Advanced Unix Programming
Lab 10
Purva Tendulkar : 111403049

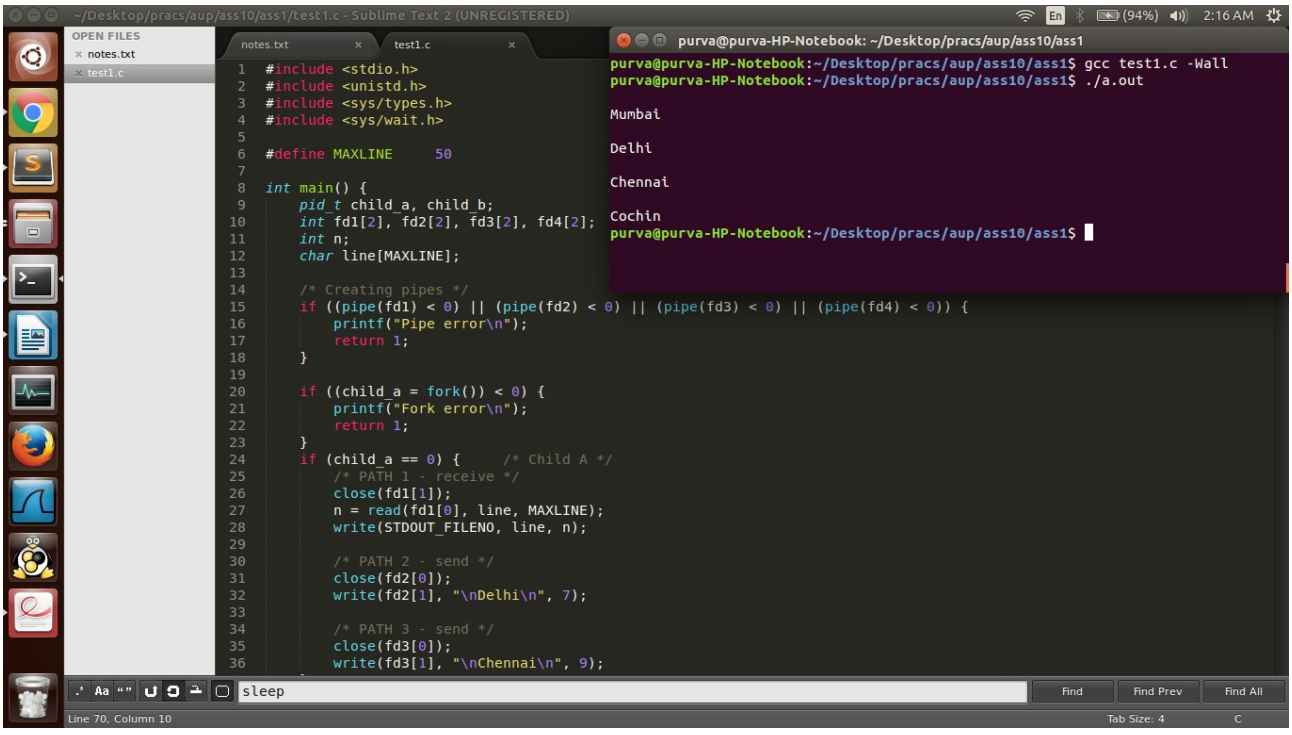
Q1. A pipe setup is given below that involves three processes. P is the parent process, and C1 and C2 are child processes, spawned from P. The pipes are named p1, p2, p3, and p4. Write a program that establishes the necessary pipe connections, setups, and carries out the reading/writing of the text in the indicated directions.

