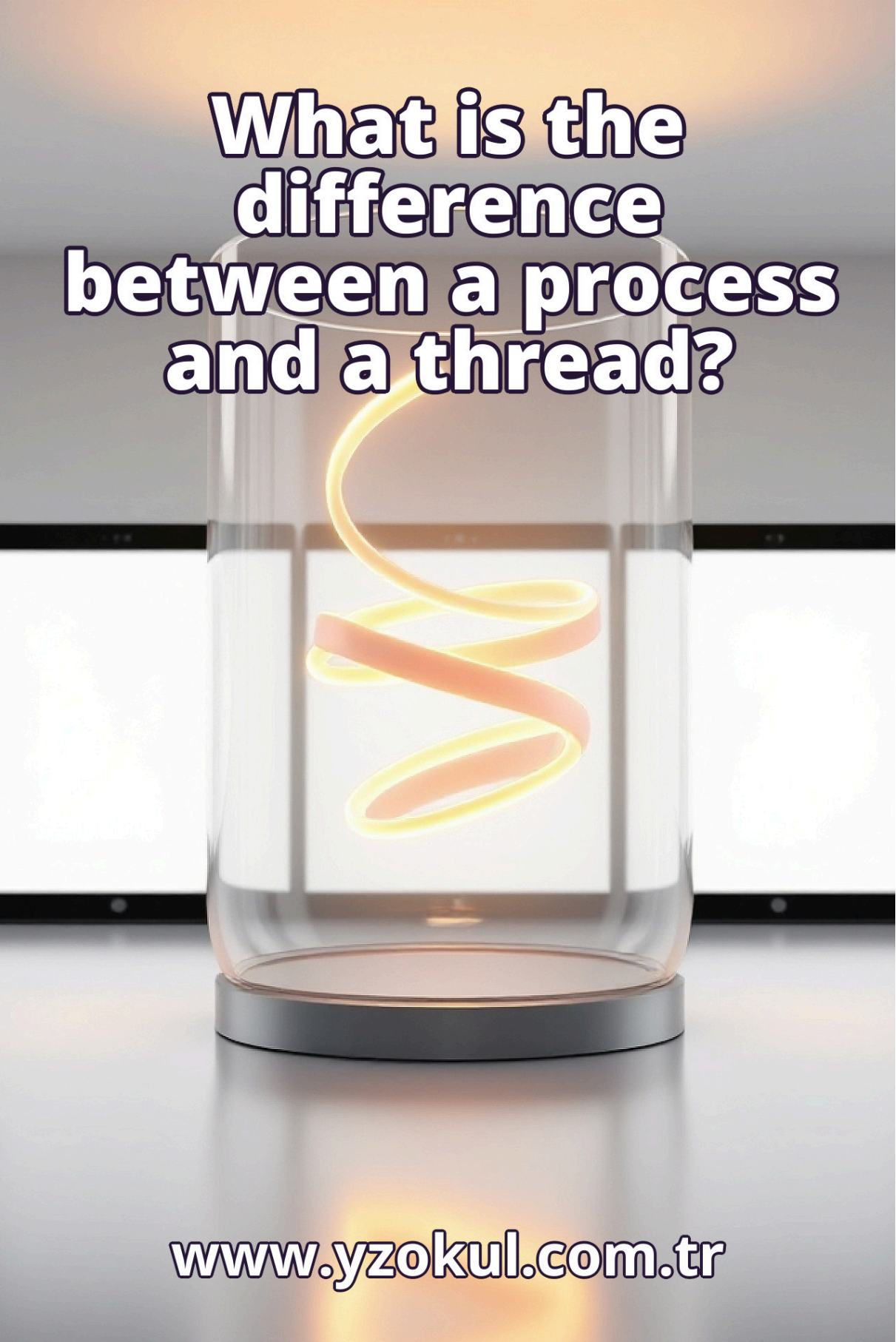
What is the difference between a process and a thread?



www.yzokul.com.tr

What is the difference between a process and a thread?



