INPUT DEVICE MONITORING AND SIMULATION

Course Title and Code – Operating System, BCSE303L

Faculty – Prof. Vallidevi K.

Slot – F1+TF1

Names of Team Members –

1. Vansh Kumar Payala [22BAI1386]
2. Harsh Singh [22BAI1372]
3. Avi Aaryan Jeet [22BAI1380]

Introduction –

Input device monitoring, specifically focused on the mouse, involves the utilization of a device driver simulator. This software tool, often known simply as a driver simulator, is instrumental in the testing and development of device drivers. These drivers serve as crucial intermediaries, facilitating seamless communication between the operating system and hardware components like keyboards, displays, graphics cards, network adapters, and, in this context, mice. The driver simulator is essential in software development because it creates a virtual environment in which developers may mimic interactions between software applications and input devices, particularly the mouse. This simulation is critical for assessing the mouse driver's performance, dependability, and compatibility over a wide range of operating systems and hardware combinations. Extensive testing in the simulator identifies any weaknesses, errors, or conflicts, allowing developers to improve and optimise the mouse driver's code before its official release. In summary, mouse driver simulators are critical components of the software development lifecycle because they provide a controlled environment for testing and validation, as well as for resolving difficulties, enhancing stability, and assuring uniform functioning across platforms and devices.

So, in this project, we have made an Input Device monitoring system in which we will monitor the actions of an input device, i.e., a Mouse in this case. We are monitoring all the functions and movements of the mouse using the 'Libevdev' library. We are using the OS concept of threads to execute the different operations which we have to execute simultaneously. We have used C language as the coding language to built the program. This project shows the exact coordinates and functions executed by the mouse with the time stamp, i.e., what is the mouse doing every second.

Applications –

1) Development and Testing: Driver simulators offer a controlled setting for developers to craft, test, and troubleshoot device drivers. By replicating authentic hardware interactions, developers can pinpoint and address errors before deploying the driver, ensuring its stability and reliable functionality.
2) Compatibility Testing: Simulators enable developers to evaluate driver compatibility across diverse operating systems, hardware configurations, and software applications. This is essential for guaranteeing seamless performance on various platforms, minimizing the risk of system crashes or hardware malfunctions.
3) Performance Optimization: Driver simulators play a crucial role in assessing the driver's performance under different conditions. Developers can analyze how the

driver responds to varied workloads, optimizing its efficiency and responsiveness to ensure the hardware device operates smoothly and without lag.

4) Error Handling: Simulators empower developers to simulate error scenarios, such as device malfunctions or unexpected user inputs. By observing the driver's reactions in these situations, developers can enhance error handling mechanisms, ensuring the driver remains stable even in challenging circumstances.
5) Training and Skill Development: Driver simulators serve as valuable tools for training developers and IT professionals. Novices can practice driver development and troubleshooting in a risk-free virtual environment, enhancing their skills and knowledge before engaging in real-world projects.
6) Security Testing: Driver simulators assist in evaluating the security features of the driver, allowing developers to pinpoint vulnerabilities and potential security threats. By simulating diverse attack scenarios, developers can reinforce the driver's security measures, safeguarding user data and maintaining system integrity.
7) Education and Training: Driver simulators are integral educational tools in computer science and engineering courses. Students can gain practical skills in device driver development, testing methodologies, and debugging techniques in a simulated environment, preparing them for industry challenges.

OS Concept used in the code –

The code utilizes multithreading, a fundamental concept in operating systems, to concurrently monitor and process mouse events. The program creates a separate thread named mouseEventThread to handle the asynchronous input from the mouse device while allowing the main thread to perform other tasks simultaneously.

To begin with, the program defines a global variable pthread_t thread to store the identifier of the mouse event monitoring thread. The pthread_create function is then employed to create this new thread. It takes four arguments: the thread identifier (&thread), default thread attributes (specified as NULL), the function to be executed by the thread (mouseEventThread), and the argument to pass to the thread function (in this case, the file descriptor of the mouse device, denoted by &fd).

