Program1: To write some data on the standard output device (by default – monitor)

//Name the program file as “write.c”

```
#include<unistd.h>
int main()
{
write(1,"hello\n",6); //1 is the file descriptor, "hello\n" is the data, 6 is the count of characters in data
}
```

Program 2

```
#include<stdio.h
#include<unistd.h>
int main()
{
int count;
count=write(1,"hello\n",6);
printf("Total bytes written: %d\n",count);
}
```

Program 3:

```
#include<unistd.h>
int main()
{
write(1,"hello\n",60); //the bytes to be printed (third parameter) are more than the data specified in 2nd parameter
}
```

Program4:

```
#include<unistd.h>
int main()
{
write(1,"hello\n",3);//the bytes to be printed (third parameter) are less than the data specified in 2nd parameter
}
```

Program5:

```
#include<unistd.h>
#include<stdio.h>
int main()
{
int count;
count=write(3,"hello\n",6); //the file descriptor is not one of the pre-specified ones i.e., 0, 1 or 2
printf("Total bytes written: %d\n",count);
}
```

Program 6: To read data from the standard input device and write it on the screen

//read.c

```
#include<unistd.h>
int main()
{
char buff[20];
read(0,buff,10);//read 10 bytes from standard input device(keyboard), store in buffer (buff)
write(1,buff,10);//print 10 bytes from the buffer on the screen
}
```

Program 7: To read data from the standard input device and write it on the screen
//read.c

```
#include<unistd.h>
int main()
{
int nread;
char buff[20];
nread=read(0,buff,10);//read 10 bytes from standard input device(keyboard), store in buffer (buff)
write(1,buff,nread);//print 10 bytes from the buffer on the screen
}
```

Program 1: Write a program using open() system call to read the first 10 characters of an existing file “test.txt” and print them on screen.

```
//open.c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
int n,fd;
char buff[50];
fd=open("test.txt",O_RDONLY); //opens test.txt in read mode and the file descriptor is saved in integer fd.
printf("The file descriptor of the file is: %d\n,fd); // the value of the file descriptor is printed.
n=read(fd,buff,10);//read 10 characters from the file pointed to by file descriptor fd and save them in buffer (buff)
write(1,buff,n); //write on the screen from the buffer
}
```

Program2: To read 10 characters from file “test.txt” and write them into non-existing file “towrite.txt”

```
//open2.c
#include<unistd.h>
#include<sys/types.h>
#include<sys/stat.h>
#include<fcntl.h>
int main()
{
int n,fd,fd1;
char buff[50];
```

```
fd=open("test.txt",O_RDONLY);
n=read(fd,buff,10);
fd1=open("towrite.txt",O_WRONLY|O_CREAT,0642);//use the pipe symbol (|) to separate O_WRONLY and O_CREAT
write(fd1,buff,n);
}
```

Program1: Program using lseek() system call that reads 10 characters from file “seeking” and print on screen. Again read 10 characters and write on screen.

```
#include<unistd.h>
#include<fcntl.h>
#include<sys/types.h
#include<sys/stat.h>
#int main()
{
int n,f;
char buff[10];
f=open("seeking",O_RDWR);
read(f,buff,10);
write(1,buff,10);
read(f,buff,10);
write(1,buff,10);
}
```

Program2: Program using lseek() system call that reads 10 characters from file “seeking” and print on screen. Skip next 5 characters and again read 10 characters and write on screen.

```
#include<unistd.h>
#include<fcntl.h>
#include<sys/types.h
#include<sys/stat.h>
int main()
{
int n,f;
char buff[10];
f=open("seeking",O_RDWR);
read(f,buff,10);
write(1,buff,10);
lseek(f,5,SEEK_CUR);//skips 5 characters from the current position
read(f,buff,10);
write(1,buff,10);
}
```

Program3: Write a program to print 10 characters starting from the 10th character from a file “seeking”.

```
//Let the contents of the file F1 be “1234567890abcdefghijxxxxxxxx”. This means we want the output to be “abcdefghij”.
//Note: the first character ‘1’ is at 0th position

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

```
int main()
{
int n,f,f1;
char buff[10];
f=open("seeking",O_RDWR);
f1=lseek(f,10,SEEK_SET);
printf("Pointer is at %d position\n",f1);
read(f,buff,10);
write(1,buff,10);
}
```

Program 1: Program for dup() system call in C to duplicate a file descriptor.

