

|  |  |
| --- | --- |
| **Intake: APD1F2409CS(CYB)** | **Module Code: CT124-3-1-ICS** |

|  |  |  |
| --- | --- | --- |
| **HAND OUT DATE: Thursday, 24 April 2025 Week 6** |  |  |
| **SUBMISSION DATE: Monday, 30 June 2025 Week 14** |  |  |

**TITLE:**

***Section B: Research & Analysis***

**Module Lecturer: Ms. Vijayalaxmi Amuthan**

**GROUP NAME 9**

|  |  |
| --- | --- |
| **AHMED MIRAHUSAIN ALVI** | **TP084807** |
| **ABDALLAH MOHAMED MAHMOUD MOHAMED** | **TP085097** |
| **SULTAN ABDULLA OMAR TAKRORI** | **TP085327** |
| **ABDUL AZIZ VAYANI** | **TP082376** |
| **MOHAMMED YOUSEF MOHAMMED MOHAMMED** | **TP085042** |
| **MAHRUS SHAMSUL AHSAN** | **TP085562** |

TABLE OF CONTENTS

[CHAPTER 1: INTRODUCTION 1](#_Toc198282387)

[CHAPTER 2: BRANCH PREDICTION & CACHING 2](#_Toc198282388)

[***2.1*** *Branch Prediction* 2](#_Toc198282389)

[***2.2*** *Caching* 4](#_Toc198282390)

[***2.3*** *Relationship between Branch Prediction and Caching* 6](#_Toc198282391)

[CHAPTER 3: COMPARATIVE STUDY OF PROGRAMMING APPROACHES 7](#_Toc198282392)

[***3.1*** *Performance* 7](#_Toc198282393)

[***3.2*** *Ease of Use* 8](#_Toc198282394)

[***3.3*** *Security and Stability* 9](#_Toc198282395)

[***3.4*** *Application in OS Development* 10](#_Toc198282396)

[CHAPTER 4: FUTURE ENHANCEMENTS & PROPOSALS 12](#_Toc198282397)

[***4.1*** *Future Enhancements* 12](#_Toc198282398)

[***4.2*** *Benefits of Future Enhancement* 13](#_Toc198282399)

[CONCLUSION 15](#_Toc198282400)

[REFERENCES 16](#_Toc198282401)

[WORKLOAD MATRIX 17](#_Toc198282402)

CHAPTER 1: INTRODUCTION

When you launch an app or browse the web, your device responds almost instantly. This flawless performance isn’t just a result of powerful hardware -it’s the outcome of intelligent system design, where both software and hardware work in balance. Techniques like branch prediction and caching allow processors to execute tasks faster by anticipating actions and reducing memory delays.

To fully optimize these hardware features, developers must choose the suitable programming technique. High-level languages simplify development, improve security, and enhance maintainability, while low-level assembly offers precise control for fine-tuned performance, especially in areas like caching and prediction. These trade-offs become critical when designing efficient, secure, and responsive operating systems.

This assignment explores the inner workings of branch prediction and caching, how they interact, and the impact of programming choices on system performance. It also presents a comparative study of programming approaches, highlighting their strengths, limitations, and roles in OS development. Finally, it looks ahead at future enhancements that could improve how operating systems respond to growing demands and security challenges. By understanding the balance between high-level abstraction and low-level control, we gain insight into how modern systems like Windows and Linux achieve their performance and how we can build even better systems moving forward.

CHAPTER 2: BRANCH PREDICTION & CACHING

## ***2.1*** *Branch Prediction*

**Branch Prediction and Caching in Modern Computer Architecture**

Sophisticated methods are the main means by which modern computer processors boost performance, speed, and efficiency. Branch prediction and caching are two of the most important methods used at the processor level. In pipelined processors, these mechanisms are essential for increasing instruction throughput and decreasing latency. With a focus on how these mechanisms improve CPU performance generally, this task will examine how branch prediction and caching work and impact contemporary computer systems.

