CHAPTER 20: MULTITHREADING AND MULTITASKING 💻🧵

In this chapter, we will break down how Windows handles **multitasking** and **multithreading**, with simple explanations, key concepts, and practical examples. We'll also include insights from Charles Petzold's book to give context.

1. Multitasking ⏱️

Multitasking is the ability of the operating system to run multiple programs at the same time.

- Windows does this by giving each program a **time slice**.
- Programs appear to run simultaneously, even if the CPU is only executing one instruction at a time.
- This improves **system responsiveness** and lets you switch between apps seamlessly.

Evolution in Windows:

- **16-bit Windows:** Used **cooperative multitasking**. Programs had to voluntarily give up CPU control. If one program "hung," the whole system could freeze.
- **32-bit Windows:** Introduced **preemptive multitasking**. The OS actively manages CPU time, making sure no program monopolizes it.

2. Multithreading 🧵

Multithreading is when a single program splits into multiple **threads**, which are lightweight units of execution within the program.

Benefits of multithreading:

- Run **background tasks** without freezing the interface.
- Keep **UI responsive** while other operations run.
- Execute **independent tasks in parallel**, improving performance on multiprocessor systems.

3. Key Terminology 📚

- **Process:** A running program with its own memory and resources.
- **Thread:** A smaller execution unit inside a process. Threads share memory and resources with their parent process.
- **Context Switching:** Saving and restoring a thread's state when switching to another thread.
- **Synchronization:** Methods to safely coordinate access to shared resources between threads, avoiding **race conditions** or data corruption.

4. Topics Covered in Chapter 20 ✅

Thread Creation and Management

- Use CreateThread to start a new thread.
- Threads can have **priorities** set, and you can **suspend**, **resume**, or **terminate** them.

Synchronization Techniques

- Prevent conflicts when threads access shared data using:
  - ✓ **Critical Sections** – fast, process-local locks.
  - ✓ **Mutexes** – system-wide locks.
  - ✓ **Semaphores** – control access to a resource with limited availability.
  - ✓ **Events** – signal between threads.

Thread-Specific Storage

- Use TlsAlloc, TlsGetValue, and TlsSetValue to give each thread its own private data.
- Useful when multiple threads need independent copies of the same type of data.

Win32 Timers

- SetTimer and KillTimer schedule recurring or one-time events.
- Timers are handy for repeating tasks without blocking the main thread.

Asynchronous Procedure Calls (APC)

- Execute code asynchronously in a separate thread using:
    - ✓ BeginThreadEx – create a thread safely in Windows.
    - ✓ QueueUserAPC – queue code to run in a specific thread.

5. Best Practices in Multithreading

- Avoid **deadlocks** by careful locking order.
- Optimize **thread performance** by limiting unnecessary threads.
- Ensure **thread safety** when multiple threads access shared resources.

MULTITASKING IN THE DOS ERA

Before Windows made multitasking smooth, DOS had **ideas** but a lot of **practical limits**.

- Users **wanted multitasking** (running multiple programs at once).
- DOS **wasn't built for it** — hardware and software got in the way.
- Early attempts were **creative but limited**.

Why DOS Struggled

I. Hardware Limits

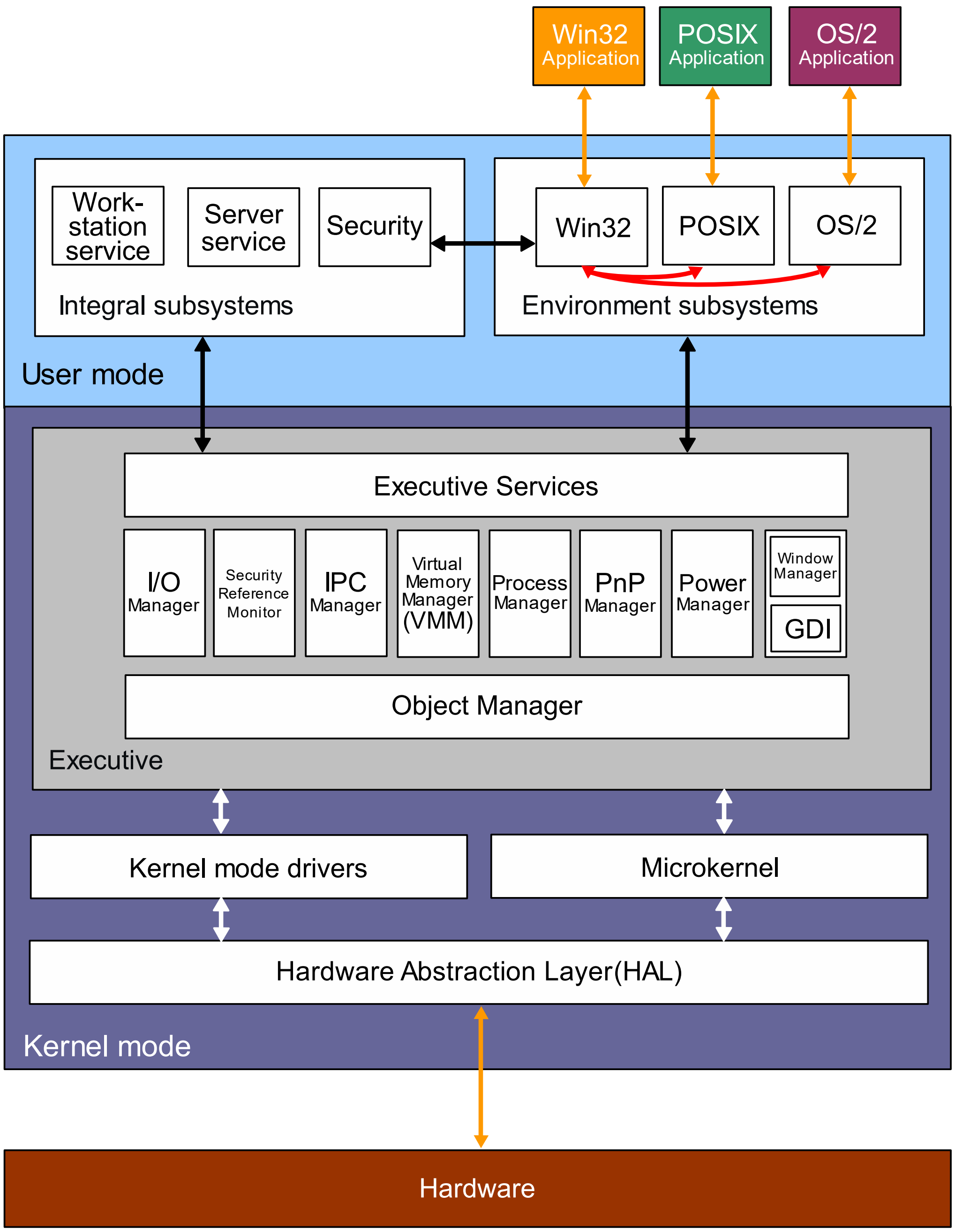
- CPUs like the **Intel 8088** had **no memory management unit (MMU)**.
- Programs had to share **one flat memory space**, making juggling apps tricky.
- Moving memory blocks to free up space was slow and hard.

II. DOS Architecture

- DOS was **simple and lean** — designed for **single-tasking**.
- APIs were basic: mostly **file access** and **program loading**.
- No robust system services → developers couldn't easily implement multitasking.

Think of DOS as a **tiny studio apartment**:

- Only one person (program) can comfortably live there at a time.
- You *could* squeeze in another, but furniture (memory) had to be constantly moved.
- Windows later gave everyone **their own rooms (protected memory & system services)** — suddenly multitasking made sense.



Creative Workarounds & Multitasking in DOS 💻✨

Even though DOS wasn't designed for multitasking, **clever programmers found ways to simulate it**.

1. TSR Programs (Terminate-and-Stay-Resident)

- TSRs stayed **in memory** after their main program closed, running in the background.
- Examples:
  - ✓ **Print spoolers:** used the hardware timer interrupt to print while you worked on another program.
  - ✓ **Borland SideKick:** temporarily paused your current app to show its interface, then returned control.
- TSRs were **early multitasking hacks**, letting small tools coexist with other programs.

2. Enhanced DOS Features

- Microsoft added **memory swapping to disk** and other tweaks.
- These **helped manage memory** better, indirectly supporting background activities.

3. Market Experiments

- **Task-switching shells** like **Quarterdeck DesqView** allowed running multiple DOS programs.
- Reality check: performance was slow, setup was tricky, adoption was limited.

4. Key Takeaways

- Multitasking was **desired by users**, even on single-user PCs.
- Programmers used **TSRs and simple task-switchers** to stretch DOS's limits.
- These efforts **highlighted the need for OS-level multitasking**, eventually realized in Windows.

5. Exploration Ideas

- Study **popular TSRs**: what they did, how they worked.
- Dive into **DOS memory management** and **context switching** challenges.
- Compare DOS multitasking hacks vs. **Windows 3.x and later**, seeing how true multitasking became practical.

