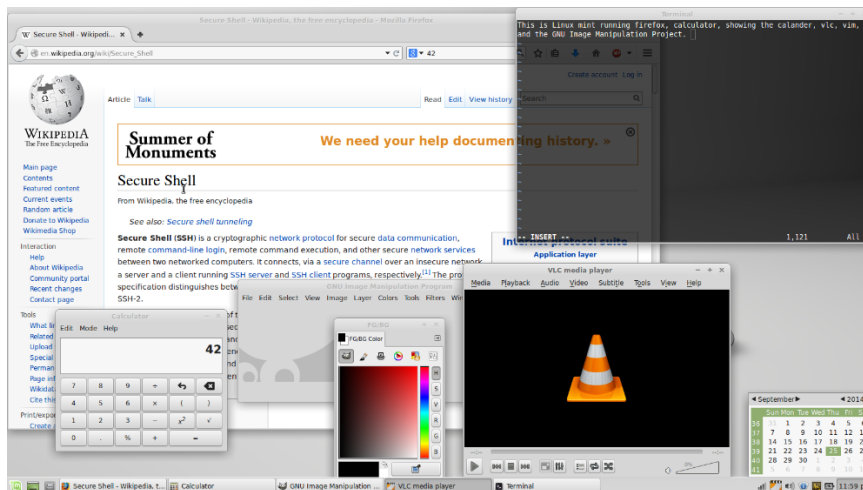


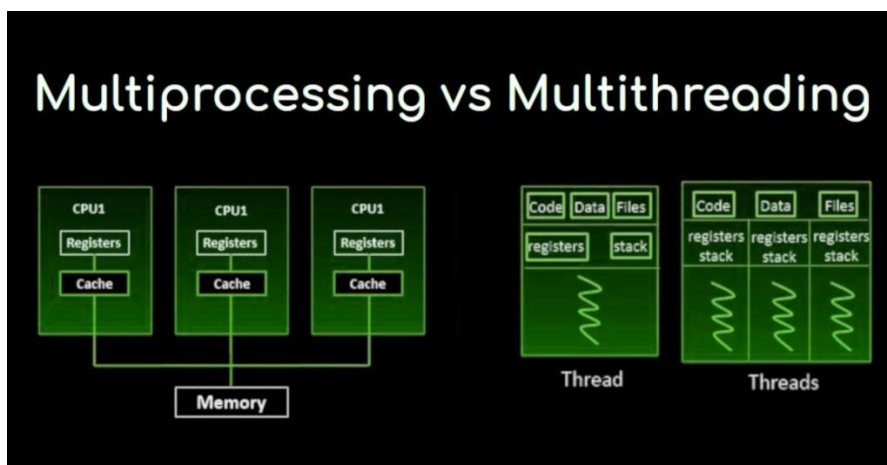
CHAPTER 20 MULTITHREADING AND MULTITASKING

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.

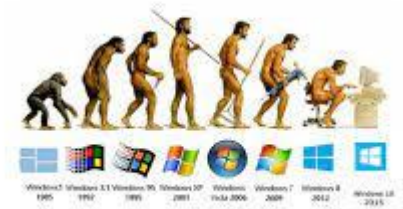


Multithreading: We will explore the ability of a single program to split its execution into multiple threads. Multithreading allows concurrent execution of tasks within the program, enabling background tasks, maintaining responsive user interfaces, and executing concurrent operations.

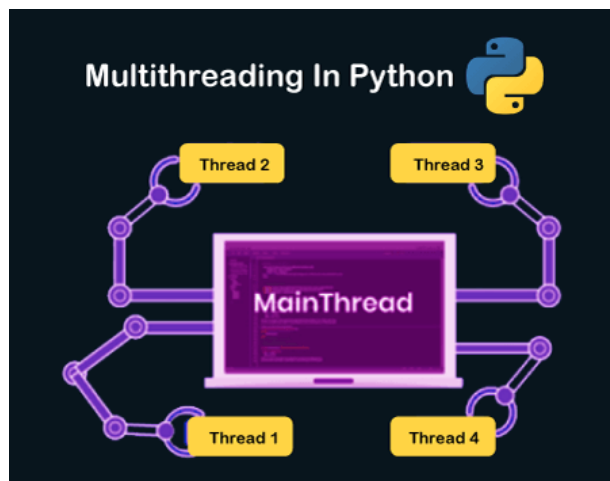


Windows Multitasking Evolution: We will examine the evolution of multitasking in Windows. In 16-bit Windows, multitasking capabilities were limited due to cooperative multitasking, where programs voluntarily yielded control to others. In 32-bit Windows, true multitasking using preemptive multitasking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

Windows Evolution



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.

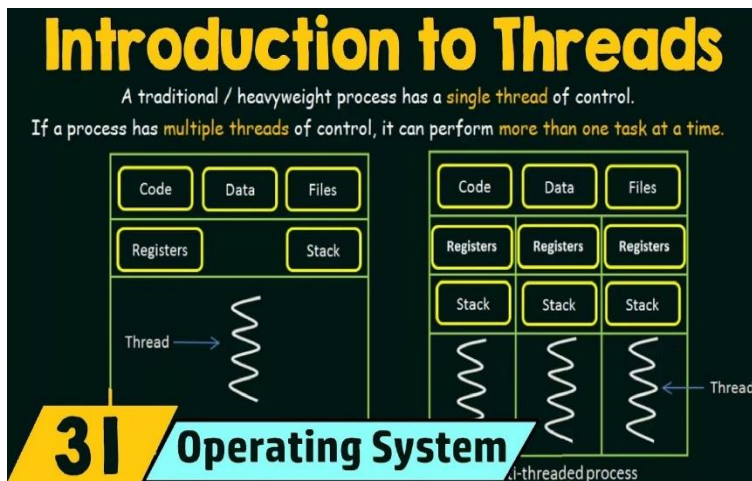


Key Terminology:

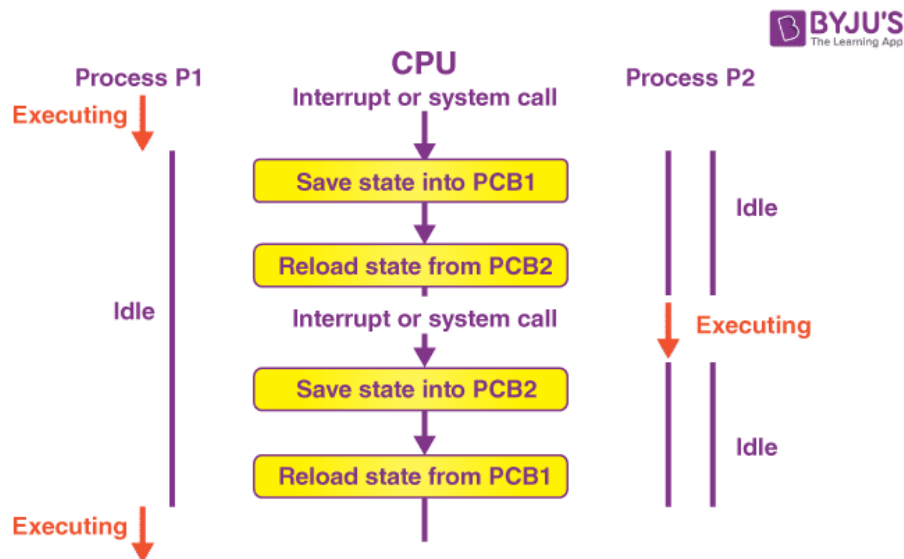
Process: A running instance of a program, with its own memory space and resources.



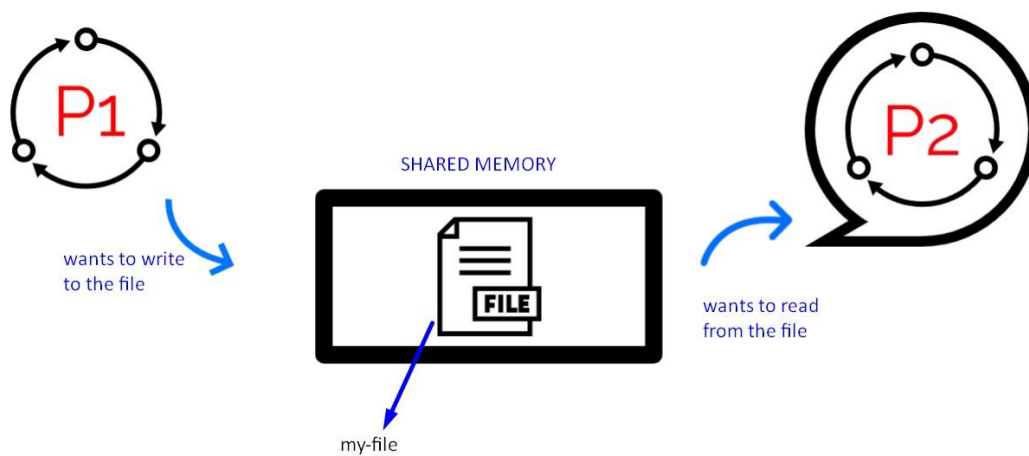
Thread: A lightweight execution unit within a process, sharing the process's memory and resources.



