CSE344 SYSTEMS PROGRAMMING

HOMEWORK 2

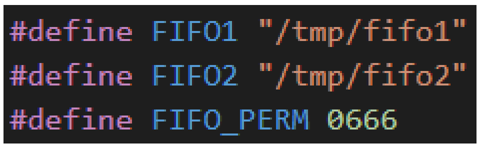
REPORT

Burak Demirkaya

210104004274

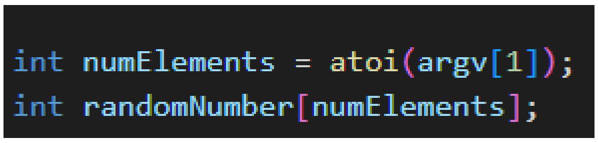
16/04/2024

In the homework, I used a makefile for running the program and setting parameters. In the makefile's run section, I manually adjusted the parameter for the number of random numbers. Also, in the clean section, I removed fifo paths to avoid their presence before running the program.



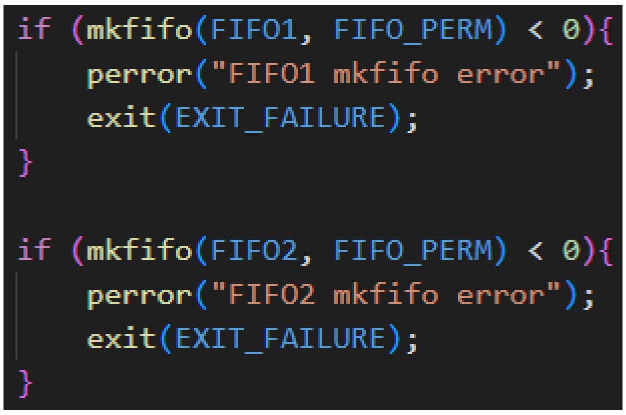
```c
#define FIFO1 "/tmp/fifo1"
#define FIFO2 "/tmp/fifo2"
#define FIFO_PERM 0666
```

I have declared my fifo paths to /tmp directory in order to ensure that the fifo permissions are granted. Also granted the permission of 0666 which means every user can read and write to them.



```c
int numElements = atoi(argv[1]);
int randomNumber[numElements];
```

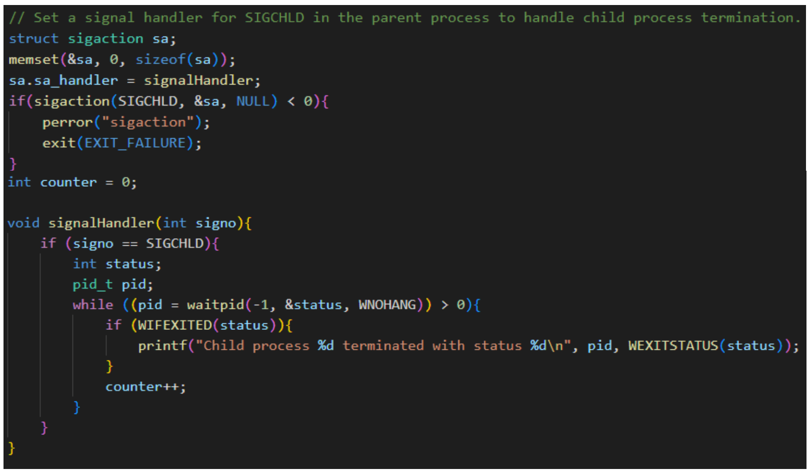
I did not use any dynamic memory allocation to create the random numbers array.



```c
if (mkfifo(FIFO1, FIFO_PERM) < 0){
    perror("FIFO1 mkfifo error");
    exit(EXIT_FAILURE);
}

if (mkfifo(FIFO2, FIFO_PERM) < 0){
    perror("FIFO2 mkfifo error");
    exit(EXIT_FAILURE);
}
```

I have opened the fifos separately to check if any of them gives an error.

