System call Implementation

System calls are essential for enabling secure and controlled communication between user applications and system resources. They allow user programs to perform operations like file handling, memory allocation, and process control through a standardized interface. In HarmonyOS, implementing a custom system call—such as kill()—requires kernel access and development tools like DevEco Studio.

The kill() system call allows one process to send signals (e.g., SIGTERM, SIGKILL) to another, enabling controlled process termination. HarmonyOS uses a microkernel architecture, and while it supports development via emulators, real testing is limited due to hardware restrictions.

HarmonyOS is not natively available on Windows PCs but can be tested through DevEco Studio's emulator.
Installation may fail midway due to system requirements, incomplete SDK setup, or lack of virtualization support. So I tried to download, install and do system call implementation through DevEco but it stopped me in half way of installing DevEco.

The following steps outline how the implementation of the kill() system call would be if installing DevEco was successful :

1. **Set Up Environment**
   Install DevEco Studio and download the HarmonyOS SDK. Ensure dependencies like Java JDK, GN, and CMake are properly configured.
2. **Define the System Call**
   In the system call header file (e.g., syscalls.h), declare the function:

```
int sys_kill(pid_t pid, int signal);
```

4. **Implement the Logic**
   In a new file (e.g., kill.c), implement the functionality:

```
int sys_kill(pid_t pid, int signal) {
    struct process *target = find_process_by_pid(pid);
    if (!target) return -ESRCH;
    return send_signal(target, signal);
}
```

10. **Register the System Call**
    Add the system call to the syscall table (e.g., syscall_table.c):

```
[SYS_KILL] = sys_kill,
```

    Update macros and constants as needed.

12. **Update User Libraries**
    Create a wrapper function in libc for user access:

```
int kill(pid_t pid, int sig) {
    return syscall(SYS_KILL, pid, sig);
```

```
}
```

Include the prototype in signal.h and recompile the user libraries.

16. **Compile and Build**
    Rebuild the kernel using:

```
make kernel
```

Ensure all new files are included and resolve any compilation errors.

18. **Test the System Call**
    Write a simple C program to test the kill() functionality:

```
#include <signal.h>
int main() {
    kill(1234, SIGTERM);
    return 0;
}
```

Test cases should include valid, invalid, and unauthorized PIDs.

24. **Document the Call**
    - **Purpose**: Sends a signal from one process to another.
    - **Parameters**: pid (target process), signal (signal type).
    - **Return**: 0 on success, -1 on error.

Example:

```
if (kill(4567, SIGKILL) == -1) {
   perror("Kill failed");
}
```