The mouseEventThread function serves as the entry point for the newly created thread. It takes the file descriptor of the mouse device as an argument, which is cast to an integer (int fd = *(int *)arg). Within this function, a loop is established to continuously read and process input events from the mouse device while the keepRunning flag is set to true.

In the mouseEventThread function, a mutex (pthread_mutex_t mutex) is initialized globally with PTHREAD_MUTEX_INITIALIZER. This mutex is crucial for thread synchronization, preventing race conditions when multiple threads attempt to access shared resources concurrently. The pthread_mutex_lock and pthread_mutex_unlock functions are employed to control access to the critical section of the code, ensuring that only one thread can execute this section at a time.

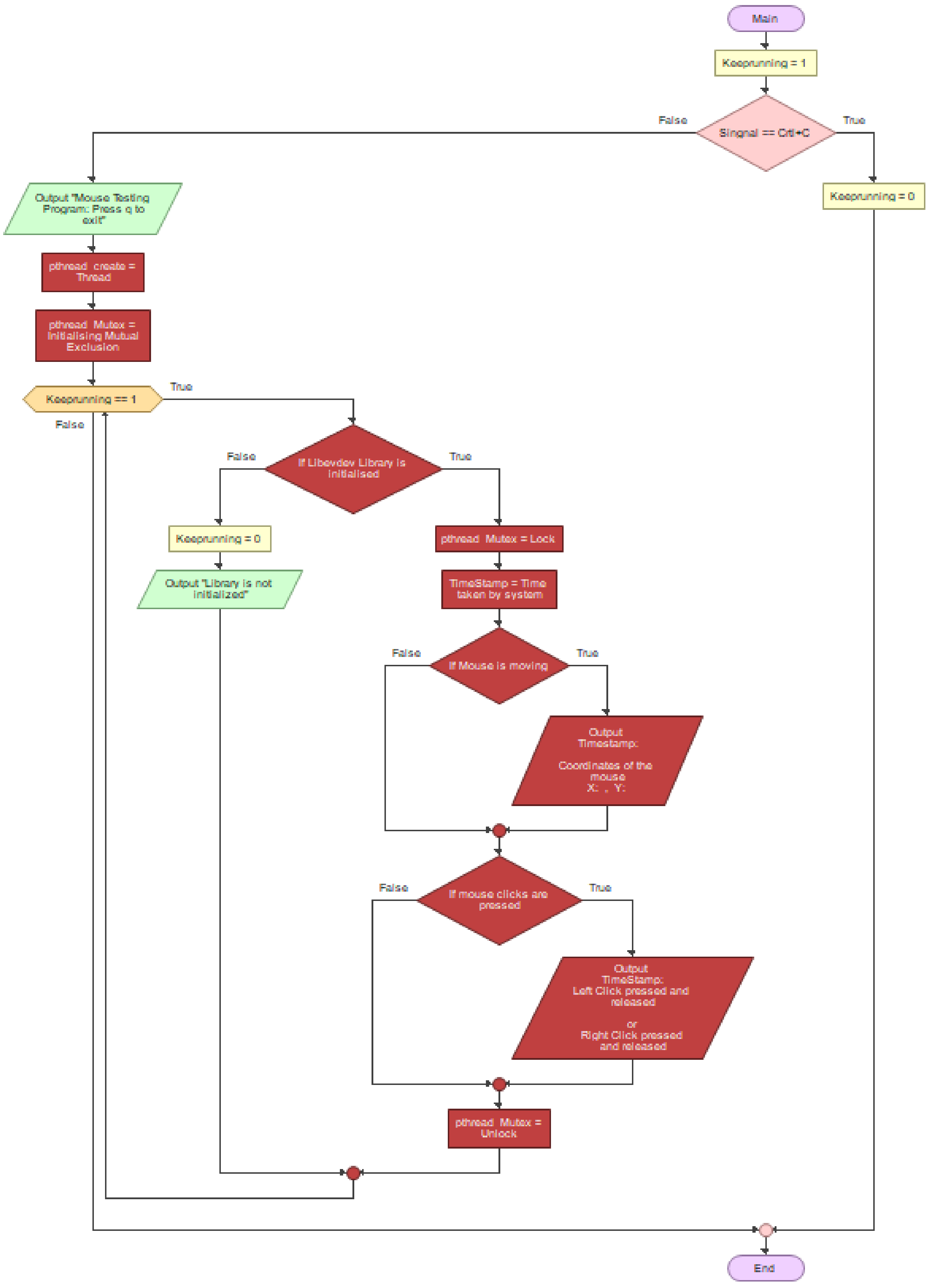
The main thread, meanwhile, continues its execution and waits for the mouse event monitoring thread to finish. This is achieved using the pthread_join function, which waits for the specified thread (in this case, thread) to terminate. The second argument (NULL) discards the exit status of the thread.

