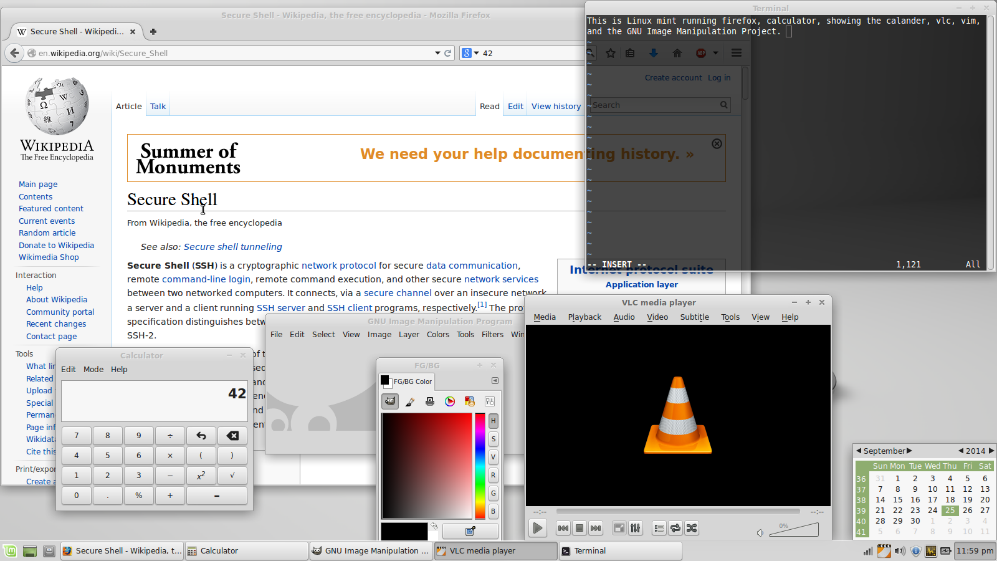
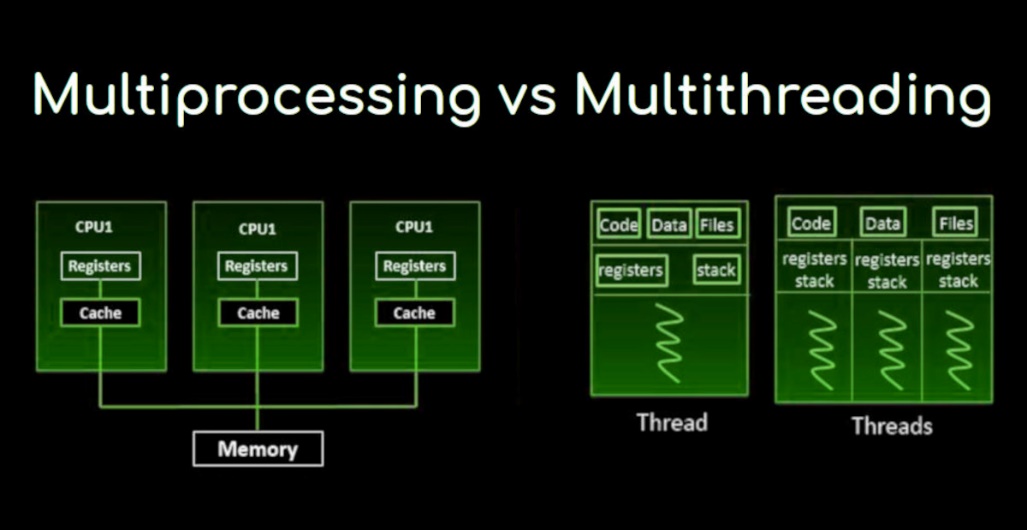
CHAPTER 20 MULTITHREADING AND MULTITASKING

In Chapter 20, we will delve into the intricacies of multitasking and multithreading in the Windows API. We will explore key concepts, provide clear explanations, relevant code examples, and insights from Charles Petzold's book. This chapter will address the following topics:

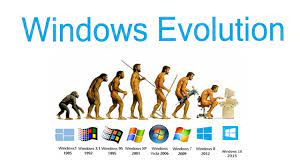
Multitasking: We will discuss the operating system's ability to run multiple programs concurrently, allocating time slices to each process. This creates the illusion of simultaneous execution and enhances system responsiveness.



