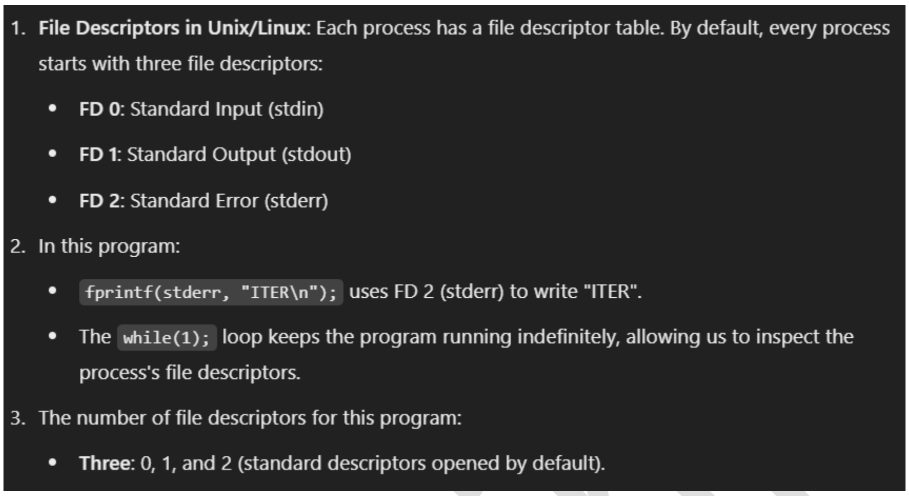
C MINOR ASSIGNMENT- 09(Inter-Process Communication using Pipes & FIFO)

1. Determine the number of file descriptors Write theirfd numbers.

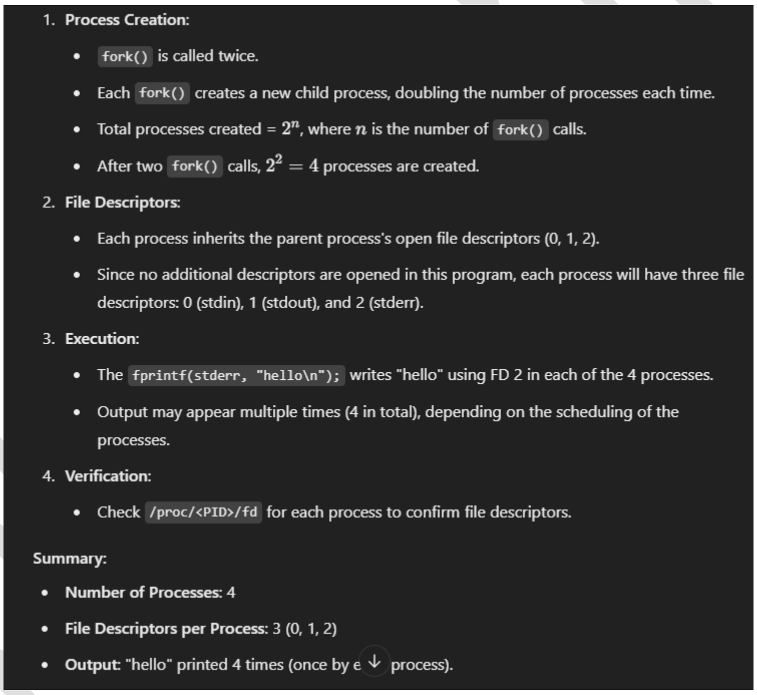
```
int main(void) {
    fprintf(stderr,"ITER\n");
    while(1);
    return 0;
}
```



1. **File Descriptors in Unix/Linux**: Each process has a file descriptor table. By default, every process starts with three file descriptors:
   - **FD 0**: Standard Input (stdin)
   - **FD 1**: Standard Output (stdout)
   - **FD 2**: Standard Error (stderr)
2. In this program:
   - `fprintf(stderr, "ITER\n");` uses FD 2 (stderr) to write "ITER".
   - The `while(1);` loop keeps the program running indefinitely, allowing us to inspect the process's file descriptors.
3. The number of file descriptors for this program:
   - **Three**: 0, 1, and 2 (standard descriptors opened by default).