Code –

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <libevdev/libevdev.h>
#include <signal.h>
#include <pthread.h>
#include <time.h>

static volatile int keepRunning = 1;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
void intHandler(int dummy) {
  keepRunning = 0;
}
void *mouseEventThread(void *arg) {
  int fd = *(int *)arg;
  while (keepRunning) {
    struct libevdev *dev = NULL;
    int err = libevdev_new_from_fd(fd, &dev);
    if (err < 0) {
      perror("Failed to initialize libevdev");
      close(fd);
      return NULL;
    }
    struct input_event ev;
    while (keepRunning && libevdev_next_event(dev, LIBEVDEV_READ_FLAG_NORMAL, &ev) == 0) {
      pthread_mutex_lock(&mutex);
      struct timespec event_time;
      clock_gettime(CLOCK_REALTIME, &event_time);
      char timestamp[64];
      strftime(timestamp, sizeof(timestamp), "%H:%M:%S", localtime(&event_time.tv_sec));

      if (ev.type == EV_REL) {
          if (ev.code == REL_X) {
              printf("[%s] Mouse moved in X direction: %d\n", timestamp, ev.value);
          } else if (ev.code == REL_Y) {
              printf("[%s] Mouse moved in Y direction: %d\n", timestamp, ev.value);
          }
      } else if (ev.type == EV_KEY) {
          if (ev.code == BTN_LEFT) {
              if (ev.value == 1) {
                  printf("[%s] Left mouse button pressed\n", timestamp);
              } else if (ev.value == 0) {
                  printf("[%s] Left mouse button released\n", timestamp);
              }
          } else if (ev.code == BTN_RIGHT) {
              if (ev.value == 1) {
                  printf("[%s] Right mouse button pressed\n", timestamp);
              } else if (ev.value == 0) {
                  printf("[%s] Right mouse button released\n", timestamp);
              }
          }
      }
      pthread_mutex_unlock(&mutex);
    }
    libevdev_free(dev);
```

```
    }
    return NULL;
}
int main() {
    signal(SIGINT, intHandler);
    int fd = open("/dev/input/event17", O_RDONLY);
    if (fd < 0) {
        perror("Failed to open input device");
        return 1;
    }
    struct libevdev *dev = NULL;
    int err = libevdev_new_from_fd(fd, &dev);
    if (err < 0) {
        perror("Failed to initialize libevdev");
        close(fd);
        return 1;
    }
    if (!libevdev_has_event_type(dev, EV_REL) || !libevdev_has_event_code(dev, EV_REL, REL_X)) {
        printf("This device is not a mouse.\n");
        libevdev_free(dev);
        close(fd);
        return 1;
    }
    int x = 0;
    int y = 0;
    pthread_t thread;
    if (pthread_create(&thread, NULL, mouseEventThread, &fd)) {
        perror("Failed to create mouse event monitoring thread");
        return 1;
    }
    printf("Mouse testing program - Press 'q' to exit\n");
    while (keepRunning) {
        int c = getchar();
        if (c == 'q') {
            keepRunning = 0;
        }
    }
    pthread_join(thread, NULL);
    libevdev_free(dev);
    close(fd);