What is the difference between a process and a thread? The short answer: a process is a running program, and a thread is one execution flow inside it. The operating system manages both. This book explains each idea step by step, starting from zero, so you can see exactly how they differ and how they work together.



A program is a set of instructions stored on disk, like a recipe that is not yet being followed. Nothing is running yet at this point. The instructions just sit there, waiting. For example, a text editor installed on your computer is a program. It only becomes active when you start it.



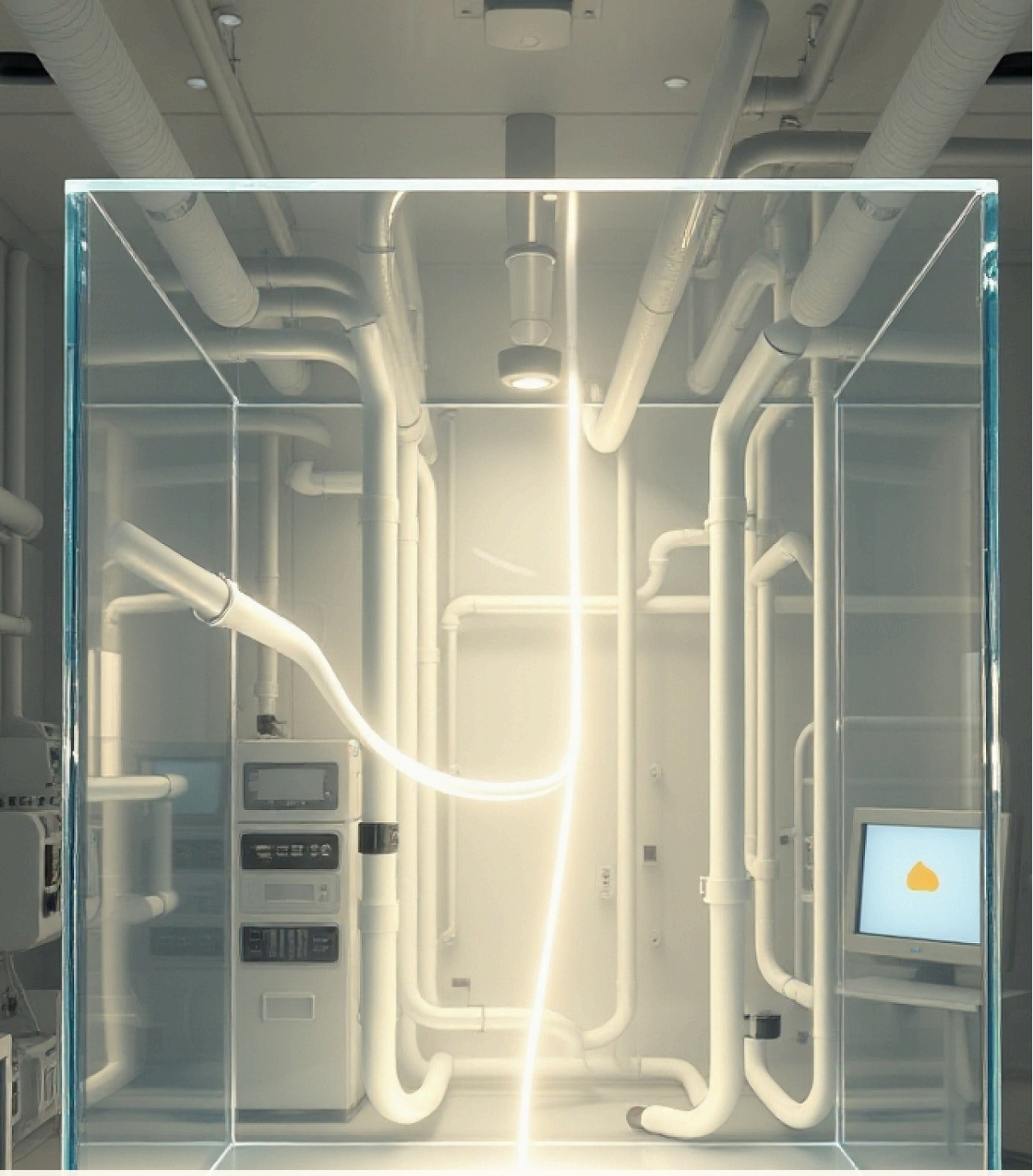
When a program starts running, the operating system turns it into a process. A process is a program in action, with its own memory and state. The operating system loads the instructions, gives the process memory, and begins executing. For example, opening your text editor creates a process that you can now use.



