OS ASSIGNMENT-4 REPORT

Lab4 - IPC

: Adarsh Anand : 2003101 :

: Prof - Dr Sharad Sinha : TA - Prachi Kashikar

QUESTIONS

1. You will write two simple programs pipe_reader.c and pipe_writer.c that use a named pipe to communicate. The pipe reader program will set up a named pipe using mkfifo() , open it read only, and read strings from it until it receives the string exit . The writer will open the named pipe file, read strings from the user and write them to the named pipe. When the user enters exit , the program will write the string to the pipe and then exit. Execution should look something like this (note that you must start the reader first):

Ans) pipe_writer.c

```c
*/
int main() {
    int fd;
    char *myfifo = "/tmp/mypipe";
    mkfifo(myfifo, 0777);
    char buf[100];
    /* open, read, and display the message from the FIFO */
    fd = open(myfifo, O_WRONLY);
    while (1) {
        printf("Enter Input: ");
        scanf("%s", buf);
        write(fd, buf, strlen(buf) + 1);
        printf("Writing buffer to pipe ... done\n");
        if (strcmp(buf, "exit") == 0) {
            break;
        }
    }
    close(fd);
    return 0;
```

pipe_reader.c

```c
int main() {
    int fd;
    char *myfifo = "/tmp/mypipe";
    char buf[100];
    /* open, read, and display the message from the FIFO */
    fd = open(myfifo, O_RDONLY);
    while (1) {
        read(fd, buf, 100);
        printf("Waiting for input ... Got it: '%s'\n", buf);
        if (strcmp(buf, "exit") == 0) {
            printf("Exiting\n");
            break;
        }
    }
    close(fd);
}
```

Output

```
}

TERMINAL
alpha@alpha-HP:~/Documents/GitHub/CS310-OS/Lab-4$ ./a.out
Waiting for input...Got it: 'ok'
Waiting for input...Got it: 'kl'
Waiting for input...Got it: 'al'
Waiting for input...Got it: 'l'
Waiting for input...Got it: 'll'
Waiting for input...Got it: 'exi'
Waiting for input...Got it: ''
Waiting for input...Got it: 'exit'
Exiting
alpha@alpha-HP:~/Documents/GitHub/CS310-OS/Lab-4$

Enter Input: l
Writing buffer to pipe...done
Enter Input: ll
Writing buffer to pipe...done
Enter Input: exi
Writing buffer to pipe...done
Enter Input: ^[[A^[[B
Writing buffer to pipe...done
Enter Input: exit
Writing buffer to pipe...done
alpha@alpha-HP:~/Documents/GitHub/CS310-OS/Lab-4$
```

Extend the above exercise to create a chat application between two processes.

User1

```
struct data {
   char buff[100];
   int status, pid1, pid2;
};

struct data *ptr;

void handler(int signum) {
   if (signum == SIGUSR1) {
       printf("Received from User2: ");
       puts(ptr->buff);
   }
}

int main() {

   int pid = getpid(); /* get process id */

   int shmid; /* shared memory id */

   int key = 12345; /* key for shared memory */

   shmid = shmget(key, sizeof(struct data), IPC_CREAT | 0666); /* create shared memory */
```

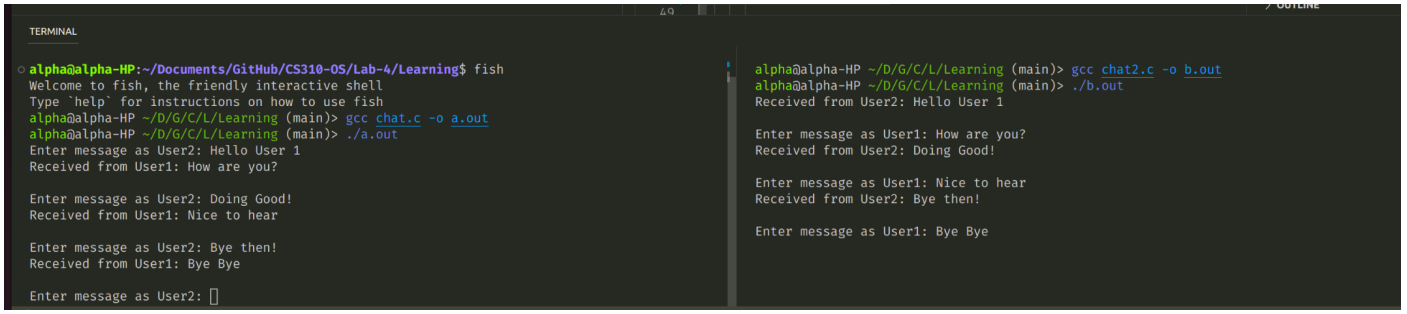
```
    ptr = (struct data *)shmat(shmid, NULL, 0); /* attach shared memory */

    ptr->pid1 = pid;
    ptr->status = -1;

    signal(SIGUSR1, handler); /* register signal handler */

    while (1) {
        while (ptr->status != 1) /* wait for signal */
            continue;
        sleep(1);

        printf("Enter message as User1: "); /* get message from user */
        fgets(ptr->buff, 100, stdin);       /* get message from user */

        ptr->status = 0; /* set status to 0 */

        kill(ptr->pid2, SIGUSR2); /* send signal to other process */
    }

    shmdt((void *)ptr);            /* detach shared memory */
    shmctl(shmid, IPC_RMID, NULL); /* delete shared memory */
    return 0;
}
```