Code :

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

#define MAXLINE		50

int main() {
	pid_t child_a, child_b;
	int fd1[2], fd2[2], fd3[2], fd4[2];
	int n;
	char line[MAXLINE];

	/* Creating pipes */
	if ((pipe(fd1) < 0) || (pipe(fd2) < 0) || (pipe(fd3) < 0) || (pipe(fd4) < 0)) {
		printf("Pipe error\n");
		return 1;
	}

	if ((child_a = fork()) < 0) {
		printf("Fork error\n");
		return 1;
	}
	if (child_a == 0) {		/* Child A */
		/* PATH 1 - receive */
		close(fd1[1]);
		n = read(fd1[0], line, MAXLINE);
		write(STDOUT_FILENO, line, n);

		/* PATH 2 - send */
		close(fd2[0]);
		write(fd2[1], "\nDelhi\n", 7);

		/* PATH 3 - send */
		close(fd3[0]);
		write(fd3[1], "\nChennai\n", 9);
	}
	else {
		if ((child_b = fork()) < 0) {
			printf("Fork error\n");
			return 1;
		}
```

```
        if (child_b == 0) {                 /* Child B */
                /* PATH 3 - receive */
                close(fd3[1]);
                n = read(fd3[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);

                /* PATH 4 - send */
                close(fd4[0]);
                write(fd4[1], "\nCochin\n", 8);
        }
        else {              /* Parent */
                /* PATH 1 - send */
                close(fd1[0]);
                write(fd1[1], "\nMumbai\n", 8);

                /* PATH 2 - receive */
                close(fd2[1]);
                n = read(fd2[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);

                /* PATH 4 - receive */
                close(fd4[1]);
                n = read(fd4[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);

                /* wait for all children to terminate */
                wait(NULL);
        }
    }

    return 0;
}
```

Input & Output Screenshots :



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 :
(A) even file

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include "ass2.h"

#define BUFFER 4
#define INGOING "OddEven.fifo"

extern int fileptr;

int main(int argc, char *argv[]) {
        int n = 2, x = 0;
   char input[BUFFER]={0};
   mkfifo(INGOING, 0666);

   int fd = open(INGOING, O_RDWR);
   fileptr = open("ass2.txt", O_APPEND | O_WRONLY);

   if (fd == -1) {
      perror("open error");
      return 1;
   }

   while (n <= 100) {
        sleep(1);
        while (x != n-1) {
             if (read(fd, input, BUFFER) == -1) {
                perror("read error");
                return 1;
             }
             x = atoi(input);
          }

          sleep(1);
      sprintf(input, "%d", n);
      if (write(fd, input, strlen(input)) == -1) {
             perror("write error");
             return 1;
          }
      //printf("\nEven : wrote %s\n", input);
      write(fileptr, input, strlen(input));
```

```
        n += 2;
    }
    printf("\n");

    close(fileptr);

    return 0;
}
```

(B) odd file

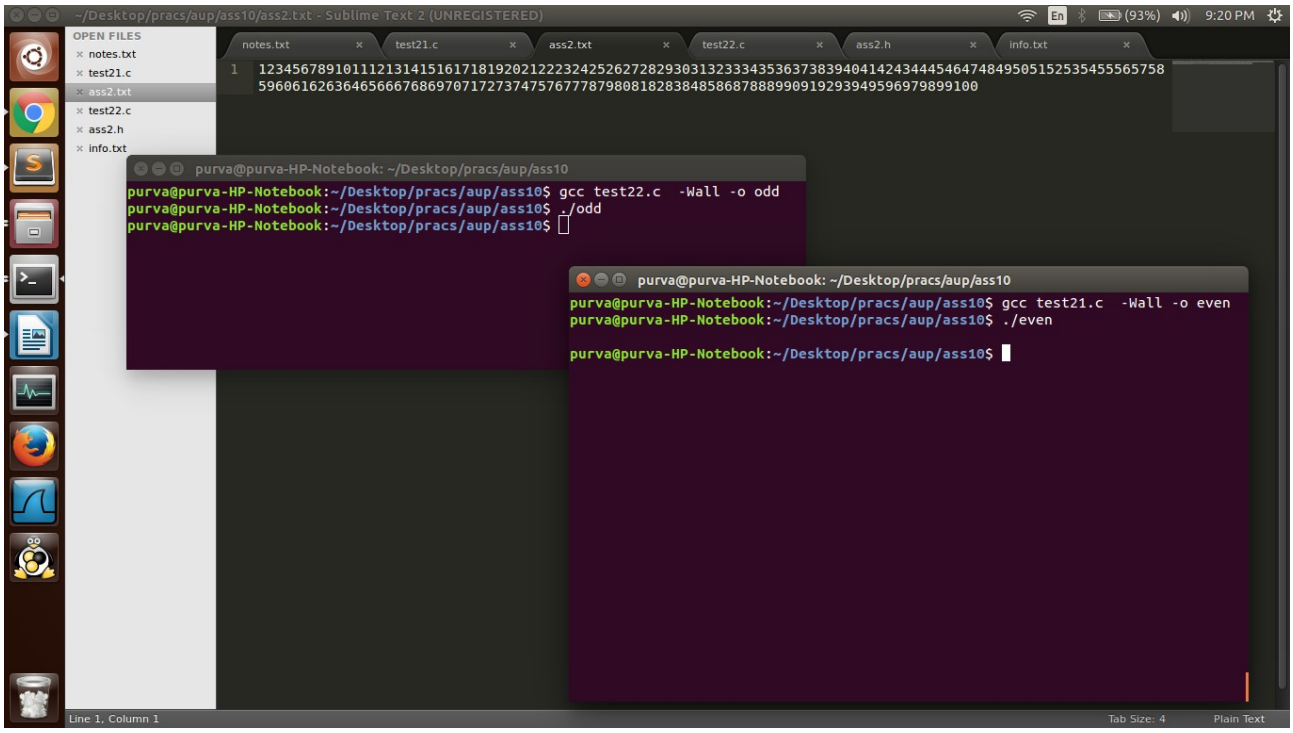
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>
#include "ass2.h"

