

# AUP : Assignment - 10 [IPC]

Aditi Rajendra Medhane 111803177

17th November 2021

## Q1

A pipe setup is given below that involves three processes. P is the parent process, and C1 and C2 are child processes, spawned from P. The pipes are named p1, p2, p3, and p4. Write a program that establishes the necessary pipe connections, setups, and carries out the reading/writing of the text in the indicated directions.

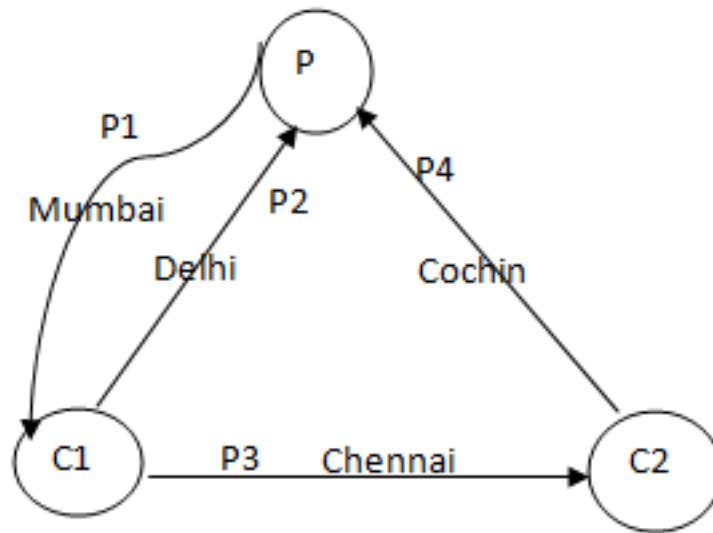


Figure 1: Pipe Setup

## Code

```
1  #include <sys/types.h>
2  #include <fcntl.h>
3  #include <unistd.h>
4  #include <errno.h>
5  #include <stdio.h>
6  #include <string.h>
7
8  #define M1 "Mumbai"
9  #define M2 "Delhi"
10 #define M3 "Chennai"
11 #define M4 "Cochin"
12
13 #define BUFLen 10
14
15 int main(void) {
16
17     int pid, ppid;
18
19     int pipes[5][2];
20
21     int n;
22     char buf[BUFLen];
23 }
```

```

24     int i;
25
26     /* open 4 pipes */
27     for (i = 1; i <= 4; i++) {
28         if (pipe(pipes[i]) == -1) {
29             perror("pipe");
30             return errno;
31         }
32     }
33
34     if ((pid = fork()) == -1) {
35         perror("fork 1 failed");
36         return errno;
37     }
38     else if (!pid) {
39         /* C1 */
40
41         /* 1 W */
42         close(pipes[1][1]);
43
44         /* 2 R */
45         close(pipes[2][0]);
46
47         /* 3 R */
48         close(pipes[3][0]);
49
50         /* 4 RW */
51         close(pipes[4][0]);
52         close(pipes[4][1]);
53
54         pid = getpid();
55         ppid = getppid();
56
57         printf("%d is child of %d\n", pid, ppid);
58
59         if ((n = read(pipes[1][0], buf, BUFLen)) == -1) {
60             perror("read 1");
61             return errno;
62         }
63         printf("%d Read %s\n", pid, buf);
64         close(pipes[1][0]);
65
66         if (write(pipes[2][1], M2, strlen(M2) + 1) == -1) {
67             perror("write 2");
68             return errno;
69         }
70         printf("%d Wrote %s\n", pid, M2);
71         close(pipes[2][1]);
72
73         if (write(pipes[3][1], M3, strlen(M3) + 1) == -1) {
74             perror("write 3");
75             return errno;
76         }
77         printf("%d Wrote %s\n", pid, M3);
78         close(pipes[3][1]);
79
80         return 0;
81     }
82
83     if ((pid = fork()) == -1) {
84         perror("fork 2 failed");
85         return errno;
86     }
87     else if (!pid) {
88         /* C2 */
89
90         /* 1 RW */