**Branch Prediction: Improving the Effectiveness of Flow Control**

Branch prediction is a crucial optimisation method in pipelined CPUs for controlling control hazards, which occur when a conditional branch's outcome is not immediately known. If the processor is uncertain which path to take, a branch (like an if-else condition or loop) may cause the pipeline to stall. Uncertainty in the control flow can interfere with the smooth operation of modern processors, which fetch and execute multiple instructions per clock cycle.

Processors employ branch predictors, which make an educated guess as to the outcome of a branch instruction before the actual outcome is known, to combat this problem. The pipeline keeps running uninterrupted if the prediction is correct. A branch misprediction, on the other hand, causes a performance penalty since the speculative instructions must be discarded in order to fetch the correct path, which adds delays and wastes processing power.

**Types of Branch Prediction**

**1. Static Branch Prediction:**

Static branch prediction makes decisions at compile time. A common example is the rule: “backward branches are taken; forward branches are not.” This kind of prediction is simple and easy to implement, but often inaccurate when dealing with dynamic workloads or complex branching behaviour during actual program execution. It does not adapt to the specific behaviour of a running program, limiting its effectiveness.

**2. Dynamic Branch Prediction:**

Dynamic prediction relies on historical behaviour of branches during runtime. It uses dedicated hardware structures like the Branch History Table (BHT) and Pattern History Table (PHT) to keep track of past outcomes. Based on this information, the processor can make more accurate predictions. Unlike static prediction, dynamic prediction continuously updates based on the program’s behaviour, making it more suitable for real-world applications where branches are not always consistent.

**3. Two-Level Predictors:**

Two-level predictors improve accuracy by using both the history of past branches and the current program counter (PC) value to predict outcomes. These include global history predictors, which track the overall behaviour of branches across the program, and local history predictors, which focus on the behaviour of individual branches. This layered strategy enables more precise predictions.

**4. Hybrid Prediction:**

Hybrid prediction combine multiple strategies to improve overall prediction accuracy. A meta-predictor is often used to decide which prediction method (e.g., global, or local) should be trusted for a particular branch. This dynamic switching helps increase efficiency, especially in programs with varying control flow patterns.

**Impact of Branch Prediction**

The accuracy of a branch predictor directly influences CPU performance. A highly accurate predictor enables near-continuous instruction flow, maximizing instruction throughput and minimizing the number of pipeline flushes. On the other hand, poor prediction accuracy leads to frequent stalls, reduced throughput, and inefficient resource usage.

Modern processors achieve up to 99% accuracy in some workloads using advanced methods such as TAGE (TAgged GEometric history length) predictors, which use variable history lengths, and neural predictors, which apply machine learning techniques to recognize patterns in branch behaviour. These advances have made branch prediction one of the most effective techniques for optimizing instruction flow in high-performance CPUs.

## ***2.2*** *Caching*

**Caching: Cutting Down on Memory Access Latency**

Caching, which helps close the speed difference between the processor and main memory (RAM), is another crucial element of contemporary computer architecture. Data and instructions that are frequently accessed are stored in caches, which are compact, quick memory units near the CPU. Caching greatly enhances system responsiveness and performance by lowering the frequency of accesses to slower main memory.

**Levels of Cache**

**L1 Cache:**

Situated right inside the CPU core, this cache is the smallest and fastest. To handle instructions and data independently, it is usually divided into an instruction cache (I-cache) and a data cache (D-cache). Although its small size (typically 32KB to 64KB) guarantees speedy access, it has a storage capacity limit.

**L2 Cache:**

Larger than L1 but slower, L2 cache acts as a backup when L1 misses occur. It is still located on the processor die and can range from 256KB to a few megabytes in size.

**L3 Cache:**

This cache is shared among all cores in a multi-core processor and is even larger (usually several megabytes). Though slower than L1 and L2, it is much faster than main memory and helps reduce inter-core data access latency.