2. State the number of FDs will be opened

```
int main(void){
  fork();
  fork();
  fprintf(stderr,"hello\n");
  return 0;
}
```



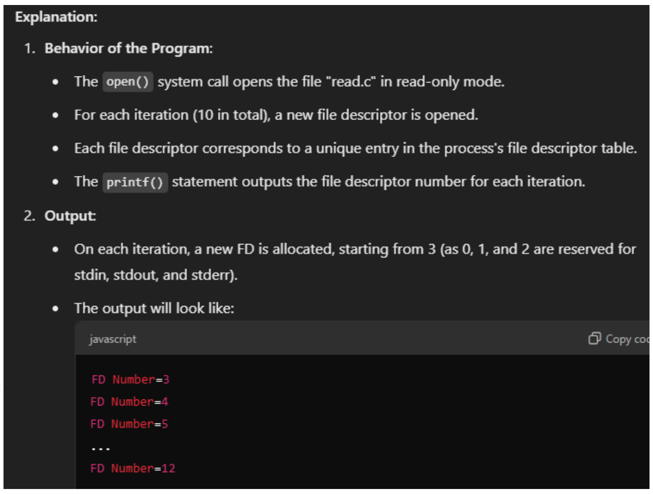
1. **Process Creation:**
   - `fork()` is called twice.
   - Each `fork()` creates a new child process, doubling the number of processes each time.
   - Total processes created = 2^n, where n is the number of `fork()` calls.
   - After two `fork()` calls, $2^2 = 4$ processes are created.
2. **File Descriptors:**
   - Each process inherits the parent process's open file descriptors (0, 1, 2).
   - Since no additional descriptors are opened in this program, each process will have three file descriptors: 0 (stdin), 1 (stdout), and 2 (stderr).
3. **Execution:**
   - The `fprintf(stderr, "hello\n");` writes "hello" using FD 2 in each of the 4 processes.
   - Output may appear multiple times (4 in total), depending on the scheduling of the processes.
4. **Verification:**
   - Check `/proc/<PID>/fd` for each process to confirm file descriptors.

Summary:

- **Number of Processes:** 4
- **File Descriptors per Process:** 3 (0, 1, 2)
- **Output:** "hello" printed 4 times (once by e ↓ process).

3. Check the directory; $ls /proc/PID/fd to find the number of FDs the following program generates when it becomes process. Assume read.c is an existing file in your PWD.

```
int main(){
  int fd,i;
  for(i=0;i<10;i++){
     fd=open("read.c",O_RDONLY);
     if(fd==-1){
        perror("Open error");
        return 1;
     }
     sleep(2);
     printf("FD Number=%d\n",fd);
  }
   return 0;
}
```



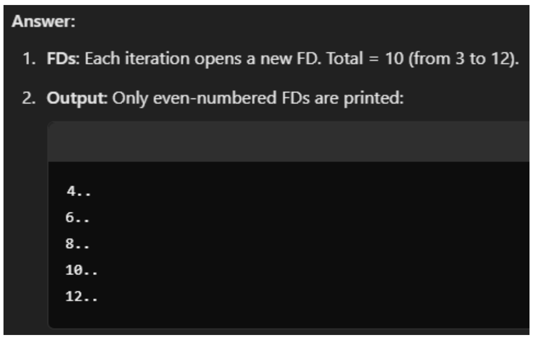
Explanation:

1. **Behavior of the Program:**
   - The `open()` system call opens the file "read.c" in read-only mode.
   - For each iteration (10 in total), a new file descriptor is opened.
   - Each file descriptor corresponds to a unique entry in the process's file descriptor table.
   - The `printf()` statement outputs the file descriptor number for each iteration.
2. **Output:**
   - On each iteration, a new FD is allocated, starting from 3 (as 0, 1, and 2 are reserved for stdin, stdout, and stderr).
   - The output will look like:

```
FD Number=3
FD Number=4
FD Number=5
...
FD Number=12
```

4. Check the directory; $ls /proc/PID/fd to find the number of FDs the following program generates when it becomes process. Assume read.c is an existing file in your PWD.

```
int main(){
  int fd,i;
  for(i=0;i<10;i++){
     fd=open("read.c",O_RDONLY);
     if(fd%2==0)
         printf("%d..\n",fd);
  }
   sleep(2);
   return 0;
}
```