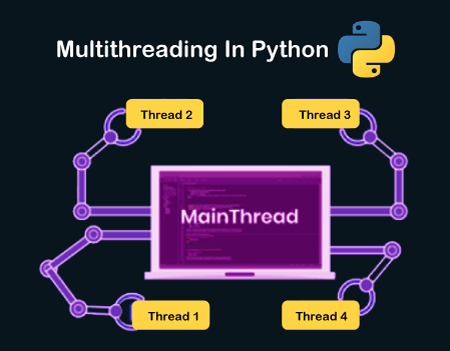
Multithreading: We will explore the ability of a single program to split its execution into multiple threads. Multithreading allows concurrent execution of tasks within the program, enabling background tasks, maintaining responsive user interfaces, and executing concurrent operations.



Windows Multitasking Evolution: We will examine the evolution of multitasking in Windows. In 16-bit Windows, multitasking capabilities were limited due to cooperative multitasking, where programs voluntarily yielded control to others. In 32-bit Windows, true multitasking using preemptive multitasking was introduced. The operating system actively assigns and revokes CPU time slices, ensuring responsiveness and preventing program monopolization.



Multithreading Benefits: We will discuss the benefits of multithreading, including the ability to perform background tasks without blocking user interaction, maintaining responsive user interfaces through separate UI update threads, and executing independent tasks simultaneously for improved performance on multiprocessor systems.

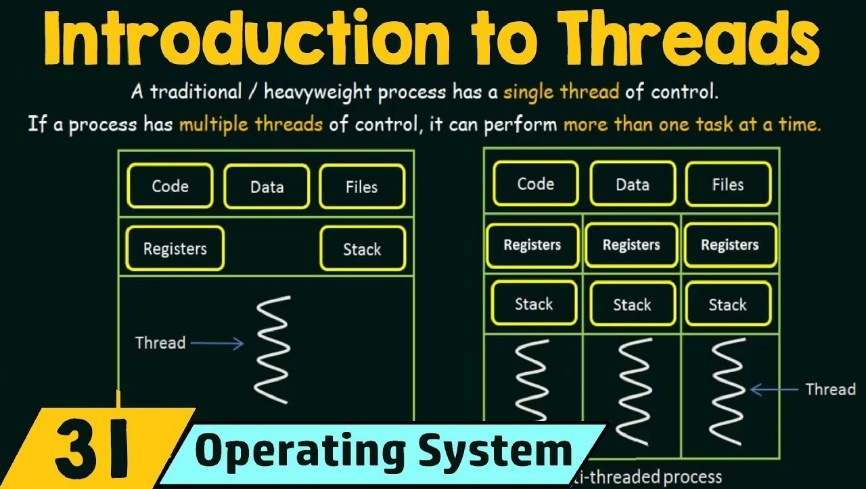


Key Terminology:

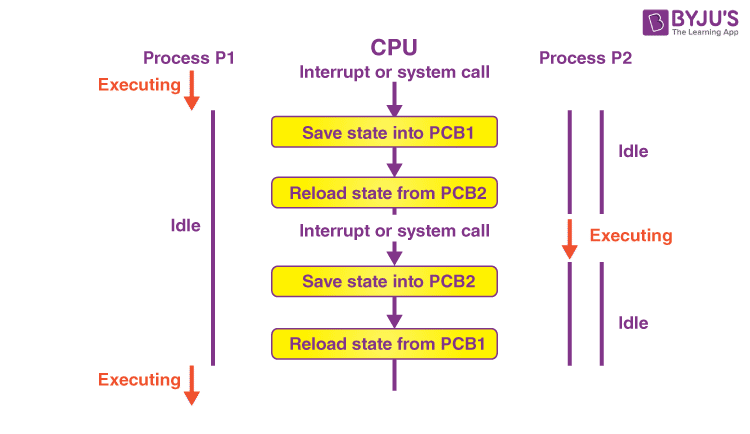
Process: A running instance of a program, with its own memory space and resources.