💡 Mental shortcut: DOS multitasking = **hacks in a single-room apartment**.

- TSRs = roommates who quietly exist in the background.
- Task-switchers = rotating chairs; clunky but functional.
- Windows = actual multiple rooms, each app gets space, fully coordinated.

Multitasking in Early Windows ⚡

Early Windows (1.0, 1985) **brought multitasking out of DOS limitations**, giving multiple programs a graphical workspace and basic coordination.

1. Windows 1.0's Multitasking Breakthrough

- **Graphical interface:** Multiple programs could run concurrently, unlike DOS command-line shells.
- **Cooperative multitasking (nonpreemptive):**
  - ✓ Programs only yield control voluntarily after processing messages.
  - ✓ Relies on well-behaved programs; a stuck program could freeze the system.
- **Message-based architecture:** Programs are idle until a message arrives.

Workarounds & Limitations

- **Preemption:** Used only for DOS programs or certain multimedia tasks (DLLs needing hardware timing).
- **Hourglass cursor:** Visual cue for busy programs.
- **Windows timer & PeekMessage:** Let programs periodically handle messages, preventing total freeze during long tasks.

2. Data Sharing Mechanisms

a) Clipboard

- **Basic & versatile:** For cut/copy/paste across applications.
- **Temporary storage:** Holds data for short-term transfer.
- **Manual operation:** User-controlled, not automatic.

b) Dynamic Data Exchange (DDE)

- **Live links:** Programs exchange data in real-time, even if idle.
- **Client-server model:** One app requests updates from another.
- **Examples:** Stock tickers, spreadsheets linked to databases.
- **Caveats:** Complex, fragile if connections fail.

c) Object Linking and Embedding (OLE)

- **Embedding objects:** One document can contain content from another program.
- **In-place editing:** Edit objects directly within the main document.
- **Example:** Edit a spreadsheet chart inside Word without launching Excel.
- **Use case:** Rich, interactive compound documents.

3. Key Points

- Clipboard → simple, manual data sharing.
- DDE → live updates between programs, more complex.
- OLE → seamless object integration for richer documents.
- 16-bit Windows used **cooperative multitasking**, with no enforced preemption.
- System responsiveness relied on programs yielding control voluntarily.

4. Further Exploration

- Investigate challenges of **preemptive multitasking** in 16-bit Windows.
- Study real examples of **application freezes or bottlenecks** under cooperative multitasking.
- Track the evolution toward **preemptive multitasking** in 32-bit Windows.
- Modern parallels: **Clipboard history, cloud sync**, **XML/JSON** for universal data exchange, and **OLE alternatives** like ActiveX or .NET components.

💡 Mental shortcut:

- DOS multitasking = roommates squeezing in one apartment.
- Windows 1.0 = a shared living room where everyone must **take turns politely**.
- Clipboard/DDE/OLE = ways to **pass notes between roommates without chaos**.

Multithreading & The Evolution of Input

1. The History Lesson: Why OS/2 Failed

Before Windows 95/NT, Microsoft and IBM built **OS/2 Presentation Manager (PM)**. It was a 32-bit operating system with a fatal flaw in how it handled user input.

The Flaw: The Serialized Message Queue In OS/2, the system had *one* single pipe for all keyboard and mouse clicks for every running program.

- **The Rule:** The system would not process Input B until the application finished processing Input A.
- **The Goal:** Predictability. If you type "ABC", the system guarantees "A" is processed before "B".
- **The Reality:** If one program crashed or hung while processing "A", the entire system froze. You couldn't click on another window because the mouse click was stuck in the queue behind "A".

The Lesson: A robust OS cannot let one bad app freeze the whole mouse.

2. The Windows Solution: Deserialized Input

Modern Windows (Win32 API) fixed this by giving every thread its own private message queue.

- **How it works:** When you click on Chrome, Windows looks at where the mouse is, determines which thread owns that window, and drops the message directly into *that specific thread's* queue.
- **The Benefit:** If Chrome hangs, Notepad keeps working. You can Alt-Tab away from a frozen program.

3. The Architecture of a Multithreaded App

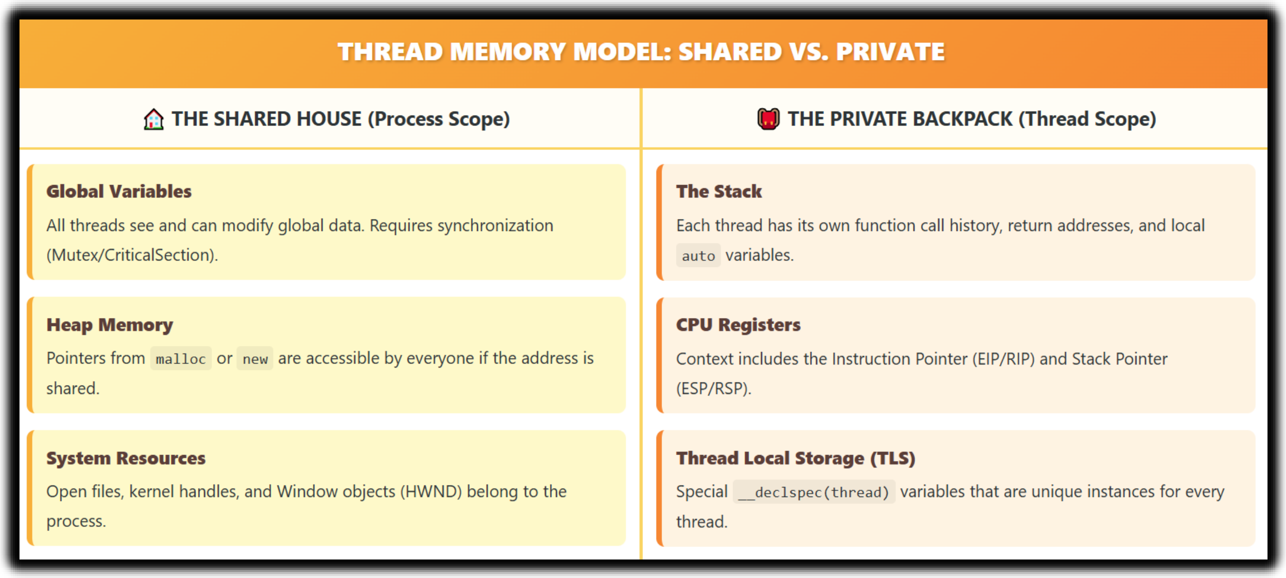
In modern Windows programming, we divide labor to keep the application responsive.

The "Governor & Staff" Model

1. **The Primary Thread (The Governor):**
   - ✓ **Job:** Creates windows, runs the Message Loop, handles the UI (WM_PAINT, buttons).
   - ✓ **Rule:** Never do heavy lifting here. If you calculate Pi to the billionth digit here, the UI freezes, and the window says "(Not Responding)".
2. **The Secondary Threads (The Staff):**
   - ✓ **Job:** Long calculations, file I/O, networking.
   - ✓ **Rule:** They do *not* own windows. They crunch numbers in the background and tell the Governor when they are done.

4. What Threads Share vs. What They Keep

When you create a new thread, it is not a separate program. It lives inside the same "house" (Process).



The Danger: Because they share global memory, two threads can try to write to the same variable at the exact same time, causing corruption (Race Conditions).

---

5. Summary Checklist

- **OS/2 PM** failed because one frozen app froze the whole system (Serialized Input).
- **Windows** succeeds because input is split per thread (Deserialized Input).
- **Primary Thread:** Handles UI/Messages.
- **Secondary Thread:** Handles heavy math/background work.
- **Threads share memory**, which makes them fast but dangerous (requires synchronization).

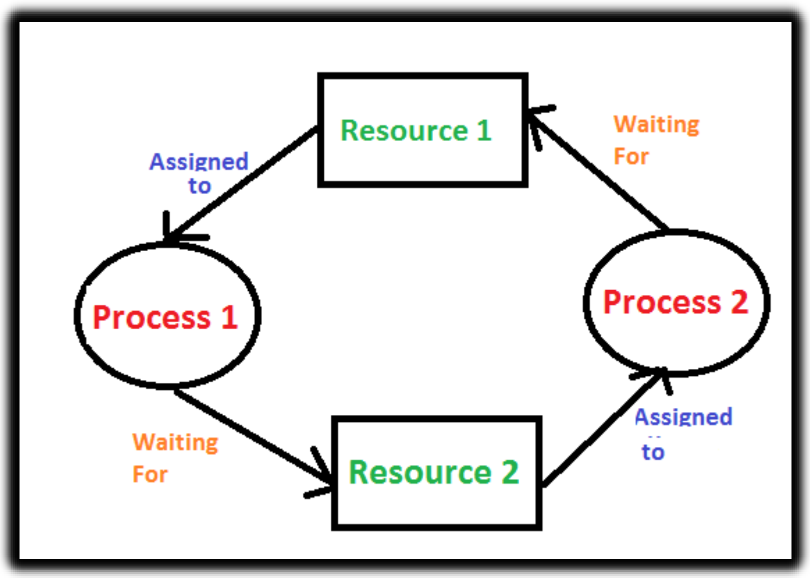
Next Step: Now that we know *why* we need threads, we need to look at **how to create them** using _beginthreadex vs CreateThread.