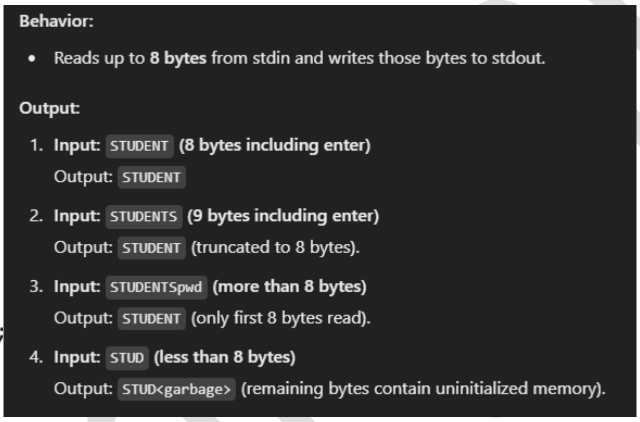
Answer:

1. **FDs:** Each iteration opens a new FD. Total = 10 (from 3 to 12).
2. **Output:** Only even-numbered FDs are printed:

```
4..
6..
8..
10..
12..
```

5. Study the use of read() and write() system calls for unformatted I/O.

```
#define BLKSIZE 8
int main(void){
  char buf[BLKSIZE];
  read(STDIN_FILENO, buf, BLKSIZE);
  write(STDOUT_FILENO, buf, BLKSIZE);
  return 0;
}
```



Behavior:

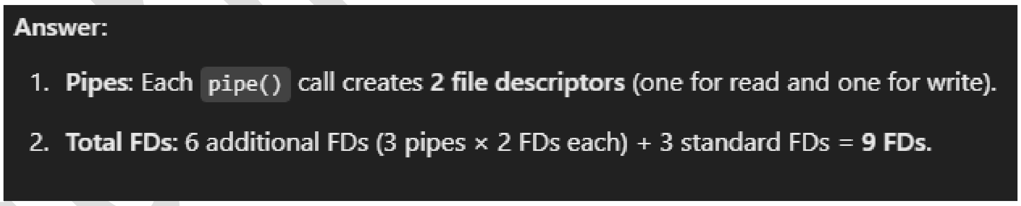
- Reads up to **8 bytes** from stdin and writes those bytes to stdout.

Output:

1. **Input:** `STUDENT` **(8 bytes including enter)**
   Output: `STUDENT`
2. **Input:** `STUDENTS` **(9 bytes including enter)**
   Output: `STUDENT` (truncated to 8 bytes).
3. **Input:** `STUDENTSpwd` **(more than 8 bytes)**
   Output: `STUDENT` (only first 8 bytes read).
4. **Input:** `STUD` **(less than 8 bytes)**
   Output: `STUD<garbage>` (remaining bytes contain uninitialized memory).

6. Write the number of file descriptors will be opened for the following code snippet. Verify the descriptor numbers by exploring the fd folder for the process in the directory /proc/PID.

```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
int main(void)
{
  int fd[2],fs[2],fds[2];
  pipe(fd);
  pipe(fs);
  pipe(fds);
  return 0;
}
```



Answer:

1. **Pipes:** Each `pipe()` call creates **2 file descriptors** (one for read and one for write).
2. **Total FDs:** 6 additional FDs (3 pipes × 2 FDs each) + 3 standard FDs = **9 FDs.**