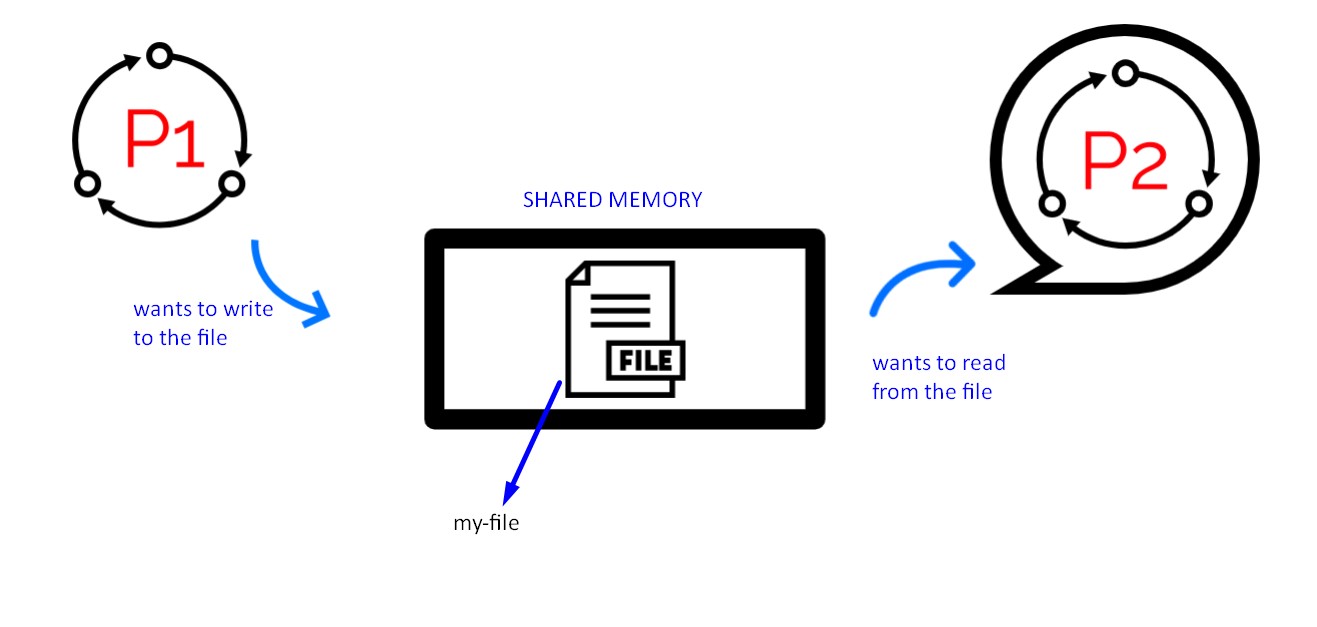
Thread: A lightweight execution unit within a process, sharing the process's memory and resources.



Context Switching: The process of saving and restoring a thread's state when switching between threads.



Synchronization: Mechanisms to coordinate access to shared resources among multiple threads, preventing data corruption and race conditions.



Topics Covered in Chapter 20:

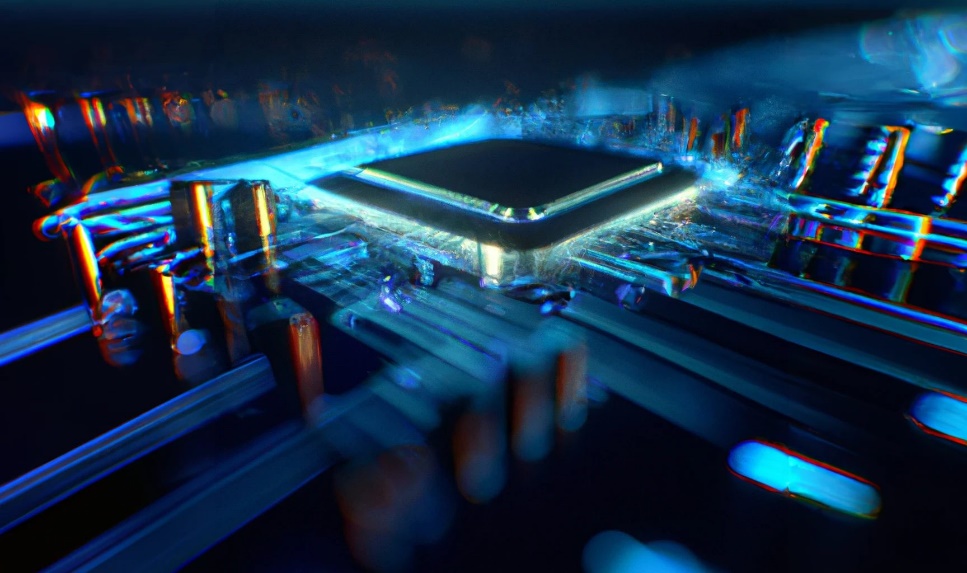
* Thread Creation and Management: We will explore the CreateThread function for creating threads, setting thread priorities, suspending and resuming threads, and terminating threads.
* Synchronization Techniques: We will discuss synchronization mechanisms such as critical sections, mutexes, semaphores, and events. These mechanisms coordinate access to shared resources among multiple threads, preventing data corruption and race conditions.
* Thread-Specific Storage: We will examine thread-specific storage using the TlsAlloc, TlsGetValue, and TlsSetValue functions. Thread-specific storage allows each thread to have its own unique data.
* Win32 Timers: We will explore the use of timers in multithreaded programming using the SetTimer and KillTimer functions. Timers allow you to schedule recurring or one-time events in your program.
* Asynchronous Procedure Calls: We will discuss asynchronous procedure calls using the BeginThreadEx and QueueUserAPC functions. These functions allow you to execute code asynchronously in a separate thread.
* Multithreaded Programming Best Practices: We will provide best practices for multithreaded programming, including avoiding deadlocks, optimizing thread performance, and ensuring thread safety.