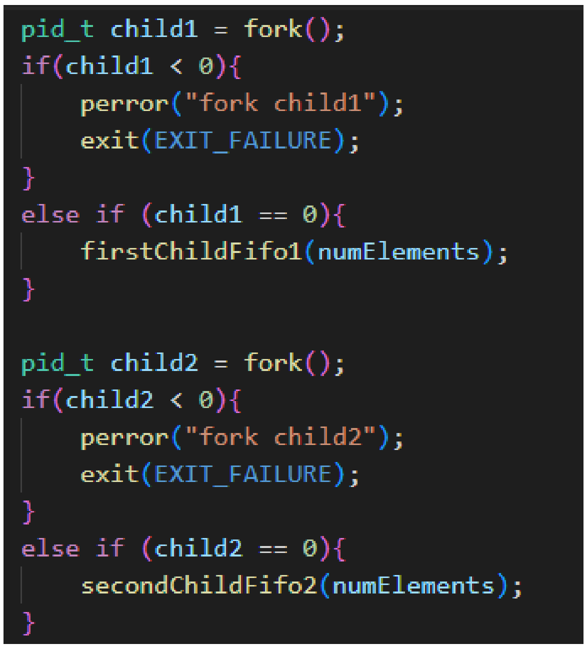


```c
// Set a signal handler for SIGCHLD in the parent process to handle child process termination.
struct sigaction sa;
memset(&sa, 0, sizeof(sa));
sa.sa_handler = signalHandler;
if(sigaction(SIGCHLD, &sa, NULL) < 0){
    perror("sigaction");
    exit(EXIT_FAILURE);
}
int counter = 0;

void signalHandler(int signo){
    if (signo == SIGCHLD){
        int status;
        pid_t pid;
        while ((pid = waitpid(-1, &status, WNOHANG)) > 0){
            if (WIFEXITED(status)){
                printf("Child process %d terminated with status %d\n", pid, WEXITSTATUS(status));
            }
            counter++;
        }
    }
}
```

I have declared my signal handler this way so that it catches SIGCHILD signal and prints the exit status of the child processes. The waitpid also ensures that there cannot exits zombie processes.

By the general idea of the fifos they have to be opened as read before written so I have first generated my child processes to read from the fifos.

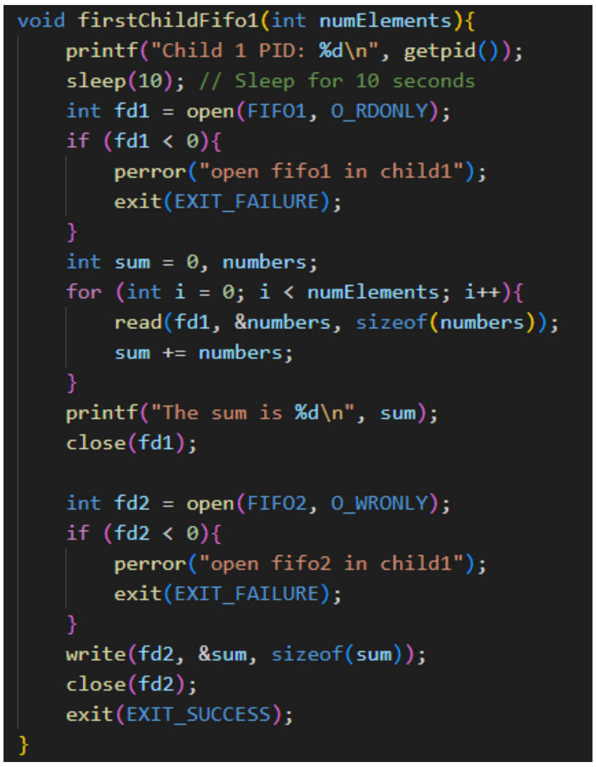


```c
pid_t child1 = fork();
if(child1 < 0){
    perror("fork child1");
    exit(EXIT_FAILURE);
}
else if (child1 == 0){
    firstChildFifo1(numElements);
}

pid_t child2 = fork();
if(child2 < 0){
    perror("fork child2");
    exit(EXIT_FAILURE);
}
else if (child2 == 0){
    secondChildFifo2(numElements);
}
```

Also I have checked their pid_t values for errors.

