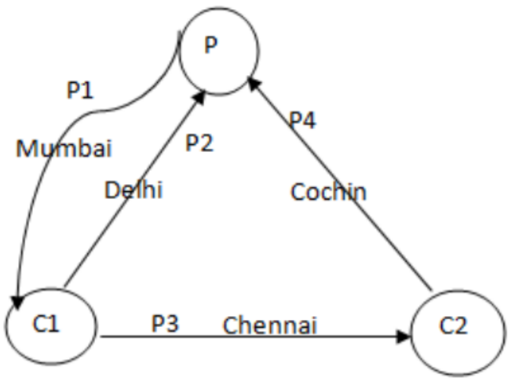
Q1)
A pipe setup is given below that involves three processes. *P* is the parent process, and *C*1 and *C*2 are child processes, spawned from *P*. The pipes are named *p*1, *p*2, *p*3, and *p*4. Write a program that establishes the necessary pipe connections, setups, and carries out the reading/writing of the text in the indicated directions.

Figure :

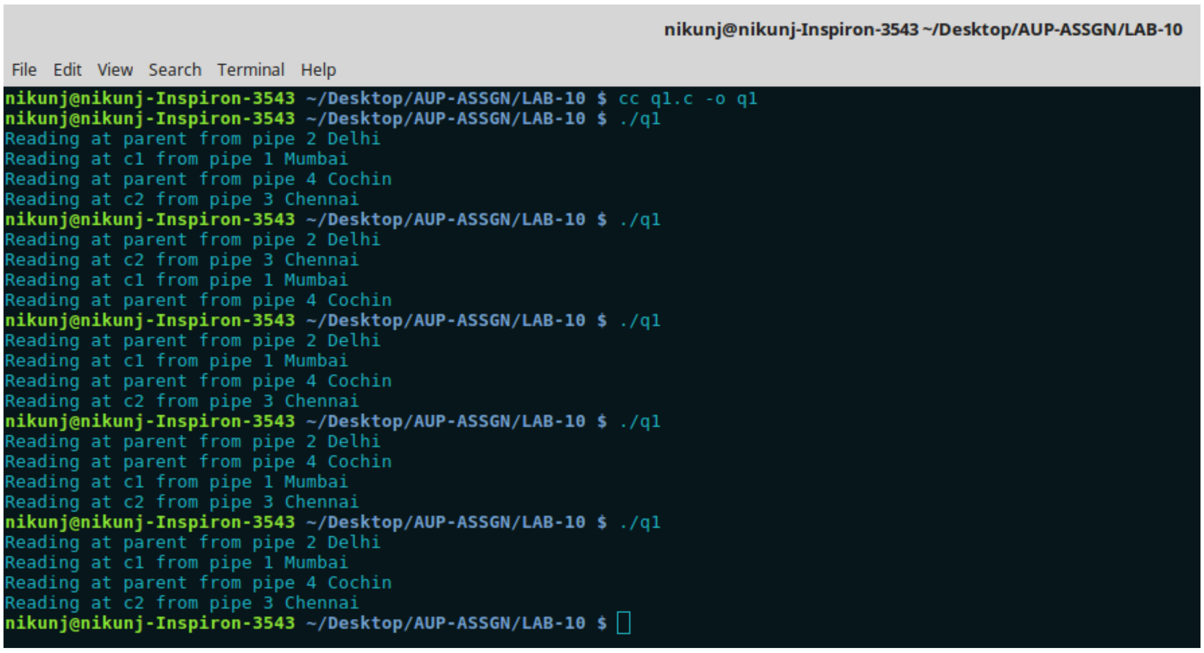


Code :

```
#include <stdio.h>
#include <unistd.h>
#define MAXLINE 1024


int main() {
	int p1[2], p2[2], p3[2], p4[2], n;
	pid_t pid;
	char line[MAXLINE];

	if(pipe(p1) < 0) {
		printf("Pipe 1 creation failed\n");
		return -1;
	}
	if(pipe(p2) < 0) {
		printf("Pipe 2 creation failed\n");
		return -1;
	}
	if(pipe(p3) < 0) {
		printf("Pipe 3 creation failed\n");
		return -1;
	}
	if(pipe(p4) < 0) {
		printf("Pipe 4 creation failed\n");
		return -1;
	}

	if((pid = fork()) > 0) {
		if(fork() > 0) {
			// Parent p
			close(p1[0]);
```

```
				close(p2[1]);
				close(p4[1]);
				write(p1[1], "Mumbai", 6);
				n = read(p2[0], line, sizeof(line));
				printf("Reading at parent from pipe 2 %s\n", line);
				n = read(p4[0], line, sizeof(line));
				printf("Reading at parent from pipe 4 %s\n", line);
			}
			else {
				// Child c2
				close(p4[0]);
				close(p3[1]);
				write(p4[1], "Cochin", 6);
				n = read(p3[0], line, sizeof(line));
				printf("Reading at c2 from pipe 3 %s\n", line);
			}
		}
		else if(pid == 0) {
			// Child c1
			close(p1[1]);
			close(p2[0]);
			close(p3[0]);H
			write(p2[1], "Delhi", 5);
			write(p3[1], "Chennai", 7);
			n = read(p1[0], line, sizeof(line));
			printf("Reading at c1 from pipe 1 %s\n", line);
		}
		else if(pid < 0) {
