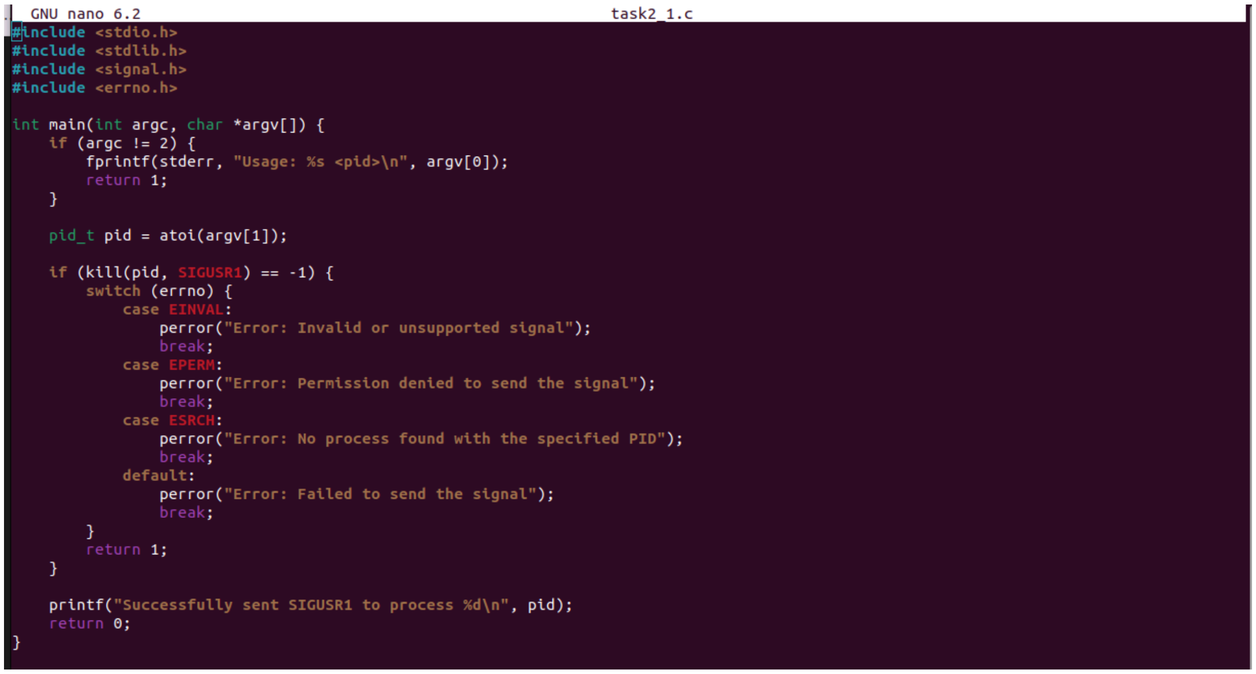
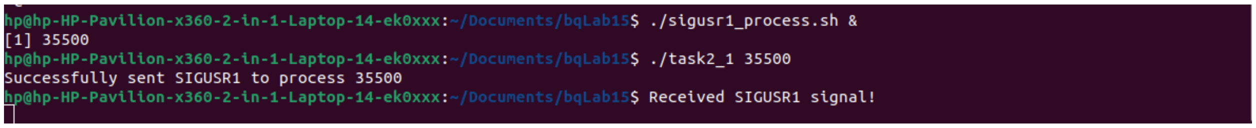
LAB 15
Breeha Qasim