Child1

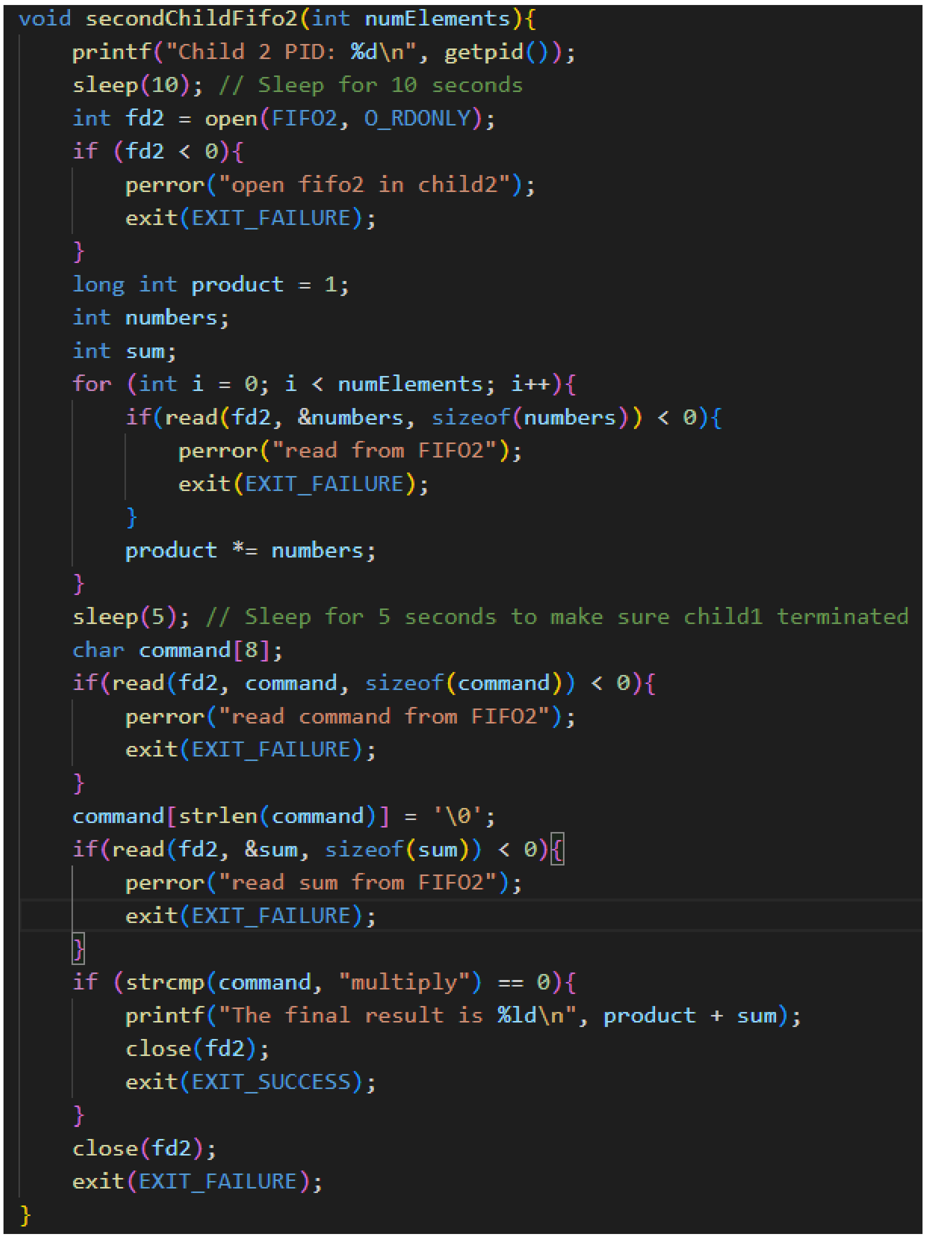


```c
void firstChildFifo1(int numElements){
    printf("Child 1 PID: %d\n", getpid());
    sleep(10); // Sleep for 10 seconds
    int fd1 = open(FIFO1, O_RDONLY);
    if (fd1 < 0){
        perror("open fifo1 in child1");
        exit(EXIT_FAILURE);
    }
    int sum = 0, numbers;
    for (int i = 0; i < numElements; i++){
        read(fd1, &numbers, sizeof(numbers));
        sum += numbers;
    }
    printf("The sum is %d\n", sum);
    close(fd1);

    int fd2 = open(FIFO2, O_WRONLY);
    if (fd2 < 0){
        perror("open fifo2 in child1");
        exit(EXIT_FAILURE);
    }
    write(fd2, &sum, sizeof(sum));
    close(fd2);
    exit(EXIT_SUCCESS);
}
```