#define BUFFER 4

extern int fileptr;

int main(int argc, char *argv[]) {
    int n = 1, x = 0;
    char input[BUFFER]={0};

    int fd = open("OddEven.fifo", O_RDWR);
    fileptr = open("ass2.txt", O_APPEND | O_WRONLY);

    if (fd == -1) {
        perror("open error");
    }

    while(n <= 100) {
        sleep(1);
        sprintf(input, "%d", n);
        if (write(fd, input, strlen(input)) == -1) {
            perror("write error");
            return 1;
        }
        //printf("\nOdd : wrote %s\n", input);
        write(fileptr, input, strlen(input));

        sleep(1);
        while (x != n+1) {
            if (read(fd, input, BUFFER) == -1) {
                perror("read error");
                return 1;
            }
            x = atoi(input);
        }
```

```
        n += 2;
    }

    return 0;
}
```

(C) included ass2.h file

```
int fileptr;
```

Input & Output Screenshots :



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 :

(A) Producer

```
#include "shared.h"
extern int *create_shared_mem_buffer();
extern int create_semaphore_set();
extern int get_buffer_size(int *sbuff);
extern void clear_buffer(int *sbuf);

void insert_item(int item, int semid, int *shared_buffer) {
  int index = get_buffer_size(shared_buffer);
  shared_buffer[index] = item;
}

int produce_item() {
  return 0xFF; // nothing dynamic just write a static integer a slot
```

```
}

int main(int argc, const char *argv[]) {
  int *shared_buffer = create_shared_mem_buffer();
  int semid = create_semaphore_set();

  clear_buffer(shared_buffer); // prepare buffer for jobs

  int item = 0;

  int i;
  while(i < 10) {
    item = produce_item();
    semop(semid, &downEmpty, 1);
    semop(semid, &downMutex, 1);
    insert_item(item, semid, shared_buffer);
    debug_buffer(shared_buffer);
    semop(semid, &upMutex, 1);
    semop(semid, &upFull, 1);
    i++;
    sleep(2);
  }

  return EXIT_SUCCESS;
}
```

(B) Consumer

```
#include "shared.h"
extern int *create_shared_mem_buffer();
extern int create_semaphore_set();
extern int get_buffer_size(int *sbuff);
extern void clear_buffer(int *sbuf);

void consume_item(int item) {
  // do something with item
}

int remove_item(int semid, int *shared_buffer) {
  int index = get_buffer_size(shared_buffer) - 1;
  int item = shared_buffer[index];
  shared_buffer[index] = 0x00;
  return item;
}

