How do apps interact with Kernel? → using system calls

Example: mkdir OS

- mkdir indirectly calls the kernel and asks the file management module to create a new directory.
- mkdir is just a wrapper of actual system calls.
- mkdir interacts with the kernel using system calls.

Example: Creating a process

- User executes a process (User space).
- Gets system call (US).
- Executes system call to create a process (KS).
- Returns to US.

Transitions from US to KS are done by software interrupts.

System calls are implemented in **C**.

A system call is a mechanism using which a user program can request a service from the kernel,
which it does not have the permission to perform.

User programs typically do not have permission to perform operations like accessing I/O devices
or communicating with other programs.

Types of System Calls

1) Process Control
a. end, abort
b. load, execute
c. create process, terminate process
d. get process attributes, set process attributes
e. wait for time
f. wait event, signal event
g. allocate and free memory

2) File Management
a. create file, delete file
b. open, close
c. read, write, reposition
d. get file attributes, set file attributes

3) Device Management
a. request device, release device
b. read, write, reposition
c. get device attributes, set device attributes
d. logically attach or detach devices

4) Information Maintenance
a. get time or date, set time or date
b. get system data, set system data
c. get process, file, or device attributes
d. set process, file, or device attributes

Examples in Windows and Unix

Category	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Management	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
	SetFileSecurity()	chmod()
	InitializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()
Device Management	SetConsoleMode(), ReadConsole(), WriteConsole()	ioctl(), read(), write()
Information Management	GetCurrentProcessID(), SetTimer(), Sleep()	getpid(), alarm(), sleep()
Communication	CreatePipe(), CreateFileMapping(), MapViewOfFile()	pipe(), shmget(), mmap()

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *src, *dest;
    char ch;
    src = fopen("input.txt", "r");
    if (src == NULL) {
        printf("Cannot open source file.\n");
        return 1;
    }
    dest = fopen("output.txt", "w");
    if (dest == NULL) {
        printf("Cannot open destination file.\n");
        fclose(src);
        return 1;
    }
    while ((ch = fgetc(src)) != EOF) {
        fputc(ch, dest);
    }
    printf("File copied successfully!\n");
    fclose(src);
    fclose(dest);
    return 0;
}
```

“If I print *Hello World* on the console, some **system call** must be made to actually write it.”

But the reason you don’t see **WriteConsole()** (or even direct WriteFile() sometimes) in **Process Monitor (ProcMon)** comes down to how Windows **abstracts console I/O** and how **C library output** works.

Let’s unpack this clearly and step-by-step — this is fascinating stuff.

🧩 Step 1: What your printf("Hello World"); actually does

When you write:

```
#include <stdio.h>

int main() {
    printf("Hello World\n");
    return 0;
}
```

Here’s the hidden call chain that happens in Windows:

```
printf()
  ↓
fwrite()   // from stdio
  ↓
WriteFile()   // Win32 API to write to the console handle
  ↓
NtWriteFile() // kernel-level system call
  ↓
Windows Kernel
```

So, **the actual system call is NtWriteFile()**, not WriteConsole().

⚙️ Step 2: Why WriteConsole() isn’t used here

WriteConsole() is part of the **Win32 API**, meant for programs that explicitly handle **console input/output using Windows handles** — not for standard C library output like printf.

- The **C runtime library** (msvcrt.dll or ucrtbase.dll) doesn’t use WriteConsole().
- Instead, it uses **WriteFile()** on the **console handle** (STD_OUTPUT_HANDLE).

Why? Because from the OS's perspective, the console is just another file-like output stream.

That's why you'll see **WriteFile()** events in ProcMon if you filter carefully — not WriteConsole().

🧠 Step 3: How the OS knows that the target is the console

When your program starts, the OS gives it three pre-opened handles:

1. STDIN → standard input (usually keyboard)
2. STDOUT → standard output (usually console)
3. STDERR → standard error (usually console)

So when printf() writes, it's simply writing bytes to the handle STDOUT —
and Windows handles the rest internally.

If you redirect output, e.g.:

a.exe > output.txt

Then STDOUT points to a file — and the *same system call chain* still works, because it's all unified as file I/O.

That's the beauty of the design.

🔍 Step 4: How to *see* what really happens in ProcMon

Here's how you can actually **trace it**:

1. Open **ProcMon**
2. Apply filters:
3. Process Name is a.exe
4. Operation is WriteFile
5. Run your program from **cmd** (not from inside ProcMon, but after filters are applied)

You'll now see entries like:

WriteFile C:\Windows\System32\conhost.exe SUCCESS Length: 12

That's your printf("Hello World\n"); writing to the console device through the console host process (conhost.exe).

🖥️ Step 5: Bonus — What is conhost.exe?

In Windows, **conhost.exe (Console Host)** is the process responsible for rendering text on the Command Prompt window.

When you write to the console handle:

YourApp → kernel32.dll → conhost.exe

So you may see **no WriteConsole()** because:

- Your program never calls it.
- The console rendering is delegated to conhost.exe, not directly visible as your app's system call.

In newer Windows (10+), console I/O is heavily optimized — often done through **pseudo-handles and internal buffering**, which may bypass visible system calls like WriteConsole().

🧩 Step 6: (Optional) For verification — call WriteConsole() explicitly

If you really want to *see* WriteConsole() in action, modify your program to explicitly use it:

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

int main() {
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    DWORD written;
    WriteConsole(hConsole, "Hello via WriteConsole!\n", 25, &written, NULL);
    return 0;
}
```

Now, when you run this version, you'll see **WriteConsole()** (and possibly its internal calls like NtDeviceIoControlFile) in Process Monitor.