```

```

91         close(pipes[1][0]);
92         close(pipes[1][1]);
93
94         /* 2 RW */
95         close(pipes[2][0]);
96         close(pipes[2][1]);
97
98         /* 3 W */
99         close(pipes[3][1]);
100
101         /* 4 R */
102         close(pipes[4][0]);
103
104         pid = getpid();
105         ppid = getppid();
106         printf("%d is child of %d\n", pid, ppid);
107
108         if ((n = read(pipes[3][0], buf, BUFLen)) == -1) {
109             perror("read 3");
110             return errno;
111         }
112         printf("%d Read %s\n", pid, buf);
113         close(pipes[3][0]);
114
115         if (write(pipes[4][1], M4, strlen(M4) + 1) == -1) {
116             perror("write 4");
117             return errno;
118         }
119         printf("%d Wrote %s\n", pid, M4);
120         close(pipes[4][1]);
121
122         return 0;
123     }
124
125     /* P */
126
127     /* close read end of 1 */
128     close(pipes[1][0]);
129
130     /* close write end of 2 */
131     close(pipes[2][1]);
132
133     /* close write end of 4 */
134     close(pipes[4][1]);
135
136     /* close both ends of 3 */
137     close(pipes[3][0]);
138     close(pipes[3][1]);
139
140     pid = getpid();
141     printf("%d is parent\n", pid);
142
143
144     if (write(pipes[1][1], M1, strlen(M1) + 1) == -1) {
145         perror("write 1");
146         return errno;
147     }
148     printf("%d Wrote %s\n", pid, M1);
149     close(pipes[1][1]);
150
151     if ((n = read(pipes[2][0], buf, BUFLen)) == -1) {
152         perror("read 2");
153         return errno;
154     }
155     printf("%d Read %s\n", pid, buf);
156     close(pipes[2][0]);
157

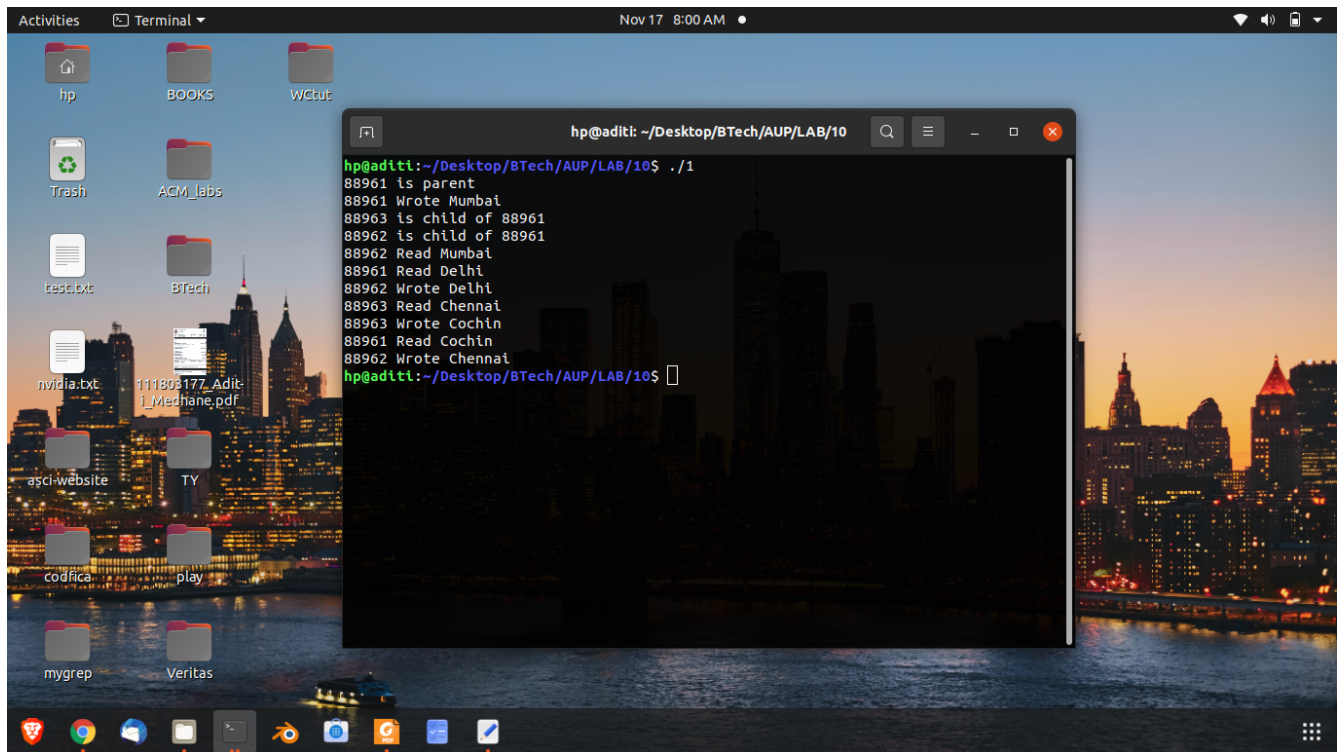
```

```

158     if ((n = read(pipes[4][0], buf, BUFLen)) == -1) {
159         perror("read 4");
160         return errno;
161     }
162     printf("%d Read %s\n", pid, buf);
163     close(pipes[4][0]);
164
165     return 0;
166 }
167