THREADING CHALLENGES & BEST PRACTICES

Multithreading is powerful, but it introduces a new class of bugs that are arguably the hardest to debug in all of computer science.

1. The Chaos: Race Conditions and Deadlocks

In a single-threaded program, you know exactly which line of code runs next. In a multithreaded program, the Operating System acts like a chaotic referee. It pauses Thread A and runs Thread B at random times—sometimes right in the middle of a calculation.



The Race Condition This happens when two threads try to change the same variable at the same time.

- *Thread A* reads Score = 10.
- *Thread B* reads Score = 10.
- *Thread A* adds 1 and writes 11.
- *Thread B* adds 1 and writes 11.
- *Result:* The score should be 12, but it is 11. Data is lost because they "raced" to write.

The Deadlock This happens when two threads stop and wait for each other.

- *Thread A* holds Key 1 and waits for Key 2.
- *Thread B* holds Key 2 and waits for Key 1.
- *Result:* They both wait forever. The program freezes.

2. The Fix: Synchronization

To stop the chaos, you need **Synchronization Primitives**. These are tools that force threads to wait their turn.

- **Critical Sections:** A block of code that only one thread can enter at a time. It is like a bathroom with a lock. If Thread A is inside, Thread B must wait outside until A leaves.
- **Semaphores:** Like a bouncer at a club. It allows a specific number of threads in at once (e.g., allowing 5 connections to a database).

3. The Hardware Reality: 16-bit vs 64-bit

The Old Days (16-bit) Processors were simple. Even adding 1 to a large number required two CPU cycles. If the OS interrupted the thread between cycle 1 and 2, the number would be corrupted. You had to lock everything.

The Modern Era (64-bit)

- **Atomicity:** Modern CPUs can read and write huge numbers (64-bit) in a single cycle. You don't need locks just to read a variable.
- **The New Danger (Optimization):** Modern CPUs are smart. They reorder your instructions to run faster. They might run Line 10 before Line 5 if they think it is safe. In multithreading, this can break your logic.
- **Takeaway:** Do not rely on hardware tricks. Always use proper Synchronization (locks) to be safe.

4. Windows Advancements

Windows evolved to make threading easier and safer.

A. Deserialized Input (The Fix for Freezing) - We covered this in the last section. Windows gives every thread its own input queue so a frozen background thread doesn't freeze the mouse cursor.

B. Thread Local Storage (TLS) Global variables are dangerous because all threads share them (leading to Race Conditions). TLS allows you to create a "Global" variable that is unique to each thread.

- If Thread A writes "Red" to *Color*, it sees "Red".
- If Thread B writes "Blue" to *Color*, it sees "Blue". They use the same variable name but look at different memory addresses.

5. The "New & Improved" Fallacy

Just because you *can* use threads doesn't mean you *should*. Threads add overhead. The CPU wastes time switching context between them.

The 1/10 Second Rule (100 Milliseconds) Use this rule of thumb to decide if you need a thread:

- **Task takes < 100ms:** Do not use a thread. Just run it. The user won't notice the tiny pause.
- **Task takes > 100ms:** Use a secondary thread. If you don't, the UI will freeze, and the user will think the app crashed.

6. Summary Checklist

1. **Race Conditions** happen when threads share data without locks.
2. **Deadlocks** happen when threads wait for each other.
3. **Critical Sections** prevent these bugs by forcing single-file access.
4. **TLS** gives threads private data.
5. **Only thread** if the task is slow (over 1/10th of a second).

The Two Ways to Spawn a Thread

a) The Raw Windows API (CreateThread)

This is the native function provided by the OS.

- **Pros:** It gives you granular control (Security attributes, Stack size).
- **Cons:** It does **not** set up the C Runtime (CRT).
- **The Danger:** If you use C functions like malloc, printf, or strtok inside a thread created with CreateThread, the program might crash or leak memory because the CRT data structures weren't initialized for that thread.

b) The C Runtime Helper (_beginthreadex)

This is the wrapper function provided by Microsoft's C library.

- **Pros:** It initializes the C Runtime, then calls CreateThread internally.
- **Cons:** Slightly different syntax.
- **The Rule:** Always use _beginthread (or _beginthreadex) if your thread uses any C library functions.

Comparison Table:

THREAD CREATION: WIN32 API VS. C RUNTIME		
FEATURE	CREATETHREAD	_BEGINTHREADEX
Origin	Windows Native API (`windows.h`)	C Runtime Library (`process.h`)
Use Case	Pure API coding (No C/C++ Libs)	**Standard C/C++ coding**
CRT Safety	**Unsafe:** May cause leaks if using `malloc`, `printf`, or `strtok`.	**Safe:** Correctlty initializes per-thread CRT data blocks.
Returns	`HANDLE`	`uintptr_t (cast to HANDLE)`
Exit Method	`ExitThread()`	`_endthreadex()`

The Random Rectangles Program (RNDRCTMT.C)

This program demonstrates the simplest possible multithreaded app:

- **Thread 1 (Main):** Handles the Window (Resizing, Closing).
- **Thread 2 (Worker):** Draws random colored rectangles on the window background forever.

I. How it works:

1. **WinMain:** Registers the class and creates the window.
2. **WM_CREATE:** The main thread calls _beginthread(Thread, ...) to spawn the worker.
3. **The Worker Loop:**
    - It sits in a while(TRUE) loop.
    - It generates random x, y, color values.
    - It calls Rectangle to draw on the screen.

4. **WM_SIZE:** When you resize the window, the Main Thread updates the global variables cxClient and cyClient. The Worker Thread reads these new values instantly and starts drawing in the new area.

II. The Compiler Setting (Crucial!)

You cannot just compile this code normally. You must tell the compiler: *"I am using threads."*

- **The Switch:** /MT (Multithreaded Static) or /MD (Multithreaded DLL).
- **The Library:** It links against LIBCMT.LIB.
- **What it does:** It changes standard functions like strtok. In a single-threaded app, strtok uses a static variable to remember where it left off. In a multithreaded app, LIBCMT replaces that static variable with **Thread Local Storage (TLS)** so two threads using strtok at the same time don't corrupt each other's strings.

III. Critical Bug Warning (Synchronization)

The RNDRCTMT.C example is simple, but it has a **Hidden Race Condition**.

The Shared Data: cxClient and cyClient (Window Size).

The Race:

- Thread 1 (Main) is writing cxClient = 500.
- Thread 2 (Worker) is reading cxClient.

It is possible for Thread 2 to read the variable *while* Thread 1 is halfway through writing it (on 16-bit systems especially).

The Fix: Real programs need a **Critical Section** to lock the variable while reading/writing.

IV. Summary Checklist

1. Use _beginthread instead of CreateThread to keep malloc safe.
2. Enable the **Multithreaded (/MT)** setting in your compiler.
3. **Global Variables** are the easiest way for threads to talk, but they are dangerous without locks.
4. **Automatic Variables** (inside functions) are safe; every thread gets its own copy on its own stack.

The Multitasking Contest (MULTI1 vs MULTI2)

This section compares two ways to solve a classic 1986 programming problem: "How do you run 4 distinct tasks in 4 separate windows at the same time?"

The Tasks:

1. Window 1: Count up (1, 2, 3...).
2. Window 2: Find Prime Numbers.
3. Window 3: Calculate Fibonacci Sequence.
4. Window 4: Draw random circles.

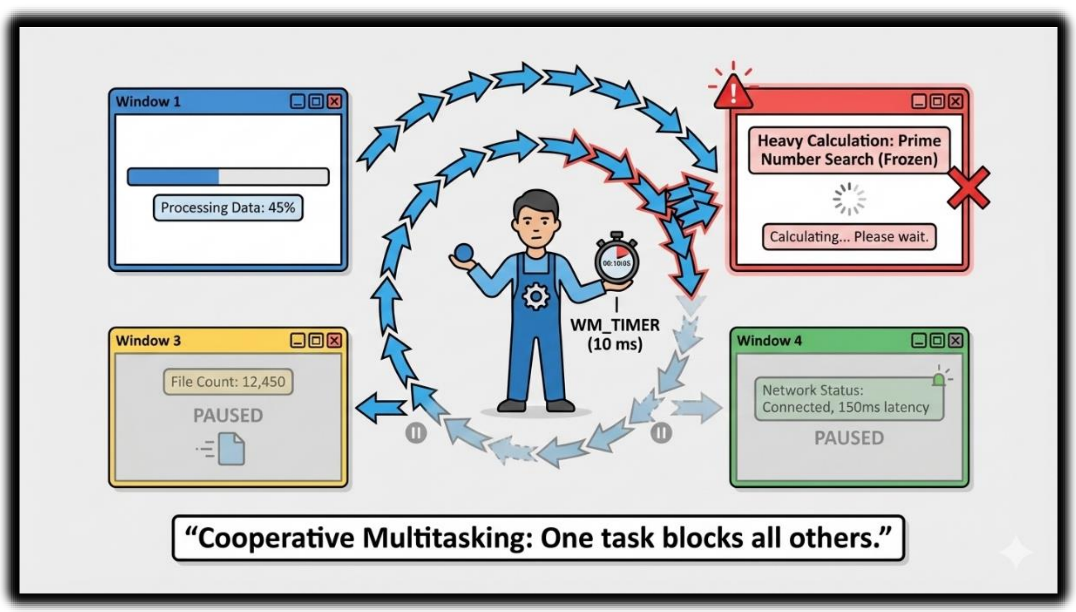
Multi1 and Multi2 Program.mp4

1. The Old Way: MULTI1 (The Simulation)

The Strategy: The Juggler MULTI1 does not use threads. It uses a **Timer**. It relies on a single main loop. Every 10 milliseconds, the timer fires a WM_TIMER message. The program catches this message and quickly updates Window 1, then Window 2, etc.