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:

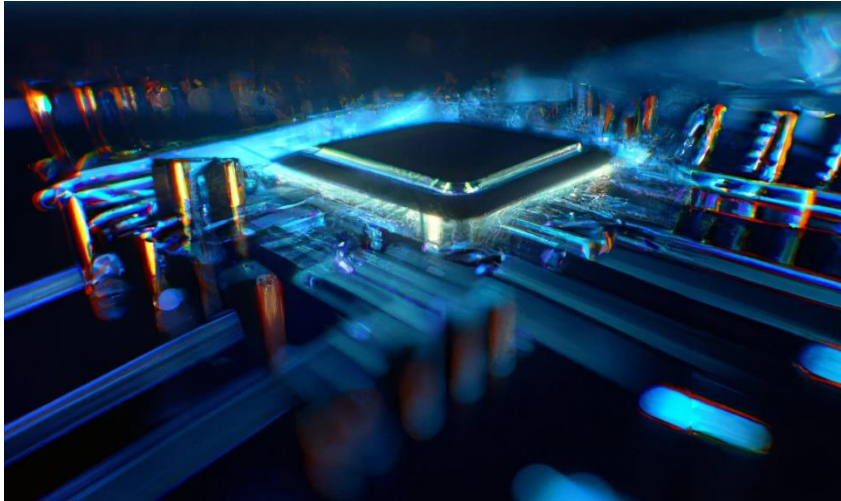
- **Thread Creation and Management:** We will explore the `CreateThread` function for creating threads, setting thread priorities, suspending and resuming threads, and terminating threads.
- **Synchronization Techniques:** We will discuss synchronization mechanisms such as critical sections, mutexes, 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

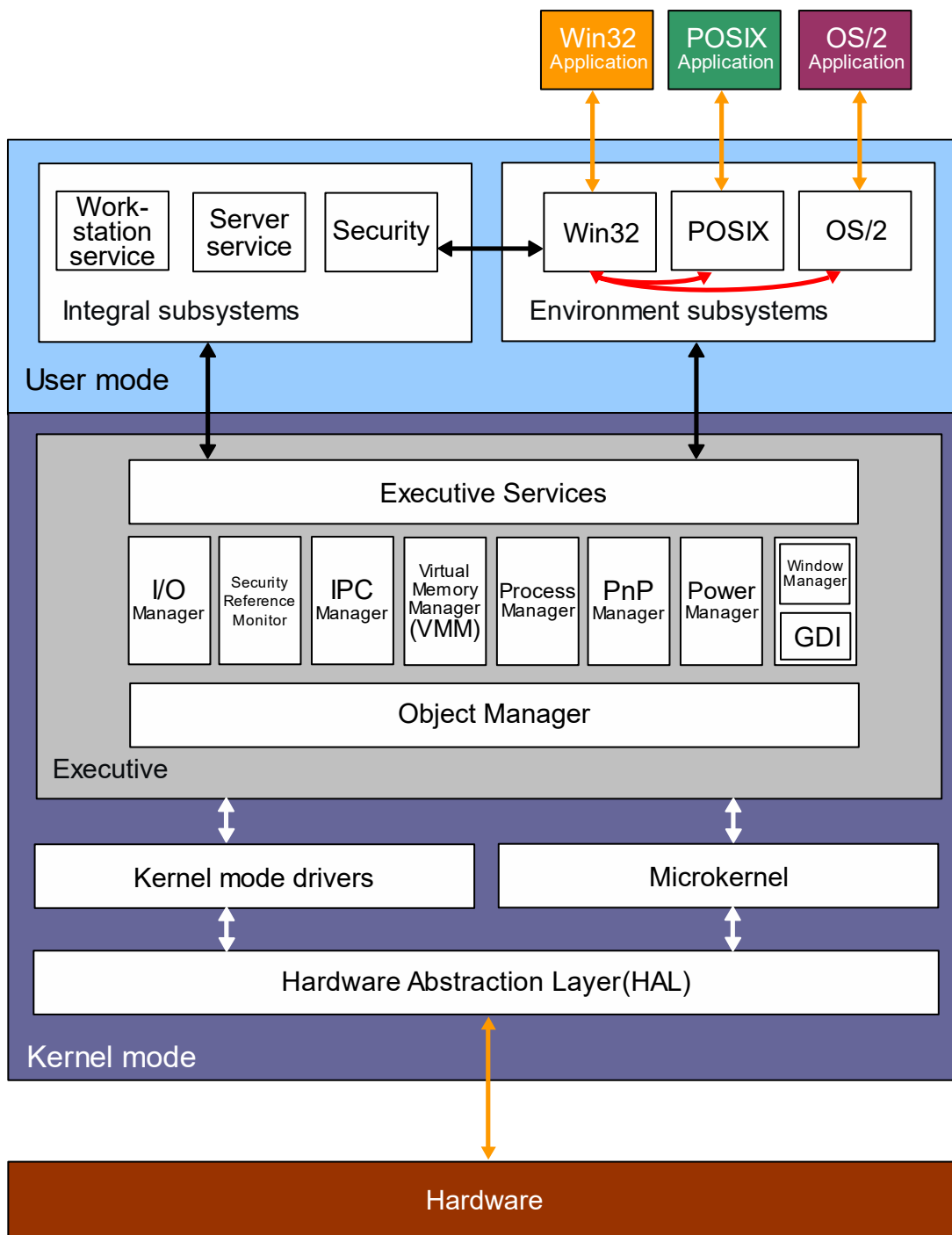
The [early days of PC computing presented a fascinating paradox](#) when it came to multitasking. While skeptics questioned its utility on single-user machines, users gradually gravitated towards its benefits, even before its complete realization. Understanding this story necessitates delving into the 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.

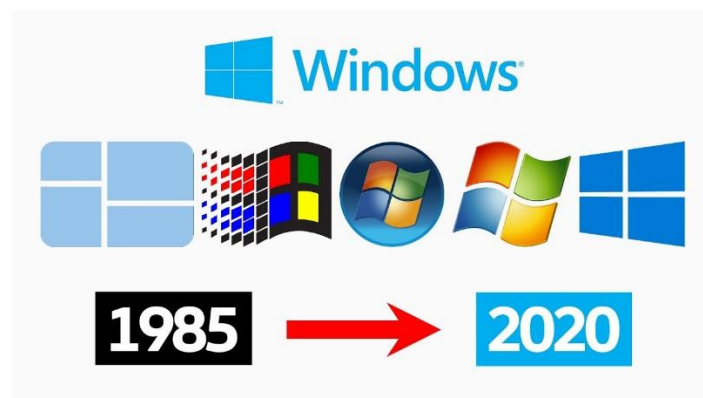


Creative Workarounds:

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.

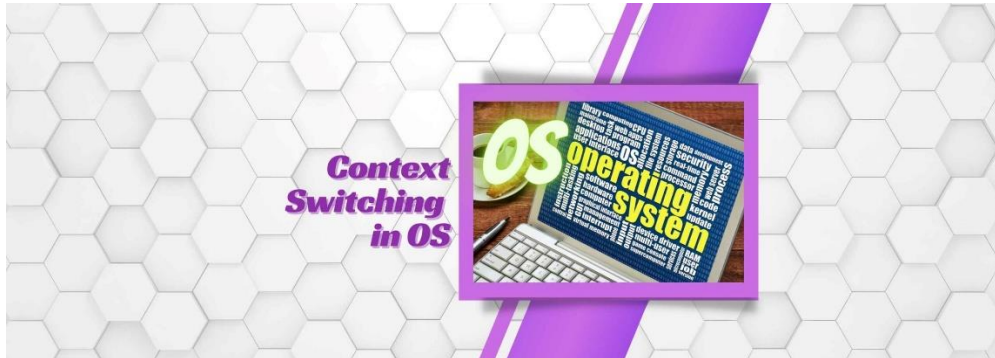


Enhanced DOS Features: Microsoft progressively added features to DOS, such as memory swapping to disk, that indirectly benefited multitasking by providing more flexible memory management.



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.

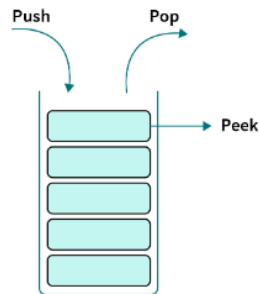


Resource Sharing and Thread Independence:

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.

KEY ADVANCEMENTS FOR MULTITHREADED PROGRAMMING:

1. Deserialized Message Queue: Unleashing True Responsiveness

No Single-Program Dominance: In contrast to serialized systems where a single program's lengthy task could stall input for all others, deserialized message queues grant each program independent processing, ensuring responsiveness even when one encounters a time-consuming operation.

Smooth User Experience: Users encounter a normal mouse cursor over responsive windows, allowing them to seamlessly switch to other programs without hindrance. This fosters a fluid and efficient workflow.

Foundation for Enhanced Multitasking: Deserializer message queues serve as a cornerstone for modern multitasking environments, enabling users to interact with multiple programs without frustrating delays.

2. Independent Message Queues for Threads: Streamlining Thread Management

Eliminating PM's Complexities: Windows NT and 98 (and later versions) dispense with the cumbersome rules regarding message-queue and non-message-queue threads that were prevalent in PM.

Simplified Communication and Coordination: Each thread possessing its own message queue streamlines communication and coordination between threads, promoting efficient collaboration and data exchange.

3. Thread Termination Function: Gaining Control

Precise Lifecycle Management: The ability to terminate a thread within the same process empowers developers with granular control over thread lifetimes and resource allocation.

Graceful Handling of Blocking Scenarios: Blocked threads can be terminated to prevent deadlocks or resource exhaustion, fostering more resilient and responsive applications.

4. Thread Local Storage (TLS): Encapsulating Thread-Specific Data

Unique Data for Shared Functions: TLS empowers threads to maintain their own unique static variables within a function shared by multiple threads, fostering encapsulation and data isolation.

C Compiler Extensions: Microsoft's C compiler enhancements make TLS usage intuitive for developers, promoting its seamless integration into multithreaded code.

5. Recommended Thread Architecture: Promoting Responsiveness and Efficiency

Separation of Concerns: The alignment with best practices, advocating for a primary thread dedicated to user interface and input handling, and secondary threads responsible for background tasks, ensures a responsive user experience and efficient resource utilization.

Windows' advancements in thread handling have revolutionized multithreaded programming, fostering more responsive, flexible, and controllable applications that deliver superior user experiences and effectively leverage system resources.

THE "NEW! IMPROVED! NOW WITH THREADS!" FALLACY

You've perfectly captured the hype and pitfalls surrounding multithreading with your "New! Improved! Now with Threads!" headline. It's crucial to avoid blindly adopting buzzwords like "Whatsit" or throwing threads into a program just because they exist.

When Threads Shine:

Responsive UI: Programs prone to hourglass freezes or relying on hacks like PeekMessage are prime candidates for multithreading. Offloading lengthy tasks to separate threads keeps the UI responsive and the user experience smooth.

Efficient Resource Utilization: For applications performing multiple heavy computations simultaneously, threads can maximize CPU and memory usage, leading to performance boosts.

When Threads Don't Make Sense:

Simple and Short Tasks: If an operation takes less than 100 milliseconds, the overhead of thread creation and context switching might outweigh any perceived benefit. The hourglass, in such cases, can be an honest reflection of processing.