MULTITASKING IN THE DOS ERA: A TALE OF CREATIVITY AND LIMITATIONS

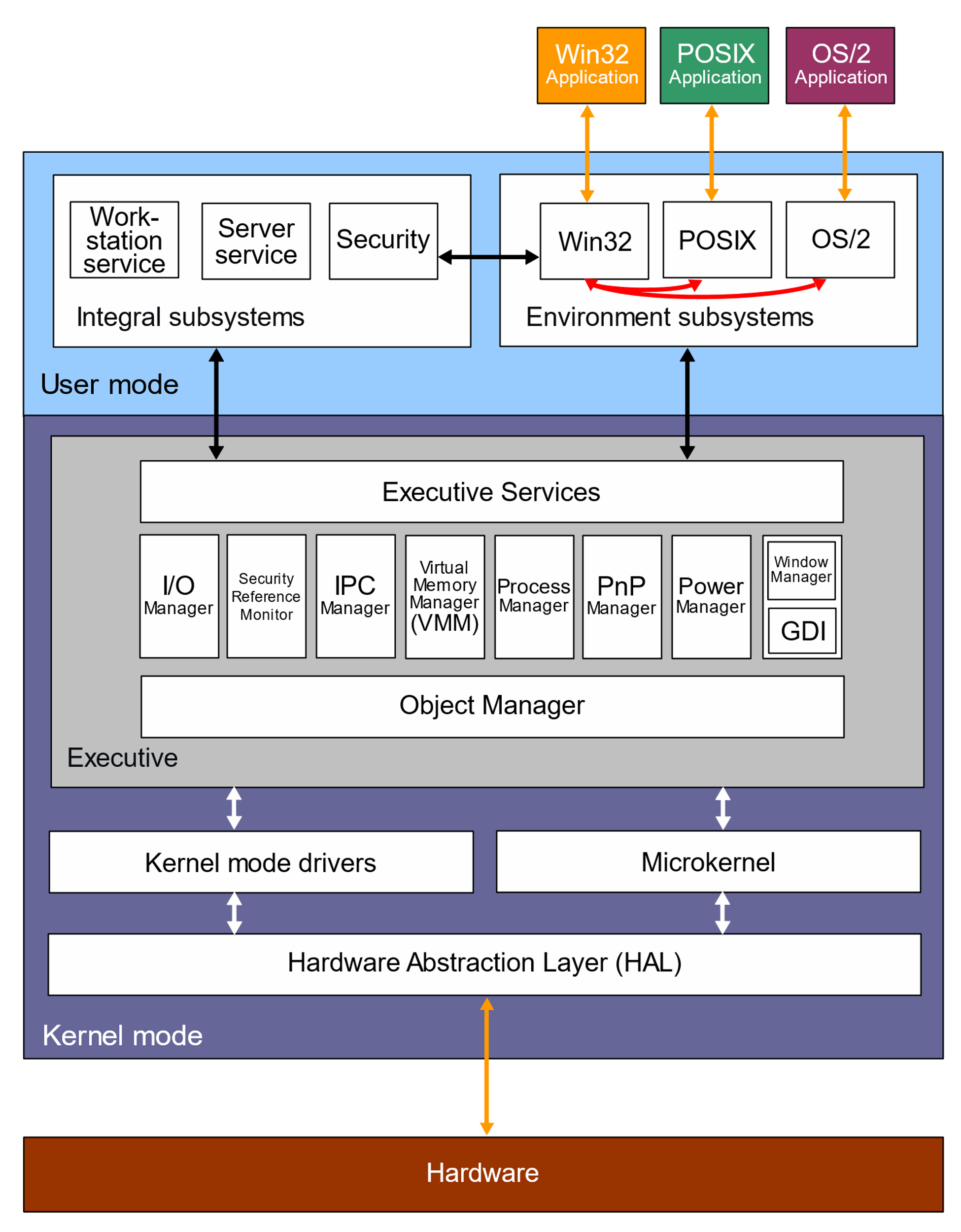
The early days of PC computing presented a fascinating paradox when it came to multitasking. While skeptics questioned its utility on single-user machines, users gradually gravitated towards its benefits, even before its complete realization. Understanding this story necessitates delving into the technical and pragmatic constraints that shaped multitasking's evolution under DOS.