The operating system is the manager of all processes. It keeps track of every process and decides when each one runs. It gives each process its own space so processes do not step on each other. For example, your text editor and your web browser run as separate processes, each with its own memory, and the operating system switches between them.



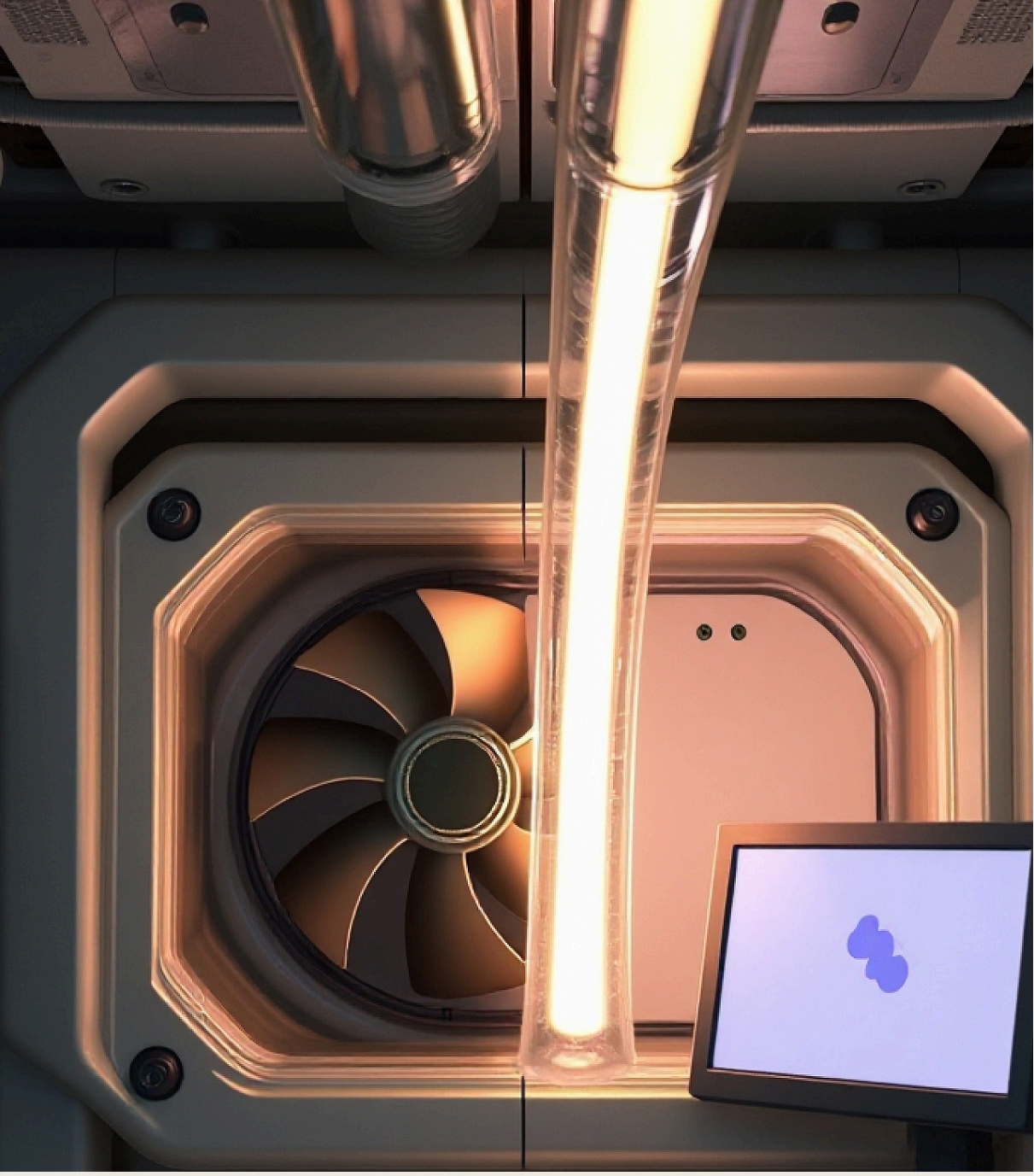
A single process can contain several execution flows running inside it. Each of these flows is called a thread. Think of a process as a workshop and threads as workers inside it, each following instructions. For example, a music player process might have one thread for playing sound and another for updating the screen. A process always has at least one thread.