```

## Output



The screenshot shows a Linux desktop with a cityscape wallpaper. A terminal window is open, displaying the output of a program. The output shows a parent process (PID 88961) and its child (PID 88962) communicating via pipes. The parent writes to the pipe, and the child reads it, then the child writes to the pipe, and the parent reads it. The output is as follows:

```

hp@aditi: ~/Desktop/BTech/AUP/LAB/10
hp@aditi:~/Desktop/BTech/AUP/LAB/10$ ./1
88961 is parent
88961 Wrote Mumbai
88962 is child of 88961
88962 is child of 88961
88962 Read Mumbai
88961 Read Delhi
88962 Wrote Delhi
88963 Read Chennai
88963 Wrote Cochin
88961 Read Cochin
88962 Wrote Chennai
hp@aditi:~/Desktop/BTech/AUP/LAB/10$

```

Figure 2: Messages read and written by processes

## Q2

Let P1 and P2 be two processes alternatively writing numbers from 1 to 100 to a file. Let P1 write odd numbers and p2, even. Implement the synchronization between the processes using FIFO.

### Code

```
1  #include <sys/types.h>
2  #include <sys/stat.h>
3  #include <unistd.h>
4  #include <fcntl.h>
5  #include <stdio.h>
6  #include <errno.h>
7  #include <string.h>
8
9
10 #define FIFO1 "/tmp/aup_fifo1"
11 #define FIFO2 "/tmp/aup_fifo2"
12 #define FILENAME "/tmp/aup_file"
13 #define BUFLen 10
14
15 int main(void) {
16
17     int pid;
18     int fr, fw, fp;
19     int i;
20     char buf[BUFLen];
21
22     if (mkfifo(FIFO1, S_IRUSR | S_IWUSR) == -1) {
23         perror("mkfifo 1");
24         return errno;
25     }
26
27     if (mkfifo(FIFO2, S_IRUSR | S_IWUSR) == -1) {
28         perror("mkfifo 2");
29         return errno;
30     }
31
32     if ((fp = open(FILENAME, O_WRONLY | O_CREAT,
33                     S_IRUSR | S_IWUSR)) == -1) {
34         perror("file");
35         return errno;
36     }
37
38     if ((pid = fork()) == -1) {