Obstacles to Multitasking under DOS:

Hardware Limitations: The Intel 8088 lacked dedicated features for efficient memory management, crucial for juggling multiple programs and their memory allocations. Moving memory blocks to consolidate free space proved challenging, hindering robust multitasking implementations.



DOS Architecture: Designed for simplicity and minimal resource consumption, DOS offered limited APIs for programs to interact with the system beyond basic file access and program loading. This lack of robust system services hampered developers' ability to implement true multitasking within the OS itself.

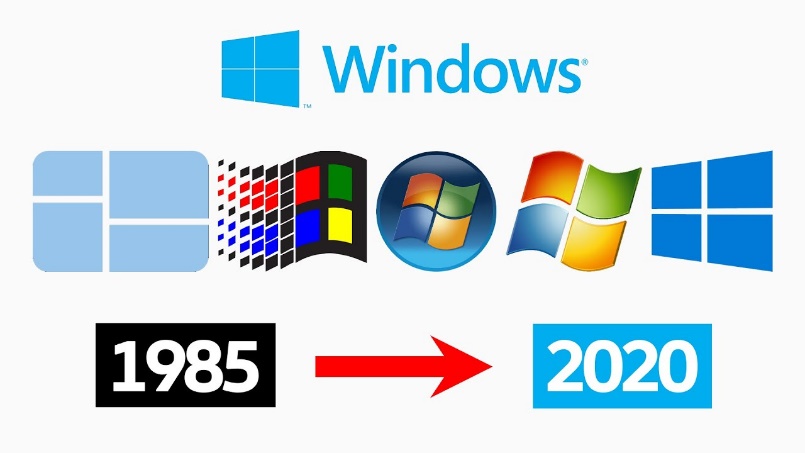


Creative Workarounds:

Terminate-and-Stay-Resident (TSR) Programs: These innovative programs occupied a small portion of memory while remaining active in the background, even after switching to another program. Some TSRs, like print spoolers, leveraged the hardware timer interrupt to perform background tasks without impacting the foreground application. Others, like SideKick, employed task switching techniques, temporarily suspending the running program while displaying their own interface.



Enhanced DOS Features: Microsoft progressively added features to DOS, such as memory swapping to disk, that indirectly benefited multitasking by providing more flexible memory management.



Market Response and Limitations:

Limited Success of Task-Switching Shells: Attempts to build task-switching environments on top of DOS, like Quarterdeck's DesqView, offered rudimentary multitasking functionalities. However, their complexity and performance limitations prevented widespread adoption.



Key Takeaways:

* Multitasking emerged as a desired user experience on PCs despite technical limitations inherent to the platform and DOS architecture.
* Creative programmers tackled these limitations through TSRs and rudimentary task-switching approaches, paving the way for more advanced solutions.
* The limitations of these workarounds highlighted the need for a dedicated operating system capable of robust and user-friendly multitasking, leading to the eventual dominance of Windows.