User Expectations: Instantaneously responding to user actions like file opening might be more important than multithreading overhead. Prioritize user expectations and avoid unnecessary complexity.

Key Takeaways:

- ✓ **Analyze Need:** Carefully evaluate if your application genuinely needs multithreading to improve performance or user experience. Don't succumb to "feature fads."
- ✓ **Weigh Costs and Benefits:** Consider the overhead of thread creation, context switching, and potential synchronization complexities against the expected gains.
- ✓ **Simplicity Matters:** Don't overcomplicate code with unnecessary threads. Sometimes, the simplest approach might be the most efficient and reliable.

The 1/10-Second Rule:

Use it as a guideline, not a hard rule. Short-lived tasks, even if exceeding 1/10th of a second, might not benefit from threading due to overhead. Prioritize context and user expectations.

Remember, multithreading is a powerful tool, but like any tool, it's best used with purpose and discretion. By **carefully considering when and how to implement threads**, you can create truly efficient and responsive applications that enhance the user experience without the overhead of unnecessary complexity.

WINDOWS MULTITHREADING: EMPOWERING RESPONSIVE AND EFFICIENT APPLICATIONS

Windows provides robust mechanisms for creating and managing multiple threads of execution within a single process, enabling applications to achieve enhanced responsiveness, improved resource utilization, and parallel execution of tasks.

Creating Threads:

1. API Approach:

The API approach to creating threads involves using the `CreateThread` function, which offers granular control over thread creation. Here are the key parameters of the `CreateThread` function:

```
HANDLE hThread = CreateThread(  
    // Security attributes (optional)  
    &security_attributes,  
    // Initial stack size (0 for default)  
    dwStackSize,  
    // Thread function  
    ThreadProc,  
    // Argument to ThreadProc  
    pParam,  
    // Flags (0 for immediate execution, CREATE_SUSPENDED for delayed)  
    dwFlags,  
    // Pointer to receive thread ID  
    &idThread  
);
```

- **SECURITY_ATTRIBUTES (optional):** This parameter allows you to specify security settings for the thread. It is often set to `NULL` when not needed.
- **dwStackSize:** This parameter specifies the initial stack size for the thread. It defaults to 0, which allows the system to allocate a suitable stack size.
- **ThreadProc:** This parameter is a pointer to the thread function. The thread function is the code that will be executed by the thread when it starts.
- **pParam:** This parameter allows you to pass an argument to the thread function. It can be a pointer to any data you want to pass to the thread.
- **dwFlags:** This parameter is usually set to 0 for normal thread execution. However, you can use the `CREATE_SUSPENDED` flag to create a thread in a suspended state, allowing for delayed execution.
- **&idThread:** This parameter is a pointer to a variable that will store the thread ID. The thread ID uniquely identifies the thread and can be used to manipulate or obtain information about the thread.

2. C Runtime Library Preference:

Many programmers prefer using the C Runtime Library approach for creating threads, as it provides a simpler interface. The `_beginthread` function from the `PROCESS.H` header is commonly used. Here are the key parameters of the `_beginthread` function:

```
#include <process.h> // Include the necessary header file

// Thread function declaration (using C Runtime Library syntax)
void __cdecl ThreadProc(void *pParam) {
    // Thread code here
}

int main() {
    // Create a thread using _beginthread
    HANDLE hThread = _beginthread(ThreadProc, 0, (void *)42); // Example argument

    // ... rest of the main thread code
}
```

- **ThreadProc:** This parameter is a pointer to the thread function, similar to the API approach.
- **uiStackSize:** This parameter specifies the initial stack size for the thread. It defaults to 0, allowing the system to allocate a suitable stack size.
- **pParam:** This parameter allows you to pass an argument to the thread function, similar to the API approach.
- **#include <process.h>:** This line includes the header file that provides the `_beginthread` function.
- **ThreadProc function:** This function has the same structure as in the API approach, but it uses the `__cdecl` calling convention and takes a `void *` argument.
- **_beginthread call:**
 - ❖ The first argument is the pointer to the thread function (`ThreadProc`).
 - ❖ The second argument specifies the initial stack size (0 for default).
 - ❖ The third argument is the argument passed to the thread function (`((void *)42` in this example).
 - ❖ The function returns a handle to the newly created thread.

Key Points:

- The `_beginthread` function simplifies thread creation compared to `CreateThread`.
- It handles thread initialization and cleanup automatically, reducing potential errors.
- The thread function syntax is consistent with other C Runtime Library functions.

Thread Function Syntax:

1. API Approach:

The thread function used in the API approach has the following syntax:

```
DWORD WINAPI ThreadProc(PVOID pParam) {  
    // Thread code here  
    return 0; // Exit code  
}
```

- **DWORD WINAPI:** This declaration specifies that the function returns a DWORD value (usually an exit code) and uses the WINAPI calling convention.
- **ThreadProc:** This is the name of the thread function, which can be customized as needed.
- **PVOID pParam:** The function takes a single argument of type PVOID (pointer to void), which allows for passing arbitrary data to the thread.
- **Thread code:** The body of the function contains the code that the thread will execute.
- **return 0:** The function typically returns 0 to indicate successful completion. You can return other values to signal different exit states.

Key Considerations:

- **Thread Safety:** Design code with thread safety in mind to prevent data corruption and unexpected behavior.
- **Synchronization:** Employ synchronization techniques like mutexes, semaphores, or critical sections to coordinate thread access to shared resources.
- **Communication:** Establish mechanisms for data exchange and coordination between threads, such as message queues or shared memory.
- **Thread Pooling:** Consider thread pools for managing and reusing threads for performance optimization.
- **Thread Cancellation:** Use `TerminateThread` cautiously due to potential resource leaks; prefer cooperative cancellation mechanisms when possible.
- **Asynchronous Operations:** Explore asynchronous I/O and overlapped structures for non-blocking operations that enhance responsiveness.

Additional Insights:

- Windows dynamically adjusts stack size as needed, mitigating initial allocation concerns.
- Suspended threads can be resumed using `ResumeThread` for delayed execution.
- Carefully manage thread priorities to balance responsiveness and resource allocation.

RNDRCTMT.C

```
1  #include <windows.h>
2  #include <process.h>
3
4  LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
5  HWND hwnd;
6  int cxClient, cyClient;
7
8  int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, PSTR szCmdLine, int iCmdShow)
9  {
10     static TCHAR szAppName[] = TEXT("RndRctMnt");
11     MSG msg;
12     WNDCLASS wndclass;
13
14     wndclass.style = CS_HREDRAW | CS_VREDRAW;
15     wndclass.lpfnWndProc = WndProc;
16     wndclass.cbClsExtra = 0;
17     wndclass.cbWndExtra = 0;
18     wndclass.hInstance = hInstance;
19     wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);
20     wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);
21     wndclass.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
22     wndclass.lpszMenuName = NULL;
23     wndclass.lpszClassName = szAppName;
24
25     if (!RegisterClass(&wndclass))
26     {
27         MessageBox(NULL, TEXT("This program requires Windows NT!"), szAppName, MB_ICONERROR);
28         return 0;
29     }
30
31     hwnd = CreateWindow(szAppName, TEXT("Random Rectangles"), WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT, NULL, NULL, hInstance, NULL);
32     ShowWindow(hwnd, iCmdShow);
33     UpdateWindow(hwnd);
34
35     while (GetMessage(&msg, NULL, 0, 0))
36     {
37         TranslateMessage(&msg);
38         DispatchMessage(&msg);
39     }
40
41     return msg.wParam;
42 }
43
44 VOID Thread(PVOID pvoid)
45 {
46     HBRUSH hBrush;
47     HDC hdc;
48     int xLeft, xRight, yTop, yBottom, iRed, iGreen, iBlue;
49
50     while (TRUE)
51     {
52         if (cxClient != 0 || cyClient != 0)
53         {
54             xLeft = rand() % cxClient;
55             xRight = rand() % cxClient;
56             yTop = rand() % cyClient;
57             yBottom = rand() % cyClient;
58             iRed = rand() & 255;
59             iGreen = rand() & 255;
60             iBlue = rand() & 255;
61
62             hdc = GetDC(hwnd);
63             hBrush = CreateSolidBrush(RGB(iRed, iGreen, iBlue));
64             SelectObject(hdc, hBrush);
65             Rectangle(hdc, min(xLeft, xRight), min(yTop, yBottom), max(xLeft, xRight), max(yTop, yBottom));
66             ReleaseDC(hwnd, hdc);
67             DeleteObject(hBrush);
68         }
69     }
70 }
71
72 LRESULT CALLBACK WndProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
73 {
74     switch (message)
75     {
76     case WM_CREATE:
77         _beginthread(Thread, 0, NULL);
78         return 0;
79     case WM_SIZE:
80         cxClient = LOWORD(lParam);
81         cyClient = HIWORD(lParam);
82         return 0;
83     case WM_DESTROY:
84         PostQuitMessage(0);
85         return 0;
86     }
87     return DefWindowProc(hwnd, message, wParam, lParam);
88 }
```