**Cache Organization and Policies**

**Cache Mapping Techniques:**

* Direct Mapped: Each memory block maps to exactly one cache line. It is simple but prone to collisions (conflicts).
* Set-Associative: Memory blocks can go to any line in a predefined set, balancing speed and flexibility.
* Fully-Associative: Any memory block can go into any cache line. This method is the most flexible but expensive in terms of hardware complexity.

**Replacement Policies:**

* Least Recently Used (LRU): Replaces the block that hasn’t been used for the longest time.
* Random: Replaces a random block, easier to implement but less efficient.
* First-In-First-Out (FIFO): Replaces the oldest block.

**Write Policies:**

* Write-Through: This ensures consistency but increases write latency by updating the cache and main memory at the same time.
* Write-Back: This technique improves performance but necessitates extra logic because writes are made to the cache and only later written to memory.

**Cache Performance**

Cache performance is mainly evaluated using hit rate and miss rate:

* Cache Hit: When the needed data is found in the cache.
* Cache Miss: When the data is not in the cache and must be fetched from main memory, resulting in delays.

**Techniques to improve cache performance**

* Prefetching: Predicting which data will be needed soon and loading it in advance.
* Victim Caches: Small buffers that store recently evicted blocks, giving them another chance before being discarded.
* Inclusive/Exclusive Caching: Determines whether higher-level caches duplicate or separate stored data across levels.

## ***2.3*** *Relationship between Branch Prediction and Caching*

Branch prediction and caching are interdependent and do not operate independently. A miscalculated branch may cause data and instructions to be fetched into the cache more often than necessary, contaminating it and possibly displacing important data. Performance suffers because of the cache hit rate being decreased.

More consistent memory access patterns and a more seamless instruction flow are guaranteed by accurate branch prediction. This helps to maximise performance and power consumption by enabling the cache to serve required data more effectively. Therefore, effective coordination between these two mechanisms is essential in contemporary processors.

Additionally, both methods support sophisticated architectural features that rely on accurate instruction flow and quick memory access, like multi-threading, out-of-order execution, and speculative execution. Processors can execute complex instruction streams while reducing unnecessary computation and memory delays thanks to the cooperation between branch prediction and caching. Because even minor adjustments can result in appreciable increases in speed, energy efficiency, and user experience, designers are constantly improving these methods.

CHAPTER 3: COMPARATIVE STUDY OF PROGRAMMING APPROACHES

## ***3.1*** *Performance*

Performance is a critical factor in operating system development, as the efficiency of the underlying code directly affects CPU usage, memory processing, and system responsiveness. The choice between assembly language and high-level programming languages directly affects the efficiency of system resource use and the performance of the operating system under different workloads.

**Assembly Language**

Assembly language provides the highest level of control over a system's hardware, allowing developers to write instructions tailored to the behaviour of the processor. This fine-grained control enables optimizations that aren't always possible with high-level programming languages. For example, developers can fine-tune register usage, minimize memory accesses, and exploit features such as instruction routing and unlooping to reduce CPU cycles. These subtle improvements can dramatically increase execution speed, especially in timing-critical programs such as interrupt handlers or context switches. Assembly language also allows direct control of the hardware, eliminating the additional overhead of layers of abstraction found in high-level programming languages.

**High-Level Programming Language**

High-level programming languages, especially C, are widely used in developing modern operating systems because they provide an ideal balance between control and performance. C allows low-level memory operations to be accessed using pointers, and supports inline compilation when needed, giving developers the flexibility to optimize only the most important parts, with most of the operating system written in readable and maintainable code. Although high-level programming languages introduce some abstraction, their cost is often insignificant compared to their benefits in productivity and reliability.

**Modern Operating Systems**

Most modern operating systems, including Linux, Windows, and Mac OS, use high-level programming languages such as C or C++ in most of their code bases. Assembly language is reserved only for performance-critical modules. For example, the Linux kernel contains small parts of assembly language for architecture-specific procedures, while the rest is implemented in C for ease of porting and maintenance.