User 2

```c
int main() {

   int pid = getpid(); /* get process id */

   int shmid; /* shared memory id */

   int key = 12345; /* key for shared memory */

   shmid = shmget(key, sizeof(struct data), IPC_CREAT | 0666); /* create shared memory */

   ptr = (struct data *)shmat(shmid, NULL, 0); /* attach shared memory */

   ptr->pid2 = pid; /* store pid in shared memory */

   ptr->status = -1;

   signal(SIGUSR2, handler);

   while (1) {
       sleep(1);

       printf("Enter message as User2: ");
       fgets(ptr->buff, 100, stdin); /* get message from user */
       ptr->status = 1;

       kill(ptr->pid1, SIGUSR1); /* send signal to other process */
```

```
        while (ptr->status == 1)
            continue;
    }

    shmdt((void *)ptr); /* detach shared memory */
    return 0;
}
```

Output



2) Compile lab4_6.c with gcc, and run it a few times. Describe the output you get, and

explain briefly how it is produced.

Ans)

1. Program starts by creating a shared memory segment using mmap() .
2. The parent process then forks a child process.
3. The parent process then enters a loop that writes the square of the loop index into the shared memory segment.
4. The parent process then sleeps for one second and then repeats the loop.
5. The child process enters a loop that reads the value from the shared memory segment, prints it, and then sleeps for one second.
6. The child process then repeats the loop. The parent process waits for the child process to finish and then exits
7. This program is typically run with two instances of the program running simultaneously. The output from the two instances is interleaved.
8. The output is shown below.

```
55    the parent process begins.
56    */
57
```

TERMINAL

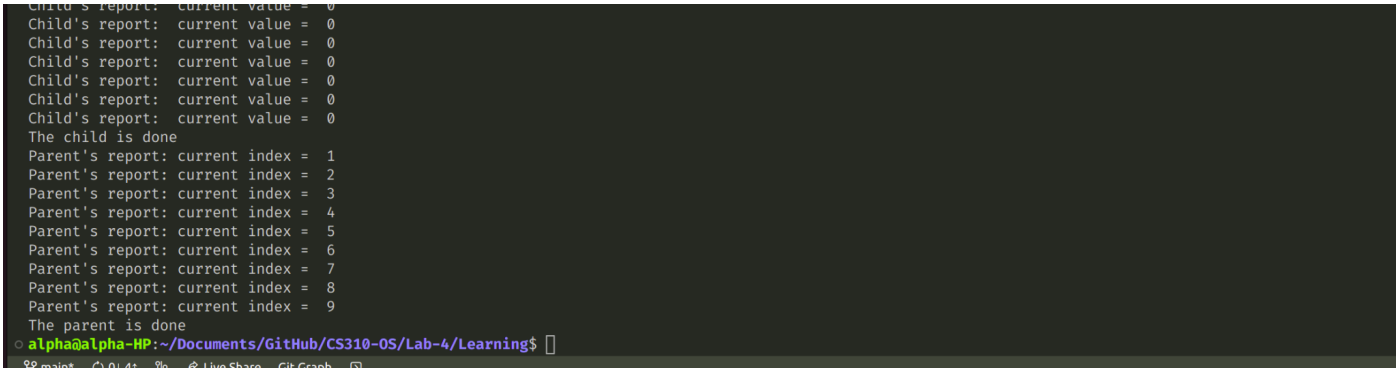
```
Parent's report: current index =  6
Child's report:  current value = 36
Parent's report: current index =  7
Child's report:  current value = 49
Parent's report: current index =  8
Child's report:  current value = 64
Parent's report: current index =  9
Child's report:  current value = 81
The child is done
The parent is done
alpha@alpha-HP:~/Documents/GitHub/CS310-OS/Lab-4/Learning$
```

3) Remove the sleep statement from the child process, rerun lab4_6.c and explain the

output produced.