			printf("Fork Failed\n");
			return -1;
		}
		return 0;
}
```

Execution :


nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10
File Edit View Search Terminal Help

```
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $ cc q1.c -o q1
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $ ./q1
Reading at parent from pipe 2 Delhi
Reading at c1 from pipe 1 Mumbai
Reading at parent from pipe 4 Cochin
Reading at c2 from pipe 3 Chennai
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $ ./q1
Reading at parent from pipe 2 Delhi
Reading at c2 from pipe 3 Chennai
Reading at c1 from pipe 1 Mumbai
Reading at parent from pipe 4 Cochin
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $ ./q1
Reading at parent from pipe 2 Delhi
Reading at c1 from pipe 1 Mumbai
Reading at parent from pipe 4 Cochin
Reading at c2 from pipe 3 Chennai
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $ ./q1
Reading at parent from pipe 2 Delhi
Reading at parent from pipe 4 Cochin
Reading at c1 from pipe 1 Mumbai
Reading at c2 from pipe 3 Chennai
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $ ./q1
Reading at parent from pipe 2 Delhi
Reading at c1 from pipe 1 Mumbai
Reading at parent from pipe 4 Cochin
Reading at c2 from pipe 3 Chennai
nikunj@nikunj-Inspiron-3543 ~/Desktop/AUP-ASSGN/LAB-10 $
```

Here p1, p2, p3, p4 are four pipes. P is the parent process. C1 and C2 are the child process spawned from P. Pipe has been created using pipe() function. p1[0] indicates the read descriptor and p1[1] indicates the write decriptor. Here the ends of the pipes are closed according to the flow of data. The read and write system calls are used and data is written as given in the figure between the processes through the pipes.