A thread is one path of execution within a process, like one worker following instructions inside the same workshop. It moves through the program's instructions one at a time. A process always has at least one thread, and it can have many. For example, when you start a simple calculator, it begins with one thread that handles your button presses.



Picture a thread moving through instructions one at a time, from start to finish. It reads an instruction, does the work, then moves to the next. This makes clear that a thread is an active flow, not a stored file. For example, a thread in a calculator app reads a number, waits for you to press a button, then shows the result.



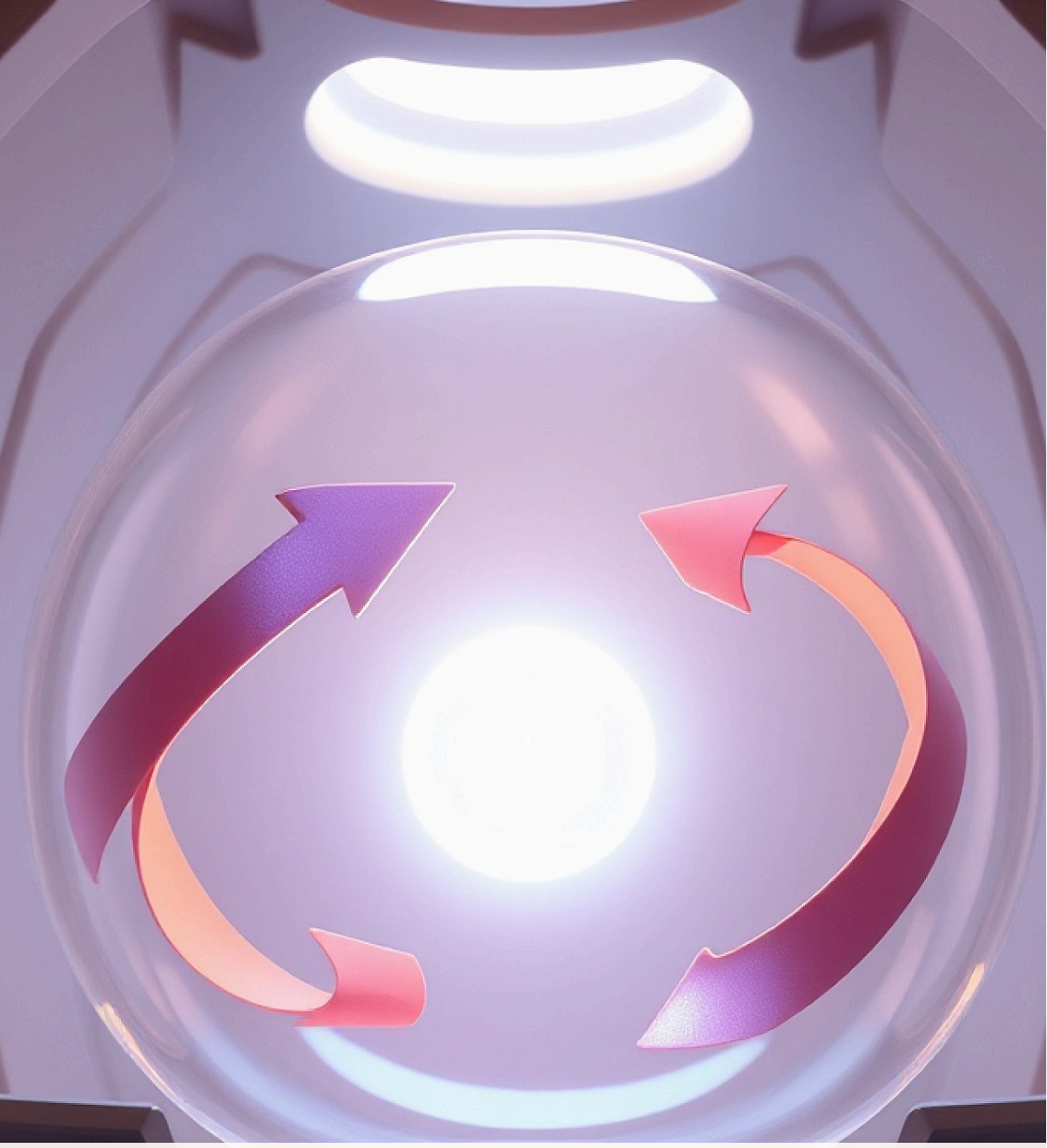
Two threads in the same process can do different jobs at the same time. One thread might handle the screen while another does background work. This shows why one process may need several threads. For example, a video player uses one thread to draw the picture and another to read the video file from disk, so the picture stays smooth.