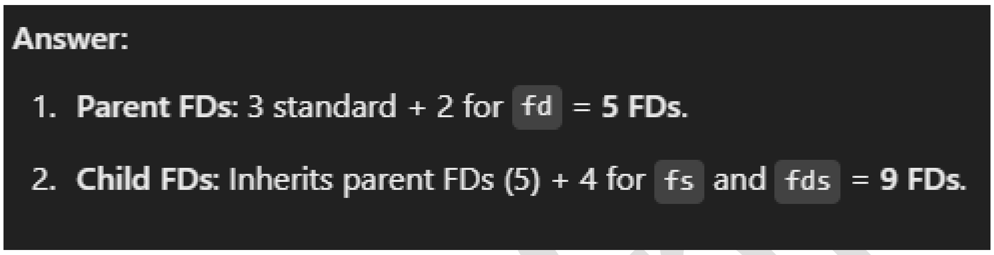
7. State the number of file descriptors will be opened for the below given code. Can you able to show the file descriptors in your machine? [Y — N]

```
#include<stdio.h>
#include<unistd.h>
#include<errno.h>
int main(void)
{
  int fd[2],fs[2],fds[2];
  if(pipe(fd) == -1){
    perror("Failed to create the pipe");
    return 1;
  }
  if(pipe(fs) == -1){
    perror("Failed to create the pipe");
    return 2;
  }
  if(pipe(fds) == -1){
    perror("Failed to create the pipe");
    eturn 3;
  }
  return 0;
}
```

ANSWER: Same as Question 6: **9 FDs** total (3 pipes × 2 FDs + 3 standard FDs).

8. Write the descriptor numbers attached to both parent and child process file descriptor table(PFDT). Verify the descriptor numbers by exploring the fd folder for the process in the directory proc.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
int main(void) {
 int fd[2],fs[2],fds[2];
 pid_t pid;
 pipe(fd);
 pid=fork();
 if(pid==0){
    pipe(fs);
    pipe(fds);
 }
 else{
   wait(NULL);
   printf("Parent waits\n");
 }
  return 0;
}
```

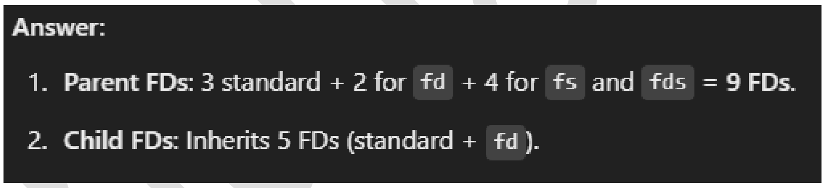


Answer:

1. **Parent FDs:** 3 standard + 2 for `fd` = **5 FDs.**
2. **Child FDs:** Inherits parent FDs (5) + 4 for `fs` and `fds` = **9 FDs.**

9. Write the descriptor numbers attached to both parent and child process file descriptor table(PFDT). Verify the descriptor numbers by exploring the fd folder for the process in the directory proc.

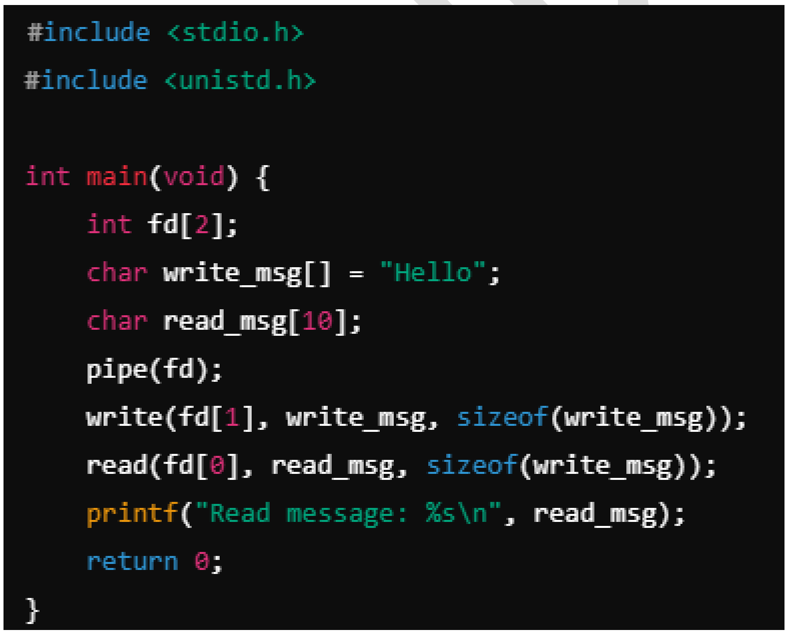
```
int main(void){int fd[2],fs[2],fds[2];
  pipe(fd);
  pid_t pid=fork();
  if(pid!=0){
     pipe(fs);
     pipe(fds);
  }
  else{
     wait(NULL);
     printf("Parent waits\n");
 }return 0;}
```



Answer:

1. **Parent FDs:** 3 standard + 2 for `fd` + 4 for `fs` and `fds` = **9 FDs.**
2. **Child FDs:** Inherits 5 FDs (standard + `fd`).

10. Develop aprogramto write and read a message using pipe() for a single process. (Hint: No need to use fork() and the main process will create and implement the pipe for both writing and reading.)

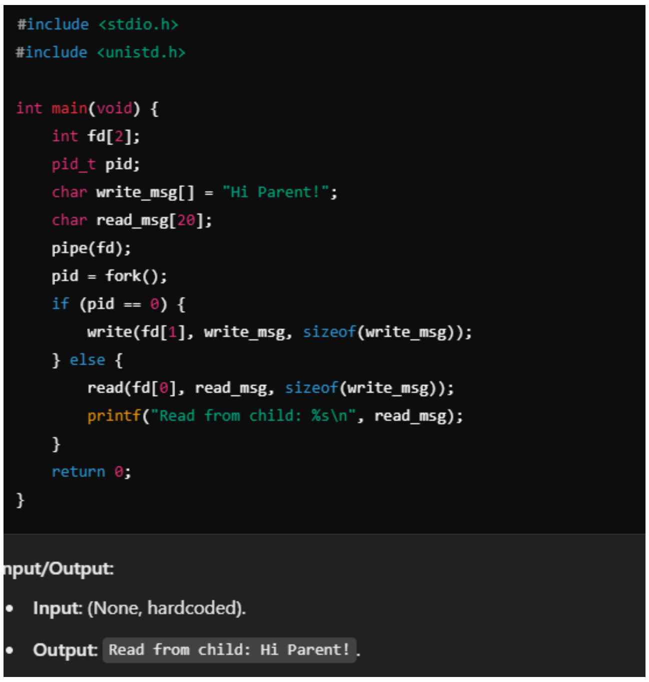