TASK 2.1



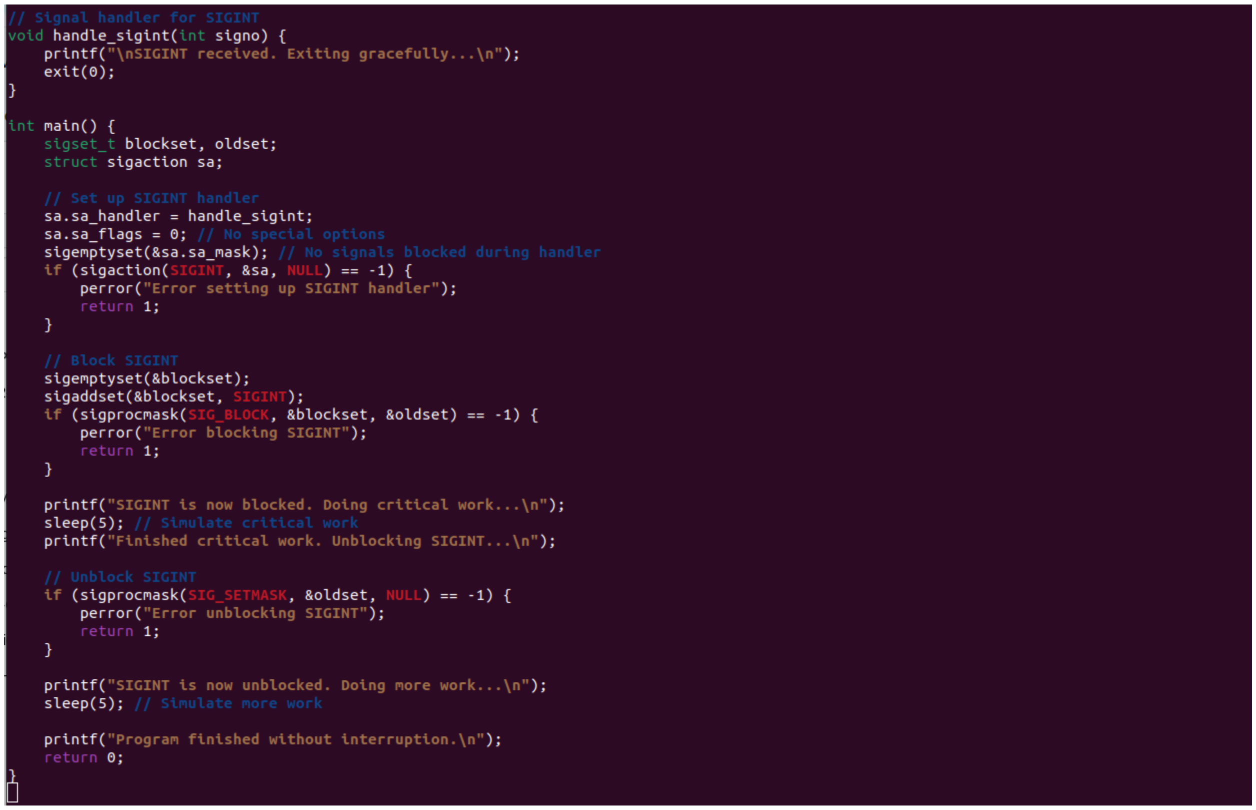
```
  GNU nano 6.2                                          task2_1.c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <errno.h>

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        return 1;
    }

    pid_t pid = atoi(argv[1]);

    if (kill(pid, SIGUSR1) == -1) {
        switch (errno) {
            case EINVAL:
                perror("Error: Invalid or unsupported signal");
                break;
            case EPERM:
                perror("Error: Permission denied to send the signal");
                break;
            case ESRCH:
                perror("Error: No process found with the specified PID");
                break;
            default:
                perror("Error: Failed to send the signal");
                break;
        }
        return 1;
    }

    printf("Successfully sent SIGUSR1 to process %d\n", pid);
    return 0;
}
```



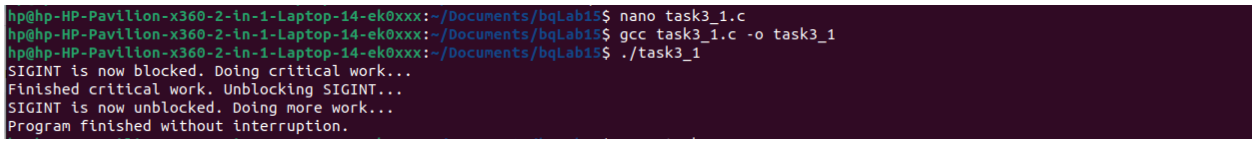
```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ ./sigusr1_process.sh &
[1] 35500
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ ./task2_1 35500
Successfully sent SIGUSR1 to process 35500
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ Received SIGUSR1 signal!
```

Explanation: The program uses the SIGUSR1 signal, a user-defined signal often used for triggering custom behavior in processes, such as configuration reloads or status updates. It also handles errors like invalid signals (EINVAL), missing processes (ESRCH), and permission issues (EPERM).