In conclusion, although assembly language offers unparalleled performance and control, its use is best limited to cases where careful optimization is critical. High-level programming languages, supported by powerful compilers and development tools, provide comparable performance in most use cases, while greatly improving productivity, code clarity, and long-term scalability. Therefore, a hybrid approach - using high-level programming languages for general operating system logic and assembly language for low-level optimizations - provides the best of both worlds in modern operating system design.

## ***3.2*** *Ease of Use*

When evaluating the usability of different programming approaches in developing operating systems, the contrast between low-level assembly programming and high-level programming languages such as C, C++, and Rust stands out noticeably. Ease of use affects not only the speed and effectiveness of a developer writing code, but also maintainability, debugging, and collaboration between teams in the long term.

Assembly language, being a low-level language, gives developers direct access to hardware components, such as central processing unit (CPU) registers, memory addresses, and peripherals. This level of control is very useful when developing critical components, such as device drivers, interrupt handlers, and bootloaders. However, this fine-grained control negatively impacts usability. Assembly language is known for being difficult to learn and use effectively. Developers must have an in-depth understanding of the target architecture, including instruction sets, memory hashing, and stack operations. Unlike high-level programming languages, assembly language does not provide built-in support for common programming structures, such as loops, conditions, and functions. Instead, developers must manage these operations manually using jump instructions and stack frames, which adds significant complexity.

Furthermore, writing even the simplest programs in assembly language requires many lines of code, making the development process time-consuming and error prone. Debugging is often more difficult due to the lack of descriptive error messages, the absence of standard debugging tools, and the need to trace execution at the instruction level.

High-level programming is more suitable for group development environments. Code written in high-level programming languages is easier to read and more self-documenting, enabling better collaboration between developers. When working on large operating system projects involving multiple contributors, this readability ensures that team members can quickly understand and modify each other's code without the need for deep architectural knowledge.

Assembly language is still necessary for specific tasks that require direct hardware manipulation or radical optimization. For example, boot loaders, context switch programs, and low-level interrupt service programs are often written in assembly language to ensure maximum efficiency and accuracy. However, in most other areas, such as process scheduling, memory management, and system calls, high-level programming languages dominate due to their simplicity and flexibility.

In conclusion, although assembly language is still important for some low-level functions, its steep learning curve and complexity make it much less user-friendly than its high-level alternatives. High-level programming languages not only speed up development and improve maintainability but also enable a wider range of developers to contribute to the development of operating systems. Improved ease of use, tool support, and collaboration capabilities make high-level programming the preferred approach for most modern operating system components.

## ***3.3*** *Security and Stability*

The emergence of speculative execution, particularly branch prediction, has significantly boosted CPU performance. However, it also introduced severe security vulnerabilities, most notably Meltdown and Spectre. These vulnerabilities have exposed critical weaknesses in how modern processors handle kernel memory and speculative execution, affecting nearly every device with a processor. While the technique is designed to improve speed by predicting and executing instructions before they are confirmed to be needed, attackers have found ways to exploit this behavior to access sensitive data like passwords and encryption keys. From a security and stability perspective, these flaws emphasize the need for exceptional system defenses and careful memory handling at the OS level.

**Meltdown and Spectre Overview**:

The emergence of speculative execution, particularly branch prediction, has significantly boosted CPU performance. However, it also introduced severe security vulnerabilities, most notably Meltdown and Spectre. These vulnerabilities have exposed critical weaknesses in how modern processors handle kernel memory and speculative execution, affecting nearly every device with a processor. While the technique is designed to improve speed by predicting and executing instructions before they are confirmed to be needed, attackers have found ways to exploit this behaviour to access sensitive data like passwords and encryption keys. From a security and stability perspective, these flaws emphasize the need for exceptional system defences and careful memory handling at the OS level.

**Security Implications:**

Meltdown primarily affects Intel and Apple processors and can be exploited to bypass the hardware barrier between user applications and sensitive parts of the system's memory. It leaks information by exploiting out-of-order execution to read kernel memory. Spectre, on the other hand, affects a broader range of devices including Intel, ARM, and AMD chips. It is a more complex and harder-to-mitigate attack that tricks applications into accessing arbitrary locations in their memory space. Unlike Meltdown, which is easier to patch, Spectre attacks require software developers and system architects to fundamentally rethink how they design secure systems. Both vulnerabilities stem from speculative execution, a mechanism that was never intended to handle security checks and thus became a critical flaw.

