1. Modify the default behavior of ctrl + c when ctrl + c is pressed, the output is Hello World! ; To achieve the same effect by the command kill

   code:

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

void handle_sigint(int sig) {
    printf("\nHello World\n", sig);
}

int main() {
    signal(SIGINT, handle_sigint);  // set handler for SIGINT

    while (1) {
        printf("Running... Press Ctrl+C to try to interrupt.\n");
        sleep(1);
    }

    return 0;
}

```

   output:

```
Hello World
Running... Press Ctrl+C to try to interrupt.
^C
Hello World
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Running... Press Ctrl+C to try to interrupt.
Killed
reyalo@PS2023YMZSXJJQ:/mnt/c/Users/Admin/Documents/bdcom_coding_zone/OS/signal$
```

```
reyalo      4326     321  0 14:14 pts/0    00:00:00 ./a.out
reyalo      4337     516  0 14:14 pts/2    00:00:00 grep --color=auto a.out
reyalo@PS2023YMZSXJJQ:/mnt/c/Users/Admin/Documents/bdcom_coding_zone$ kill -9 4326
```

2. Write a program to simulate the try/catch/finally in C. It's able to handle both file inexistence and segment fault.

   code:

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

jmp_buf jump_buf;
int exception_code = 0;

#define TRY     if ((exception_code = setjmp(jump_buf)) == 0)
#define CATCH   else
#define FINALLY if (1)
#define THROW(code) longjmp(jump_buf, code)

#define FILE_NOT_OPENED 1001

void signal_handler(int signum) {
    THROW(signum);  // Using the macro instead of calling longjmp directly
}

int main() {
    // Set up signal handlers
    signal(SIGSEGV, signal_handler);  // Invalid memory access

    FILE *file = NULL;

    TRY {
        printf("In TRY block\n");

        // Try to open a file
        file = fopen("non_existing_file.txt", "r");
        if (file == NULL) {
            printf("File cannot be opened, throwing exception...\n");
            THROW(FILE_NOT_OPENED);  // 1 = File open error
        }

        printf("File opened successfully\n");
        fclose(file);

        // Uncomment to simulate divide by zero:
        // int x = 5 / 0;
    }
    CATCH {
```

```
    CATCH {
        if (exception_code == FILE_NOT_OPENED) {
            printf("Caught exception: File open error\n");
        } else if (exception_code == SIGSEGV) {
            printf("Caught signal: Segmentation fault\n");
        } else {
            printf("Caught unknown exception (code: %d)\n", exception_code);
        }
    }
    FINALLY {
        printf("In FINALLY block - this always runs\n");
    }

    return 0;
}

```

output:

```
reyalo@PS2023YMZSXJJQ:/mnt/c/Users/Admin/Documents/bdcom_coding_zone/OS/signal$ rr p23.c
In TRY block
File cannot be opened, throwing exception...
Caught exception: File open error
reyalo@PS2023YMZSXJJQ:/mnt/c/Users/Admin/Documents/bdcom_coding_zone/OS/signal$
```

3. Implement the function sleep() and a periodic timer using alarm() and pause() simulations. And wake up the process from the sleep state in advance in the timer periodically

   code:

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


#define min(a, b) ((a) < (b) ? (a) : (b))
#define max(a, b) ((a) > (b) ? (a) : (b))


volatile sig_atomic_t time_rem = 0;
int period = 6; // Period for the alarm in seconds


void alarm_handler(int signum) {
    alarm(period);

}

void my_sleep(int seconds) {
    time_rem = seconds;

    signal(SIGALRM, alarm_handler);


    alarm( min(time_rem, 6));               // Set the first alarm for the smaller of time_rem time or

    while (time_rem > 0){
        pause();                            // Wait for next alarm signal

        period = min(time_rem, period);     // Update the period for the next alarm
        time_rem -= period;                 // Decrease the remaining time by the period

        printf("Alarm! Wake up after %d seconds... Remaining: %d seconds\n",period, time_rem);
    }
}

int main() {
    printf("Sleeping for 20 seconds with periodic alarm every 6 seconds...\n");
    my_sleep(20);
    printf("Done sleeping!\n");

    return 0;
```

Output:

```
reyalo@PS2023YMZSXJJQ:/mnt/c/Users/Admin/Documents/bdcom_coding_zone/OS/signal$ rr p33.c
Sleeping for 20 seconds with periodic alarm every 6 seconds...
Alarm! Wake up after 6 seconds... Remaining: 14 seconds
Alarm! Wake up after 6 seconds... Remaining: 8 seconds
Alarm! Wake up after 6 seconds... Remaining: 2 seconds
Alarm! Wake up after 2 seconds... Remaining: 0 seconds
Done sleeping!
reyalo@PS2023YMZSXJJQ:/mnt/c/Users/Admin/Documents/bdcom_coding_zone/OS/signal$
```

4. What issues does signal may bring to the program? Illustration

   Here are some issues signals may bring, with brief illustrations:

   I. Race Conditions :Signals can interrupt at **any time**, even during non-atomic operations.

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

int counter = 0;

void handler(int sig) {
    counter++;  // 🔥 Racy access to shared variable
}

int main() {
    signal(SIGALRM, handler);
    alarm(1);

    while (1) {
        counter++;  // 🔥 This can be interrupted
    }
}
```

   II. Non-Reentrant Functions: Only certain functions are **safe in signal handlers**

   Unsafe: printf(), malloc(), fopen()

Safe: write(), _exit()

III. Global State Corruption

```
volatile sig_atomic_t flag = 0;
void handler(int sig) { flag = 1; }
```

IV. Unmasked Signals : If not masked, signals may **interrupt critical code**.

Use sigprocmask() to temporarily block signals.

V. Nested Interrupts : Handlers can be interrupted by another signal unless handled carefully.