    return 0;
}
```

Flowchart –



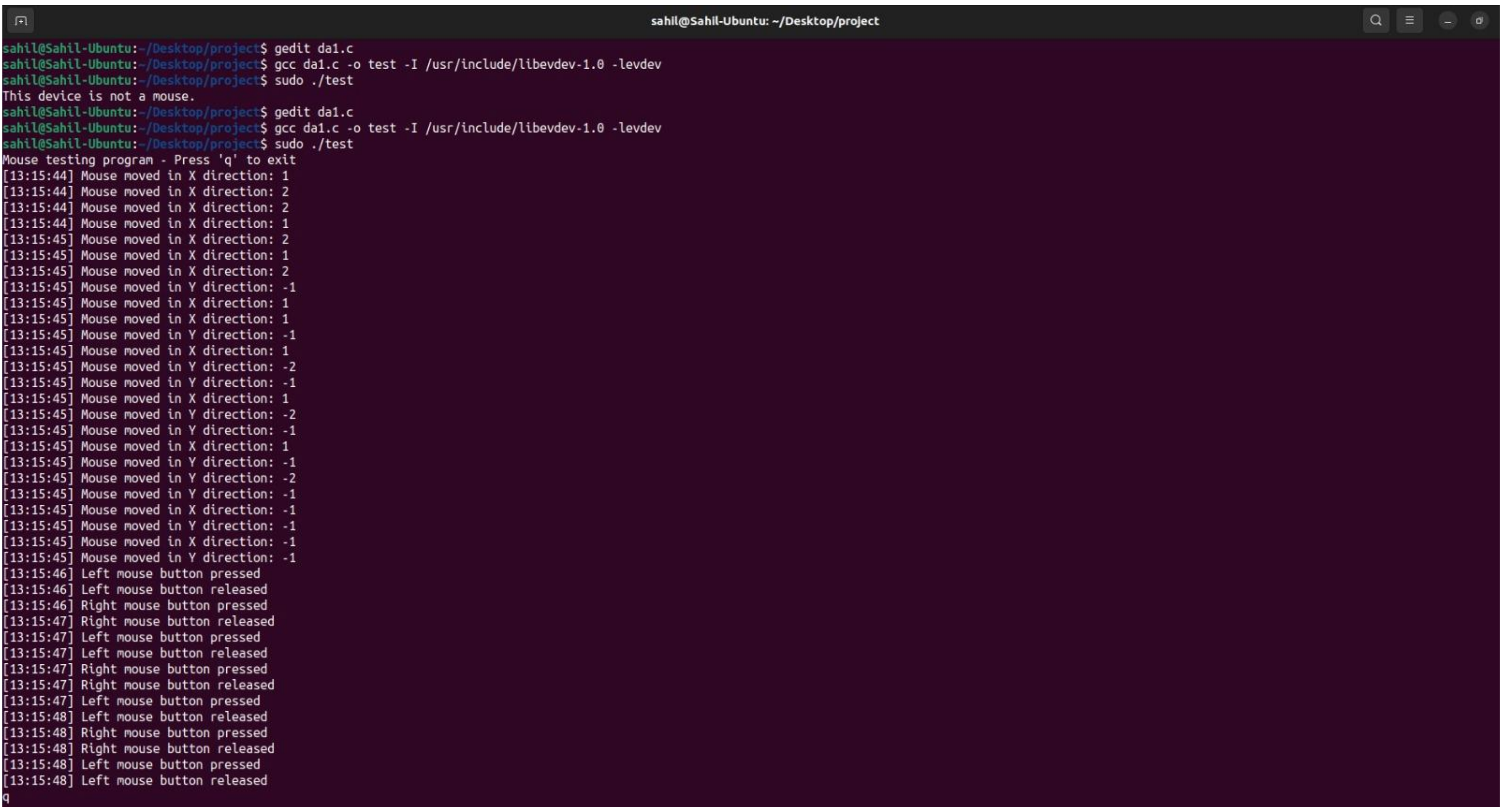
Main
Keeprunning = 1
Singnal == Crtl+C
False
True
Keeprunning = 0
Output "Mouse Testing Program: Press q to exit"
pthread_create = Thread
pthread_Mutex = Initialising Mutual Exclusion
Keeprunning == 1
True
False
If Libevdev Library is initialised
False
True
Keeprunning = 0
Output "Library is not initialized"
pthread_Mutex = Lock
TimeStamp = Time taken by system
If Mouse is moving
False
True
Output Timestamp: Coordinates of the mouse X: , Y:
If mouse clicks are pressed
False
True
Output TimeStamp: Left Click pressed and released or Right Click pressed and released
pthread_Mutex = Unlock
End

Functioning of Code **–**

1) Included Libraries:
   - This program includes necessary header files like <stdio.h> for standard input/output operations, <fcntl.h> for defines the values that can be specified for the Command and Argument parameters of the fcntl subroutine and for the Oflag parameter of the open subroutine, <libevdev.h> - It is a wrapper library for evdev devices. it moves the common tasks when dealing with evdev devices into a library and provides a library interface to the caller, <signal.h> for defining the following macros that are used to refer to the signals that occur in the system, <pthread.h> contains function declarations and mappings for threading interfaces and defines a number of constants used by those functions, <time.h> that contains definitions of functions to get and manipulate date and time information.
2) Global Variables and Signal Handling:
   - static volatile int keepRunning = 1; This variable is used as a flag to control the execution of both the main program and the mouse event monitoring thread. It is marked as volatile to indicate that its value may change due to a signal (Ctrl-C). The code will keep executing until this variable's value is 1. If the value of this variable turns 0, then the code will stop.
   - pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER; A mutex (mutual exclusion) is initialized to synchronize access to shared resources between the main program and the thread.
3) Signal Handler:
   - void intHandler(int dummy): This function is a signal handler for Ctrl-C (SIGINT). When we give the input Ctrl-C, it sets the variable keepRunning to 0, indicating that the program should exit gracefully.
4) Mouse Event Monitoring Thread:
   - void *mouseEventThread(void *arg): This function is the entry point of the thread which will continuously monitor mouse events. It takes the file descriptor of the mouse input device as an argument.