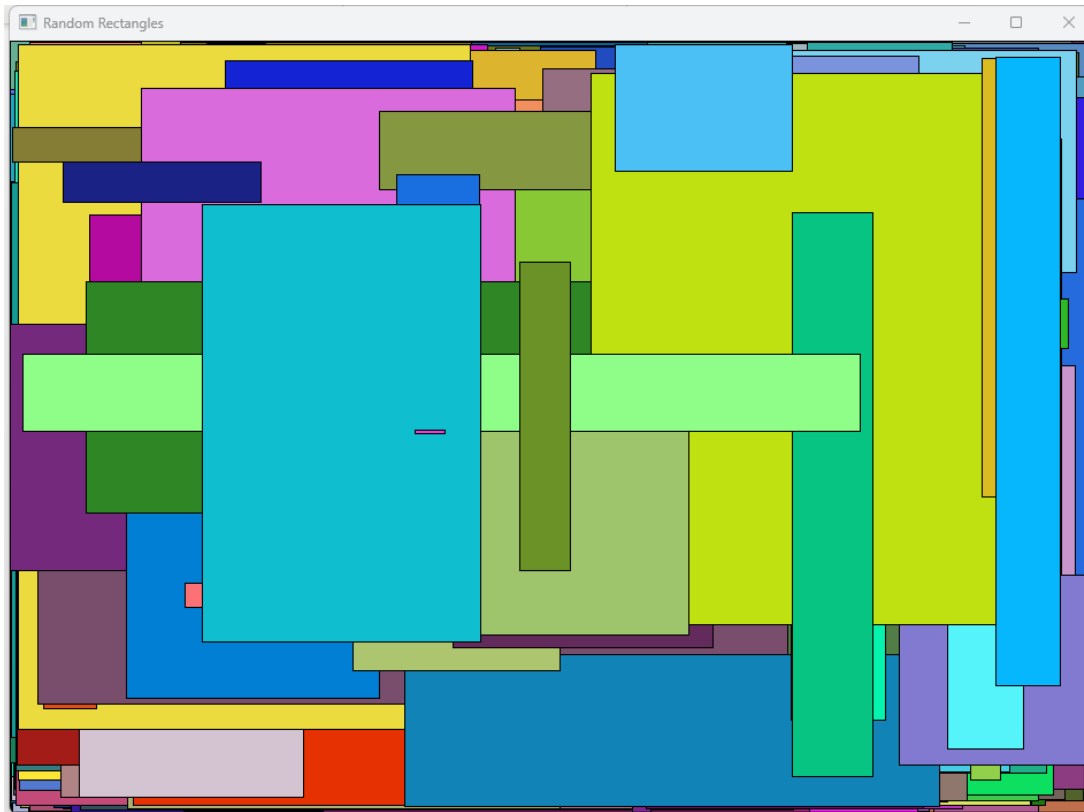
The [provided C code](#) represents a [multithreaded Windows application](#) that generates and displays random rectangles using the WinAPI.

- The application [employs a main window](#), and its functionality is divided into a window procedure (WndProc) and a secondary thread (Thread) responsible for rendering the random rectangles.
- In the [WinMain function](#), the program initializes the window class and creates the main window. The window class is registered with the necessary attributes, and the main window is created with the title "Random Rectangles." The window's size and position are set, and the program enters the message loop to handle user interactions.
- The [Thread function](#), executed as a separate thread, continuously generates random coordinates and colors for rectangles within the client area of the main window.
- It [uses the rand\(\) function](#) to obtain random values for the left, right, top, and bottom coordinates, as well as random RGB values for the rectangle's color. The GDI functions are then employed to draw the rectangle on the window's device context.
- The [WndProc function handles messages](#) for the main window. Upon receiving the WM_CREATE message, it initiates the secondary thread (Thread) using `_beginthread`.
- The [WM_SIZE message updates the dimensions of the client area](#), allowing the secondary thread to generate random rectangles within the updated size. The WM_DESTROY message posts a quit message to terminate the application.

This code showcases a [simple yet effective integration of multithreading in a WinAPI application](#), with one thread dedicated to window management and the other responsible for rendering random rectangles concurrently. The multithreaded approach enhances responsiveness and user experience, especially in scenarios where continuous background tasks need to run independently.

Program Behavior in short:

- The main thread creates the window and enters the message loop.
- When the window is created, the Thread function starts running in a separate thread.
- The Thread function continuously generates and draws random rectangles.
- The main thread handles window messages, including resizing the window and terminating the program.
- The Thread function runs independently until the program terminates.



Key Steps:

Compiler Configuration:

Select the [Multithreaded run-time library](#) in the Project Settings dialog box.

This sets the [/MT compiler flag](#), instructing the compiler to link with LIBCMT.LIB for thread-safe C library functions.

Thread-Safe Functions:

The [LIBCMT.LIB library](#) provides thread-safe versions of C library functions, ensuring proper behavior in multithreaded environments.

Functions like strtok that maintain static data have separate instances for each thread to avoid conflicts.

Thread Creation:

The `_beginthread` function, declared in `PROCESS.H`, is used to create a new thread of execution.

The `_MT` identifier (defined by the `/MT` flag) enables this function's declaration.

Thread Function:

The `Thread` function serves as the entry point for the new thread.

It receives a `VOID*` argument (unused in this example) for potential data passing.

It runs concurrently with the main thread, executing its code independently.

Thread Communication:

`Global variables` like `hwnd`, `cxClient`, and `cyClient` are shared between threads for communication.

The `main thread` updates `cxClient` and `cyClient` in the `WM_SIZE` handler, and the `Thread` function uses them for drawing.

Automatic vs. Static Variables:

`Automatic variables` (local to functions) are unique to each thread.

`Static variables` (declared with `static`) are shared across all threads, enabling communication but requiring caution for synchronization.

Thread-Local Storage (TLS):

Windows 98 introduced `TLS for storing persistent data` unique to each thread.

It's an `alternative to static variables` when thread-specific data is needed without potential conflicts.

Potential Issues and Improvements:

- **Synchronization:** While not explicitly implemented in this code, synchronization mechanisms like mutexes or critical sections are crucial in more complex scenarios to prevent race conditions and ensure consistent data access when multiple threads modify shared variables.
- **Error Handling:** Implement proper error handling for thread creation and termination to ensure program robustness.
- **Resource Management:** Ensure proper release of resources like device contexts and brushes to avoid memory leaks.
- **Thread Pooling:** Explore thread pools for efficient management and reuse of threads, especially when creating and destroying threads frequently.
- **Consider TLS:** If you need persistent data unique to each thread, evaluate using TLS instead of static variables to avoid potential conflicts.

THE PROGRAMMING CONTEST CHALLENGE: A DEEPER DIVE

This passage explores a [fascinating programming contest from 1986](#), providing a glimpse into the early days of multitasking and offering valuable insights into how technology and programming practices have evolved. Here's a deeper look at the key elements:



The Challenge:

Multitasking Simulation: Create a program simulating four windows displaying distinct tasks:

- **Window 1:** Increasing numbers (sequential)
- **Window 2:** Increasing prime numbers
- **Window 3:** Fibonacci sequence
- **Window 4:** Randomly sized circles

Multitasking Environment: While still a novelty in 1986, the problem demanded consideration of parallel execution and efficient resource utilization.

Time Constraint: Designed for completion within 30 minutes, it emphasized swift programming and efficient solutions.

Contestant Approaches:

DOS-based Solutions:

Screen Division: Most used a loop to sequentially update each window, emulating multitasking within a single screen.

High CPU Usage: Programs consumed 100% CPU time due to the single-threaded model.

Windows 1.0 Perspective: Though not explicitly addressed, the passage provides a hypothetical scenario of how the program might have looked adapted for the nascent Windows environment.

MULTI Program: The included sample code showcases a more robust solution leveraging Windows' multitasking capabilities.

Insights and Evolution:

Technological Leap: The contest highlights the significant progress made in multitasking technology since the late 1980s.

Shifting Paradigms: From single-threaded loops to true multitasking, the challenge demonstrates the evolution of programming approaches.

Abstraction and Efficiency: Modern operating systems and programming languages now provide powerful tools for efficient and elegant multitasking solutions.

Additional Aspects to Consider:

- **Concurrency Mechanisms:** Different approaches to implementing true multitasking in the hypothetical Windows version, such as threads or processes.
- **Resource Management:** Strategies for balancing CPU usage and data sharing between threads to optimize performance.
- **User Interface Design:** Enhancing the visual representation of the four tasks and user interaction in a Windows-based environment.

Overall, this passage offers a thought-provoking reminder of the rapid advancement in multitasking technology and programming practices. It serves as a valuable case study for understanding the challenges and potential of creating concurrent applications, making it relevant not only for historical context but also for ongoing advancements in software development.

MULTI1.C PROGRAM

While the code doesn't explicitly create separate threads, it leverages Windows' message-driven architecture and timers to achieve a simulated multitasking effect. Here's a breakdown of how it works:

1. Single-Threaded Main Loop:

The **WinMain function** initializes the main window and enters a message loop using `GetMessage`, `TranslateMessage`, and `DispatchMessage`.

This loop handles events like window resizing, timer notifications, and user input.

2. Child Windows for Tasks:

The **WndProc function** creates four child windows, each with its own window procedure (`WndProc1`, `WndProc2`, `WndProc3`, `WndProc4`).

Each child window is responsible for displaying its specific content (numbers, primes, Fibonacci sequence, or circles).

3. Timer for Simulated Concurrency:

A timer is set using `SetTimer` with a 10-millisecond interval.

When the timer fires, it sends a `WM_TIMER` message to the main window.

The main window iterates through the child windows and sends a `WM_TIMER` message to each, effectively "waking them up" to perform their tasks.

4. Asynchronous Updates:

Each child window's `WndProc` handles the `WM_TIMER` message by updating its content asynchronously.

This means that the main thread doesn't wait for each window to finish updating before moving on to the next one.

This creates the illusion of simultaneous execution, even though there's only a single thread.

Key Points:

- **No Explicit Threads:** The code doesn't create separate threads using mechanisms like `_beginthread` or `CreateThread`.
- **Message-Driven Multitasking:** It simulates multitasking within the constraints of the single-threaded message loop.
- **Timer for Triggering Updates:** The timer acts as a regular "tick" to signal the child windows to perform their tasks.
- **Asynchronous Window Procedures:** Each child window's window procedure handles its updates independently, contributing to the multitasking appearance.
- **Remember:** This approach is a simulation of multitasking, not true multithreading. In a multithreaded environment, multiple threads would genuinely execute concurrently, potentially offering performance benefits and a more responsive user experience.

Key Points:

Single-Threaded Simulation: The program creates a multitasking-like effect using a single thread and timers, not true multithreading.

Child Window Structure: The main window creates four child windows, each responsible for its own content.

Timer-Driven Updates: A timer triggers periodic updates to the child windows, simulating concurrent execution.

Inefficient Prime Number Check: The prime number generator in WndProc2 could be optimized for better performance.

Potential Performance Constraints: The timer-based approach might limit performance on faster machines.

Additional Considerations:

WM_PAINT Handling: The program doesn't store window content for WM_PAINT messages, which might lead to flickering or artifacts if redrawing is delayed.