Further Exploration:

* Investigate specific examples of popular TSRs and their functionalities.
* Analyze the technical challenges of memory management and context switching in the DOS environment.
* Compare and contrast the limitations of early DOS-based multitasking solutions with the capabilities of Windows 3.1 and beyond.

By understanding the ingenuity and constraints of the DOS era, we gain deeper appreciation for the advancements in multitasking that laid the groundwork for modern computing experiences.

MULTITASKING IN THE EARLY WINDOWS ERA:

Windows 1.0's Breakthrough: Introduced in 1985, Windows 1.0 offered a more sophisticated multitasking solution than TSRs or task-switching shells, even within the constraints of real mode.

Graphical Interface for Multitasking: Windows' graphical environment distinguished itself from command-line systems like UNIX by enabling multiple programs to run concurrently on the same screen, facilitating seamless switching and data exchange.

Nonpreemptive Multitasking: Cooperation Required:

Message-Based Architecture: Windows programs are primarily driven by messages, often originating from user input. They typically remain idle until a message arrives.

No Preemptive Time Slicing: 16-bit Windows did not enforce time-based task switching. Instead, control switched only when a program voluntarily returned control to Windows after processing a message.

Cooperative Nature: This reliance on programs to "play fair" and yield control earned it the name "cooperative multitasking." A poorly designed or unresponsive program could monopolize the system.

Exceptions and Workarounds:

Preemption for DOS and Multimedia: Windows did employ preemptive multitasking for running DOS programs and for multimedia tasks within DLLs, which required timely responses to hardware events.

Coping with Limitations:

Hourglass Cursor: A visual signal to the user that a program was busy, but not a true solution.

Windows Timer: Allowed periodic execution of code for tasks like animation and clocks.

PeekMessage Function: Enabled programs to periodically check for messages and relinquish control voluntarily, preventing complete unresponsiveness during long operations.

1. Clipboard:

Foundation for Data Sharing: The most basic and widely used method for transferring data between Windows programs.

Cut, Copy, Paste Operations: Enables users to seamlessly move text, images, or other data between applications.

Temporary Storage: Acts as a temporary holding area for copied or cut data, accessible to most Windows programs.

2. Dynamic Data Exchange (DDE):

Establishing Links: Allows two programs to establish a link and exchange data in real time, even if they're not actively running.

Client-Server Model: One program (client) requests data from another (server), enabling dynamic updates.

Common Use Cases: Stock market tickers, spreadsheets linked to databases, and other scenarios requiring live data exchange.

Limitations: Can be complex to implement and prone to errors if links break.

3. Object Linking and Embedding (OLE):

Embedding Objects: Enables one document to contain objects created in other programs, maintaining their original formatting and functionality.

In-Place Editing: Users can often edit embedded objects within the main document, without launching the original application.

Example: Embedding a spreadsheet chart within a Word document, allowing direct editing of the chart without leaving Word.

Compound Documents: Facilitates the creation of rich, interactive documents with components from various sources.

Key Points:

* Each mechanism offers distinct advantages for different data sharing scenarios.
* The clipboard is versatile but requires manual copying and pasting.
* DDE excels for live data exchange but can be more complex.
* OLE provides seamless integration of objects, fostering richer document experiences.
* 16-bit Windows relied on nonpreemptive multitasking, a cooperative model that depended on programs to yield control regularly.
* This model had limitations, as a single program could potentially block others.
* Windows provided some mechanisms to mitigate these issues, but true preemptive multitasking would require a more robust foundation.

Further Exploration:

* Consider the challenges of implementing preemptive multitasking within the constraints of 16-bit Windows.
* Research specific examples of how nonpreemptive multitasking affected user experience and application design in early Windows programs.
* Explore the evolution of multitasking techniques in subsequent Windows versions, leading to the full-fledged preemptive model in 32-bit Windows.
* Universal Data Exchange Formats: Consider the role of formats like XML and JSON in modern data interchange.
* Clipboard Enhancements: Explore features like clipboard history and cloud-based syncing for enhanced clipboard functionality.
* OLE Alternatives: Investigate technologies like ActiveX and .NET components for object-based integration.

