# ASSIGNMENT 2A

## Question

Program for using `signal` system call.

## Solution

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

void signalHandler(int signal){
    if(signal == SIGINT){ /* When SIGINT is caught, the message is printed*/
        printf("Ha Ha, Not Stopping\n");
    }
}

int main(){
    if(signal(SIGINT, signalHandler) == SIG_ERR){
        perror("Error occured in catching SIGINT.\n");
    }

    while(1){ /* Running an infinite loop to demonstrate the interrupt*/
        printf("Team 1 process running\n");
        sleep(1);
    }

    return 0;
}
```

## Explaination

This C code demonstrates how to set up a signal handler for the **SIGINT** signal and handle it within a program.

- The code includes the necessary headers for standard library functions (<stdlib.h>), POSIX operating system services (<unistd.h>), and signal handling (<signal.h>)

### Signal Handler Function:

```c
void signalHandler(int signal){
    if(signal == SIGINT){ /* When SIGINT is caught, the message is printed*/
        printf("Ha Ha, Not Stopping\n");
    }
}
```

- This code defines a function named `signalHandler`, which will be called when the program receives a **`SIGINT`** signal (usually triggered by pressing **Ctrl+C** in

the terminal).
- The function checks if the received signal is **SIGINT** and, if so, prints a message, **"Ha Ha, Not Stopping\n"**.

### Signal Registration:

```c
    if(signal(SIGINT, signalHandler) == SIG_ERR){
     perror("Error occurred in catching SIGINT.\n");
    }
```

In this section, the code sets up a signal handler for the SIGINT signal (Ctrl+C).

- It calls the signal() function to register the `signalHandler` function to be executed when **`SIGINT`** is received.
- If an error occurs during this registration, it prints an error message using `perror().`

## Working of `signal():

The `signal()` function in C sets up a signal handler for a specific signal, but it doesn't continuously keep track of when the signal is being generated. Instead, it establishes a one-time association between the specified signal and the provided signal handler function.

Here's how it works:

1. **Registration:** When we call `signal()` with a specific signal (e.g., `SIGINT`) and a signal handler function (e.g., `signalHandler`), we are telling the operating system to invoke `signalHandler` when that particular signal is generated.

2. **Signal Handling:** When the specified signal (in this case, `SIGINT`) is generated, the operating system interrupts the normal execution of our program and transfers control to the signal handler function (`signalHandler`).

3. **Execution of Signal Handler:** The signal handler function (`signalHandler`) is executed in response to the generated signal. It can perform custom actions or respond to the signal in a specific way. In our code, it prints a message when `SIGINT` is received.

4. **Return to Normal Execution:** After the signal handler function completes its execution, control returns to the point in our program where it was interrupted. This allows our program to continue running.

5. **Repeatable:** The association between the signal and the signal handler persists until we explicitly change it. If we don't reset it or if we don't block or ignore the signal, our program can continue to receive and respond to that signal as long as it is running.

In our code, once `signal(SIGINT, signalHandler)` is called, it sets up the signal handler function `signalHandler` to be executed whenever a `SIGINT` signal is generated. The operating system takes care of monitoring for `SIGINT` signals, and when one is received (e.g., when the user presses Ctrl+C in the terminal), it invokes the `signalHandler` function.

## Why its better to use `write()` than `printf()` in the signal handler function?

1. **Async-Signal Safety:**
    - Signal handlers, including those used for **SIGINT**, should be designed to be "async-signal safe."
        - This means they should only use functions and operations that are guaranteed to be safe to use in the context of an asynchronous signal.
        - `write()` is considered async-signal safe, while `printf()` is not.
    - When a signal is delivered, it interrupts the normal flow of the program and can occur at any point in your code.
        - `printf()` involves complex operations like memory allocation and file I/O, which are not guaranteed to be safe in a signal handler context and can lead to undefined behavior.

2. **Avoiding Deadlocks:**
    - In a signal handler, it's crucial to avoid operations that may lead to deadlocks or conflicts with the rest of the program.
    - Functions like `printf()` can internally acquire locks, and if the program is already holding those locks when the signal is received, it can result in a deadlock.

3. **Minimal Resource Usage:**
    - Signal handlers should be as lightweight as possible.
    - printf() can be relatively heavy in terms of resource usage compared to write(), especially if it involves formatting and writing to multiple streams.
    - In contrast, write() is a simple system call that writes data directly to a file descriptor without any formatting or buffering.

4. **Predictable Output:**
    - `printf()` may buffer its output, which means the printed message might not appear immediately when the signal handler is invoked.
        - This can lead to confusion and make it less clear when the signal handler is executed.
    - write(), on the other hand, typically writes data immediately, providing more predictable behavior in a signal handler.

**Usage**

```
void signalHandler(int signal){
    if(signal == SIGINT){
        const char* message = "Ha Ha, Not Stopping\n";
        write(STDOUT_FILENO, message, strlen(message));
    }
}
```

In this example, `write()` is used to directly write the message to the standard output (file descriptor **STDOUT_FILENO**) in the signal handler, ensuring that the output is immediate and avoiding any potential issues associated with using `printf()` in a signal handler.