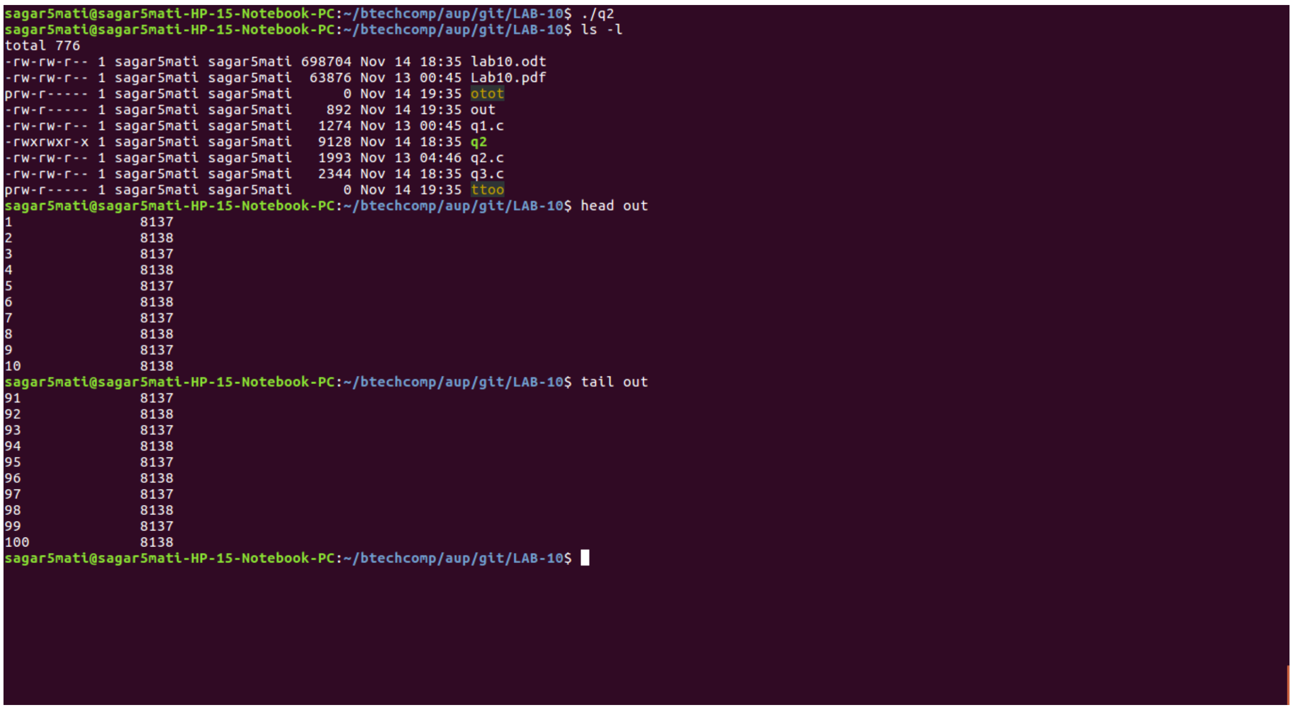
Q2) Let P1 and P2 be two processes alternatively writing numbers from 1 to 100 to a file. Let P1 write odd numbers and p2, even. Implement the synchronization between the processes using FIFO.

Code:

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

int main() {
        int out;
        pid_t odd, even;

        if((out = open("out", O_WRONLY | O_CREAT | O_TRUNC,
                        S_IRUSR | S_IWUSR | S_IRGRP)) == -1) {
                perror("Can't create output file");
                return -1;
        }
        if((mkfifo("otot", S_IRUSR | S_IWUSR | S_IRGRP)) == -1) {
                perror("Can't create FIFO file");
                return -1;
        }
        if((mkfifo("ttoo", S_IRUSR | S_IWUSR | S_IRGRP)) == -1) {
                perror("Can't create FIFO file");
                return -1;
        }

        if((odd = fork()) < 0) {
                perror("Unable to create child process");
                return -1;
        }
        if(odd != 0) {
                if((even = fork()) < 0) {
                        perror("Unable to create child process");
                        return -1;
                }
        }

        if(odd == 0) {
                int otot, ttoo, counter;
                char next;
                pid_t podd;
                podd = getpid();
                next = 'a';

                if((otot = open("otot", O_WRONLY)) == -1) {
                        perror("Unable to open FIFO for writing");
                        return -1;
                }
                if((ttoo = open("ttoo", O_RDONLY)) == -1) {
                        perror("Unable to open FIFO for reading");
                        return -1;
                }

                counter = 1;

                do {
                        dprintf(out, "%d\t\t%d\n", counter, podd);
                        write(otot, &next, sizeof(char));
                        counter += 2;
                }while(read(ttoo, &next, sizeof(char)) && counter != 101);
                close(otot);
                close(ttoo);

                return 0;
        }

        if(even == 0) {
                int otot, ttoo, counter;
                char next;
                pid_t peven;

                peven = getpid();
                next = 'a';

                if((otot = open("otot", O_RDONLY)) == -1) {
                        perror("Unable to open FIFO for writing");
                        return -1;
                }
                if((ttoo = open("ttoo", O_WRONLY)) == -1) {
                        perror("Unable to open FIFO for reading");
                        return -1;
                }

                counter = 2;

                while(read(otot, &next, sizeof(char)) && counter != 102) {
                        dprintf(out, "%d\t\t%d\n", counter, peven);
                        write(ttoo, &next, sizeof(char));
                        counter += 2;
                }
                close(otot);
                close(ttoo);
```

```
		return 0;
	}

	if(odd && even) {
		waitpid(odd, NULL, 0);
		waitpid(even, NULL, 0);
		close(out);
		return 0;
	}

	return 0;
}
```

Execution:



```
sagar5mati@sagar5mati-HP-15-Notebook-PC:~/btechcomp/aup/git/LAB-10$ ./q2
sagar5mati@sagar5mati-HP-15-Notebook-PC:~/btechcomp/aup/git/LAB-10$ ls -l
total 776
-rw-rw-r-- 1 sagar5mati sagar5mati 698704 Nov 14 18:35 lab10.odt
-rw-rw-r-- 1 sagar5mati sagar5mati  63876 Nov 13 00:45 Lab10.pdf
prw-r----- 1 sagar5mati sagar5mati      0 Nov 14 19:35 otot
-rw-r----- 1 sagar5mati sagar5mati    892 Nov 14 19:35 out
-rw-rw-r-- 1 sagar5mati sagar5mati   1274 Nov 13 00:45 q1.c
-rwxrwxr-x 1 sagar5mati sagar5mati   9128 Nov 14 18:35 q2
-rw-rw-r-- 1 sagar5mati sagar5mati   1993 Nov 13 04:46 q2.c
-rw-rw-r-- 1 sagar5mati sagar5mati   2344 Nov 14 18:35 q3.c
prw-r----- 1 sagar5mati sagar5mati      0 Nov 14 19:35 ttoo
sagar5mati@sagar5mati-HP-15-Notebook-PC:~/btechcomp/aup/git/LAB-10$ head out
1               8137
2               8138
3               8137
4               8138
5               8137
6               8138
7               8137
8               8138
9               8137
10              8138
sagar5mati@sagar5mati-HP-15-Notebook-PC:~/btechcomp/aup/git/LAB-10$ tail out
91              8137
92              8138
93              8137
94              8138
95              8137
96              8138
97              8137
98              8138
99              8137
100             8138
sagar5mati@sagar5mati-HP-15-Notebook-PC:~/btechcomp/aup/git/LAB-10$ █
```

Explanation:

Two FIFO files have been used one in each direction i.e. from process 1 to process 2 and vice versa. Process 1 starts by outputting ‘1’ and indicating the other process by writing to the FIFO file. The 2nd process then writes its output and indicates the process 1 using the other FIFO file. This continues and synchronizes the process to write the output. Finally both process close the file and indicate EOF on the FIFO which ends the process.

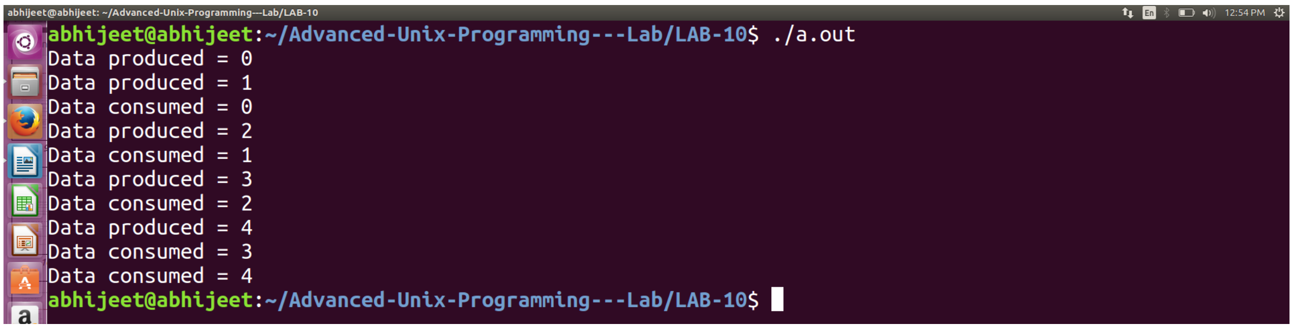
Q3)
Implement a producer-consumer setup using shared memory and semaphore. Ensure that data doesn’t get over-written by the producer before the consumer reads and displays on the screen. Also ensure that the consumer doesn’t read the same data twice.

Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <limits.h>
#define SIZE 4
#define ITERATION 16
union semun {
	int val;
	struct semid_ds *buf;
	unsigned short *array;
	struct seminfo *__buf;
};
int toggle_sem(int semid, int sem_no, int flag, int sem_op) {
	struct sembuf sb;
	sb.sem_num = sem_no;
	sb.sem_op = sem_op;
	sb.sem_flg = flag;
	return semop(semid, &sb, 1);
}
void consumer(key_t key, int shmid) {
	int *data, semid, count;
	data = (int *) shmat(shmid, 0, 0);
	semid = semget(key, 3, IPC_EXCL | 0666);
	if(semid == -1) {
		perror("semget failed ");
		exit(1);
	}
	for(count = 0; count < ITERATION; count++) {
		toggle_sem(semid, 2, 0, -1); //wait till the buffer has at least one element
		toggle_sem(semid, 0, 0, -1); //lock on shared memory
		printf("Data consumed = %d\n", data[count % SIZE]);
		toggle_sem(semid, 0, 0, 1);
		toggle_sem(semid, 1, 0, 1);
	}
}
void producer(key_t key, int shmid) {
	int *data, count;
	data = (int *) shmat(shmid, 0, 0);
	int semid = semget(key, 3, IPC_EXCL | 0666);
	if(semid == -1) {
		perror("Semget failed");
		exit(1);
	}
	for(count = 0; count < ITERATION; count++) {
		toggle_sem(semid, 1, 0, -1);		//wait until the buffer has at least one empty
space
		toggle_sem(semid, 0, 0, -1);		//Lock on shared memory
		data[count % SIZE] = count;
		printf("Data produced = %d\n", data[count % SIZE]);
		toggle_sem(semid, 0, 0, 1);
		toggle_sem(semid, 2, 0, 1);
	}
}
int main(int argc, char *argv[]) {
	int shmid, pid, semid;
	key_t key = 5683;
	shmid = shmget(key, SIZE * sizeof(int), IPC_CREAT | 0600);
	if(shmid == -1) {
		perror("Error while creating Shared memeory: ");
		return 1;
	}
	semid = semget(key, 3, IPC_EXCL | IPC_CREAT | 0666);
	if(semid >= 0) {
		union semun arg;
		arg.val = 1;
		semctl(semid, 0, SETVAL, arg);	//Semaphore for buffer
		arg.val = SIZE;
		semctl(semid, 1, SETVAL, arg); //number of empty elements in buffer
		arg.val = 0;
		semctl(semid, 2, SETVAL, arg); //number of filled elements in buffer
		pid = fork();
		if(pid == 0) {
			consumer(key, shmid);
		}
		else if(pid != -1) {
			producer(key, shmid);
		}
		else {
			perror("Fork failed: ");
		}
	}
	else {
		perror("Faled to get semaphore ");
	}
	semctl(semid, 0, IPC_RMID);
	shmctl(shmid, 0, IPC_RMID);
	return 0;
}
```