**OS-Level Responses and Mitigation:**

Operating system developers responded rapidly by releasing patches aimed at mitigating the vulnerabilities. Apple released a patch for OSX, Microsoft launched a Windows update, and Linux developers scrambled to implement similar fixes. The most effective solution to Meltdown was the introduction of Kernel Page Table Isolation (KPTI), which separates kernel and user space memory to prevent unauthorized access. However, this separation comes at a cost. Depending on the system and the tasks being performed, the performance slowdown can range between 5–30%. This trade-off reveals a long-standing issue in cybersecurity: the delicate balance between performance and protection. While these patches strengthen security and reduce risks of memory leaks and data breaches, they also impact usability, especially on older systems.

**Kernel Mode vs. User Mode Separation**

* **Restricted Access and Control**: Kernel mode allows full access to system resources, while user mode restricts applications from directly interacting with hardware. This separation helps prevent unauthorized access to critical parts of the system.
* **Improved System Stability**: By keeping regular applications in user mode, any crashes or bugs are contained. This means the entire operating system remains stable even if a single program fails or behaves unexpectedly.
* **Stronger Protection Against Attacks**: Separating user and kernel modes reduces the risk of exploits like buffer overflows. It ensures that malicious code running in user mode cannot gain high-level privileges or access sensitive memory areas.

In conclusion, speculative execution provides speed but introduces deep-rooted security risks. Addressing Meltdown and Spectre requires both hardware awareness and strong OS-level strategies, such as memory isolation and strict access controls, to ensure long-term system security and stability.

## ***3.4*** *Application in OS Development*

Complex responsibilities including memory management, process scheduling, device driver handling, and system call management are all part of developing an operating system (OS). The choice between low-level assembly language and high-level programming languages has a big influence on portability, performance, and maintainability.

**1. Kernel Programming:**

Traditionally, OS kernel components that require precise hardware control have used low-level assembly language. In the early phases of OS initialisation, assembly allows developers to control interrupts, work with CPU registers, and manage bootstrapping routines. For example, Linux uses assembly to write the basic bootloader and interrupt handling code. Windows also uses assembly in its boot loader and for context switching.

However, high-level languages like C is part of the bulk of kernel development due to better readability, portability, and developer productivity. For example, most of the Linux kernel is written in C, allowing complex operations with a lot fewer lines of code than assembly and better compiler optimizations.

**2. System Calls:**

System calls, which provides an interface between user applications and kernel services, also shows the use of both approaches. The actual logic behind system calls (like read(), write(), and fork()) is used in high-level languages, even though the system call entry and exit points often involve low-level assembly for register manipulation and transition between user and kernel modes.

**3. Driver Development:**

Device drivers need direct access to hardware, often require fine-tuned control using memory-mapped I/O and port I/O operations. while assembly can provide this level of accuracy, the majority of most of the drivers are written in C or C++ to balance between maintainability and performance. For example, Windows drivers usually use the Windows Driver Model (WDM) in C or C++, and Linux drivers are written in C. Assembly is only used for performance-critical paths or hardware-specific changes.

Since the shape generation system in the case study focusses on resource and performance limitations, assembly language is essential for low-level hardware interface and bootstrapping in OS development. High-level languages, on the other hand, are better at abstraction, maintainability, and productivity, which makes them more appropriate for the wider range of OS development, particularly as systems get more complicated.

CHAPTER 4: FUTURE ENHANCEMENTS & PROPOSALS

## ***4.1*** *Future Enhancements*

In performance-sensitive systems like the Shape Generation System, where efficient CPU and memory usage is vital, targeted improvements in branch prediction and caching can unlock significant speed gains. Future upgrades should encompass both hardware refinements and intelligent software techniques. Future enhancements should focus on both hardware and software aspects.