Why it works: Computers are fast. If you switch between tasks quickly enough, it *looks* like they are happening at the same time.

The Flaw: This is "Cooperative Multitasking." If Window 2 gets stuck calculating a massive Prime Number, Windows 3 and 4 stop updating. The entire application freezes until the calculation is done.



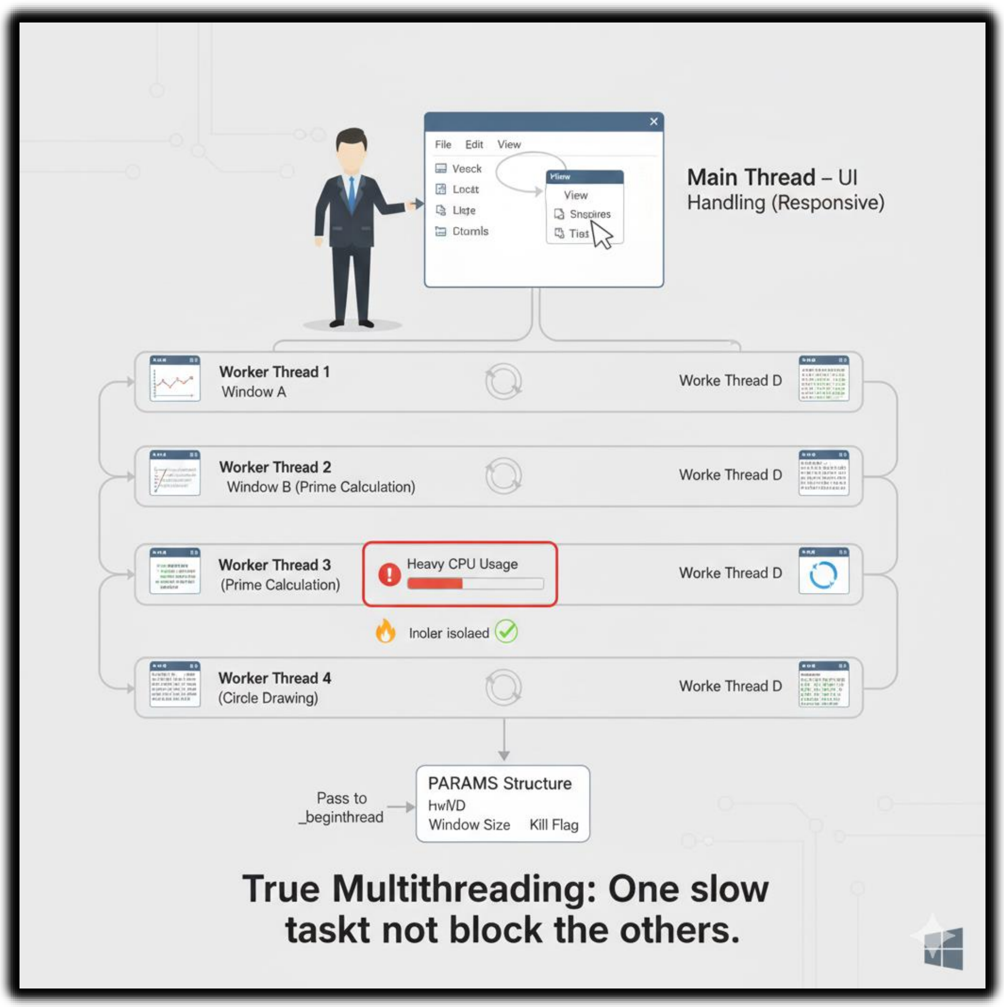
2. The New Way: MULTI2 (True Multithreading)

The Strategy: The Team MULTI2 creates 5 threads total.

1. **Main Thread:** Handles the UI (moving windows, clicking menus).
2. **4 Worker Threads:** Each window gets its own dedicated thread created with _beginthread.

The Differences:

- **No Timer:** The worker threads don't wait for a "tick." They run while loops as fast as the CPU allows.
- **Struct Passing:** Since _beginthread only accepts one argument, we pack the data (Window Handle, Window Size, Kill Flag) into a PARAMS structure and pass the pointer.
- **Responsiveness:** If the Prime Number thread gets stuck on a hard calculation, the Circle thread keeps drawing. The Main thread keeps responding to the mouse.



3. The Danger: The "bKill" Race Condition

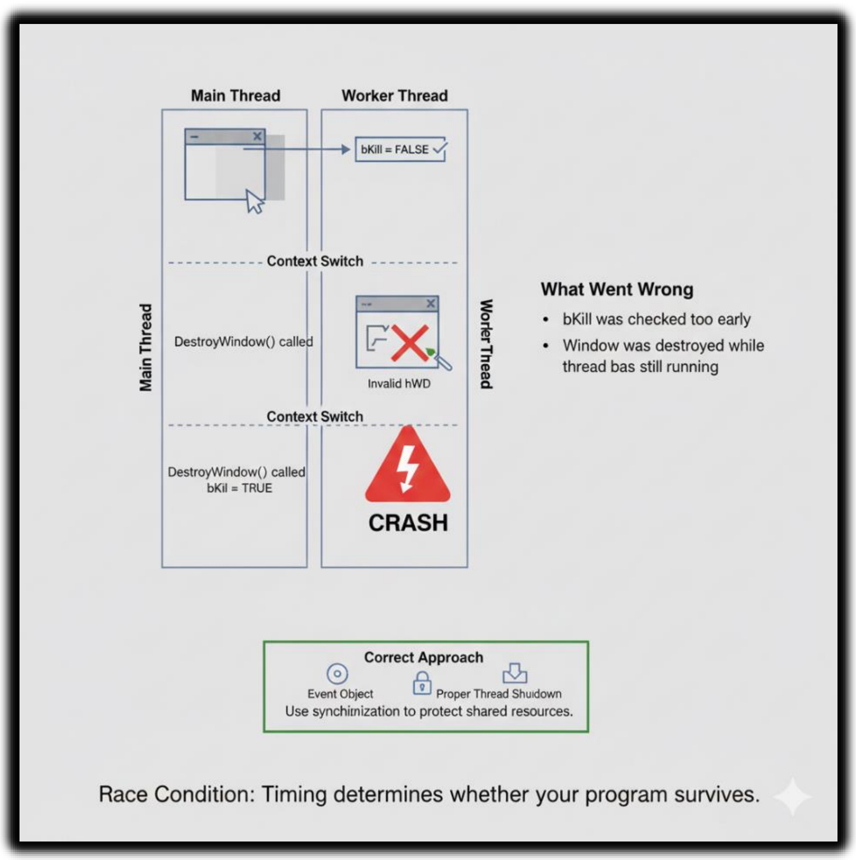
MULTI2 introduces a serious bug that MULTI1 didn't have.

The Scenario: You have a boolean flag called bKill. When the user closes the window, the Main Thread sets bKill = TRUE. The Worker Thread checks this flag; if it is TRUE, it stops.

The Crash (Race Condition):

1. Worker Thread checks bKill. It is **FALSE**.
2. *Context Switch happens.* The OS pauses the Worker and runs the Main Thread.
3. User closes the window. Main Thread destroys the window handle and sets bKill = TRUE.
4. *Context Switch happens.* The OS resumes the Worker.
5. Worker Thread (thinking bKill is still false) tries to draw on the window.
6. **CRASH.** The window handle is invalid.

The Lesson: You cannot rely on simple boolean flags to stop threads safely. In a real application, you must use **Synchronization Objects** (like Event Objects or Critical Sections) to ensure the window isn't destroyed while a thread is painting on it.