int main(int argc, const char *argv[]) {
  int *shared_buffer = create_shared_mem_buffer();
  int semid = create_semaphore_set();

  int item = 0;

  int i;
  while(i < 10) {
```

```
    sleep(1);
    semop(semid, &downFull, 1);
    semop(semid, &downMutex, 1);
    item = remove_item(semid, shared_buffer);
    debug_buffer(shared_buffer);
    semop(semid, &upMutex, 1);
    semop(semid, &upEmpty, 1);
    consume_item(item);
    i++;
    sleep(3);
  }

  return EXIT_SUCCESS;
}
```

(C) shared.c

```
#include "shared.h"

/**
 * returns current size of shared buffer
 */
int get_buffer_size(int *sbuff) {
  int i = 0;
  int counter = 0;
  for (i = 0; i < BUFFER_SIZE; ++i) {
    if (sbuff[i] == 0xFF) {
      counter++;
    }
  }
  return counter;
}

void debug_buffer(int *sbuff) {
  int i = 0;
  for (i = 0; i < BUFFER_SIZE; ++i) {
    if (sbuff[i] == 0xFF) printf("1");
  }
  printf("\n");
}

/**
 * returns a pointer to a shared memory buffer that the
 * producer can write to.
 */
int *create_shared_mem_buffer() {
  int *shmaddr = 0; /* buffer address */
  key_t key = SHM_KEY; /* use key to access a shared memory segment */

  int shmid = shmget(key, BUFFER_SIZE, IPC_CREAT | SHM_R | SHM_W); /* give create, read
and write access */
  if (errno > 0) {
    perror("failed to create shared memory segment");
```

```
    exit (EXIT_FAILURE);
  }

  shmaddr = (int*)shmat(shmid, NULL, 0);
  if (errno > 0) {
    perror ("failed to attach to shared memory segment");
    exit (EXIT_FAILURE);
  }

  // clean out garbage memory in shared memory
  return shmaddr;
}

/**
 * only used in the producer to clean out garbage memory when
 * constructing initial buffer.
 */
void clear_buffer(int *sbuff) {
  int i = 0;
  for (i = 0; i < BUFFER_SIZE; ++i) sbuff[i] = 0x00;
}

/**
 * create FULL and EMPTY semaphores and MUTEX
 */
int create_semaphore_set() {
  key_t key = ftok(SEM_KEY, 'E');

  int semid = semget(key, NSEM_SIZE, 0600 | IPC_CREAT);
  if (errno > 0) {
    perror("failed to create semaphore array");
    exit (EXIT_FAILURE);
  }

  semctl(semid, FULL_ID, SETVAL, 0);
  if (errno > 0) {
    perror("failed to set FULL semaphore");
    exit (EXIT_FAILURE);
  }

  semctl(semid, EMPTY_ID, SETVAL, BUFFER_SIZE);
  if (errno > 0) {
    perror("failed to set EMPTY sempahore");
    exit (EXIT_FAILURE);
  }

  semctl(semid, MUTEX_ID, SETVAL, 1);
  if (errno > 0) {
    perror("failed to create mutex");
  }

  return semid;
```

}

(D) shared.h

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define BUFFER_SIZE 5
#define EMPTY_ID 0
#define FULL_ID 1
#define MUTEX_ID 2
#define NSEM_SIZE 3

#define SHM_KEY 9
#define SEM_KEY "."

static struct sembuf downEmpty = { EMPTY_ID, -1, 0 };
static struct sembuf upEmpty = { EMPTY_ID, 1, 0 };
static struct sembuf upFull = { FULL_ID, 1, 0 };
static struct sembuf downFull = { FULL_ID, -1, 0 };
static struct sembuf downMutex = { MUTEX_ID, -1, 0 };
static struct sembuf upMutex = { MUTEX_ID, 1, 0 };

int *create_shared_mem_buffer();
int create_semaphore_set();
int get_buffer_size(int *sbuff);
void clear_buffer(int *sbuf);
```

Input & Output Screenshots :