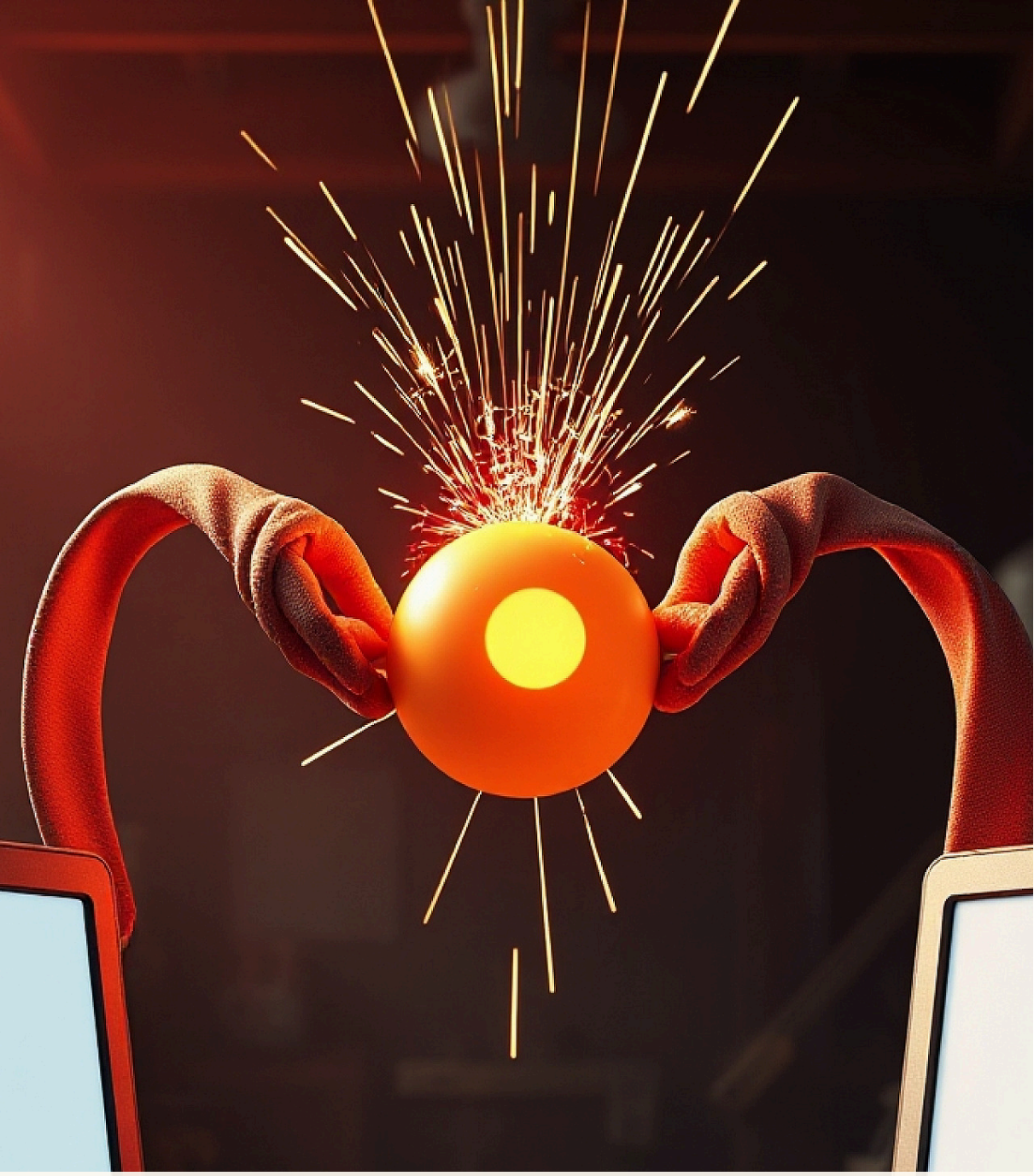
Threads in the same process share the process's memory and resources. They can read and use the same data without making copies. This is called memory sharing. For example, two threads in a music player can both access the list of songs already loaded in memory, so one thread can play a song while another shows its title.