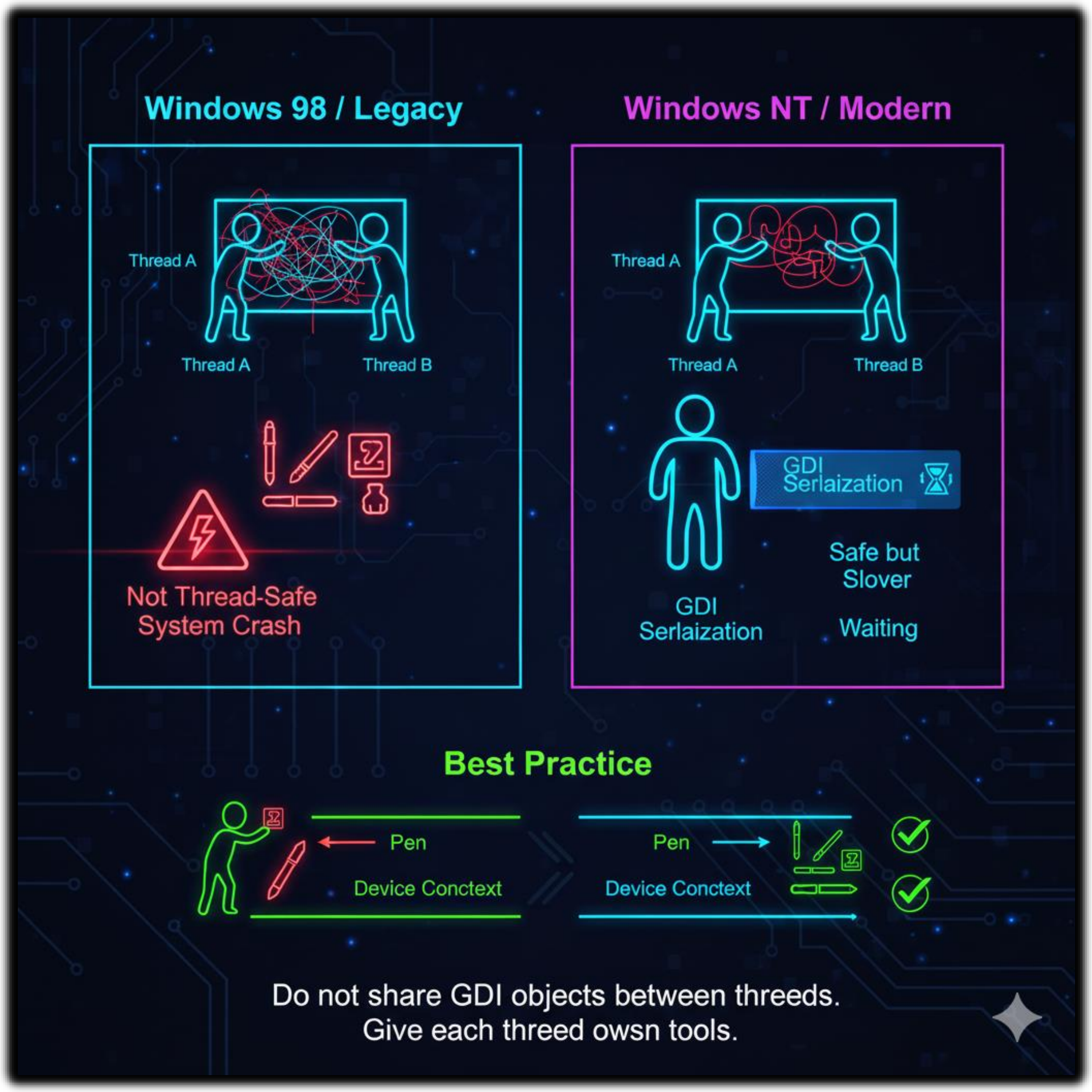


4. GDI and Threads

Windows GDI (Graphics Device Interface) has specific rules for threads:

- **Windows 98/Legacy:** Not thread-safe. If two threads try to draw at the exact same time, the system might crash.
- **Windows NT/Modern:** The OS "Serializes" GDI calls. It forces Thread B to wait if Thread A is drawing. This prevents crashes but slows performance.

Best Practice: Do not share GDI Objects (Pens, Brushes, Device Contexts) between threads. Give every thread its own tools.



Summary Checklist

1. **MULTI1** fakes multitasking using WM_TIMER. It is safe but can freeze if one task is heavy.
2. **MULTI2** uses _beginthread for real multitasking. It is fast but dangerous.
3. **The Struct:** Use a structure to pass multiple arguments to a new thread.
4. **The Race:** Checking a variable (bKill) and acting on it is not an "Atomic" operation. The OS can interrupt you in the middle.
5. **The Fix:** Real multithreading requires **Synchronization** to prevent accessing dead memory.

SLEEP, SYNCHRONIZATION, AND CRITICAL SECTIONS

We established that multithreading causes chaos (Race Conditions). Now, let's look at the tools we use to tame that chaos.

1. The Sleep Function: The "Pause" Button

Sleep(milliseconds) is the simplest way to control a thread.

- **What it does:** It tells the Operating System: *"Stop running me for X milliseconds. Let other threads use the CPU."*
- **The Sleep(0) Trick:** If you call Sleep(0), you tell the OS: *"I don't need to pause, but I am willing to give up the rest of my turn if anyone else is waiting."* It is a polite way to yield the processor.

When to use it:

- **Good:** In a background thread loop to prevent it from eating 100% CPU (e.g., check for new email every 5 seconds).
- **Bad:** In the Main Thread. Calling Sleep here freezes your window. The user cannot click anything until it wakes up.

2. The Traffic Jam: Why We Need Locks

Imagine a bank account with $100.

- **Thread A** tries to withdraw $10.
- **Thread B** tries to withdraw $10.

If they run at the exact same time:

1. Thread A reads balance ($100).
2. Thread B reads balance ($100).
3. Thread A writes new balance ($90).
4. Thread B writes new balance ($90).

- **Result:** You withdrew $20, but the balance only dropped by $10. The bank lost money.

We need a way to force Thread B to wait until Thread A is completely finished.

3. The Solution: Critical Sections

A Critical Section is a traffic light for code. It protects a specific block of memory.

How it works (The 4 Steps):

1. **Initialize:** Create the "Traffic Light" object (CRITICAL_SECTION).
2. **Enter:** Before you touch the shared data, call EnterCriticalSection.
   - ✓ *Effect:* If the light is Green, you pass. The light turns Red.
   - ✓ *Effect:* If the light is Red (someone else is inside), your thread **sleeps** immediately. You wait until it turns Green.
3. **Leave:** When you are done, call LeaveCriticalSection.
   - ✓ *Effect:* The light turns Green. If another thread was waiting, it wakes up and enters.
4. **Delete:** When the program ends, clean up the object.

The Code Example:

```
CRITICAL_SECTION cs; // The Traffic Light

// ... Initialization somewhere ...

void UpdateBankAccount() {
    EnterCriticalSection(&cs);  // STOP! Wait your turn.

    // --- SAFE ZONE ---
    // Only one thread can be here at a time.
    int balance = ReadBalance();
    balance = balance - 10;
    WriteBalance(balance);
    // ----------------

    LeaveCriticalSection(&cs); // Go ahead, next person.
}
```

4. Alternative Tools

- **Mutex (Mutual Exclusion):** Similar to a Critical Section, but it works across *different programs* (e.g., syncing Word and Excel). Slower than Critical Sections.
- **Semaphores:** A counter. Instead of "Only 1 person," it allows "Up to 5 people." Good for limiting database connections.
- **Events:** A starting gun. Thread A sleeps until Thread B fires the "Event" signal.

5. Summary Checklist

1. **Sleep** pauses a thread to save CPU.
2. **Race Conditions** corrupt data when threads fight over memory.
3. **Critical Sections** lock a piece of code so only one thread runs it at a time.
4. **Enter/Leave:** You must always pair EnterCriticalSection with LeaveCriticalSection. If you forget to Leave, the program hangs forever (Deadlock).

UNDERSTANDING CRITICAL SECTIONS

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.

1. Key Functions

Initializing a Critical Section:

```
#include <windows.h>

CRITICAL_SECTION cs;  // Declare a global critical section object

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

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
```

Deleting a Critical Section:

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

BigJob1
program.mp4

Events and Thread Signaling (BIGJOB1 & BIGJOB2)

We now understand how to *lock* threads (Critical Sections). Now, let's learn how to *talk* to them.

1. The Core Concept: Signaling

In a multithreaded app, threads often need to wait for each other.

- **The Main Thread:** "I need the Worker Thread to start calculating now."
- **The Worker Thread:** "I am finished. Here is the result."

We could use a while loop to constantly check a variable (Polling), but that wastes 100% of the CPU. Instead, we use **Event Objects**. An Event Object is like a **Start Gun**.

- **Unsignaled State:** The gun is raised. Everyone waits.
- **Signaled State:** *BANG!* The gun fires. The waiting thread wakes up and runs immediately.