Ans)

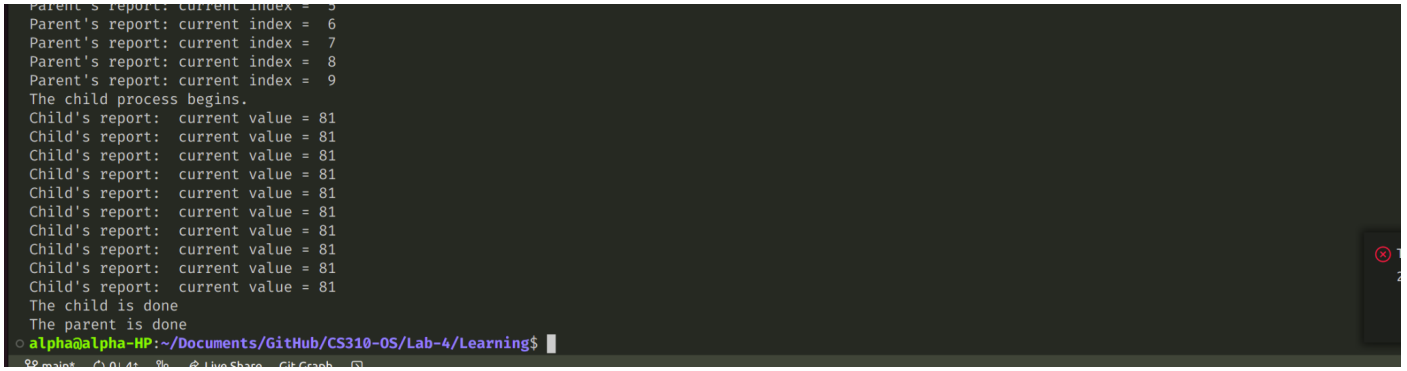
1. When the sleep statement is removed from the child process, the child process reads the value from the shared memory segment before the parent process has written a value to the shared memory segment.
2. The child process then prints the value that was read from the shared memory segment.
3. For all the iterations of the loop, the value read by the child process is 0.
4. After the child is done executing, the parent process starts executing.
5. The parent process writes the loop index into the shared memory segment.
6. The parent process then repeats the loop. The output is shown below.



4) Restore the sleep statement from the previous step, and remove it from the parent process. Again, rerun lab4_6.c , and explain the output produced.

Ans)

1. When the sleep statement is removed from the parent process, the parent process writes the loop index into the shared memory segment before the child process has read the value from the shared memory segment.
2. The parent process then repeats the loop.
3. After the parent is done executing, the child process starts executing.
4. The child process reads the value from the shared memory segment.
5. The child process then prints the value that was read from the shared memory segment ie 9.
6. The child process then repeats the loop. The output is shown below.



5) Rather than rely upon sleep statements to synchronize the two processes, consider the use of spinlocks. In this approach, the parent will write to shared memory when the memory location contains the value -1, and the child will read when the memory location is not -1.

Ans) Initialize the shared memory location in main memory to -1 before the fork operation.

```
// Initialize the shared memory location in main memory to -1 before the fork operation.
   shared_memory = mmap(-1, SIZE, PROT_READ | PROT_WRITE,
                        MAP_ANONYMOUS | MAP_SHARED, -1, 0); // -1 is the initial value
```

Replace the sleep statement for the child by a spinlock that checks that the shared memory contains a nonnegative value. At the end of the child's loop, the shared memory location should be reset to -1.

Child Process

```
if (0 == pid) { /* processing for child */
       printf("The child process begins.\n");
       for (i_child = 0; i_child < run_length; i_child++) {
           while (*shared_memory == -1) {
               // spinlock
           }
           value = *shared_memory;
           printf("Child's report:  current value = %2d\n", value);
           *shared_memory = -1;
       }
       printf("The child is done\n");
   }
```

Parent Process

```
else { /* processing for parent */

       printf("The parent process begins.\n");

       for (i_parent = 0; i_parent < run_length; i_parent++) {

           while (*shared_memory != -1) {

               // spinlock

           }

           *shared_memory = i_parent * i_parent;

           printf("Parent's report: current index = %2d\n", i_parent);

       }

       wait(pid); /* wait for child to finish */

       printf("The parent is done\n");

   }

}
```

Output

```
Child's report:  current value =  0
Parent's report: current index =  0
Child's report:  current value =  1
Parent's report: current index =  1
Parent's report: current index =  2
Child's report:  current value =  4
Child's report:  current value =  9
Parent's report: current index =  3
Parent's report: current index =  4
Child's report:  current value = 16
Child's report:  current value = 25
Parent's report: current index =  5
Child's report:  current value = 36
Parent's report: current index =  6
Parent's report: current index =  7
Child's report:  current value = 49
Child's report:  current value = 64
Parent's report: current index =  8
The child is done
Parent's report: current index =  9
The parent is done
alpha@alpha-HP:~/Documents/GitHub/CS310-OS/Lab-4$
```