority inversion can create situations where high priority tasks must wait on low priority tasks to complete execution. For instance if a low priority task is executing and locks a resource that a higher-priority task attempts to access or lock the high priority task enters a blocked state and waits for the resource to become available. This can cause real time constraint violations unless the low priority task finishes quickly. Further there is unbounded priority inversion where the duration of the priority inversion also depends on actions of other unrelated tasks which is unpredictable. One real world example of priority inversion was with the Mars Pathfinder mission in which the software began having total system resets and losing data upon each reset. The problem was traced back to priority inversion where a timer was initialized and if the timer expired the integrity of the system was compromised therefore stopping all processes, resetting devices, reloading software, testing the systems and then the system finally starts over which takes until the following day.

In order to avoid unbounded priority inversion developers use priority inheritance protocol and priority ceiling protocol approaches. With priority inheritance, a low priority task inherits the priority of any higher priority task also waiting for the same resource. The change takes place as soon as the higher-priority task blocks and ends when the resource is released by the initial lower-priority task. In the priority ceiling approach each resource is assigned a priority which is always one priority level higher than that of the highest priority task that will use the resource. Following the scheduler will give the same priority to any task trying to access the resource and when the task finishes its priority will return to the original. (Stalling 12)



(b) Use of priority inheritance

G. Contention Aware Scheduling

Zhuravlev's paper goes into great depth on contention-aware scheduling on multicore systems, he identifies several assumptions used by contention-aware schedulers. Assumption one is the scheduler will space-share as opposed to time share the machine, space sharing is deciding how to assign threads to neighboring cores and distant cores and time sharing is multiplexing time on a single cpu among multiple threads. Assumption two is that the architecture of the system must contain multiple cores, where subsets of cores share different resources. Assumption three is that all resource sharing is destructive interference, this will weed out threads that resource share constructively. All contention-aware-schedulers will consist of four major components or building blocks, the objective, the prediction, the decision, and the enforcement. The objective is the metric the scheduler is optimizing. The prediction is how the scheduler will create thread-to-core maps to facilitate the objective and the search space will be too vast for trial and error. The decision is how the scheduler will select the thread-to-core map that will be used. The enforcement is what will bind the threads to the cores specified by the placement. (Zhuravlev, 14)

Zhuravlev then goes on to analyze whether contention-aware-scheduling is effective and practical. In practice they are good at mitigating shared resource contention, improving performance and predictability. They however do not solve the issue of shared resource contention but actively avoid it because of this there are limits to what these algorithms can do. Zhuravlev then states it is unrealistic to think that contention-aware-schedulers will solve the issue of resource contention in multiprocessors on their own but any solution will most likely include contention-aware-scheduling. (Zhuravlev, 14)

VI. Conclusion

Throughout our studies we were able to explore algorithms that act as solutions for the complex problems inherent in multiprocessor, multicore, and real-time scheduling. We compared and contrasted how each algorithm would work for a particular situation and what things need to be considered in order to make proper implementation decisions. Within our paper we discussed algorithms that would approximate the most optimized solution, adaptations that could prevent problems inherent in a given design, and showed that certain algorithmic solutions for modern scheduling share some overlap in their approaches. We also go on to look at a brief history of how Linux has tackled these hurdles, as well as a close look into the current implementation and ways Linux provides a robust and scalable scheduling method.

We learned about scheduling complications such as missing deadlines, priority inversion and fail-soft operations as well as attempts to solve these problems. For example, how priority inversion can be resolved through priority inheritance. Comparisons were made between a basic operating system and real-time systems, which differ in what characteristics they require to function correctly. As some tasks are more heavily regarded as critical during run-time, the system takes these deadlines into account during it's scheduling. We also learned about nuances specific to multicore systems such as shared resource contention and cooperative resource sharing. We then discussed a potential solution in the form of a contention aware scheduler and how it helps solve the issue of resource contention but will likely be used in a solution to this issue.

VII. References

(1) Kwok, Y., & Ahmad, I. (1999). Static scheduling algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys, 31(4), 406-471. doi:10.1145/344588.344618

(2) Chovatiya, V. (2020, May 09). How Program Gets Run: Linux. Retrieved November 11, 2020, from http://www.vishalchovatiya.com/program-gets-run-linux/

(3) "EARLIEST DEADLINE FIRST (EDF) SCHEDULING ALGORITHM." 29 June 2018. Web. 14 Nov. 2020 from https://microcontrollerslab.com/earliest-deadline-first-scheduling/

(4) Hou, E. S., Ansari, N., & Ren, H. (1994). A Genetic Algorithm for Multiprocessor Scheduling. *TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, 5*(2), 113-120.

(5) Jones, M. T. (2007, October 31). Anatomy of Linux synchronization methods. Retrieved November 14, 2020, from https://www.ibm.com/developerworks/library/l-linux-synchronization/index.html

(6) Jones, M. (2009, December 15). How the Linux kernel runs a program. Retrieved November 10, 2020, from https://0xax.gitbooks.io/linux-insides/content/SysCall/linux-syscall-4.html

(7) Kalin, M. (2019, February 05). CFS: Completely fair process scheduling in Linux. Retrieved November 11, 2020, from https://opensource.com/article/19/2/fair-scheduling-linux

(8) Lindh, Fredrik, Jessica Wennerstrom, and Thomas Otnes. "Scheduling Algorithms for Real-Time Systems." Web. 10 Nov. 2020 from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.623.107

(9) Linus Torvalds (2015). Linux (v5.10-rc3) [Operating system]. Retrieved from https://github.com/torvalds/linux/releases/tag/v5.10-rc3

(10) Mallawaarachchi, V. (2017, July 07). Introduction to Genetic Algorithms - Including Example Code. Retrieved November 14, 2020, from https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

(11) Sih, G., & Lee, E. (1993). Declustering: A new multiprocessor scheduling technique. *IEEE Transactions on Parallel and Distributed Systems, 4*(6), 625-637. doi:10.1109/71.242160

(12) Stallings, William. "Chapter 10." *Operating Systems: Internals and Design Principles 9e*. Web. 1 Nov. 2020 from https://sgp1.digitaloceanspaces.com/proletarian-library/books/5b0c7356751347504e2f6ce19e42d218.pdf

(13) Zapata, O. U. P., & Alvarez, P. M. (2005). EDF and RM multiprocessor scheduling algorithms: Survey and performance evaluation. *Seccion de Computacion Av. IPN*, *2508*.

(14) Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., & Prieto, M. (2012). Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys, 45*(1), 1-28. doi:10.1145/2379776.2379780