2. The BIGJOB1 Program (The "Create/Destroy" Method)

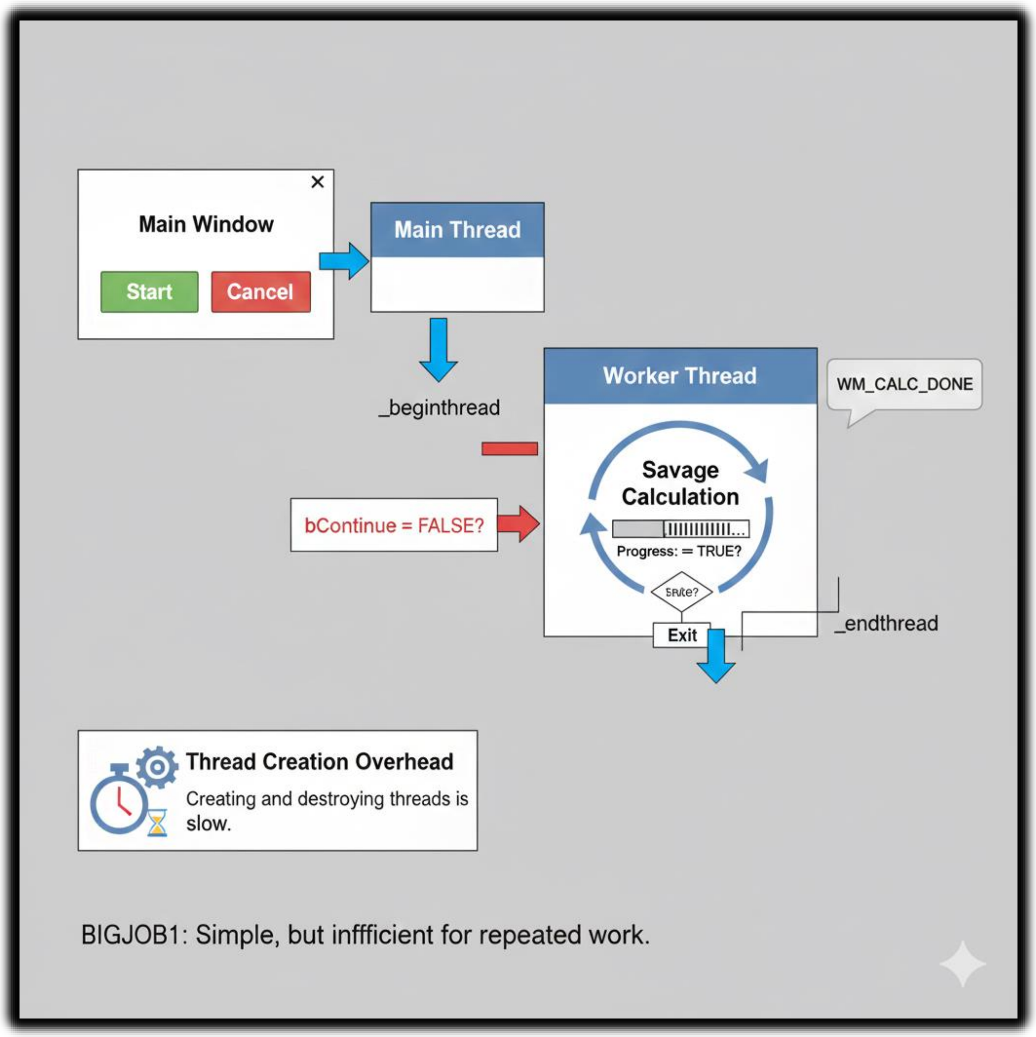
This program calculates a complex math benchmark ("Savage").

The Strategy:

1. **Start:** When you click "Start", the Main Thread calls _beginthread.
2. **Run:** The new thread runs the calculation loop.
3. **Finish:** When the calculation is done, the thread sends a message (WM_CALC_DONE) to the Main Window and then **destroys itself** (_endthread).

The Communication Trick: How does the Main Thread stop the calculation if the user clicks "Cancel"? It sets a boolean flag bContinue = FALSE. The Worker Thread checks this flag inside its for loop. If it sees FALSE, it aborts.

The Flaw: Creating and destroying a thread every time you click "Start" is slow (Thread Creation Overhead). It is better to keep the thread alive and just put it to sleep.



3. The BIGJOB2 Program (The "Event" Method)

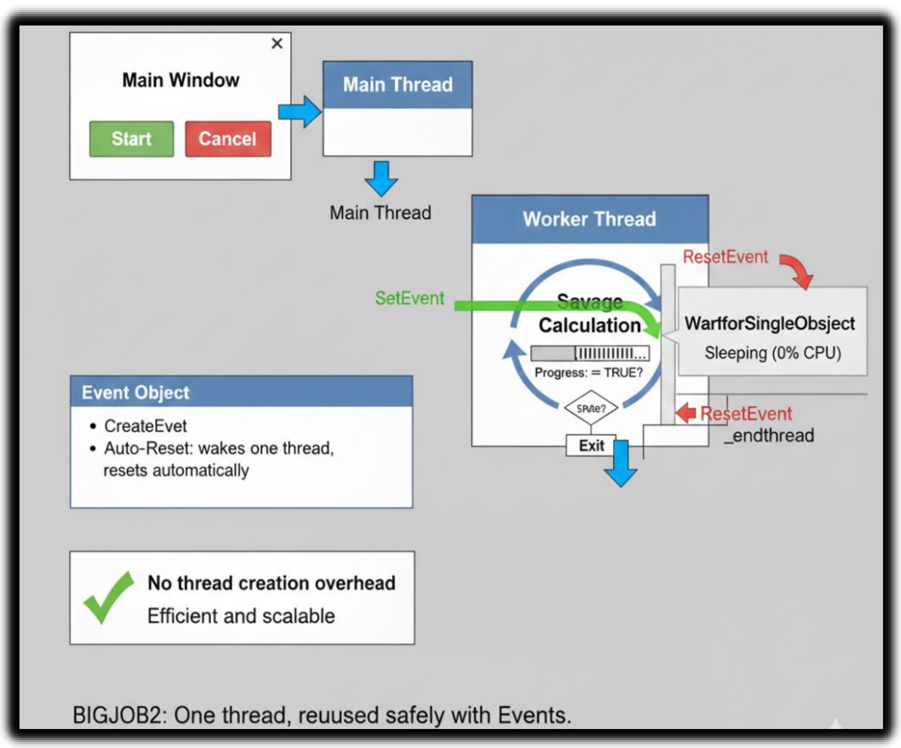
This version is much smarter. It creates the Worker Thread *once* when the program starts.

The Strategy:

1. **Sleep:** The Worker Thread sits in an infinite loop, waiting at a WaitForSingleObject call. It consumes 0% CPU.
2. **Wake Up:** When you click "Start", the Main Thread calls SetEvent.
3. **Run:** The WaitForSingleObject sees the signal, wakes up, runs the calculation *once*, and then loops back to the top to wait again.

The Key Functions:

- **CreateEvent:** Makes the object.
    - ✓ *Auto-Reset:* Once one thread wakes up, the event automatically turns off (Unsignaled) so the thread doesn't run twice.
    - ✓ *Manual-Reset:* The event stays On until you manually turn it off.
- **SetEvent:** Fires the gun. Wakes up the waiting thread.
- **ResetEvent:** Reloads the gun (makes it Unsignaled).
- **WaitForSingleObject:** The thread pauses here until the Event is Signaled.



4. Comparison: Polling vs. Events

Bad (Polling):

```
// Worker Thread
while (bNotDone) {
    if (bReady) { DoWork(); } // Wastes CPU checking this 1,000,000 times/sec
}
```

Good (Events):

```
// Worker Thread
while (TRUE) {
    WaitForSingleObject(hEvent, INFINITE); // Sleeps. CPU usage = 0%
    DoWork();
}
```

5. Summary Checklist

1. **Events** are the best way to coordinate threads without wasting CPU.
2. **SetEvent** wakes up a waiting thread.
3. **WaitForSingleObject** puts a thread to sleep until the signal comes.
4. **Auto-Reset Events** are safer for "Start Work" signals because they turn themselves off automatically.
5. **BIGJOB2** is better than BIGJOB1 because it reuses the thread instead of recreating it.

PERSISTENT THREAD MODEL 🧵

Why a Persistent Thread?

Instead of creating and destroying threads repeatedly, the program keeps **one worker thread alive for the entire lifetime** of the app.

I. Why This Model Is Used

This approach avoids the cost of repeatedly creating and destroying threads and provides a more stable execution environment for background work.

II. Practical Benefits

- **Reduced overhead:** Thread creation and teardown are expensive operations.
- **Improved responsiveness:** The thread is already available when work begins.
- **Predictable performance:** Memory and scheduling behavior remain consistent.

III. Ideal Use Cases

- Repeated or long-running background tasks
- Work triggered by user input or window messages
- Applications where startup latency matters

Synchronization Reality

I. The Moment Things Get Dangerous

A persistent thread is simple **only** when it works alone. The moment shared data is introduced, synchronization becomes mandatory.

II. Shared State Problems

When multiple threads access shared variables:

- Race conditions become unavoidable without protection.
- Bugs become timing-dependent and hard to reproduce.