   - Inside the thread: A libevdev structure is initialized using libevdev_new_from_fd() to handle input events. A loop (while (keepRunning)) continuously reads and processes input events using libevdev_next_event(). For each event, a mutex is locked to ensure synchronized printing of event details. The timestamp is obtained using the current system time. Depending on the type and code of the event, relevant information (e.g., mouse movement or button press/release) is printed to the console. The mutex is unlocked after printing the event details.
5) Main Program:
   - int main(): The main function of the program.

- Signal handling: signal(SIGINT, intHandler): Sets up the signal handler for recognizing the signal Ctrl-C.
- Opening the mouse input device: int fd = open("/dev/input/event17", O_RDONLY): Opens the device file corresponding to the mouse input (assuming it's event17) for further simulation.
- Checking the device and initializing libevdev: A libevdev structure is created to handle input events. It checks whether the device is a mouse by checking the mouse event location. Then if the event is found to be a mouse, then we verified the presence of relative motion events like x,y coordinates of the mouse using the command EV_REL, X-axis motion REL_X, Y-axis motion REL_Y. Then we verified the functions of the mouse like left click and right click using the commands EV_KEY, BTN_LEFT and BTN_RIGHT.
- Thread creation: A new thread is created using pthread_create(). The thread runs the mouseEventThread function and receives the file descriptor of the mouse device as an argument. It creates 2 threads in which 1 thread executes the main flow of the program and for the other thread mutex is locked so that there is only 1 thread that accesses the shared resources and can execute the mouse event operations concurrently.
- User interaction: The main program displays a message indicating that it's a mouse testing program and prompts the user to press 'q' to exit. It continuously waits for user input using getchar() in a loop.
- Thread joining and cleanup: When the user presses 'q', the keepRunning flag is set to 0, indicating the program should exit. The main program waits for the mouse event monitoring thread to finish using pthread_join(). Memory allocated for the libevdev structure is freed, and the mouse input device is closed.
- The program returns 0 to indicate successful execution.

Terminal Output –