Execution:

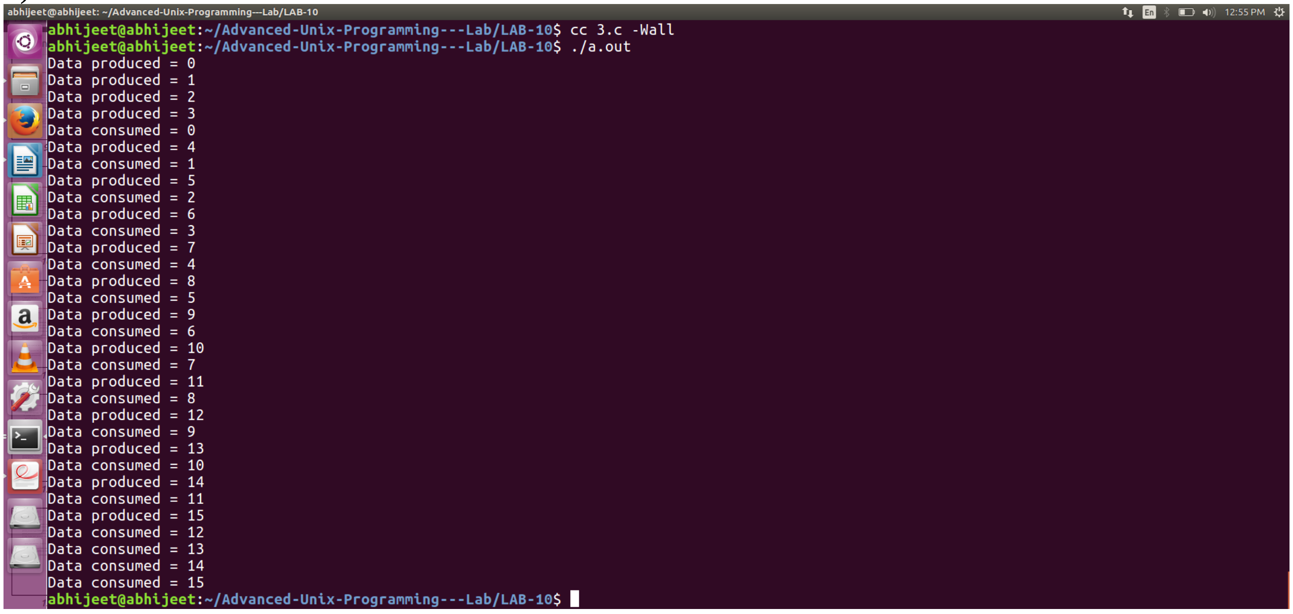
a)



```
abhijeet@abhijeet:~/Advanced-Unix-Programming---Lab/LAB-10$ ./a.out
Data produced = 0
Data produced = 1
Data consumed = 0
Data produced = 2
Data consumed = 1
Data produced = 3
Data consumed = 2
Data produced = 4
Data consumed = 3
Data consumed = 4
abhijeet@abhijeet:~/Advanced-Unix-Programming---Lab/LAB-10$
```