Child1 simply reads the random numbers and add them up. After that it sends the result to the second fifo which will be read by the second child.

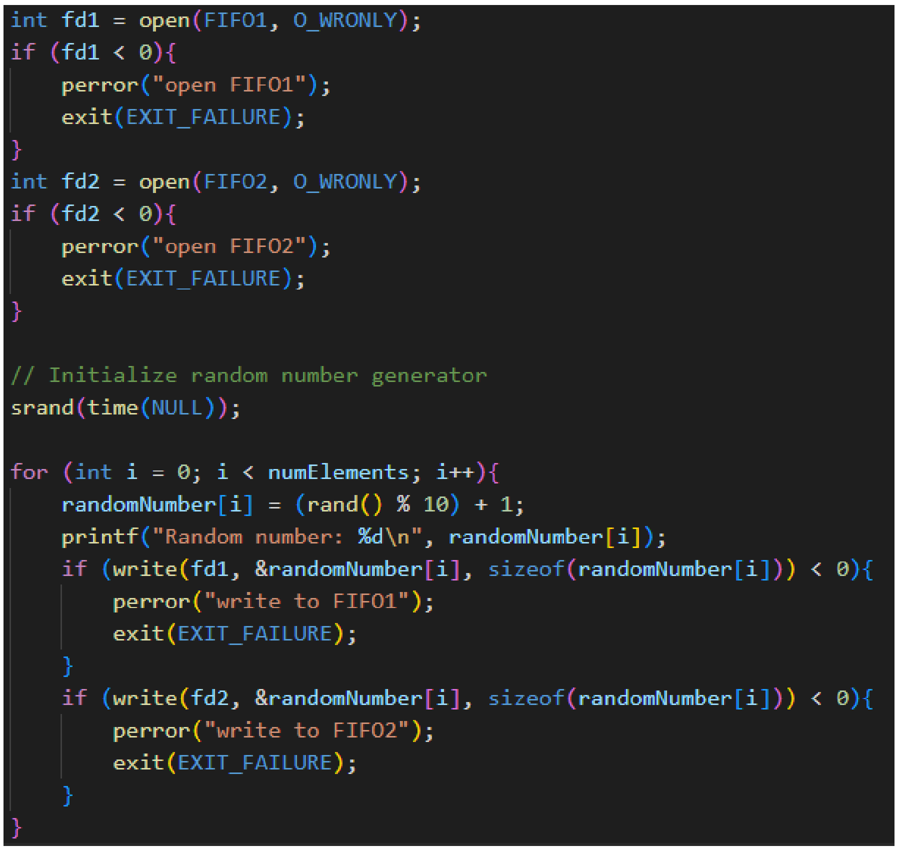
Child2



```c
void secondChildFifo2(int numElements){
    printf("Child 2 PID: %d\n", getpid());
    sleep(10); // Sleep for 10 seconds
    int fd2 = open(FIFO2, O_RDONLY);
    if (fd2 < 0){
        perror("open fifo2 in child2");
        exit(EXIT_FAILURE);
    }
    long int product = 1;
    int numbers;
    int sum;
    for (int i = 0; i < numElements; i++){
        if(read(fd2, &numbers, sizeof(numbers)) < 0){
            perror("read from FIFO2");
            exit(EXIT_FAILURE);
        }
        product *= numbers;
    }
    sleep(5); // Sleep for 5 seconds to make sure child1 terminated
    char command[8];
    if(read(fd2, command, sizeof(command)) < 0){
        perror("read command from FIFO2");
        exit(EXIT_FAILURE);
    }
    command[strlen(command)] = '\0';
    if(read(fd2, &sum, sizeof(sum)) < 0){
        perror("read sum from FIFO2");
        exit(EXIT_FAILURE);
    }
    if (strcmp(command, "multiply") == 0){
        printf("The final result is %ld\n", product + sum);
        close(fd2);
        exit(EXIT_SUCCESS);
    }
    close(fd2);
    exit(EXIT_FAILURE);
}
```