III. Required Synchronization Tools

- Critical sections
- Mutexes
- Events

IV. Rule of Thumb

One thread → simple
Two threads touching shared data → synchronize or suffer

EVENT-DRIVEN ARCHITECTURE 🔔

1. Core Design Principle

I. Event-Driven vs Polling

The architecture is event-driven, meaning threads sleep until something meaningful happens, rather than constantly checking conditions.

II. Signal Flow

- The **window procedure** signals an event when work should begin.
- The **worker thread** waits on that event.
- While waiting, the thread consumes no CPU time.
- Once signaled, the thread wakes up, performs its task, and returns to waiting.

III. Benefits of This Model

- Zero busy-waiting
- Clean separation between UI and background work
- High responsiveness even under load

2. Event Types and Their Meaning

I. Auto-Reset Events

- Automatically reset after waking one thread
- Ideal for one-time notifications
- Common in producer/consumer patterns

II. Manual-Reset Events

- Remain signaled until explicitly reset
- Useful when multiple actions depend on the same signal
- Require careful control to avoid logic errors

III. Design Awareness

Choosing the correct event type is not optional — it directly affects correctness and performance.

USER INTERACTION AND STATUS FEEDBACK 🖱

1. Input Design

I. Mouse-Driven Control

- **Left mouse click:** Starts the calculation
- **Right mouse click:** Aborts the calculation

This design is minimal, intentional, and avoids unnecessary UI complexity.

2. Status Communication

I. Information Presented to the User

- Current program state (idle, running, aborted)
- Time taken for calculations

II. Why This Matters

A responsive UI is not a luxury. If users cannot tell what the program is doing, the program is effectively broken.

III. Possible Enhancements

- Progress bars
- Activity indicators
- Periodic progress messages from the worker thread

CODE STRUCTURE AND RESPONSIBILITIES 🧠

1. Window Procedure (WndProc)

I. Primary Responsibilities

- Handle window creation and cleanup
- Process mouse input
- Signal events to the worker thread
- Receive completion or abort messages
- Paint status information

II. Mental Model

The window procedure is a **coordinator**, not a worker.

2. Worker Thread Logic

I. Core Execution Pattern

- Infinite loop
- Wait for event
- Perform calculation
- Check a control flag (bContinue) to allow abortion
- Notify the window of completion or cancellation

II. Mental Model

The worker thread is a **sleeping worker**, not a CPU spinner.

KEY TAKEAWAYS 🧠

1. Architectural Lessons

- Persistent threads improve responsiveness and efficiency
- Events are the backbone of clean WinAPI multithreading
- UI responsiveness is part of program correctness

2. Design Discipline

- UI logic and worker logic must remain separated
- Abort paths must be treated as first-class logic
- Sleeping threads are superior to polling threads

THREAD LOCAL STORAGE (TLS) 🧬

1. Purpose of TLS

I. What TLS Provides

Thread Local Storage allows each thread to store its own private data, even though all threads execute the same code.

II. When TLS Is Necessary

- Each thread needs its own state
- Global variables would conflict
- Passing context through function parameters becomes unmanageable

2. TLS Lifecycle

I. Define a Data Structure

Create a structure representing the per-thread data.

II. Allocate a TLS Index

Use TlsAlloc once and store the index globally.

III. Assign Data Per Thread

Allocate memory for the structure and associate it with the TLS index using TlsSetValue.

IV. Access TLS Data

Retrieve the data anywhere in the thread using TlsGetValue.

V. Cleanup Per Thread

Free the allocated memory when the thread exits.

VI. Free the TLS Index

Release the TLS index after all threads are finished.

3. TLS Rules That Matter

I. Isolation Advantage

TLS data is thread-private, so locking is unnecessary for that data.

II. Responsibility Reminder

TLS does not manage memory for you — allocation and cleanup are still your job.

III. Common Failure

Forgetting cleanup leads to silent memory leaks that are difficult to trace.

FINAL MENTAL MODEL ⚙️

1. Core Ideas to Remember

- **Persistent thread** → always ready
- **Event-driven signaling** → no wasted CPU
- **WndProc** → coordinator
- **Worker thread** → sleeper
- **TLS** → per-thread memory, not shared state

This is **real WinAPI architecture**, not textbook decoration.

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

1. Understanding the Need for TLS

I. Shared Memory Reality in Multithreaded Programs

In a multithreaded application, threads typically share:

- Global variables
- Heap-allocated memory

This shared access is powerful, but it also introduces risk.

II. The Core Problem TLS Solves

Some data **must not be shared**, even though the code accessing it is identical across threads. Examples include:

- Per-thread counters
- Thread-specific state or context
- Temporary buffers used during calculations

If such data is shared, threads can overwrite each other's values, leading to corruption and unpredictable behavior.

III. Why Locks Are Not Always the Answer

Using critical sections or mutexes for every access:

- Adds overhead
- Complicates logic
- Can still lead to deadlocks if misused

TLS avoids this entirely by giving **each thread its own private copy of data**.

IV. What TLS Provides

Thread Local Storage allows each thread to associate data with itself, rather than with the process as a whole.
Each thread sees **only its own value**, even though all threads use the same TLS index.

2. Conceptual Model of TLS

I. TLS as a Per-Thread Slot Table

You can think of TLS as:

- A table indexed by a TLS index
- Each thread has its own version of this table

The same index points to **different data** depending on which thread is running.

II. Key Design Advantage

- No synchronization required for TLS data
- No accidental cross-thread modification
- Clean separation of thread-specific state

3. Windows API Functions for TLS Management

I. TlsAlloc

Allocates a free TLS index.

- The returned index is process-wide
- It acts as an identifier for thread-local data

This is typically done once during program initialization.

II. TlsSetValue

Associates a value with a TLS index **for the calling thread only**.

- The value is usually a pointer to a dynamically allocated structure
- Other threads using the same index will not see this value

This step is usually performed when a thread starts.

III. TlsGetValue

Retrieves the value associated with a TLS index **for the calling thread**.

- Safe to call from anywhere in the thread
- Returns NULL if no value has been set

This allows thread-specific data to be accessed without passing parameters through function calls.

IV. TlsFree

Releases a previously allocated TLS index.

- Should be called only after all threads are done using it
- Does not automatically free per-thread memory

Freeing the TLS index too early can cause undefined behavior.

4. Lifetime and Responsibility Considerations

I. Memory Management Is Manual

TLS does **not** manage memory for you.

- You allocate memory for each thread
- You must free it when the thread exits

Failure to do so results in memory leaks that are easy to miss.

II. Thread Exit Cleanup

Each thread should:

- Retrieve its TLS value
- Free any allocated memory
- Optionally clear the TLS slot

This is commonly done just before the thread terminates.

5. When TLS Is the Right Tool

I. Ideal Use Cases

- Per-thread error information
- Thread-specific caches
- Independent execution contexts
- Libraries that must be thread-safe without global locks

II. When TLS Is Overkill

- Data that must be shared anyway
- Simple single-threaded applications
- Cases where passing parameters is simpler and clearer

6. Key Takeaways 🧠

I. What TLS Gives You

- True thread isolation
- Cleaner multithreaded design
- Reduced synchronization complexity

II. What TLS Does Not Do

- It does not replace synchronization for shared data
- It does not manage memory automatically
- It does not protect poorly designed thread logic

III. Mental Rule

Shared data → synchronize
Thread-specific data → TLS

7. Example Code

(Implementation follows to demonstrate allocation, assignment, access, and cleanup of TLS data in a multithreaded WinAPI application.)

```
// Allocate a TLS index for thread-specific data
DWORD dwTlsIndex = TlsAlloc();

// Within each thread:
PDATA pdata = (PDATA)GlobalAlloc(GPTR, sizeof(DATA));
pdata->a = /* thread-specific value */;
pdata->b = /* another thread-specific value */;
TlsSetValue(dwTlsIndex, pdata);

// Access thread-local data:
PDATA pdata = (PDATA)TlsGetValue(dwTlsIndex);
// Use pdata->a and pdata->b as needed

// Before thread termination:
GlobalFree(TlsGetValue(dwTlsIndex));

// When all threads using the TLS index are done:
TlsFree(dwTlsIndex);
```

THREAD LOCAL STORAGE (THE EASY WAY)

We previously learned that Global variables are dangerous (shared by everyone) and Local variables are temporary (die when the function ends).

Thread Local Storage (TLS) is the magic middle ground: A global variable that gives each thread its own private copy.

The old way (using TlsAlloc, TlsGetValue) is painful. Microsoft gave us a shortcut.

1. The Magic Keyword: __declspec(thread)

This is a compiler trick that handles all the hard work for you. You don't need to call API functions. You just declare the variable.

The Syntax:

```
__declspec(thread) int iGlobal = 1; // Lives globally, but unique per thread
```

How it works:

- **Thread A** reads iGlobal, it sees **1**. It adds 5. Now it sees **6**.
- **Thread B** reads iGlobal, it still sees **1**. It adds 100. Now it sees **101**.
- **Thread A** reads iGlobal again, it still sees **6**.

2. Initialization and Usage

The compiler automatically gives every new thread its own fresh copy of the variable when the thread starts.