SIZE = 3, ITERATION = 5

b)



```
abhijeet@abhijeet:~/Advanced-Unix-Programming---Lab/LAB-10$ cc 3.c -Wall
abhijeet@abhijeet:~/Advanced-Unix-Programming---Lab/LAB-10$ ./a.out
Data produced = 0
Data produced = 1
Data produced = 2
Data produced = 3
Data consumed = 0
Data produced = 4
Data consumed = 1
Data produced = 5
Data consumed = 2
Data produced = 6
Data consumed = 3
Data produced = 7
Data consumed = 4
Data produced = 8
Data consumed = 5
Data produced = 9
Data consumed = 6
Data produced = 10
Data consumed = 7
Data produced = 11
Data consumed = 8
Data produced = 12
Data consumed = 9
Data produced = 13
Data consumed = 10
Data produced = 14
Data consumed = 11
Data produced = 15
Data consumed = 12
Data consumed = 13
Data consumed = 14
Data consumed = 15
abhijeet@abhijeet:~/Advanced-Unix-Programming---Lab/LAB-10$
```

SIZE = 4, ITERATION = 16

The above program has two macros: SIZE and ITERATION. SIZE is the number of integers that the shared memory should be able to hold, and ITERATION is the number of data that the producer/consumer should produce/consume. Three semaphores have been used.

1) sem_no = 0; this semaphore is used as a mutex lock for the shared memory buffer. Semval is initialized to 1.
2) sem_no = 1; this semphore is used to indicate the number of elements that are empty or have been consumed in the buffer. Semval is initialized to SIZE, since the buffer is empty initially.
3) sem_no = 2; this semaphore is used to indicate the number of elements that are present in the buffer, such that they have not been consumed. Semval is initialized to 0, since the buffer is empty initially.

A toggle_sem function has been created to carry out the semop function call.

After creating the shared memory and the semaphores a fork is called, with the child process acting as the consumer and the parent being the prodcuer.

The producer, first fetches the shared memory and the semaphores, after which it starts producing and filling up the buffer. It first does a -1 on the second semphore (sem_no = 1). This means that if the buffer is full, the producer will wait until at least one block is empty in the buffer. Then it does a -1 on the first semaphore (sem_no = 0) to acquire a lock on the buffer. After producing the data, the lock for the buffer is released by doing a 1 operation on the first semaphore and then a 1 is done on the third semaphore (sem_no = 2) stating that one more element has been filled in the buffer.

The consumer process is similar to that of the producer. During consumption, it frist does a -1 on the third operation, stating that at least one element has to be present in the buffer for it to consume. After this it acquires a lock on the buffer with the first semaphore, consumes the data and then release the lock. Then it does a 1 on the second semaphore, stating that one more element has been consumed and that the buffer has one more empty element.

The shared memory and the semphores are removed at the end of the program.