Child2 reads the random numbers from the parent and calculates the product of them. Then it waits for 5 seconds in order to ensure that the first child has calculated the summation and sent it to the second fifo. After that it reads the command that is sent by the parent. Later it reads the summation result from the first child1. Finally if the command is “multiply” it prints out the final result which is the summation of the child1 result and child2 result. If the command has been read incorrect it exits with a failure status.

Parent process

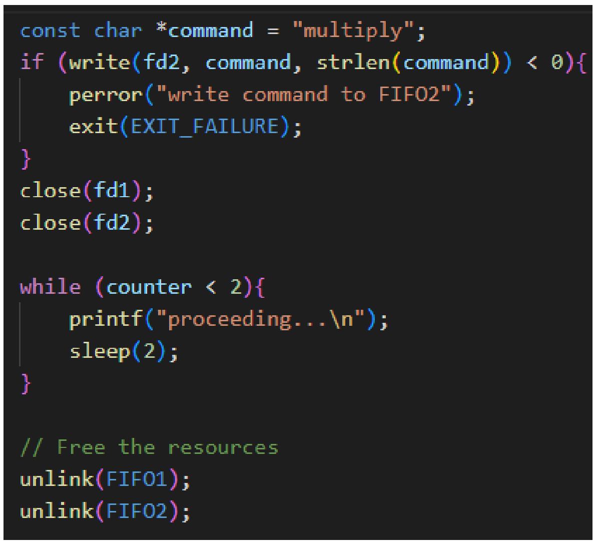


```c
int fd1 = open(FIFO1, O_WRONLY);
if (fd1 < 0){
    perror("open FIFO1");
    exit(EXIT_FAILURE);
}
int fd2 = open(FIFO2, O_WRONLY);
if (fd2 < 0){
    perror("open FIFO2");
    exit(EXIT_FAILURE);
}

// Initialize random number generator
srand(time(NULL));

for (int i = 0; i < numElements; i++){
    randomNumber[i] = (rand() % 10) + 1;
    printf("Random number: %d\n", randomNumber[i]);
    if (write(fd1, &randomNumber[i], sizeof(randomNumber[i])) < 0){
        perror("write to FIFO1");
        exit(EXIT_FAILURE);
    }
    if (write(fd2, &randomNumber[i], sizeof(randomNumber[i])) < 0){
        perror("write to FIFO2");
        exit(EXIT_FAILURE);
    }
}
```

Parent process opens the fifos as write only mode and writes the random numbers generated to the fifos. In order to prevent that the random numbers can be zero, I have added one to them.