Different processes have their own separate memory spaces. One process cannot simply reach into another process's data. The operating system enforces this separation. For example, your web browser cannot directly read the memory of your text editor. If they need to exchange information, they must use a special channel provided by the operating system.



Sharing is convenient because threads can pass information quickly and work on the same task together. They do not need to copy data or ask the operating system for help. This makes threads lighter and faster to coordinate than separate processes. For example, two threads in a photo editor can work on the same image at once, one applying a filter while the other updates the preview.



When two threads touch the same data at the same time, the result can become wrong or confusing. One thread might change a value while the other is still reading it. Shared data must be accessed carefully and in an orderly way. For example, if two threads in a banking app update the same balance at once, the final amount could be incorrect.



Access to shared data must be managed so threads take turns instead of colliding. A common way is to use a lock: a thread locks the data while it works, and other threads wait until it unlocks. This keeps the shared memory useful without causing errors. For example, a thread updating a shopping cart can lock the cart, change the quantity, then unlock it so another thread can safely read it.



A common mistake is to treat a thread as the same thing as a process. A thread lives inside a process; the process is the container, and the thread is a flow within it. One process can have many threads, but each thread belongs to exactly one process. For example, a web browser is one process, and each tab may run in its own thread inside that process.



Another common mistake is to assume shared memory is always harmless. Sharing brings speed but also requires careful management to avoid clashes. Without care, threads can overwrite each other's changes. For example, two threads adding items to the same list at the same time might lose an item. Always use proper coordination when threads share data.



To sum up: a process is a running program managed by the operating system, and a thread is one execution flow inside it. Threads in the same process share memory, which is convenient but must be managed carefully. Processes have separate memory, while threads share theirs. Remember: the process is the container, and the thread is the flow within it.