TASK 3.1



```
// Signal handler for SIGINT
void handle_sigint(int signo) {
    printf("\nSIGINT received. Exiting gracefully...\n");
    exit(0);
}

int main() {
    sigset_t blockset, oldset;
    struct sigaction sa;

    // Set up SIGINT handler
    sa.sa_handler = handle_sigint;
    sa.sa_flags = 0; // No special options
    sigemptyset(&sa.sa_mask); // No signals blocked during handler
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Error setting up SIGINT handler");
        return 1;
    }

    // Block SIGINT
    sigemptyset(&blockset);
    sigaddset(&blockset, SIGINT);
    if (sigprocmask(SIG_BLOCK, &blockset, &oldset) == -1) {
        perror("Error blocking SIGINT");
        return 1;
    }

    printf("SIGINT is now blocked. Doing critical work...\n");
    sleep(5); // Simulate critical work
    printf("Finished critical work. Unblocking SIGINT...\n");

    // Unblock SIGINT
    if (sigprocmask(SIG_SETMASK, &oldset, NULL) == -1) {
        perror("Error unblocking SIGINT");
        return 1;
    }

    printf("SIGINT is now unblocked. Doing more work...\n");
    sleep(5); // Simulate more work

    printf("Program finished without interruption.\n");
    return 0;
}
```



```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ nano task3_1.c
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ gcc task3_1.c -o task3_1
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ ./task3_1
SIGINT is now blocked. Doing critical work...
Finished critical work. Unblocking SIGINT...
SIGINT is now unblocked. Doing more work...
Program finished without interruption.
```

Explanation: This program demonstrates blocking and unblocking the SIGINT signal (Ctrl+C). It blocks SIGINT during a critical section using sigprocmask, ensuring uninterrupted execution, and then unblocks it afterward. A custom signal handler gracefully terminates the program when SIGINT is unblocked and received.

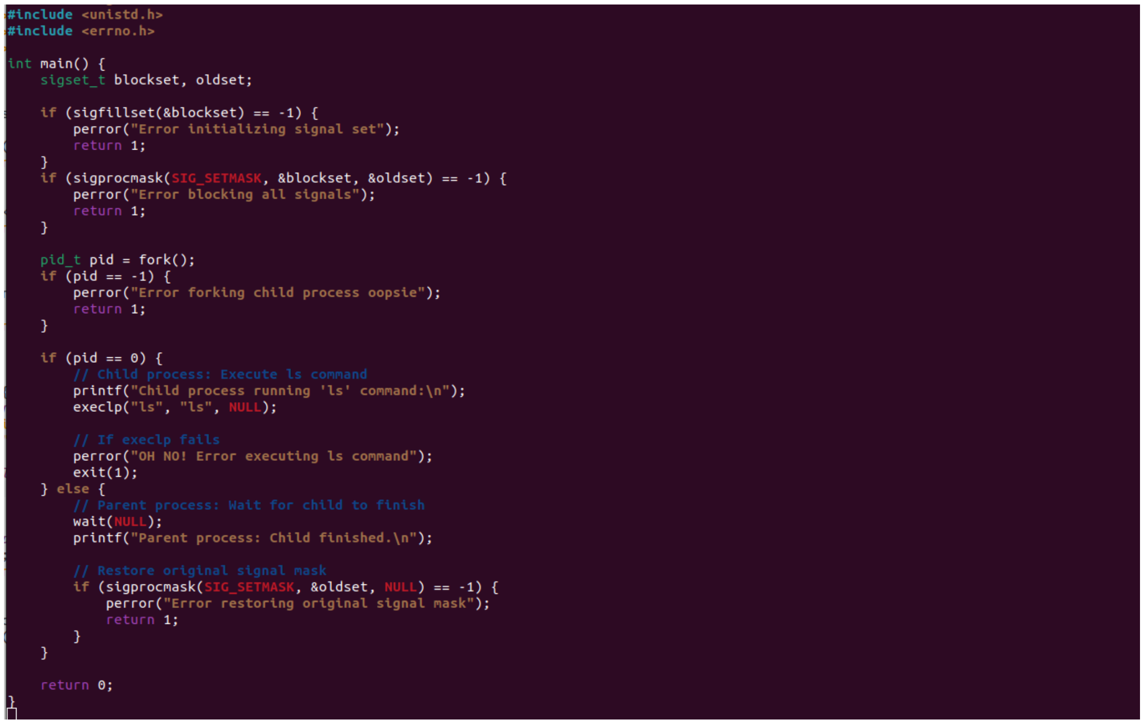
TASK 3.2 Is it possible that after a call to ‘makepair’ ‘pipe1’ exists but ‘pipe2’ does not?
Indeed, that is possible. The function uses mkfifo for pipes 1 and 2 in an attempt to generate two named pipes (FIFOs). The function won't delete pipe1 before returning if pipe1 is created successfully but pipe2 fails (for example, because of insufficient space or permissions, which could set errno to something other than EEXIST). Pipe1 would be there in this situation, but pipe2 would not.