```
#include <stdio.h>
#include <unistd.h>

int main(void) {
    int fd[2];
    char write_msg[] = "Hello";
    char read_msg[10];
    pipe(fd);
    write(fd[1], write_msg, sizeof(write_msg));
    read(fd[0], read_msg, sizeof(write_msg));
    printf("Read message: %s\n", read_msg);
    return 0;
}
```

- **Input**: (None, hardcoded).
- **Output**: `Read message: Hello`.

11. Here, use the fork() system call to create a child process. The child process will write a message into the pipe and the parent process will read the message from the pipe. The parent process will display the message on stdout. Design a program to establish the communication using pipe between the processes.



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

int main(void) {
    int fd[2];
    pid_t pid;
    char write_msg[] = "Hi Parent!";
    char read_msg[20];
    pipe(fd);
    pid = fork();
    if (pid == 0) {
        write(fd[1], write_msg, sizeof(write_msg));
    } else {
        read(fd[0], read_msg, sizeof(write_msg));
        printf("Read from child: %s\n", read_msg);
    }
    return 0;
}
```

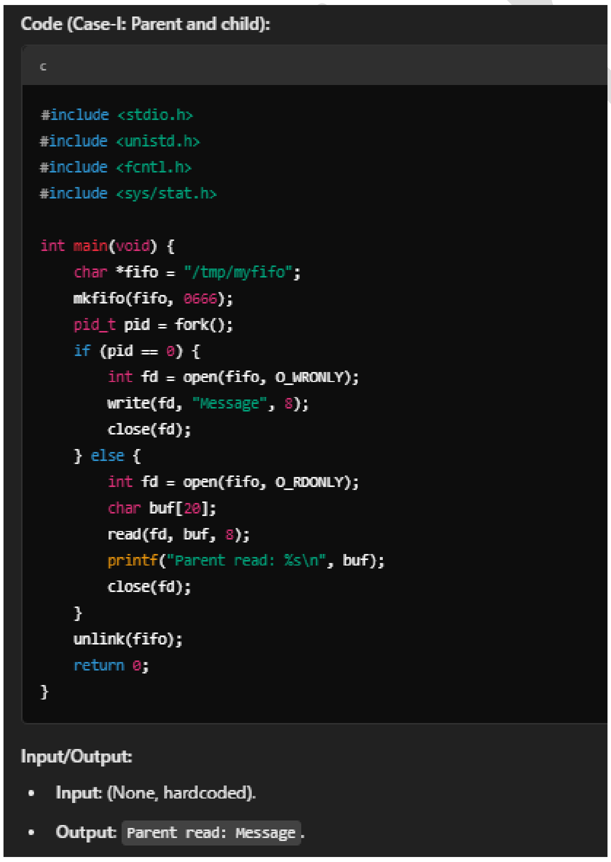
nput/Output:

- **Input:** (None, hardcoded).
- **Output:** `Read from child: Hi Parent!`.

12. Develop a program to communicate between two processes using a named pipe (FIFO). The program will demonstrate how data is written into a FIFO and how data is read from a FIFO. Implement FIFO in two aspects;

CASE-I: Between parent process and child process (co-operative processes)

CASE-II: Between two different process (i.e. two independent processes)



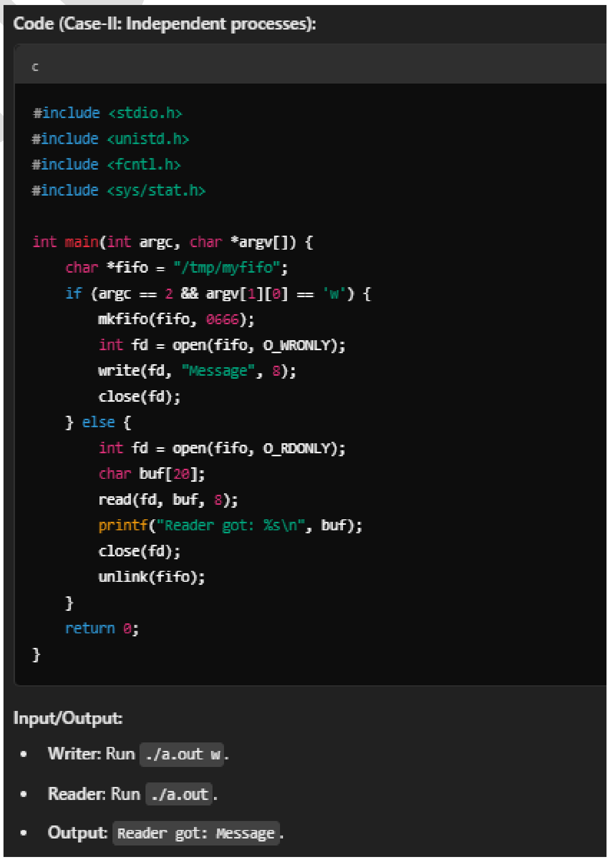
Code (Case-I: Parent and child):

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(void) {
    char *fifo = "/tmp/myfifo";
    mkfifo(fifo, 0666);
    pid_t pid = fork();
    if (pid == 0) {
        int fd = open(fifo, O_WRONLY);
        write(fd, "Message", 8);
        close(fd);
    } else {
        int fd = open(fifo, O_RDONLY);
        char buf[20];
        read(fd, buf, 8);
        printf("Parent read: %s\n", buf);
        close(fd);
    }
    unlink(fifo);
    return 0;
}
```

Input/Output:

- **Input:** (None, hardcoded).
- **Output:** `Parent read: Message`.



Code (Case-II: Independent processes):

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(int argc, char *argv[]) {
    char *fifo = "/tmp/myfifo";
    if (argc == 2 && argv[1][0] == 'w') {
        mkfifo(fifo, 0666);
        int fd = open(fifo, O_WRONLY);
        write(fd, "Message", 8);
        close(fd);
    } else {
        int fd = open(fifo, O_RDONLY);
        char buf[20];
        read(fd, buf, 8);
        printf("Reader got: %s\n", buf);
        close(fd);
        unlink(fifo);
    }
    return 0;
}
```

Input/Output:

- **Writer:** Run `./a.out w`.
- **Reader:** Run `./a.out`.
- **Output:** `Reader got: Message`.