**Branch Prediction Enhancements:**

* **Hardware-Level Improvements:** Integrating simplified branch prediction mechanisms into low-end CPUs or microcontrollers could reduce the performance penalty of mis predicted branches. A small branch history table could track recent outcomes of conditional jumps, particularly useful in shape-drawing loops where predictable patterns exist.
* **Static and Profile-Guided Optimizations:** Compilers could apply static branch prediction techniques by analysing common execution paths and reordering code accordingly. Profile-guided optimization (PGO) could be used during development to analyse real-world usage patterns and optimize branch-heavy routines.
* **Loop Prediction in Shape Rendering:** Since shape generation often involves repeated patterns (drawing lines, scanning rows), predictive execution paths could be hardcoded into rendering loops, reducing the cost of evaluating conditions repeatedly.

**Caching Enhancements:**

* **Instruction and Data Caches:** On systems that allow it, adding a minimal instruction cache for frequently used routines (e.g., drawing functions) can save execution time. A small data cache could hold recently used pixel data or rendering parameters.
* **Software-Based Caching:** If hardware support is limited, software caching strategies—such as using lookup tables for common trigonometric calculations or storing precomputed shape templates—can reduce computation time and memory accesses.
* **Smart Cache Policies:** Implementing policies like Least Recently Used (LRU) or frequency-based eviction strategies for memory buffers (even in software) can prioritize critical drawing routines and data during runtime, keeping the system responsive.

## ***4.2*** *Benefits of Future Enhancement*

Enhancing branch prediction and caching not only boosts execution speed but also extends the system’s capabilities on low-power platforms. These improvements are particularly beneficial for embedded devices and microcontrollers, where resources are inherently constrained. Future improvements should address both hardware-level upgrades and smarter software strategies.

**Optimized Resource Usage:**

* Performance-critical routines like pixel plotting, hardware communication, or loop unrolling should remain in assembly for maximum speed.
* High-level languages like C can manage control logic, shape definitions, and system configuration, reducing the complexity of maintaining large blocks of assembly.

**Improved Maintainability:**

* High-level languages provide structure, error handling, and portability. By using them for non-performance-critical features—like user interaction or configuration handling—the overall system becomes easier to read, debug, and extend.
* Modular design allows developers to swap in new shapes, rendering modes, or UI elements without rewriting the entire core.

**Better Memory and Process Management:**

* High-level languages offer abstractions for dynamic memory allocation and resource tracking, which can help in managing frame buffers or input queues more efficiently.
* When paired with low-level memory operations in assembly, this creates a more flexible and controlled environment for managing limited resources.

**Platform Flexibility and Portability:**

* Using a high-level wrapper around low-level routines enables easier porting to new hardware. Only the assembly portion would need adjustment for different microcontrollers or display drivers.
* This hybrid design supports future expansion, like adding colour rendering or animation, by allowing high-level modules to control features while still leveraging fast, low-level execution where it matters most.

**Easier Testing and Debugging:**

* High-level code can include diagnostic routines, logging mechanisms, or test harnesses that monitor the low-level engine without interfering with performance.
* This separation makes it possible to identify and fix issues faster during development and testing.

CONCLUSION

The comparative study concludes by highlighting the important trade-offs that must be done when developing operating systems between high-level programming and low-level assembly. High-level languages improve development speed, maintainability, and security, while assembly provides control and efficiency for hardware interfacing and performance optimisations like branch prediction and caching. Both strategies are carefully used in modern operating systems like Windows and Linux to optimise versatility and performance. This hybrid model will help future developments in hardware and software design, especially in caching and branch prediction, guaranteeing stable, effective, and flexible operating systems that can handle changing security threats and computing demands.