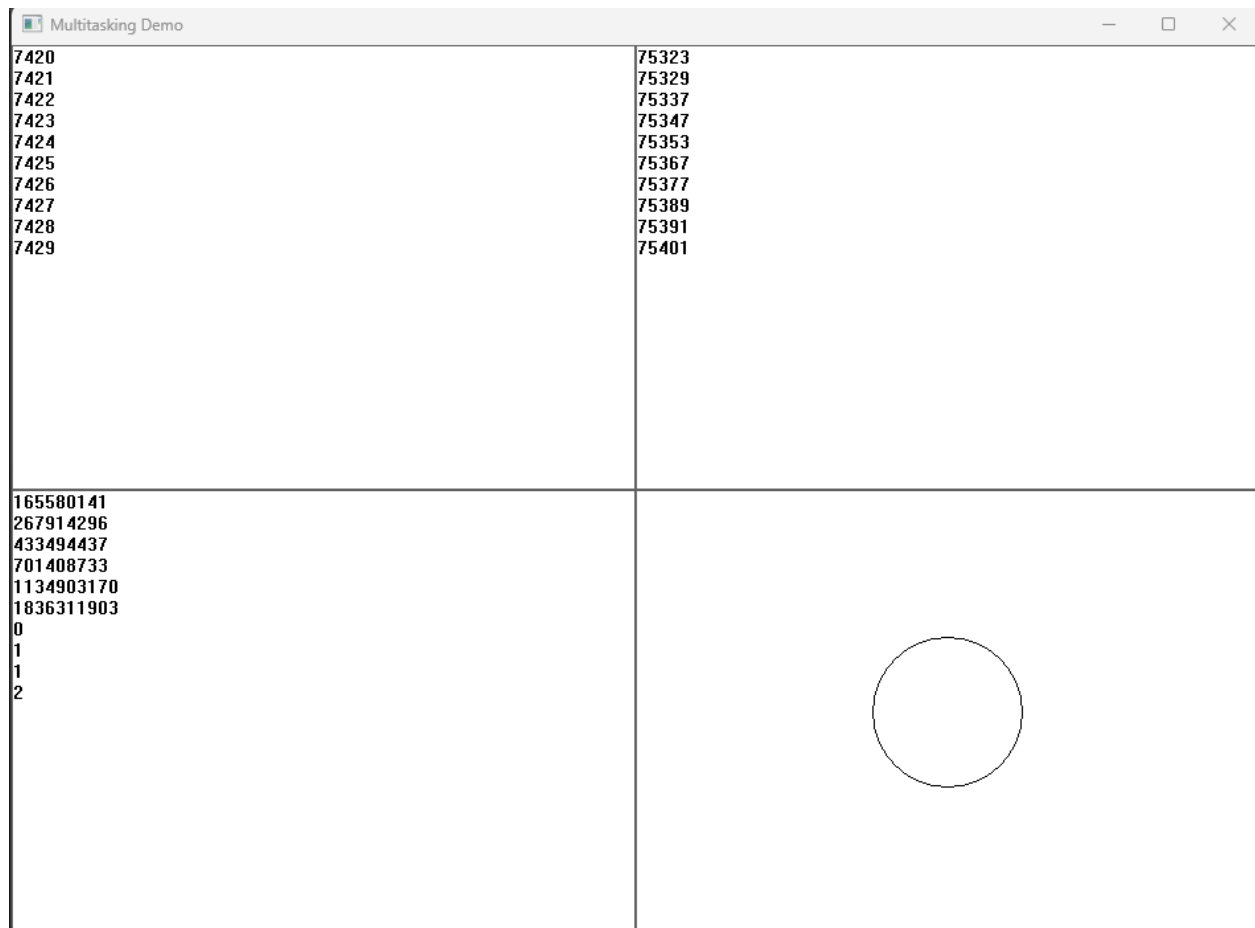
Hardware Dependence: Tuning updates based on machine speed can be challenging and might lead to compatibility issues across different systems.

Alternative Approaches:

True Multithreading: Using separate threads for each window's tasks could offer genuine concurrency and potentially better performance on multiprocessor systems.

Dynamic Updates: Allowing windows to update independently based on their own needs, rather than relying solely on a timer, could improve responsiveness and resource utilization.

Message Queues: Employing message queues for communication between windows could enhance flexibility and decoupling.



All the 4 subwindows above do different things.

In Conclusion:

While the [MULTI1 program](#) demonstrates a viable approach to simulating multitasking within a single-threaded Windows application, it's essential to consider its limitations and potential performance bottlenecks. For more demanding scenarios, exploring true multithreading or alternative strategies for handling window updates might be necessary to achieve optimal performance and responsiveness.

KEY DIFFERENCES BETWEEN MULTI2 AND MULTI1:

Multithreading Paradigm:

MULTI1 simulates multitasking within a single thread using timers and asynchronous window procedures.

MULTI2 embraces true multithreading, creating separate threads for each child window's tasks, enabling genuine concurrent execution and potentially better performance.

Update Mechanism:

MULTI1 relies on a timer to periodically trigger updates in child windows.

MULTI2 removes this timer dependency. Threads autonomously handle updates, potentially enhancing responsiveness and avoiding timer-related constraints.

Code Structure:

MULTI2's main window code is simplified as it no longer manages timers or sends WM_TIMER messages.

Thread functions encapsulate each child window's logic, promoting better code organization and maintainability.

Thread Communication:

MULTI2 introduces a PARAMS structure to facilitate communication between threads, containing window handles, dimensions, and a termination flag.

Additional Insights:

Performance Implications:

True multithreading in MULTI2 can often lead to better performance, especially on multiprocessor systems, as tasks can execute concurrently.

However, effective thread management and synchronization are crucial to avoid potential bottlenecks or race conditions.

Synchronization Considerations:

While MULTI2 doesn't demonstrate explicit synchronization mechanisms, more complex multithreaded scenarios often require them to ensure data consistency and prevent race conditions.

Error Handling:

Robust multithreaded applications should incorporate error handling within threads to gracefully manage unexpected issues and prevent application crashes.

Thread Priority:

MULTI2 doesn't explore thread priority settings, but adjusting priority levels can be useful in certain scenarios to prioritize critical tasks or prevent resource contention.

Design Trade-offs:

True multithreading can introduce complexity in terms of thread management, synchronization, and potential for race conditions.

Developers need to carefully assess the benefits and challenges of multithreading in the context of specific application requirements.

Alternative Approaches:

Beyond timers and multithreading, other techniques for handling concurrent tasks within Windows applications exist, such as message queues or asynchronous I/O, each with its own advantages and considerations.



Race Conditions:

Window Destruction: The bKill flag might not prevent a thread from attempting to access a destroyed window's device context if thread switching occurs at inopportune moments. While Windows 98 might handle this gracefully, it's not ideal to rely on such behavior.

Concurrent Drawing: Multiple threads drawing on the same window without synchronization could potentially lead to visual artifacts or inconsistencies. However, Windows 98 seems to serialize graphics calls, mitigating this issue.

GDI Object Handling: Windows 98 doesn't serialize access to GDI objects, so threads could inadvertently destroy objects being used by others. This requires careful synchronization or avoidance of object sharing.

Synchronization Needs:

Critical Sections: While MULTI2 doesn't use them, critical sections (using EnterCriticalSection and LeaveCriticalSection) are essential tools for protecting shared data and resources in multithreaded environments. They ensure that only one thread can access a critical section at a time, preventing race conditions.

Best Practices:

- Use critical sections to protect shared data structures and resources.
- Avoid sharing GDI objects between threads whenever possible.
- If GDI object sharing is unavoidable, implement proper synchronization mechanisms.

Key Takeaways:

- **Multithreaded programming** introduces complexity and potential for race conditions.
- **Careful synchronization** is crucial for ensuring data consistency and preventing unexpected behavior.
- Understanding operating system behavior and **API thread safety** is essential for robust multithreaded applications.
- Even when an operating system handles certain situations gracefully, it's best to **implement explicit synchronization for clarity, maintainability, and portability** across different environments.

UNDERSTANDING THE SLEEP FUNCTION'S ROLE IN MULTITHREADED WINDOWS PROGRAMMING:

Purposeful Pauses: The Sleep function serves as a strategic tool for temporarily halting a specific thread's execution within a multithreaded Windows application. This intentional pause enables developers to control the timing of tasks, create delays, and even implement basic animation effects, often within secondary threads.

Mechanism and Usage:

Voluntary Suspension: When a thread calls Sleep, it willingly suspends its own progress for a designated time interval specified in milliseconds.

Resource Management: During this sleep period, the thread relinquishes its processor time slices, refraining from consuming significant resources. However, it still necessitates minimal system attention for periodic checks to determine when to resume execution.

Yielding Time Slices: Passing an argument of 0 to Sleep causes the thread to forfeit any remaining time within its current time slice, allowing other threads to potentially gain priority.

Impact on Thread Concurrency:

Selective Suspension: Crucially, Sleep only affects the thread that directly invokes it. Other threads within the process, as well as those residing in separate processes, continue to operate undisturbed, maintaining overall system responsiveness.

Best Practices and Considerations:

Ideal for Secondary Threads:

Animation Without Windows: In scenarios where secondary threads lack windows, Sleep emerges as a viable technique for implementing simple animation sequences.

Pacing Background Tasks: It's often employed to regulate the pace of background tasks or lengthier jobs, ensuring they don't monopolize system resources.

Caution in Primary Threads:

Potential Responsiveness Issues: While Sleep can be utilized within primary threads, it's generally discouraged due to the risk of hindering message processing for windows, potentially compromising application responsiveness.

Alternatives for Primary Threads: In primary threads, timers or asynchronous operations often provide more suitable approaches for timing-related tasks.

Beyond Basic Timing:

- **Synchronization for Complex Scenarios:** While Sleep proves valuable for fundamental timing needs, more intricate multithreaded scenarios frequently necessitate sophisticated synchronization mechanisms.
- **Exploring Alternatives:** Developers should carefully consider alternatives like timers, events, and manual wait states using synchronization objects, depending on the specific requirements of their multithreaded applications and the interactions between threads.

In essence, **Sleep offers a fundamental yet valuable tool for managing thread execution** within multithreaded Windows programs. By understanding its purpose, usage, impact on concurrency, and best practices, developers can effectively leverage it to achieve desired timing behaviors and resource management, while ensuring overall application responsiveness.