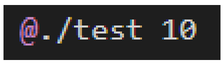
```c
const char *command = "multiply";
if (write(fd2, command, strlen(command)) < 0){
    perror("write command to FIFO2");
    exit(EXIT_FAILURE);
}
close(fd1);
close(fd2);

while (counter < 2){
    printf("proceeding...\n");
    sleep(2);
}

// Free the resources
unlink(FIFO1);
unlink(FIFO2);
```

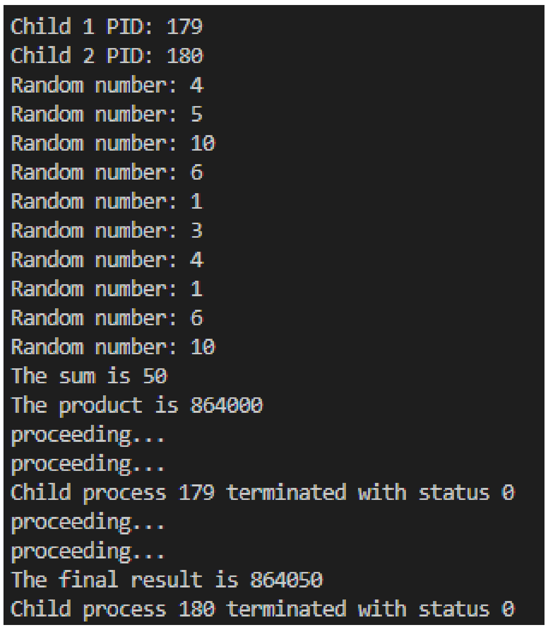
Parent process sends the command “multiply” to the second fifo and prints proceeding… every two seconds until the initial counter is equal to 2. Finally it frees the fifo resources by unlinking them.

Output Examples

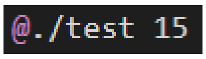


```
@./test 10
```

Run command

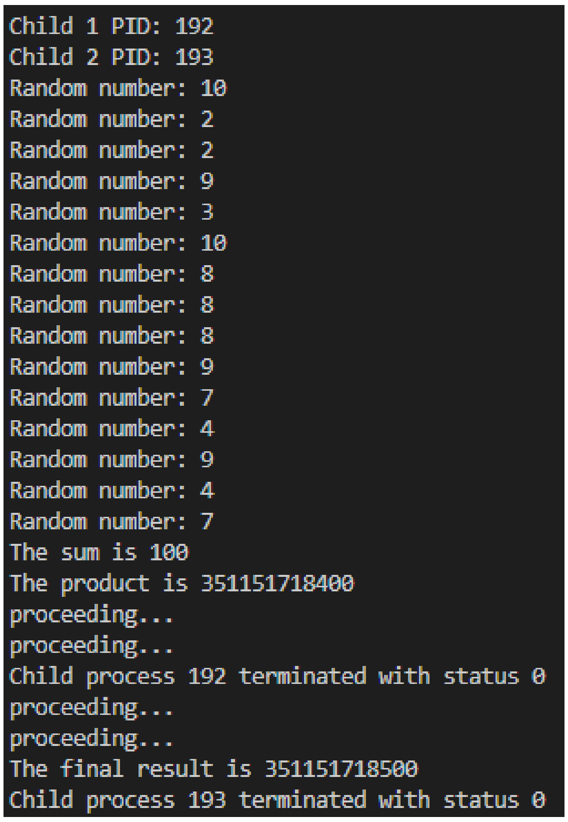


```
Child 1 PID: 179
Child 2 PID: 180
Random number: 4
Random number: 5
Random number: 10
Random number: 6
Random number: 1
Random number: 3
Random number: 4
Random number: 1
Random number: 6
Random number: 10
The sum is 50
The product is 864000
proceeding...
proceeding...
Child process 179 terminated with status 0
proceeding...
proceeding...
The final result is 864050
Child process 180 terminated with status 0
```



```
@./test 15
```

Run command



```
Child 1 PID: 192
Child 2 PID: 193
Random number: 10
Random number: 2
Random number: 2
Random number: 9
Random number: 3
Random number: 10
Random number: 8
Random number: 8
Random number: 8
Random number: 9
Random number: 7
Random number: 4
Random number: 9
Random number: 4
Random number: 7
The sum is 100
The product is 351151718400
proceeding...
proceeding...
Child process 192 terminated with status 0
proceeding...
proceeding...
The final result is 351151718500
Child process 193 terminated with status 0
```

I printed the ID's of the child's so that when they terminated the signal handler can print the details of their exit status.