```
sahil@Sahil-Ubuntu:~/Desktop/project$ gedit da1.c
sahil@Sahil-Ubuntu:~/Desktop/project$ gcc da1.c -o test -I /usr/include/libevdev-1.0 -levdev
sahil@Sahil-Ubuntu:~/Desktop/project$ sudo ./test
This device is not a mouse.
sahil@Sahil-Ubuntu:~/Desktop/project$ gedit da1.c
sahil@Sahil-Ubuntu:~/Desktop/project$ gcc da1.c -o test -I /usr/include/libevdev-1.0 -levdev
sahil@Sahil-Ubuntu:~/Desktop/project$ sudo ./test
Mouse testing program - Press 'q' to exit
[13:15:44] Mouse moved in X direction: 1
[13:15:44] Mouse moved in X direction: 2
[13:15:44] Mouse moved in X direction: 2
[13:15:44] Mouse moved in X direction: 1
[13:15:45] Mouse moved in X direction: 2
[13:15:45] Mouse moved in X direction: 1
[13:15:45] Mouse moved in X direction: 2
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in X direction: 1
[13:15:45] Mouse moved in X direction: 1
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in X direction: 1
[13:15:45] Mouse moved in Y direction: -2
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in X direction: 1
[13:15:45] Mouse moved in Y direction: -2
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in X direction: 1
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in Y direction: -2
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in X direction: -1
[13:15:45] Mouse moved in Y direction: -1
[13:15:45] Mouse moved in X direction: -1
[13:15:45] Mouse moved in Y direction: -1
[13:15:46] Left mouse button pressed
[13:15:46] Left mouse button released
[13:15:46] Right mouse button pressed
[13:15:47] Right mouse button released
[13:15:47] Left mouse button pressed
[13:15:47] Left mouse button released
[13:15:47] Right mouse button pressed
[13:15:47] Right mouse button released
[13:15:47] Left mouse button pressed
[13:15:48] Left mouse button released
[13:15:48] Right mouse button pressed
[13:15:48] Right mouse button released
[13:15:48] Left mouse button pressed
[13:15:48] Left mouse button released
q
```

Conclusion –

In conlusion, this project effectively demonstrated the integration of operating system concepts in the development of a comprehensive mouse event monitoring system on the Linux platform. The code effectively captures and analyses input events from a particular mouse device by using the C programming language and the libevdev library. The usage of pthreads allows for concurrent execution, allowing the programme to watch mouse events asynchronously while remaining responsive. The addition of signal handling capabilities provides a layer of human involvement, allowing the monitoring threads to be gracefully terminated. Furthermore, the use of a mutex assures thread safety and avoids any race circumstances when accessing shared resources. The importance of threading, signal handling, and synchronisation in the context of operating systems is vividly highlighted in this project, which provides a realistic application of these principles in real-world circumstances.

In a larger sense, the project demonstrates the adaptability and applicability of operating system principles in tackling real-world problems. The ability to capture and understand mouse events not only serves as a core notion for developing user interfaces, but it also demonstrates the versatility of operating system concepts in a variety of applications. This project establishes the groundwork for future research and development of more complicated systems, emphasising the relevance of thread management, signal handling, and synchronisation in the creation of efficient and responsive software. It provides significant insights into the practical implementation of operating system ideas as an educational endeavour, giving students hands-on experience in designing software that communicates directly with hardware components. Overall, this project accomplishes not just its goal of monitoring mouse events, but it also serves as a practical and instructional investigation of operating system ideas in action.