RUSH HOUR AT THE MEMORY BANK: NAVIGATING THE CROSSROADS OF THREADS WITH TRAFFIC LIGHTS

Imagine [bustling Times Square during rush hour](#), but instead of pedestrians, it's a throng of cars, all rushing to access the same bank at the intersection. This, my friend, is the world of multithreaded programming, where multiple threads, like those impatient drivers, want to access shared resources (think bank accounts) at the same time.

[Without any order, it's pure chaos.](#) Cars collide, accounts get mixed up, and the whole system grinds to a halt. This is exactly what happens in a multithreaded program without synchronization. Threads trying to update the same information simultaneously can lead to corrupted data, unexpected crashes, and buggy behavior.

Enter the [critical section, the traffic light of the multithreaded world](#). It's a designated zone – a specific block of code – where only one car (thread) can enter at a time. Just like a crossing with flashing yellow arrows, the critical section uses synchronization mechanisms, like traffic cops or stop signs, to act as gatekeepers.

Here's how it works:

- **Car Approaching:** A car (thread) arrives at the critical section, like reaching the bank entrance.
- **Requesting Entry:** The driver (thread) signals the cop (synchronization mechanism), like waiting in line.
- **Checking the Line:** The cop (mechanism) checks if the bank counter is occupied, like looking for an open lane.
- **Granting Access:** If it's free, the car (thread) gets the green light and enters the critical section, like stepping into the bank.
- **Exclusive Zone:** While inside, the car (thread) has exclusive access to the counter (shared data), like safely withdrawing or depositing money.
- **Exiting Gracefully:** Once done, the driver (thread) leaves the bank (critical section), signaling the cop (mechanism) the lane is free.

This controlled access prevents race conditions, those moments where threads compete for the same data, potentially leading to errors. Imagine two cars trying to reach the same teller at once – chaos ensues!

By [strategically placing traffic lights \(critical sections\)](#) in your multithreaded program, you ensure smooth traffic flow, data integrity, and ultimately, a well-functioning system. You've transformed the frantic rush hour into a coordinated queue, where each thread gets its turn without the risk of a multi-car pileup.

Just like cars at an intersection, **threads are independent paths of execution** within a program. They can access shared resources like variables, files, or data structures. Without proper synchronization, race conditions can occur, leading to unpredictable and potentially disastrous outcomes.

Race conditions are like cars approaching the intersection simultaneously without any traffic rule in place. Both threads might try to modify the same resource at the same time, resulting in incorrect or corrupted data.

Synchronization mechanisms act as the traffic lights, ensuring controlled access to critical sections and preventing collisions. Here are some common approaches:



- **Mutual exclusion:** This guarantees that only one thread can access a critical section at a time. Imagine a stop sign or a single traffic light at the intersection, allowing one car to proceed at a time.
- **Semaphores:** These act like traffic lights with multiple lanes. A semaphore variable tracks the number of resources available (green lights) and the number of threads waiting (red lights).
- **Mutex with condition variables:** This allows threads to wait for specific conditions before entering a critical section. Think of it as a traffic light combined with a sensor that detects approaching cars, ensuring there's no waiting when the intersection is clear.

Choosing the right synchronization mechanism depends on various factors like the nature of the shared resources, the expected traffic (thread concurrency), and the desired behavior.

The analogy also highlights the **importance of reliable synchronization mechanisms**. Just like malfunctioning traffic lights can lead to chaos, bugs in synchronization code can cause serious problems in your program.

Ultimately, **thread synchronization is about establishing order and ensuring predictable behavior** in your multithreaded program. By employing the right strategies and understanding the potential dangers, you can create programs that run smoothly and avoid the traffic jams of data collisions.

CRITICAL SECTIONS: PROTECTING DATA INTEGRITY IN MULTITHREADED ENVIRONMENTS

In multithreaded operating systems, multiple threads of execution can run concurrently within a single program, sharing resources and working towards shared goals. However, this concurrency introduces challenges when multiple threads attempt to access and modify shared data simultaneously. To prevent data corruption and ensure program stability, we introduce the concept of critical sections.

Understanding the Need for Coordination:

- **Single-Tasking Simplicity:** In single-tasking environments, programs operate without interference, eliminating coordination concerns.
- **Multitasking Challenges:** Even in multitasking systems, issues can arise when multiple programs access shared resources like files. Operating systems offer mechanisms like shared file access and record locking to address these challenges.
- **Multithreading Complexities:** Multithreading within a single program significantly increases coordination needs. Threads often share data, and modifications by one thread might conflict with operations by others, potentially leading to data inconsistencies and program crashes.

Critical Sections: Enforcing Exclusive Access:

- **Definition:** A critical section is a block of code designed to execute atomically, meaning it must complete without interruption from other threads. This ensures data integrity and prevents race conditions.
- **Mechanics:** Synchronization mechanisms, analogous to traffic lights, govern access to critical sections. They ensure only one thread enters a critical section at a time, preventing conflicts and maintaining data consistency.

Common Synchronization Mechanisms:

- **Mutual Exclusion (Mutexes):** These act as binary locks, allowing only one thread to hold the lock and enter the critical section at a time.
- **Semaphores:** Semaphores generalize mutexes, enabling control over multiple resources or resource pools.
- **Monitors:** High-level constructs that encapsulate shared data and provide built-in synchronization mechanisms.

Importance of Correct Implementation:

- **Reliable Synchronization:** Just as malfunctioning traffic lights lead to chaos, bugs in synchronization code can cause serious program errors.
- **Careful Design and Testing:** Proper design and testing of critical sections and synchronization mechanisms are crucial for ensuring multithreaded program correctness and reliability.

In essence, critical sections act as safeguards for multithreaded programs, preventing data corruption and ensuring predictable behavior by enforcing exclusive access to shared data when necessary. By understanding their purpose and the associated synchronization mechanisms, developers can build robust and efficient multithreaded applications.

UNDERSTANDING CRITICAL SECTIONS IN-DEPTH:

Critical sections are synchronization mechanisms that enforce exclusive access to shared resources or code blocks by multiple threads within a process. This prevents race conditions and ensures data consistency.

Key Functions:

1. Initializing a Critical Section:

```
#include <windows.h>

CRITICAL_SECTION cs; // Declare a global critical section object

// Initialize the critical section before use:
InitializeCriticalSection(&cs);
```

2. Entering a Critical Section:

```
EnterCriticalSection(&cs); // Acquire ownership of the critical section

// Code that accesses shared resources or executes critical code

LeaveCriticalSection(&cs); // Release ownership
```

3. Deleting a Critical Section:

```
// When no longer needed:
DeleteCriticalSection(&cs); // Free up associated resources
```

Important Considerations:

Opaque Structure: The CRITICAL_SECTION structure is internal to Windows, so don't modify its fields directly.

Ownership Semantics: Only one thread can "own" a critical section at a time. Other threads attempting to enter will be blocked until the owner releases it.

Blocking Behavior: EnterCriticalSection blocks the calling thread if another thread already owns the critical section.

Resource Management: Always initialize critical sections before use and delete them when no longer needed to release system resources.

Additional Notes:

- **Initialization:** Initialize critical sections only once, typically during program startup.
- **Error Handling:** Check the return values of these functions for potential errors.
- **Alternatives:** For more complex synchronization scenarios, consider mutexes, semaphores, or other synchronization primitives.

Best Practices:

- Use critical sections judiciously to protect specific critical sections of code or resources.
- Avoid overusing critical sections, as excessive synchronization can impact performance.
- Consider alternative synchronization mechanisms if critical sections don't meet specific requirements.

MUTUAL EXCLUSION AND CRITICAL SECTIONS:

Exclusive Access: Critical sections enforce mutual exclusion, meaning only one thread can execute within a critical section at a time. This prevents race conditions and ensures data consistency when multiple threads access shared resources or code.

Ownership Semantics: A thread "owns" a critical section when it enters it, blocking other threads attempting to enter until the owner exits.

Multiple Critical Sections:

Distinct Resources: You can define multiple critical sections to protect different shared resources or code blocks within a process. This allows for fine-grained control of thread synchronization.

Example: Four threads could use cs1 to synchronize access to one shared data structure and cs2 to synchronize access to a different structure.

Main Thread Considerations:

Potential Hangs: Be cautious when using critical sections in the main thread, as a secondary thread holding a critical section for too long could block the main thread's progress.

Optimized Access: Secondary threads might only need brief critical section access to copy shared data to local variables, minimizing blocking times.

Limitations and Mutexes:

Process Boundaries: Critical sections are limited to coordinating threads within a single process.

Cross-Process Synchronization: For coordinating threads across multiple processes, use mutex objects (mutual exclusion objects). These provide similar mutual exclusion guarantees but operate at the system level, allowing synchronization between threads in different processes.

Key Takeaways:

- Critical sections are essential tools for enforcing mutual exclusion in multithreaded Windows programs, ensuring data integrity and preventing race conditions.
- Understanding their ownership semantics, appropriate usage patterns, and limitations is crucial for effective multithreaded programming.
- For cross-process synchronization, mutex objects offer the necessary capabilities.

EVENT SIGNALING: ORCHESTRATING THREADS IN HARMONY

In the [symphony of multithreaded programming](#), event signaling serves as the conductor, coordinating threads' interactions and ensuring harmonious execution. It empowers threads to communicate significant occurrences, such as task completion or the need for coordination.

Common Scenarios for Event Signaling:

- [Task Completion Notification](#): A secondary thread signals its completion to the primary thread, allowing the latter to proceed with subsequent actions or initiate new tasks.
- [Data Availability Signaling](#): A producer thread signals to a consumer thread that data is ready for processing, optimizing data flow and resource utilization.
- [Error Signaling](#): Threads can signal error conditions to other threads, enabling error handling and corrective actions.
- [Synchronization Signaling](#): Threads can use events to signal when shared resources become available or to coordinate access to critical sections, preventing data races and ensuring integrity.

Windows Event Objects: A Signaling Mechanism

- [Creation and Management](#): Windows provides `CreateEvent` and `CloseHandle` functions to create and manage event objects.
- [Signaling and Waiting States](#): Events can exist in signaled or non-signaled states.
- [SetEvent](#): Signals the event, moving it to the signaled state.
- [WaitForSingleObject](#) or [WaitForMultipleObjects](#): Allow a thread to wait for an event to become signaled.