PM AND THE SERIALIZED MESSAGE QUEUE

Contextualizing OS/2 and PM:

Early Multitasking Exploration: OS/2 and PM, products of Microsoft and IBM's collaboration, represented a significant step in the quest for robust multitasking within a graphical interface reminiscent of Windows.

Preemptive Kernel: The underlying OS/2 operating system embraced preemptive multitasking, enabling the system to forcibly switch between tasks based on time slices.

Serialized Message Queue in PM:

Mouse and Keyboard Input Serialization: PM introduced a serialized message queue specifically for user input messages originating from the keyboard and mouse. This design imposed a strict sequence on their processing.

Implications: PM would withhold delivery of a new keyboard or mouse message until the program had fully processed the previous one, including any secondary messages it triggered, such as menu commands.

Rationale and Trade-offs:

Predictable Input Focus: The primary motivation for serialization was to safeguard predictable type-ahead and mouse-ahead behavior—allowing users to type or click ahead of the system's visual response.

Focus Shift Handling: This predictability was crucial when a message caused a change in input focus between windows. Serialization ensured that subsequent keyboard messages would accurately target the newly focused window.

Responsiveness Sacrifice: However, this design trade-off came at the cost of potential system-wide unresponsiveness. A program struggling to process messages could effectively stall the entire PM environment, preventing other programs from receiving any user input.

Modern Messaging Paradigms:

Deserialization for Responsiveness: Recognizing the limitations of serialization, 32-bit Windows versions embraced deserialized message queues. This approach prioritizes responsiveness and task isolation, preventing a single program from monopolizing user input.

Seamless Input Focus Switching: Even if a program is occupied with a lengthy operation, users can seamlessly switch input focus to a different program, maintaining system responsiveness.

Key Takeaways:

* Serialized message queues prioritized predictable input behavior, but often at the expense of system responsiveness.
* Modern operating systems favor deserialized message queues to ensure responsiveness and prevent single programs from impeding multi-tasking experiences.

Further Exploration:

* Balancing Predictability and Responsiveness: Delve deeper into the ongoing challenge of balancing these competing design goals in user interface systems.
* Alternative Input Handling Approaches: Explore alternative strategies for ensuring predictable input behavior without resorting to strict message serialization.
* Impact on User Experience: Examine the tangible effects of different message queue designs on user experience, productivity, and overall system satisfaction.

*In the context of threading, multitasking, and WinAPI, PM refers to the Presentation Manager, a graphical user interface (GUI) system that was part of the OS/2 operating system.*

* Developed by Microsoft and IBM: PM was initially a joint effort between the two companies in the late 1980s.
* GUI for OS/2: It served as the primary GUI for OS/2, providing a visually rich environment for applications.
* Similarities to Windows: PM shared many design elements and concepts with early Windows versions, such as windows, menus, icons, and a message-driven architecture.
* Serialized Message Handling: One of PM's distinctive features was its serialized message queue for keyboard and mouse input, which aimed to ensure predictable input behavior but could potentially impact responsiveness.
* Eventually Replaced: While OS/2 continued to evolve, PM was eventually superseded by other GUI systems, such as Workplace Shell and, later, Windows NT.

Relevance to WinAPI:

* Historical Context: Understanding PM's design and its approach to message handling provides valuable insights into the evolution of multithreading and multitasking concepts within Windows API.
* Threading and Multitasking Principles: PM's serialized message queue illustrates a specific approach to managing user input in a multitasking environment, highlighting the trade-offs between predictability and responsiveness.
* Impact on Windows API: The lessons learned from PM's design likely influenced the development of more robust and responsive threading and multitasking features in subsequent Windows API versions.

While PM is not directly used in modern Windows development, its legacy provides valuable context for understanding the evolution of multithreading and multitasking concepts in the WinAPI.

MULTITHREADED ARCHITECTURE:

Primary Thread: Assign this thread the exclusive responsibility of user interactions and interface management. It creates windows, holds window procedures, and processes messages. This ensures responsiveness and prevents UI freezes due to lengthy background tasks.

Secondary Threads: Dedicate these threads to background tasks that don't directly involve user interactions. They diligently perform computations, data processing, or other resource-intensive operations without compromising the user experience.

Governor and Staff Analogy:

Governor (Primary Thread): Acts as the public face of the program, handling external communication (user input and messages) and delegating tasks to staff members.