39         perror("fork");
40         return errno;
41     }
42     else if (pid) {
43         /* P1 */
44
45         if ((fw = open(FIFO1, O_WRONLY)) == -1) {
46             perror("write 1");
47             return errno;
48         }
49
50         if ((fr = open(FIFO2, O_RDONLY)) == -1) {
51             perror("read 2");
52             return errno;
53         }
54
55         i = 1;
56         while (i <= 100) {
57             sprintf(buf, "%d\n", i);
58
59             if (write(fp, buf, strlen(buf)) == -1) {
60                 perror("odd write");
```

```

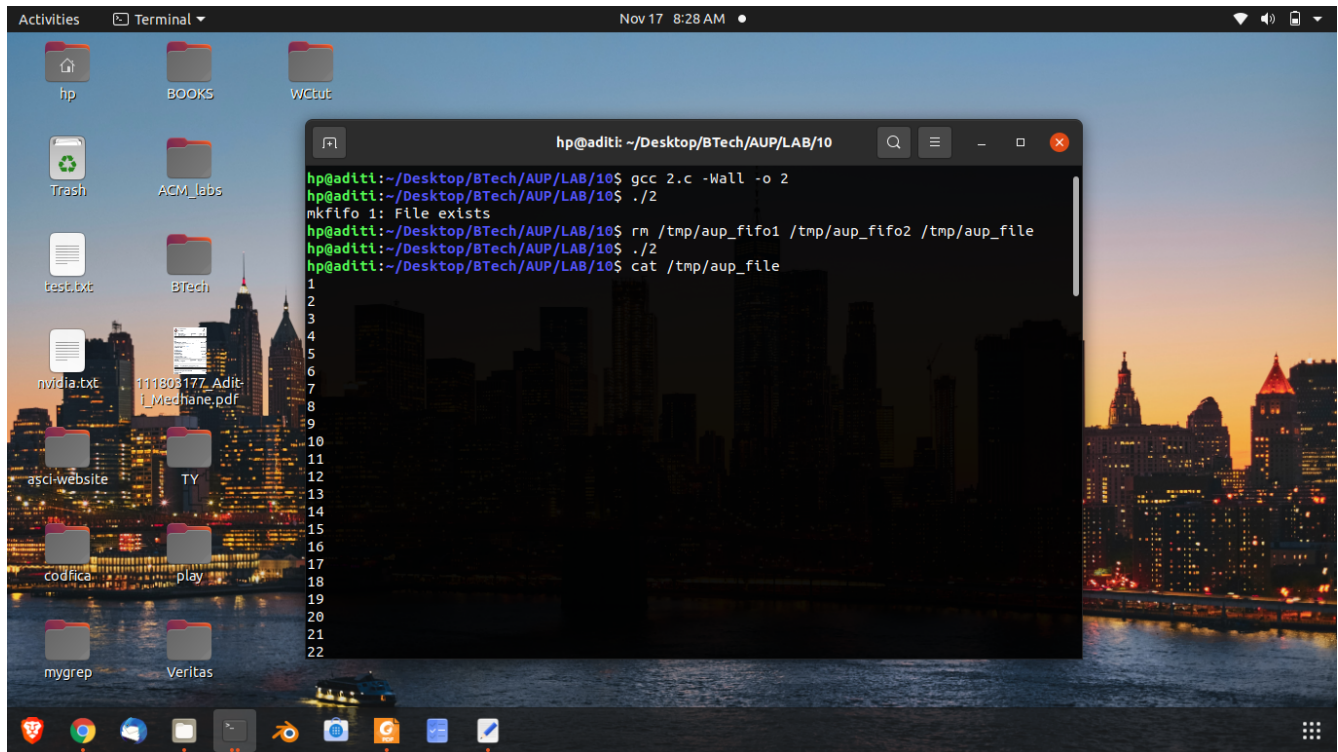
61         return errno;
62     }
63
64     if (write(fw, "*", 1) == -1) {
65         perror("sync write odd");
66         return errno;
67     }
68
69     if (read(fr, buf, 1) == -1) {
70         perror("sync read odd");
71         return errno;
72     }
73
74     i += 2;
75 }
76
77 close(fp);
78 close(fw);
79 close(fr);
80
81 return 0;
82 }
83 else {
84     /* P1 */
85
86     if ((fr = open(FIFO1, O_RDONLY)) == -1) {
87         perror("read 1");
88         return errno;
89     }
90
91     if ((fw = open(FIFO2, O_WRONLY)) == -1) {
92         perror("write 2");
93         return errno;
94     }
95
96     i = 2;
97     while (i <= 100) {
98         if (read(fr, buf, 1) == -1) {
99             perror("sync read even");
100             return errno;
101         }
102
103         sprintf(buf, "%d\n", i);
104
105         if (write(fp, buf, strlen(buf)) == -1) {
106             perror("odd write");
107             return errno;
108         }
109
110         if (write(fw, "*", 1) == -1) {
111             perror("sync write even");
112             return errno;
113         }
114
115         i += 2;
116     }
117
118     close(fp);
119     close(fw);
120     close(fr);
121
122     return 0;
123 }
124 }