Additional Considerations:

- **Alternative Signaling Mechanisms:** Semaphores, mutexes, condition variables, and even simple flags or shared variables can also facilitate inter-thread signaling, each with its own advantages and use cases.
- **Synchronization and Data Integrity:** Always employ appropriate synchronization mechanisms when threads access shared data to safeguard data consistency and prevent race conditions.
- **Thread Termination Caution:** While the `TerminateThread` function forcibly terminates a thread, its use is generally discouraged due to potential resource leaks and unpredictable behavior. Prefer cooperative thread termination methods whenever possible.

Awaiting BIGJOB1 Code for Deeper Exploration

To delve into the specifics of event signaling within the BIGJOB1 program, please provide its code for analysis. This will enable a thorough examination of its thread creation, signaling mechanisms, and potential abortion handling strategies.



Thread Creation and Management:

`_beginthread`: The program employs `_beginthread` to spawn the secondary thread responsible for the lengthy calculation.

Thread Function: The `Thread` function encapsulates the calculation logic and performs the following:

Initializes a loop counter and a timer.

Executes the "savage" benchmark calculation `REP` times, checking the `bContinue` flag within the loop to allow for potential abortion.

Upon completion or abortion, sends a custom message (`WM_CALC_DONE` or `WM_CALC_ABORTED`) to the main window using `SendMessage`.

Terminates using `_endthread`.

Window Procedure:

- **Message Handling:** The WndProc function handles various events and messages:
- **Mouse Clicks:** Initiates or aborts the calculation based on left and right mouse button clicks, respectively.
- **Custom Messages:** Processes completion or abortion messages from the secondary thread, updating status and triggering a window repaint.
- **WM_PAINT:** Displays status and results (or instructions) in the window's client area.
- **Other Messages:** Passes unhandled messages to the default window procedure.

User Experience:

- **Status Messages:** Clear status messages guide the user: "Ready," "Working," and "Done" (with elapsed time).
- **Abortion Control:** The user can directly abort a running calculation with a right mouse click.
- **Error Beep:** A beep signals an attempt to start a calculation while one is already in progress.

Synchronization Considerations:

- **Potential for Race Conditions:** While not explicitly shown in this simplified example, multithreaded programs often require careful synchronization to prevent race conditions, especially when multiple threads access shared data.
- **Synchronization Mechanisms:** Windows provides tools like critical sections, mutexes, events, and semaphores for thread synchronization.

Thread Safety:

- **Volatile Keyword:** The volatile keyword is used for pparams->bContinue to ensure correct memory access and visibility between threads, as this variable might be modified by the main thread while being read by the secondary thread.



Additional Explorations:

- ✓ **Alternative Signaling Mechanisms:** Explore using other synchronization mechanisms like events or semaphores for thread communication.
- ✓ **Complex Calculations:** Investigate performance implications of using threads for more computationally intensive tasks.
- ✓ **Error Handling:** Consider incorporating error handling to gracefully handle potential thread issues or calculation errors.

BIGJOB1's primary purpose is to demonstrate multithreading concepts, specifically:

Offloading Lengthy Tasks to Secondary Threads: It showcases how computationally intensive tasks can be assigned to separate threads to avoid blocking the main thread, thus maintaining program responsiveness.

Thread Communication and Signaling: It illustrates how threads can communicate with each other using custom Windows messages to signal completion or abortion of tasks.

User-Initiated Thread Control: It allows users to initiate and abort the calculation using mouse clicks, demonstrating a basic level of user interaction with threads.

Performance Considerations: It highlights the potential performance benefits of multithreading, though the specific calculation used (the "savage" benchmark) is somewhat artificial and might not fully reflect real-world scenarios.

While not directly applicable to practical tasks, it serves as an educational tool for:

Understanding Thread Creation and Behavior: It provides a concrete example of thread creation, execution, and communication.

Exploring Thread Synchronization: It serves as a starting point for understanding synchronization challenges and techniques in multithreaded programming.

Experimenting with Thread Management: It allows for experimentation with different thread management strategies, such as abortion and signaling.

Creating an Event Object:

```
HANDLE hEvent = CreateEvent(NULL, /* Security attributes */
                           FALSE, /* Auto-reset event */
                           FALSE, /* Initially unsignaled */
                           NULL); /* No name */
```

Signaling an Event:

```
SetEvent(hEvent);
```

Waiting for an Event (with a timeout):

```
DWORD waitResult = WaitForSingleObject(hEvent, 5000); /* Wait for up to 5 seconds */
if (waitResult == WAIT_OBJECT_0) {
    // Event was signaled
} else if (waitResult == WAIT_TIMEOUT) {
    // Timeout occurred
} else {
    // Error handling
}
```

Resetting an Event (for manual-reset events):

```
ResetEvent(hEvent);
```

Example of Thread Coordination using an Event:

```
// Thread 1:
void ThreadFunction1() {
    // Perform some tasks...
    SetEvent(hEvent); // Signal completion
}

// Thread 2:
void ThreadFunction2() {
    WaitForSingleObject(hEvent, INFINITE); // Wait for Thread 1's signal
    // Proceed with tasks that depend on Thread 1's completion
}
```

Remember that these are simplified examples. Actual multithreaded applications often involve more complex synchronization patterns and error handling.

Performance Considerations:

Benchmark Context: The "savage" benchmark is somewhat artificial and might not fully reflect real-world performance gains achievable with multithreading.

Complex Calculations: Further exploration of multithreading's impact on computationally intensive tasks is encouraged.

Additional Explorations:

Alternative Signaling Mechanisms: Investigate the use of events or semaphores for thread communication and synchronization.

Error Handling: Incorporate error handling to gracefully manage potential thread issues or calculation errors.

Complex Applications: Explore applying these concepts to more sophisticated multithreaded applications, addressing practical challenges and best practices.

EVENT OBJECTS AS THREAD SYNCHRONIZATION HUBS:

Binary State Machines: Event objects function as simple binary flags, existing in either a "signaled" (set) or "unsignaled" (reset) state. They serve as fundamental building blocks for thread coordination and synchronization in multithreaded applications.

Communication Channels: Threads can signal events to notify other threads of important occurrences, trigger specific actions, or coordinate access to shared resources.

Creating Event Objects:

CreateEvent Function: Event objects are created using the CreateEvent function, which offers a variety of configuration options:

- ✓ **Security Attributes:** Indicate whether the event can be shared between processes or is restricted to the current process.
- ✓ **Reset Type:** Determine whether the event is "manual-reset" (remains signaled until explicitly reset) or "automatic-reset" (unsignals automatically after a thread waits on it).
- ✓ **Initial State:** Specify whether the event starts in a signaled or unsignaled state.
- ✓ **Event Name:** Optionally assign a name to the event for identification and debugging purposes.

Signaling and Unsignaling:

Triggering Actions: Threads signal events using the SetEvent function, effectively setting the event's state to "signaled" and potentially waking up threads that are waiting on it.

Restoring Passivity: The ResetEvent function is used to set an event's state back to "unsignaled," typically for manual-reset events that require explicit clearance before being signaled again.

Waiting for Events:

Synchronizing Actions: Threads can wait for events to become signaled using the WaitForSingleObject function, allowing them to pause execution until a specific condition or action occurs in another thread.

Timeout Option: This function supports a timeout parameter, enabling threads to wait for a limited duration before proceeding, preventing indefinite blocking.

Automatic Reset Events:

Convenient State Management: Automatic-reset events simplify event management by automatically transitioning back to the unsignaled state after a single thread successfully waits on them. This eliminates the need for explicit calls to ResetEvent in many scenarios.

BIGJOB2 Integration:

Persistent Thread Model: BIGJOB2 demonstrates a programming model that leverages a persistent thread throughout the program's lifetime, utilizing an event object to activate the calculation only when necessary.

Event-Driven Efficiency: This approach can potentially improve responsiveness and resource utilization, as the thread idles until signaled, rather than being repeatedly created and destroyed for each calculation.

Code Analysis: While the specific code for BIGJOB2 is not provided, examining its implementation would reveal concrete examples of event object usage and how they integrate with other program components.

Additional Considerations:

Reset Type Selection: Carefully choose between manual-reset and automatic-reset events based on the specific synchronization patterns and thread interactions required in your application.

Synchronization Best Practices: Event objects are often used in conjunction with other synchronization mechanisms, such as critical sections or mutexes, to protect shared data and ensure thread safety in complex multithreaded scenarios.

Robust Error Handling: Incorporate proper error handling techniques when working with event objects to gracefully manage potential issues like timeouts or failure to signal, enhancing the overall resilience of your application.

PERSISTENT THREAD MODEL:

Highlighting Advantages: Emphasizes how a single, persistent thread throughout program execution can lead to:

- Reduced overhead compared to repeated thread creation and destruction.
- Enhanced responsiveness due to the thread's immediate availability.
- Potential for improved performance through better resource utilization.

Exploring Synchronization Implications: Discusses the importance of synchronization mechanisms (e.g., critical sections, mutexes) in more complex scenarios where multiple threads might access shared data, ensuring thread safety and preventing race conditions.

Event-Driven Architecture:

Diving Deeper into Event Handling: Elaborates on the role of event objects in coordinating thread actions:

- Details how the window procedure signals the event object to trigger calculations.
- Explains how the worker thread waits for events to become signaled, entering a suspended state when idle and resuming upon notification.

Considering Alternative Event Types: Introduces the concept of manual-reset events, which remain signaled until explicitly reset, and their potential use cases in different synchronization patterns.

User Interaction and Status Display:

User Experience Focus: Emphasizes the importance of user-friendly interaction and clear feedback:

- Outlines how left mouse clicks initiate calculations and right mouse clicks abort them.
- Describes the informative status messages displayed in the window, indicating program state and calculation time.

Design Considerations: Proposes potential UI enhancements, such as progress bars or visual indicators, to further improve user experience and provide real-time feedback during calculations.

Code Structure and Components:

Window Procedure Deep Dive: Provides a more granular breakdown of the WndProc function's responsibilities:

- Explains how it handles various window messages, including creation, mouse events, custom messages from the worker thread, and window painting.
- Describes the management of the event object, PARAMS structure, and thread communication.

