Running The Programme

```
acar@acar:~/HW2$ make clean
rm -f *.o program
acar@acar:~/HW2$ make
gcc -Wall -g -c main.c
gcc -Wall -g main.o -o program
acar@acar:~/HW2$ ./program 4
```

Creation and Cleanup of FIFO Files

This section deletes any existing FIFO files and creates new ones. If the FIFO creation fails, it outputs an error message and terminates the program.

```c
// Remove any existing FIFOs with the same name
unlink(fifo1);
unlink(fifo2);

// Create two FIFOs for inter-process communication
if (mkfifo(fifo1, 0666) == -1 || mkfifo(fifo2, 0666) == -1) {
    perror("Failed to create FIFOs");
    exit(EXIT_FAILURE);
}
printf("FIFOs created successfully.\n");
```

Setting Up the Signal Handler

This configures a signal handler to catch the SIGCHLD signal, which is used to monitor the termination of child processes.

```c
// Setup signal handling for child process termination
struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sa.sa_handler = child_handler;
sigemptyset(&sa.sa_mask);
sigaction(SIGCHLD, &sa, NULL);
```

Creation and Execution of Child Processes

This section creates two child processes. The first child process reads data from the first FIFO, processes it, and writes the results to the second FIFO. The second child process reads and processes data from the second FIFO and waits for 10 seconds to ensure data is ready.

```
// Fork the first child process
pid1 = fork();
if (pid1 == 0) {  // Child 1: Process data from fifo1 and send results to fifo2
    // Data processing and writing results to fifo2
}

// Fork the second child process
pid2 = fork();
if (pid2 == 0) {  // Child 2: Process data from fifo2
    sleep(10);  // Ensure this child process waits for data to be ready in fifo2
    // Data reading from fifo2 and processing
}
```

Monitoring Status of the Parent Process and "Proceeding" Messages

The parent process waits for both child processes to finish, printing a "Proceeding" message every two seconds during the wait.

```
// Wait for both children to exit, printing "Proceeding" every 2 seconds
while (child_exit_count < 2) {
    printf("Proceeding\n");
    sleep(2);
}
```

Program Termination and Resource Cleanup

it deletes the FIFO files and prints messages indicating that all child processes have completed and the parent has finished waiting.

```
// Clean up by removing FIFO files
unlink(fifo1);
unlink(fifo2);
printf("FIFO files have been deleted.\n");

printf("All child processes have completed.\n");
printf("Parent finished waiting for children.\n");
```

ERROR HANDLİNG MECHANİSMS

1. FIFO Creation

```
if (mkfifo(fifo1, 0666) == -1 || mkfifo(fifo2, 0666) == -1) {
    perror("Failed to create FIFOs");
    exit(EXIT_FAILURE);
}
```

The **mkfifo** function returns -1 if there is an error during the creation of FIFO files. If any error occurs, the program prints an error message and exits with **EXIT_FAILURE**. This ensures that the program terminates safely if it cannot initiate the required IPC mechanisms.,

2. Fıle Opening

```
int fd_read = open(fifo1, O_RDONLY);
if (fd_read == -1) {
    perror("Failed to open fifo1 for reading");
    exit(EXIT_FAILURE);
}
```

The **open** function attempts to open a file at the specified path and returns -1 if it fails. In the event of failure, it prints an explanatory error message and terminates the program with an error status code. This check ensures that resources are properly opened during the process.

3. Data Reading

```
if (read(fd_read, buffer, sizeof(buffer)) != sizeof(buffer)) {
    perror("Failed to read data from fifo1");
    close(fd_read);
    exit(EXIT_FAILURE);
}
```

The **read** function returns a different result than expected if it cannot read the specified number of bytes. If a complete read cannot be performed, an error message is printed, the opened file is closed, and the program terminates with an error. This control is critical for maintaining data integrity.

4. Data Writing

```
if (write(fd_write, &sum, sizeof(sum)) != sizeof(sum) ||
    write(fd_write, &product, sizeof(product)) != sizeof(product) ||
    write(fd_write, "multiply", strlen("multiply") + 1) != strlen("multiply") + 1) {
    perror("Failed to write data to fifo2");
    close(fd_write);
    exit(EXIT_FAILURE);
}
```

The **write** function returns a different result than expected if it cannot write the specified number of bytes. If an error occurs during the writing process, an error message is printed, the file is closed, and the program terminates with an error. This control prevents data loss and ensures the correct flow of data through the IPC channel.

BONUS SECTIONS

1. Zombie Process Protection:

```c
void child_handler(int sig) {
    pid_t pid;
    int status;
    // Process all terminated children without blocking
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("Child with PID %ld exited with status %d.\n", (long)pid, status);
        child_exit_count++;
        if (child_exit_count == 2) {
            printf("All child processes have completed.\n");
        }
    }
}
```

The application utilizes a mechanism that handles the SIGCHLD signal to prevent child processes from becoming zombie processes. The SIGCHLD signal is sent from the operating system to the parent process when a child process terminates. In our application, this signal is caught by a signal handler (child_handler), and the status of child processes is queried using the waitpid() function. This approach ensures that terminated child processes are promptly cleaned up, preventing any process from remaining in a zombie state. This method prevents wastage of system resources and enables the application to operate more efficiently.

2. Reporting of Process Exit Statuses:

```c
// Within the same child_handler function, process exit statuses are also logged.
printf("Child with PID %ld exited with status %d.\n", (long)pid, status);
```

The exit status of each child process is extensively logged within the child_handler function. Exit statuses contain important information about how processes terminated (normal exit, exit with error, etc.). These pieces of information help us understand whether processes exhibit the expected behavior. Additionally, they provide critical data for identifying possible error conditions and making improvements in future versions of the application. This feature enhances the overall reliability and fault tolerance of the application.

OUTPUT EXAMPLES

```
acar@acar:~/HW2$ ./program 4
FIFOs created successfully.
Random array generated:
7 5 6 7
Proceeding
Proceeding
Proceeding
Proceeding
Proceeding
Child 2 received command: multiply
Child 2 received sum: 25, multiplication result: 1470, final value(sum+multiplication): 1495.
Child with PID 118821 exited with status 0.
Child with PID 118822 exited with status 0.
All child processes have completed.
FIFO files have been deleted.
All child processes have completed.
Parent finished waiting for children.
```

```
acar@acar:~/HW2$ ./program 10
FIFOs created successfully.
Random array generated:
9 2 3 7 6 8 7 4 5 2
Proceeding
Proceeding
Proceeding
Proceeding
Proceeding
Child 2 received command: multiply
Child 2 received sum: 53, multiplication result: 5080320, final value(sum+multiplication): 5080373.
Child with PID 118824 exited with status 0.
Child with PID 118825 exited with status 0.
All child processes have completed.
FIFO files have been deleted.
All child processes have completed.
Parent finished waiting for children.
```

```
acar@acar:~/HW2$ ./program 6
FIFOs created successfully.
Random array generated:
5 1 8 6 7 5
Proceeding
Proceeding
Proceeding
Proceeding
Proceeding
Proceeding
Child with PID 118829 exited with status 0.
Child 2 received command: multiply
Child 2 received sum: 32, multiplication result: 8400, final value(sum+multiplication): 8432.
Proceeding
Child with PID 118830 exited with status 0.
All child processes have completed.
FIFO files have been deleted.
All child processes have completed.
Parent finished waiting for children.
```