Program Presentation

In the program, the parent process generates 2 FIFOs and random numbers. Child 1 collects the elements using FIFO1 and sends them to FIFO2. Child 2 multiplies the elements in FIFO2 and adds them with the incoming total and prints them on the screen.

-Makefile

Make all arg=<number> Program starts with this command.

Example: make all arg=3

Parent Process:

-Creating two FIFOs and Error if not created

-Signal Handling with waitpid

```c
int counter = 0;

void sigchld_handler(int signo)
{
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    {
        printf("Child %d terminated\n", pid);
        counter++;
    }
}

void createFifo(char *fifo)
{
    if (mkfifo(fifo, 0666) == -1)
    {
        perror("mkfifo");
        exit(1);
    }

}
```

We use FIFO creation function. If FIFOs cannot be created successfully, we print an error message and terminate the program.
After receiving the SIGCHLD signal, terminated child processes are cleared by calling the waitpid() function.

-Argument and Random Numbers

```c
int main(int argc, char *argv[])
{
    srand(time(NULL));
    if (argc != 2)
    {
        printf("Usage: %s <array size>\n", argv[0]);
        exit(1);
    }
    int arraySize = atoi(argv[1]);
    if (arraySize <= 0)
    {
        printf("Invalid array size: %d\n", arraySize);
        exit(1);
    }
    int numbers[arraySize];
    for (int i = 0; i < arraySize; i++)
    {
        numbers[i] = rand() % 100;
    }
    printf("Initial numbers: ");
    for (int i = 0; i < arraySize; i++)
    {
        printf("%d ", numbers[i]);
    }
    printf("\n");
```

We get the argument using argc and argv. If it is incorrect, print is made.
Random numbers are generated and initial numbers are printed.

-Creating Child Processes using fork

```c
int id1 = fork();
if (id1 == -1)
{
    perror("Failed to fork child process 1");
    exit(1);
}
if (id1 == 0)
{
    childProcess1(arraySize);
}
else{
    int id2 = fork();
    if (id2 == -1)
    {
        perror("Failed to fork child process 2");
        exit(1);
    }
    if (id2 == 0)
    {
        childProcess2(arraySize);
    }
```

-Zombie Process and exit status for all processes

```c
struct sigaction sa;
sigemptyset(&sa.sa_mask);
sa.sa_handler = sigchld_handler;
sa.sa_flags = 0;
sigaction(SIGCHLD, &sa, NULL);
```

By handling the SIGCHLD signal, this piece of code tracks the completion of the parent process, child processes, and cleans them up before they become zombie (dead) processes. This way, the parent process can monitor the exit statuses of child processes and clear them appropriately so that system resources are not wasted.

```c
void sigchld_handler(int signo)
{
    int status;
    pid_t pid;
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0)
    {
        if (WIFEXITED(status))
        {
            printf("Child %d exited with status %d\n", pid, WEXITSTATUS(status));
        }
        else
        {
            printf("Child %d exited abnormally\n", pid);
        }
        counter++;

    }
}
```

Normal termination and abnormal termination are detected via signal.

-Proceeding printing every two seconds

```c
while (counter < 2)
{
    printf("Proceeding\n");
    sleep(2);

}
```

-Some Outputs

```
Initial numbers: 4 2 0 9 2 7 3 5 0 8
Proceeding
Proceeding
Proceeding
Proceeding
Proceeding
Processing sum: 40
Processing multiplication: 0
Final result: 40
Child 18035 exited with status 0
Child 18036 exited with status 0
Program completed successfully
```

```
Initial numbers: 4 7 5 3 8 6 6 5 9 2
Proceeding
Proceeding
Proceeding
Proceeding
Proceeding
Processing sum: 55
Processing multiplication: 10886400
Final result: 10886455
Child 18375 exited with status 0
Child 18376 exited with status 0
Program completed successfully
```

-Errors in Child’s Processes

```c
void childProcess2(int arraySize)
{
    int fd2, sum, result = 1;
    char command[9];

    fd2 = open(FIFO2, O_RDONLY);
    if (fd2 == -1)
    {
        perror("Failed to open FIFO2");
        exit(1);
    }
    sleep(10);
    if (read(fd2, command, sizeof(command)) == -1)
    {
        perror("Failed to read command from FIFO2");
        exit(1);
    }
    if (strcmp(command, "multiply") != 0)
    {
        printf("Invalid command: %s\n", command);
        exit(1);
    }
    for (int i = 0; i < arraySize; i++)
    {
        int number;
        if (read(fd2, &number, sizeof(int)) == -1)
        {
            perror("Failed to read data from FIFO2");
            exit(1);
        }
        result *= number;
    }
    if (read(fd2, &sum, sizeof(int)) == -1)
    {
        perror("Failed to read sum from FIFO2");
        exit(1);
    }
    close(fd2);

    printf("Processing multiplication: %d\n", result);
    printf("Final result: %d\n", result + sum);

    exit(0);
}
```

```c
void childProcess1(int arraySize)
{
    int fd1, sum = 0;
    int numbers[arraySize];

    fd1 = open(FIFO1, O_RDONLY);
    if (fd1 == -1)
    {
        perror("Failed to open FIFO1");
        exit(1);
    }
    sleep(10);
    if (read(fd1, numbers, arraySize * sizeof(int)) == -1)
    {
        perror("Failed to read data from FIFO1");
        exit(1);
    }
    close(fd1);

    for (int i = 0; i < arraySize; i++)
    {
        sum += numbers[i];
    }
    printf("Processing sum: %d\n", sum);

    int fd2 = open(FIFO2, O_WRONLY);
    if (fd2 == -1)
    {
        perror("Failed to open FIFO2");
        exit(1);
    }
    if (write(fd2, &sum, sizeof(int)) == -1)
    {
        perror("Failed to write data to FIFO2");
        exit(1);
    }
    close(fd2);

    exit(0);
}
```

We defined a function named childProcess1.We created FIFOs so then open FIFO1 in read mode and perform addition by reading random numbers. Then, we are writing the result to FIFO2.

Method of childProcess2. After opening in read mode, we read the "multiply" command. Then, after reading random numbers and multiply them with each other. After ,sum sent from child1 to FIFO2 is sum with the multiplication result and print the final result to the screen.