Worker Thread Breakdown: Delves into the Thread function's logic:

- Details its infinite loop structure, waiting for events and performing calculations.
- Explains the use of the bContinue flag to control abortion.
- Clarifies the communication of completion or abortion messages back to the window procedure.

Key Takeaways and Further Exploration:

Reinforcing Efficient Thread Handling: Reiterates the benefits of BIGJOB2's persistent thread approach for resource efficiency and responsiveness.

Stressing Event-Driven Benefits: Emphasizes the versatility of event objects for coordinating thread actions and enabling responsiveness in event-driven applications.

Expanding Horizons: Encourages further exploration of:

- Advanced synchronization techniques for complex multithreaded scenarios.
- Robust error handling strategies to gracefully manage potential thread failures or signaling issues.
- Performance optimization techniques, including alternative synchronization mechanisms and thread scheduling adjustments.

THREAD LOCAL STORAGE

Allows for persistent storage that is unique to each thread within a multithreaded program.

Necessary when multiple threads need to access and modify data independently without affecting other threads. The key functions are:

```
// Allocate a TLS index
DWORD TlsAlloc();

// Set a value for a TLS index in the current thread
BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue);

// Get the value of a TLS index for the current thread
LPVOID TlsGetValue(DWORD dwTlsIndex);

// Free a TLS index
BOOL TlsFree(DWORD dwTlsIndex);
```

Steps for Using TLS:

Define a Data Structure: Define a data structure that represents the data you want to store in TLS. For example:

```
typedef struct {  
    int a;  
    int b;  
} DATA, *PDATA;
```

Allocate a TLS Index: Use the TlsAlloc function to allocate a TLS index. This index will be used to access the TLS data for each thread. Store the returned index in a DWORD variable. For example:

```
DWORD dwTlsIndex = TlsAlloc();
```

Set TLS Value in Each Thread: Before starting a thread, allocate memory for the TLS data using GlobalAlloc or any other memory allocation function. Cast the allocated memory to the data structure type and set it as the TLS value using the TlsSetValue function. For example:

```
PDATA pdata = (PDATA)GlobalAlloc(GPTR, sizeof(DATA));  
TlsSetValue(dwTlsIndex, pdata);
```

Get TLS Value in Any Function: In any function where you need to access the TLS data, retrieve the TLS value using the TlsGetValue function and cast it to the data structure type. You can then access and use the data as needed. For example:

```
PDATA pdata = (PDATA)TlsGetValue(dwTlsIndex);  
// Access and use pdata->a and pdata->b
```

Free TLS Memory: When a thread finishes its execution, you should free the memory allocated for the TLS data using the appropriate memory deallocation function. You can retrieve the TLS value using TlsGetValue and then free the memory. For example:

```
GlobalFree(TlsGetValue(dwTlsIndex));
```

Free TLS Index: When you no longer need the TLS index, use the TlsFree function to free it. This should be done after all threads have finished using TLS. For example:

```
TlsFree(dwTlsIndex);
```

Remember:

- TLS provides a mechanism for thread-specific data, ensuring data integrity and isolation in multithreaded environments.
- Use it for variables that need to maintain unique values for each thread throughout their execution.
- Properly allocate, set, get, and free TLS resources to avoid memory leaks and errors.

THREAD LOCAL STORAGE (TLS): ENABLING THREAD-SPECIFIC DATA IN MULTITHREADED ENVIRONMENTS

Understanding the Need for TLS:

- In multithreaded programs, multiple threads often share access to global variables and heap memory.
- However, scenarios arise where threads need to maintain unique copies of certain variables, ensuring data integrity and preventing conflicts.
- TLS addresses this challenge by providing a mechanism to store data that is specific to each thread, ensuring isolation and preventing unintended modifications by other threads.

Windows API Functions for TLS Management:

- **TlsAlloc:** Grants an unused TLS index, serving as a unique identifier for a thread-local variable.
- **TlsSetValue:** Associates a value (typically a pointer to a memory block) with a given TLS index for the current thread.
- **TlsGetValue:** Retrieves the value associated with a TLS index for the current thread.
- **TlsFree:** Releases a previously allocated TLS index.

Example Code:

```
1 // Allocate a TLS index for thread-specific data
2 DWORD dwTlsIndex = TlsAlloc();
3
4 // Within each thread:
5 PDATA pdata = (PDATA)GlobalAlloc(GPTR, sizeof(DATA));
6 pdata->a = /* thread-specific value */;
7 pdata->b = /* another thread-specific value */;
8 TlsSetValue(dwTlsIndex, pdata);
9
10 // Access thread-local data:
11 PDATA pdata = (PDATA)TlsGetValue(dwTlsIndex);
12 // Use pdata->a and pdata->b as needed
13
14 // Before thread termination:
15 GlobalFree(TlsGetValue(dwTlsIndex));
16
17 // When all threads using the TLS index are done:
18 TlsFree(dwTlsIndex);
```

Microsoft C Extension: `__declspec(thread)`

Simplifies TLS usage by directly declaring thread-local variables:

```
__declspec(thread) int iGlobal = 1; // Global thread-local variable

void myFunction() {
    __declspec(thread) static int iLocal = 2; // Thread-local variable within a function
}
```

The `__declspec(thread)` extension in Microsoft C/C++ simplifies the usage of Thread Local Storage (TLS) by [allowing you to directly declare thread-local variables](#). This extension is supported by the Microsoft C/C++ compiler and provides a convenient way to define thread-local variables without explicitly using the TLS API.

Here's how you can use the `__declspec(thread)` extension:

Declare a Thread-Local Variable:

To declare a thread-local variable, you can use the `__declspec(thread)` specifier before the variable declaration. For example:

```
__declspec(thread) int a;  
...
```

In the above example, the variable `a` is declared as a thread-local variable. Each thread will have its own separate copy of `a`, and changes made to `a` in one thread will not affect its value in other threads.

Use the Thread-Local Variable:

You can use the thread-local variable `a` in the same way as any other variable. For example:

```
a = 42; // Assign a value to the thread-local variable  
int result = a + 10; // Use the thread-local variable in an expression  
...
```

Each thread will have its own independent copy of `a`, and you can read from and write to `a` within each thread without worrying about thread synchronization or data interference.

It's worth noting that the `__declspec(thread)` extension can be used with various types of variables, including primitive types, structures, and pointers.

Benefits of `__declspec(thread)`:

- Using the `__declspec(thread)` extension simplifies the usage of TLS by eliminating the need to explicitly allocate and manage TLS indices using the TLS API functions. It provides a more straightforward and concise syntax for declaring thread-local variables
- With `__declspec(thread)`, the compiler takes care of allocating and managing the TLS storage for thread-local variables, making it easier to work with TLS in your code.

However, keep in mind that this extension is specific to the Microsoft C/C++ compiler and may not be portable across different compilers or platforms.

Tell me more. How does the `__declspec(thread)` extension handle the initialization of thread-local variables? Are there any limitations or considerations when using the `__declspec(thread)` extension in Microsoft C/C++? Can you provide an example of how the `__declspec(thread)` extension can be used with structures or pointers?

When it comes to the **initialization of thread-local variables with the `__declspec(thread)` extension**, each thread will have its own separate copy of the variable, and the initialization will occur automatically when a thread is created.

The **initialization is performed once for each thread**, ensuring that each thread starts with its own initialized copy of the variable. Here's an example to illustrate the initialization behavior of thread-local variables:

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  __declspec(thread) int a = 0;
5
6  void ThreadFunction()
7  {
8      // Each thread will have its own copy of 'a'
9      printf("Thread ID: %lu, a: %d\n", GetCurrentThreadId(), a);
10     a++; // Modify 'a' in this thread
11 }
12
13 int main()
14 {
15     HANDLE threads[3];
16
17     // Create three threads
18     for (int i = 0; i < 3; i++) {
19         threads[i] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadFunction, NULL, 0, NULL);
20     }
21
22     // Wait for all threads to finish
23     WaitForMultipleObjects(3, threads, TRUE, INFINITE);
24
25     return 0;
26 }
```

In this example, **the `a` variable is declared as thread-local using `__declspec(thread)`**. Each thread will have its own copy of `a`, which is initialized to 0 automatically. The `ThreadFunction` is called from each thread, and they all access and modify their respective copies of `a`. The output will show that each thread has its own unique value of `a`.

Considerations and Limitations:

- The `__declspec(thread)` extension is specific to the Microsoft C/C++ compiler and **may not be supported by other compilers or platforms**. It is not part of the C/C++ language standard.
- Thread-local variables declared with `__declspec(thread)` are **not suitable for inter-thread communication or synchronization**. Each thread operates on its own independent copy of the variable, and changes made in one thread do not affect the copies in other threads.
- It's important to note that the initialization of thread-local variables with complex constructors or destructors is not supported with `__declspec(thread)`. The initialization is limited to simple types or types with trivial constructors.

Example with Structures and Pointers:

```
1  #include <stdio.h>
2  #include <windows.h>
3
4  typedef struct {
5      int x;
6      char* name;
7  } ThreadData;
8
9  __declspec(thread) ThreadData* threadData = NULL;
10
11 void ThreadFunction()
12 {
13     threadData = (ThreadData*)malloc(sizeof(ThreadData));
14     threadData->x = GetCurrentThreadId();
15     threadData->name = "Thread Local Data";
16
17     printf("Thread ID: %lu, x: %d, name: %s\n", GetCurrentThreadId(), threadData->x, threadData->name);
18 }
19
20 int main()
21 {
22     HANDLE threads[3];
23
24     for (int i = 0; i < 3; i++) {
25         threads[i] = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)ThreadFunction, NULL, 0, NULL);
26     }
27
28     WaitForMultipleObjects(3, threads, TRUE, INFINITE);
29
30     return 0;
31 }
```

In this example, we declare a **thread-local variable** `threadData` of type `ThreadData*` (a pointer to a structure). Each thread will have its own separate copy of `threadData`. The `ThreadFunction` allocates memory for `threadData` and initializes its members. Each thread can access and modify its own copy of `threadData`.

And that's it for chapter 20 Multithreading...now to the final chapters.