Staff (Secondary Threads): Perform their assigned work meticulously behind the scenes, reporting back to the governor for further instructions. They avoid direct interactions with the outside world, ensuring a unified and responsive user experience.

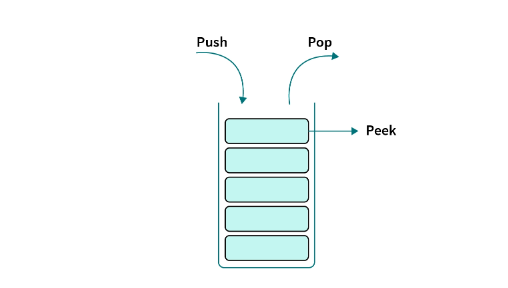


Resource Sharing and Thread Independence:

Shared Resources: Threads within a process share resources like memory, open files, and static variables. This enables efficient data exchange and collaboration.



Separate Stacks and Processor State: Each thread maintains its own stack, ensuring automatic variables remain unique to that thread. They also have independent processor states, allowing the operating system to seamlessly switch between them without context loss.



Key Considerations:

* Task Delineation: Carefully identify tasks that can be executed independently and delegate them to secondary threads to maximize responsiveness and resource utilization.
* Communication Mechanisms: Establish effective communication channels between threads for data exchange, synchronization, and coordination.
* Synchronization: Implement appropriate synchronization techniques (e.g., mutexes, semaphores) to prevent race conditions and ensure data integrity when multiple threads access shared resources.
* Thread Safety: Design code with thread safety in mind to avoid unexpected behavior or crashes in multithreaded environments.

THREADING CHALLENGES AND COMPLEXITIES:

Unpredictable Interactions and Debugging Difficulties:

Preemptive multitasking systems can interrupt threads at any point, leading to unexpected and often intermittent interactions that can be challenging to reproduce and debug.

Race conditions, deadlocks, and subtle timing-related bugs are common in multithreaded environments, requiring careful design and synchronization techniques to prevent.

Synchronization and Coordination:

Semaphores: Allow programmers to halt a thread's execution until signaled by another thread, facilitating coordination and preventing data access conflicts.

Critical Sections: Sections of code that cannot be interrupted, ensuring atomic operations and preventing data corruption in multithreaded contexts.

Deadlocks: Occur when threads mutually block each other's execution, requiring careful design and synchronization strategies to avoid.

Architectural Considerations:

Task Decomposition: Partitioning tasks into independent threads can enhance responsiveness and resource utilization but necessitates thoughtful synchronization and communication mechanisms.

Thread Safety: Ensuring code functions correctly in multithreaded environments requires careful design and synchronization to protect shared data and resources.

Platform-Specific Considerations:

32-bit Advantages: 32-bit programs often exhibit greater immunity to certain thread-related issues compared to 16-bit programs due to differences in instruction sets and compiler optimizations.

Critical Sections: While 16-bit programs might require critical sections for seemingly simple operations like incrementing 32-bit variables, such measures might be unnecessary in 32-bit environments.

64-bit Considerations:

* Data Size and Atomicity: 64-bit architectures can handle larger data types and often perform 64-bit operations atomically, further reducing the need for explicit synchronization in some cases.
* Compiler Optimizations: Modern compilers can generate more efficient and thread-safe code, potentially mitigating some threading challenges.
* Cache Coherency: 64-bit systems often employ more sophisticated cache coherency protocols, which can improve performance but also introduce new synchronization complexities.
* Instruction Reordering: Advanced processors might reorder instructions for optimization, potentially impacting thread behavior. Memory barriers become crucial to ensure correct execution order in such cases.

*Important Note:*

While 64-bit systems offer potential benefits, thread safety and synchronization remain essential. Don't assume operations are atomic simply because they're performed on 64-bit variables. Always consult language and platform-specific documentation for accurate guidance on thread-safe practices.

Best Practices:

* Minimize Shared State: Reduce shared data and resources among threads to minimize synchronization overhead and potential conflicts.
* Prefer Fine-Grained Synchronization: Isolate critical sections to the smallest possible code blocks to enhance concurrency and reduce blocking.
* Thorough Testing: Employ rigorous testing under various load conditions to expose potential threading issues, as they often manifest intermittently.
* Prioritize Thread Safety: Design code with thread safety in mind from the outset to avoid costly refactoring later.