Of course. When a program in Linux reads a file, it's a well-defined process that bridges the gap between the user's request and the physical data on a storage device.[1] The process involves a conversation between the user's program (user space) and the operating system's kernel.

Here are the steps followed:

Step 1: The open() System Call 📞

Everything starts when a program needs to access a file. It doesn't just grab it from the disk; it must first ask the kernel for permission and a handle.

1. **Application Request:** A program, like `cat` or your own Python script, calls the `open()` function.[2] For example: `open("/home/user/report.txt", O_RDONLY)`.[3]

2. **System Call:** This function call is a wrapper that triggers a **system call**, transferring control from the user program to the Linux kernel. The kernel now takes over.

Step 2: Path Traversal (Namei) 🗺️

The kernel receives the path string /home/user/report.txt and must find the file it represents. This process is called path traversal or namei (name to inode).

1. **Start at the Root:** The kernel starts at the root directory (/). It knows the inode number of the root directory.
2. **Parse Component by Component:** The kernel breaks the path down: home, user, report.txt.
3. **Find home:** The kernel reads the data blocks of the root (/) directory. A directory is just a special file containing a list of filenames and their corresponding **inode numbers**. It searches this list for the entry "home" and retrieves its inode number.
4. **Find user:** Now, using the inode for home, the kernel reads its data blocks to find the entry for "user" and get its inode number.
5. **Find report.txt:** Finally, using the inode for user, it reads its data blocks to find the entry for "report.txt" and retrieves its final inode number.

Step 3: Inode Table and Permission Check 🔐

The kernel now has the unique inode number for report.txt.

1. **Inode Lookup:** The kernel uses this number as an index to find the file's **inode** in the filesystem's **inode table** on the disk. The inode is a data structure that stores all the metadata about the file.
2. **Load Inode:** The inode is loaded from disk into memory. It contains crucial information:
    - File type (e.g., regular file, directory)
    - Permissions (read, write, execute for owner, group, and others)
    - Owner's user ID (UID) and group ID (GID)
    - File size in bytes[4]
    - Timestamps (last accessed, last modified)
    - **Pointers to the actual data blocks** on the disk where the file's content is stored.
3. **Permission Check:** The kernel checks the file's permissions (from the inode) against the user's credentials and the requested access mode (O_RDONLY). If the user doesn't have read permission, the open() call fails and returns a "Permission Denied" error.

Step 4: Allocating a File Descriptor 🎫

If the path is valid and permissions are granted, the kernel creates several entries to manage the open file.

1. **System-Wide Open File Table:** The kernel creates an entry in a system-wide table of all open files. This entry contains information like the current read/write offset (initially 0) and a pointer to the in-memory inode.
2. **Per-Process File Descriptor Table:** The kernel then adds an entry to the specific process's own file descriptor table. This entry points to the entry in the system-wide open file table.
3. **Return File Descriptor:** The index of this entry in the per-process table is a small integer called the **file descriptor** (e.g., 3, 4, 5...). This number is returned to the user program by the open() system call.

From now on, the program doesn't use the file's name; it uses this simple integer file descriptor to refer to the open file.

Step 5: The read() System Call 📖

The program now has a file descriptor and can read the file's contents.

1. **Application Request:** The program calls read(fd, buffer, count), providing the file descriptor (fd), a memory buffer to store the data, and the number of bytes (count) it wants to read.[5]

2. **Kernel Action:** This triggers another system call. The kernel uses the fd to quickly look up the file in the process's file descriptor table, which leads it to the system-wide table entry.
3. **Locate Data:** From the system-wide table entry, the kernel gets the current file offset and the in-memory inode. Using the **pointers in the inode** and the offset, it calculates exactly which data blocks on the disk it needs to read.
4. **Disk to Page Cache:** The kernel issues commands to the disk controller to read the required data blocks from the physical disk into the kernel's **page cache** (a part of RAM). Using the cache prevents re-reading the disk for frequently accessed data.
5. **Cache to User Buffer:** The kernel copies the data from its page cache into the buffer that the user's program provided.
6. **Update Offset:** The kernel updates the file offset in the system-wide open file table by the number of bytes that were successfully read.[6]

7. **Return Value:** The read() call returns the number of bytes copied into the buffer.[7] This value might be less than count if the end of the file was reached.[8] A return value of 0 indicates the end of the file.[9]

Step 6: The close() System Call 👋

When the program is finished with the file, it calls close(fd).[10]

1. **Resource Cleanup:** The kernel receives the file descriptor and removes the corresponding entry from the per-process file descriptor table.
2. **Decrement Link Count:** It decrements the link count on the entry in the system-wide open file table. If the count reaches zero (meaning no other process has this file open), that entry is also removed, and the in-memory inode may be freed. This ensures system resources are properly managed.