Example 1: Simple Integer

```c
#include <windows.h>
#include <stdio.h>

// 1. Declare it with the magic keyword
__declspec(thread) int iCount = 0;

void ThreadFunc() {
    // 2. Use it like a normal variable
    iCount++;
    printf("Thread ID: %d, Count: %d\n", GetCurrentThreadId(), iCount);
}
// If you run 3 threads, they ALL print "Count: 1". None of them affect the others.
```

Example 2: Complex Structs (Pointers) You can use it for pointers too, but you must allocate the memory yourself inside the thread.

```c
typedef struct { int id; char* name; } UserData;

// The POINTER is thread-local.
// Thread A has its own pointer, Thread B has its own pointer.
__declspec(thread) UserData* pData = NULL;

void ThreadFunc() {
    // Each thread allocates its own memory block
    pData = (UserData*)malloc(sizeof(UserData));
    pData->id = GetCurrentThreadId();
    // ... use it ...
}
```

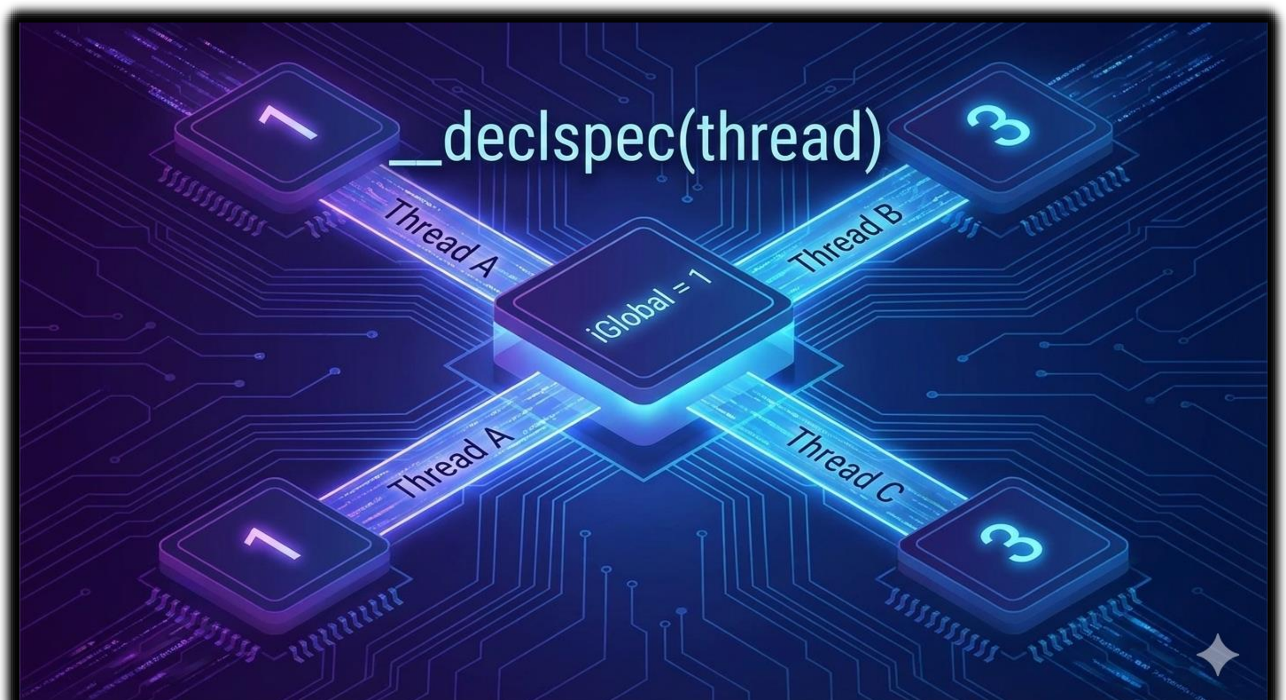
3. The Catch (Limitations)

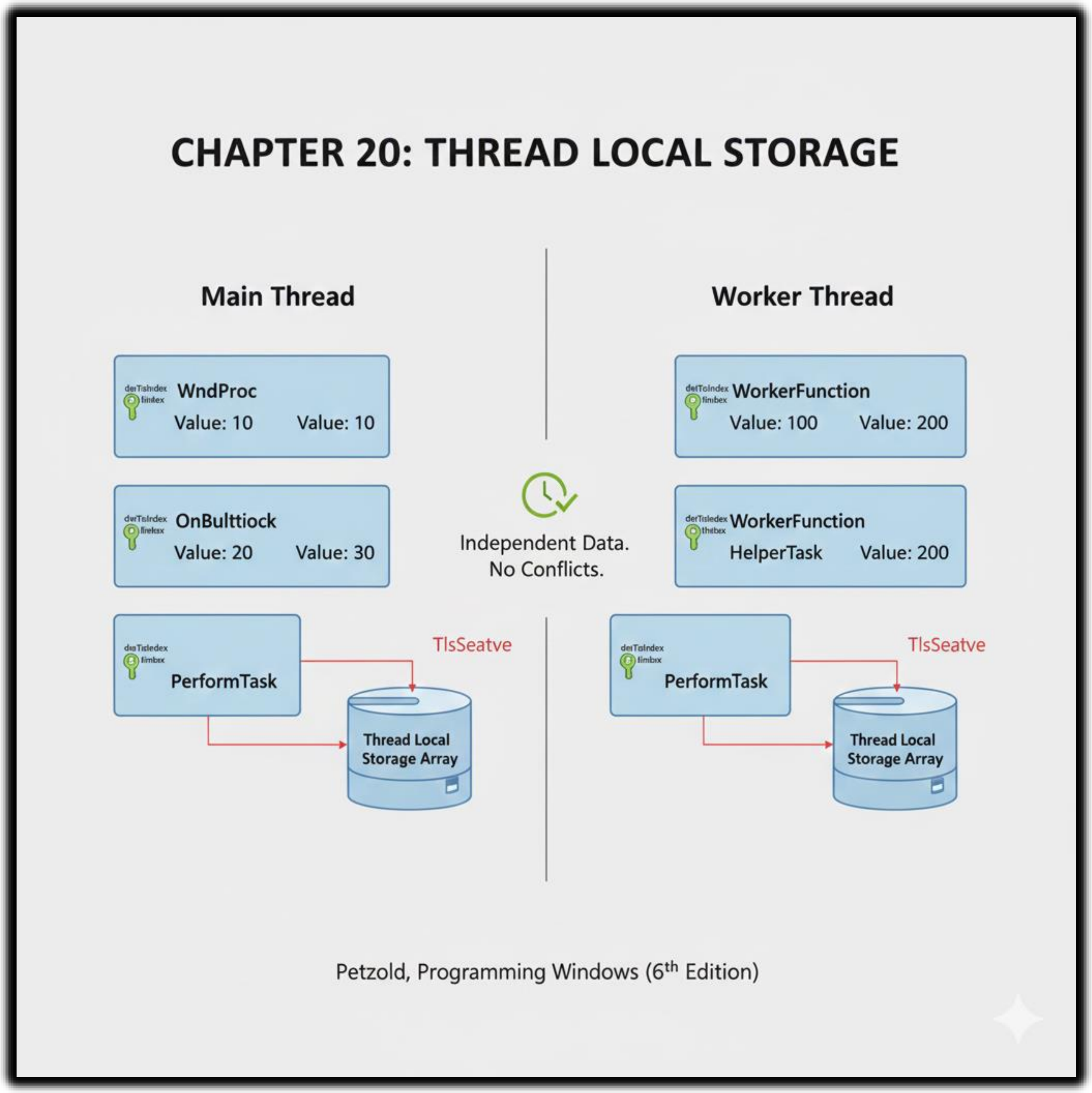
This magic comes with rules:

1. **Microsoft Only:** This is not standard C++. It won't work on Linux or GCC (unless they have similar extensions).
2. **No Complex Constructors:** In C++, you generally cannot use this for classes that need complex initialization code (constructors/destructors) to run automatically. It works best for simple types (int, float, pointers).
3. **DLL Issues:** If you are writing a DLL, __declspec(thread) can cause crashes on older Windows versions (pre-Vista) if the DLL is loaded dynamically (LoadLibrary).

Summary Checklist for Chapter 20

1. **Multithreading** allows apps to do two things at once (UI + Math).
2. **_beginthread** is better than CreateThread for C programs.
3. **Race Conditions** kill programs. Use **Critical Sections** to stop them.
4. **Events** allow threads to signal each other ("I'm done!").
5. **TLS (__declspec(thread))** gives every thread its own private "global" variable.




CHAPTER 20: THREAD LOCAL STORAGE
Main Thread
Worker Thread
WndProc
Value: 10
Value: 10
OnBulttiock
Value: 20
Value: 30
PerformTask
TlsSeatve
Thread Local Storage Array
WorkerFunction
Value: 100
Value: 200
WorkerFunction
HelperTask
Value: 200
PerformTask
TlsSeatve
Thread Local Storage Array
Independent Data.
No Conflicts.
Petzold, Programming Windows (6th Edition)