✅ In summary

Concept	Explanation
printf()	Goes through C runtime, not Win32 console API
Actual system call	NtWriteFile (writing to STDOUT handle)

Concept	Explanation
WriteConsole()	Used only when you explicitly call it
Why not visible	Because C runtime treats console as a regular file
How to trace	Filter WriteFile operations in ProcMon
Console renderer	conhost.exe handles screen display

So in short:

You didn’t see WriteConsole() in ProcMon because printf() doesn’t use it — it uses WriteFile() on the console handle instead. The actual kernel-level system call under the hood is NtWriteFile.

Class 8: Kernel Data Structures

Topic: Kernel Data Structures – Essential for Operating System Implementation

The topic of **Kernel Data Structures** is central to the implementation of operating systems. This note describes several fundamental data structures used extensively in OS kernels.

---

1. Lists, Stacks, and Queues

1.1 Arrays

- **Definition:** A simple data structure where each element can be accessed directly.
- **Limitations:**
    - Inefficient when storing items of varying sizes.
    - Removing an item requires shifting other items to preserve order.
- **Example in OS:** Main memory is essentially organized as an array.

1.2 Lists

- **Definition:** A collection of data values stored sequentially. Items are accessed in order, unlike arrays.
- **Implementation:** Typically implemented as **linked lists**, which link items via pointers.
- **Advantages:**
    - Can handle items of varying sizes.
    - Allows easy insertion and deletion.
- **Disadvantages:** Linear-time search; accessing a specific item may require traversing all n elements.
- **Uses:** Kernel algorithms or building other structures like stacks and queues.
- **Types of Linked Lists:**

1. **Singly linked list:** Each item points to its successor.
2. **Doubly linked list:** Each item points to its predecessor and successor.
3. **Circularly linked list:** Last element points back to the first element.

- **Example in OS:** Linux kernel maintains process lists using doubly linked lists.

1.3 Stacks

- **Definition:** Sequentially ordered data structure following **LIFO (Last In, First Out)** principle.
- **Operations:**
    - **Push:** Insert an item.

    - **Pop:** Remove an item.
- **Example in OS:** Function call management – parameters, local variables, and return addresses are pushed onto the stack when a function is invoked and popped off upon return.

1.4 Queues

- **Definition:** Sequentially ordered data structure following **FIFO (First In, First Out)** principle.
- **Example in OS:**
    - Printer jobs are processed in submission order.
    - Tasks waiting for CPU execution are managed in queues.

---

2. Trees

- **Definition:** Hierarchical data structure with **parent–child relationships**.
- **Types:**
    1. **General tree:** Parent may have unlimited children.
    2. **Binary tree:** Parent has at most two children (left and right).
    3. **Binary search tree (BST):** Left child ≤ right child.
        - Worst-case search performance: O(n).
        - **Balanced BST:** Ensures at most log(n) levels → better worst-case performance.
- **Example in OS:** Linux uses **Red-Black trees** (a type of balanced BST) for CPU scheduling.

---

3. Hash Functions and Maps

3.1 Hash Functions

- **Definition:** Converts input data into a numeric value for indexing a table (typically an array).
- **Efficiency:** Retrieval can be near O(1), compared to O(n) for lists.
- **Challenge: Hash collisions** occur when two inputs map to the same table index.
    - Handled using linked lists at each table location.
    - Efficiency decreases as collisions increase.
- **Example in OS:** Used for fast process table lookup or file descriptor tables.

3.2 Hash Maps

- **Definition:** Uses hash functions to map **key → value** pairs.
- **Example in OS:** Password authentication systems map user names (key) to passwords (value).

---

4. Bitmaps

- **Definition:** String of n **binary digits** (bits) representing the status of n items.
- **Example:**
    - 0 = available, 1 = unavailable (or vice versa).
    - Used for tracking the availability of disk blocks.
- **Advantage:** Space-efficient; a single bit uses 1/8th the space of an 8-bit Boolean.
- **Example in OS:** Linux uses bitmaps to track free and allocated disk blocks.

---

5. Linux Kernel Data Structures

Data Structure	Linux Implementation	File/Reference	Example in OS
Linked Lists	Doubly linked lists	<linux/list.h>	Maintaining process lists
Queues	FIFO queues (kfifo)	kfifo.c	IO buffers, device queues
Balanced BSTs	Red-black trees	<linux/rbtree.h>	CPU scheduling trees

---

Summary:
Kernel data structures such as lists, stacks, queues, trees, hash maps, and bitmaps are essential for efficient OS operations. Linux provides specific implementations of these structures in its kernel source code.