```

## Explanation

The program uses two FIFOs, `/tmp/aup_fifo1` and `/tmp/aup_fifo2`. The file which is used for writing the numbers is `/tmp/aup_file`.

## Output



```
hp@aditi: ~/Desktop/BTech/AUP/LAB/10
hp@aditi:~/Desktop/BTech/AUP/LAB/10$ gcc 2.c -Wall -o 2
hp@aditi:~/Desktop/BTech/AUP/LAB/10$ ./2
mkfifo 1: File exists
hp@aditi:~/Desktop/BTech/AUP/LAB/10$ rm /tmp/aup_fifo1 /tmp/aup_fifo2 /tmp/aup_file
hp@aditi:~/Desktop/BTech/AUP/LAB/10$ ./2
hp@aditi:~/Desktop/BTech/AUP/LAB/10$ cat /tmp/aup_file
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
```

Figure 3: Synchronized writes to shared file

### Q3

Implement a producer-consumer setup using shared memory and semaphore. Ensure that data doesn't get over-written by the producer before the consumer reads and displays on the screen. Also ensure that the consumer doesn't read the same data twice.

#### Code

```
1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <sys/mman.h>
4  #include <sys/stat.h>
5  #include <fcntl.h>
6  #include <semaphore.h>
7  #include <errno.h>
8  #include <stdio.h>
9
10 #define BUF_SIZE 5
11 #define N_ITEMS 10
12
13 int main(void) {
14
15     int *buf;
16     sem_t *sem_fill;
17     sem_t *sem_empty;
18     int pid;
19     int i;
20
21     printf("Maximum number of elements in buffer: %d\n", BUF_SIZE);
22     printf("Number of items to be produced and consumed: %d\n", N_ITEMS);
23
24     if ((buf = (int *)mmap(NULL,
25         BUF_SIZE * sizeof(int),
26         PROT_READ | PROT_WRITE,
27         MAP_SHARED | MAP_ANONYMOUS,
28         -1,
29         0)) == (void *)-1) {
30         perror("mmap 1");
31         return errno;
32     }
33
34     if ((sem_fill = (sem_t *)mmap(NULL,
35         sizeof(sem_t),
36         PROT_READ | PROT_WRITE,
37         MAP_SHARED | MAP_ANONYMOUS,
38         -1,
39         0)) == (void *)-1) {
40         perror("mmap fill");
41         return errno;
42     }
43
44     if (sem_init(sem_fill, 1, 0) == -1) {
45         perror("init fill");
46         return errno;
47     }
48
49     if ((sem_empty = (sem_t *)mmap(NULL,
50         sizeof(sem_t),
51         PROT_READ | PROT_WRITE,
52         MAP_SHARED | MAP_ANONYMOUS,
53         -1,
54         0)) == (void *)-1) {
55         perror("mmap 2");
56         return errno;
57     }
58
59     if (sem_init(sem_empty, 1, BUF_SIZE) == -1) {
```



```

60         perror("init empty");
61         return errno;
62     }
63
64     if ((pid = fork()) == -1) {
65         perror("fork");
66         return errno;
67     }
68     else if (pid) {
69         /* parent, producer */
70
71         for (i = 0; i < N_ITEMS; i++) {
72             if (sem_wait(sem_empty) == -1) {
73                 perror("wait in producer");
74                 return errno;
75             }
76
77             buf[i % BUF_SIZE] = i;
78             printf("Writing %d into buffer\n", i);
79
80             if (sem_post(sem_fill) == -1) {
81                 perror("post in producer");
82                 return errno;
83             }
84         }
85     }
86     else {
87         /* child, consumer */
88
89         for (i = 0; i < N_ITEMS; i++) {
90
91             if (sem_wait(sem_fill) == -1) {
92                 perror("wait in consumer");
93                 return errno;
94             }
95
96             printf("Read %d from buffer\n", buf[i % BUF_SIZE]);
97
98             if (sem_post(sem_empty) == -1) {
99                 perror("post in consumer");
100                 return errno;
101             }
102         }
103     }
104
105     return 0;
106 }