REFERENCES

Andrii, V. (2024, March 18). *Writing a Windows Driver Model Driver: A Step-by-Step Guide*. Retrieved from apriorit: https://www.apriorit.com/dev-blog/791-driver-windows-driver-model

Ballejos, L. (2024, December 5). *What Is a High-Level Programming Language?* Retrieved from ninjaOne: https://www.ninjaone.com/it-hub/it-service-management/what-is-a-high-level-programming-language/

coursera Staff. (2024, October 10). *Low-Level vs. High-Level Programming Languages*. Retrieved from coursera: https://www.coursera.org/articles/high-level-programming-languages

Coursera Staff. (2025, January 7). *Memory Management: 2025 Overview*. Retrieved from Coursera: https://www.coursera.org/articles/memory-management

Dorner, C. (2024, September 4). *Understanding Linux Kernel Programming: An In-Depth Guide with Coding Examples*. Retrieved from Linkedin: https://www.linkedin.com/pulse/understanding-linux-kernel-programming-in-depth-guide-charles-dorner-ggone/

Gaurav, S. (2023, October 15). *Interrupt Handling*. Retrieved from Scaler: https://www.scaler.com/topics/operating-system/interrupt-handling/

GeeksforGeeks. (2023, November 19). *What is a Low Level Language?* Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/what-is-a-low-level-language/

GeeksforGeeks. (2024, December 20). *Caching - System Design Concept.* Retrieved from GeeksforGeeks: https://www.geeksforgeeks.org/caching-system-design-concept-for-beginners/

Kanade, V. (2023, June 15). *What Is Assembly Language? Working, Features, and Advantages*. Retrieved from spiceworks: https://www.spiceworks.com/tech/tech-general/articles/what-is-assembly-language/

Kaplarevic, V. (31, August 2023). *System Calls in Operating System Explained*. Retrieved from phoenixNAP: https://phoenixnap.com/kb/system-call

Karolina Rusinowicz, D. J. (2022, March 24). *An introduction to low-level programming*. Retrieved from codilime: https://codilime.com/blog/introduction-to-low-level-programming/

Majkowski, M. (2021, May 6). *Branch predictor: How many "if"s are too many? Including x86 and M1 benchmarks!* Retrieved from CloudFlare: https://blog.cloudflare.com/branch-predictor/

Root, E. (2022, February 1). *Spectre vulnerability: 4 years after discovery*. Retrieved from kaspersky: https://www.kaspersky.com/blog/spectre-meltdown-in-practice/43525/

*Spectre and Meltdown explained: A comprehensive guide for professionals*. (2019, May 15). Retrieved from TechRepublic: https://www.techrepublic.com/article/spectre-and-meltdown-explained-a-comprehensive-guide-for-professionals/

The University of Edinburgh. (2018, February 19). *Computer Architecture: Branch Prediction.* Retrieved from The University of Edinburgh: https://www.inf.ed.ac.uk/teaching/courses/car/Notes/2017-18/lecture05-handling\_hazards.pdf

Watkins, D. (2022, March 28). *Scheduling tasks with the Linux cron command*. Retrieved from opensource: https://opensource.com/article/22/3/scheduling-tasks-linux-cron

WORKLOAD MATRIX

|  |  |  |
| --- | --- | --- |
| NAME | TASK | SIGNATURE |
| AHMED MIRAHUSAIN ALVI (TP084807) | * **Workload Matrix** * **Branch Prediction** | *AHMED ALVI* |
| MOHAMMED YOUSEF MOHAMMED MOHAMMED (TP085042) | * **Performance** * **Ease of Use** | **MO** |
| SULTAN ABDULLA OMAR TAKRORI (TP085327) | * **Introduction** * **Security and Stability** | **Stk** |
| MAHRUS SHAMSUL AHSAN (TP085562) | * **Application in OS Development** * **Conclusion** | ***~~MSA~~*** |
| ABDALLAH MOHAMED MAHMOUD MOHAMED MAHMOUD (TP085097) | * **Future Enhancement** * **Benefits of Future Enhancement** | **Abdo** |
| ABDUL AZIZ VAYANI (TP080822) | * **Caching** * **References** * **Table of Contents** | ***Aziz*** |