TASK 3.3

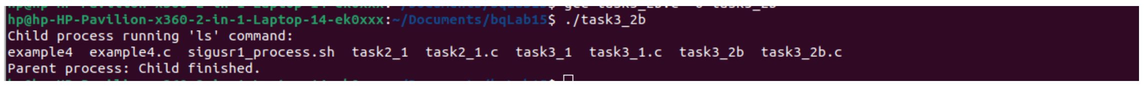
1. **Does a makepair return value of 0 guarantee that FIFOs corresponding to pipe1 and pipe2 are available on return?**

   It is true that both FIFOs corresponding to pipes 1 and 2 have been correctly established and are usable if makepair returns a value of 0. Only after both mkfifo calls are successful does the function return 0 (ignoring the situation in which either FIFO already exists, as shown by errno == EEXIST). The function unlinks any FIFOs that may have been generated during this call and returns -1 if any phase of the FIFO creation process or the signal mask restoration with sigprocmask fails.

2. **Write a program in which the parent blocks all signals before forking a child process to execute an ls command.**



```c
#include <unistd.h>
#include <errno.h>

int main() {
    sigset_t blockset, oldset;

    if (sigfillset(&blockset) == -1) {
        perror("Error initializing signal set");
        return 1;
    }
    if (sigprocmask(SIG_SETMASK, &blockset, &oldset) == -1) {
        perror("Error blocking all signals");
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("Error forking child process oopsie");
        return 1;
    }

    if (pid == 0) {
        // Child process: Execute ls command
        printf("Child process running 'ls' command:\n");
        execlp("ls", "ls", NULL);

        // If execlp fails
        perror("OH NO! Error executing ls command");
        exit(1);
    } else {
        // Parent process: Wait for child to finish
        wait(NULL);
        printf("Parent process: Child finished.\n");

        // Restore original signal mask
        if (sigprocmask(SIG_SETMASK, &oldset, NULL) == -1) {
            perror("Error restoring original signal mask");
            return 1;
        }
    }

    return 0;
}
```



```
hp@hp-HP-Pavilion-x360-2-in-1-Laptop-14-ek0xxx:~/Documents/bqLab15$ ./task3_2b
Child process running 'ls' command:
example4  example4.c  sigusr1_process.sh  task2_1  task2_1.c  task3_1  task3_1.c  task3_2b  task3_2b.c
Parent process: Child finished.
```

Explanation: This program demonstrates blocking all signals in a parent process using sigprocmask before forking a child process. The child executes the ls command with execlp, while the parent waits for the child to finish using wait. After the child completes, the parent restores the original signal mask, ensuring controlled signal management during execution.

TASK 4.1

Using async-signal-safe functions is crucial when it comes to signal handlers. Invoking a signal handler disrupts the program's regular execution flow. Recalling such routines from a signal handler may result in undefined behavior or corrupted data. Since fprintf and strlen are not async-signal-safe functions, they may interfere with buffered input/output or provide unexpected outcomes in the event that a signal is received. Since write is async-signal-safe and communicates with file descriptors directly without buffering, it is utilized in its place.