```

## Explanation

There is 1 producer and 1 consumer, 10 items are sent through the shared memory in total, capacity of the shared memory is 5 items. ### Output

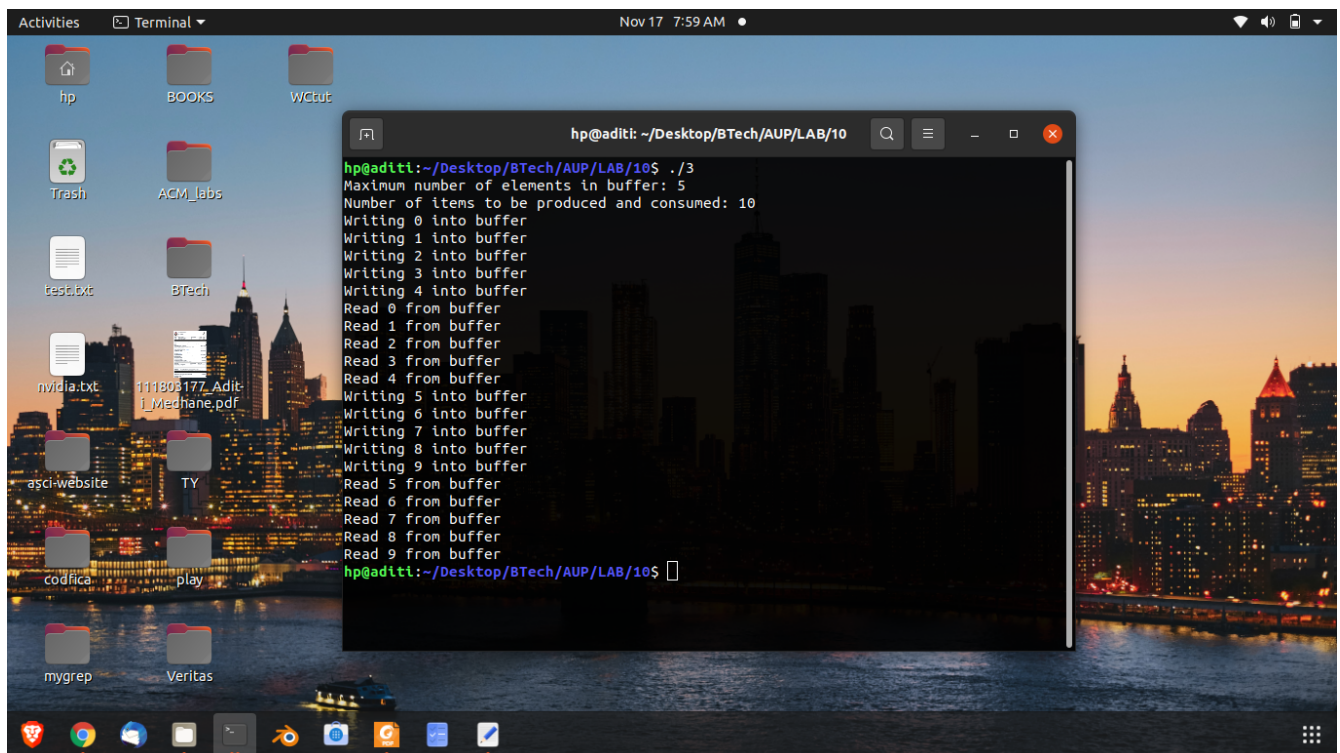


Figure 4: Items written and read without deadlock