```
//dup.c
#include<unistd.h>
#include<stdio.h>
3include<fcntl.h>
int main()
{
int old_fd, new_fd;
old_fd=open("test.txt",O_RDWR);
printf("File descriptor is %d\n",old_fd);
new_fd=dup(old_fd);
printf("New file descriptor is %d\n",new_fd);
}
```

Program 2: Program to use dup2() system call in linux to duplicate a file descriptor.

```
//dup2.c
#include<unistd.h>
#include<stdio.h>
3include<fcntl.h>
int main()
{
int old_fd, new_fd;
old_fd=open("test.txt",O_RDWR);
printf("File descriptor is %d\n",old_fd);
new_fd=dup2(old_fd,7);
printf("New file descriptor is %d\n",new_fd);
}
```

Program 3: Program to show that both file descriptor point to the same file and same pointer position is maintained

```
//create a file test.txt with the content "1234567890abcdefghij54321"
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
int old_fd, new_fd;
char buff[10];
old_fd=open("test.txt",O_RDWR);
read(old_fd,buff,10);//read first 10 characters using old file descriptor
write(1,buff,10);//prints them on screen
new_fd=dup(old_fd);//duplicates file descriptor
read(old_fd,buff,10);//this read will read the next 10 characters even if new file descriptor is used
write(1,buff,10);
```

```
}
```

Program for fork() system call

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else{
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
}
```

Program for wait() system call which makes the parent process wait for the child to finish.

```
#include<unistd.h>
#include<sys/types.h>
#include<stdio.h>
#include<sys/wait.h>
int main()
{
pid_t p;
printf("before fork\n");
p=fork();
if(p==0)//child
{
printf("I am child having id %d\n",getpid());
printf("My parent's id is %d\n",getppid());
}
else//parent
{
wait(NULL);
printf("My child's id is %d\n",p);
printf("I am parent having id %d\n",getpid());
}
printf("Common\n");
}
```

Program 2: Program to create an orphan process

```
#include<stdio.h
#include<unistd.h>
#include<sys/types.h>
int main()
{
```

```
pid_t p;
p=fork();

if(p==0)
{
sleep(5); //child goes to sleep and in the mean time parent terminates
printf("I am child having PID %d\n",getpid());
printf("My parent PID is %d\n",getppid());
}
else
{
printf("I am parent having PID %d\n",getpid());
printf("My child PID is %d\n",p);
}
}
```

```
//zombie.c
#include<stdio.h>
#include<unistd.h>
int main()
{
pid_t t;
t=fork();
if(t==0)
{
printf("Child having id %d\n",getpid());
}
else
{
printf("Parent having id %d\n",getpid());
sleep(15); // Parent sleeps. Run the ps command during this time
}
}
```

```
//execl.c
#include<stdio.h>
#include<unistd.h>
int main()
{
printf("Before execl\n");
execl("/bin/ps","ps","-a",NULL);//
printf("After execlp\n");
}
```

```
//threads
#include<pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,void *(*start_routine) (void *),
void arg);
```

Program 1: Program to create threads in linux. Thread prints 0-4 while the main process prints 20-24

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

```
#include<unistd.h>
#include<pthread.h>
void *thread_function(void *arg);
  int i,j;
  int main() {
  pthread_t a_thread;    //thread declaration

  pthread_create(&a_thread, NULL, thread_function, NULL);
//thread is created
  pthread_join(a_thread, NULL); //process waits for thread to finish . //Comment this line to see the difference
  printf("Inside Main Program\n");
  for(j=20;j<25;j++)
  {
  printf("%d\n",j);
  sleep(1);
  }
  }
void *thread_function(void *arg) {
// the work to be done by the thread is defined in this function
  printf("Inside Thread\n");
  for(i=0;i<5;i++)
  {
  printf("%d\n",i);
  sleep(1);
  }
  }
```

Program 2: Program to create a thread. The thread prints numbers from zero to n, where value of n is passed from the main process to the thread. The main process also waits for the thread to finish first and then prints from 20-24.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<pthread.h>
#include<string.h>
void *thread_function(void *arg);
  int i,n,j;
  int main() {
  char *m="5";
  pthread_t a_thread;    //thread declaration
  void *result;
  pthread_create(&a_thread, NULL, thread_function, m); //thread is created
  pthread_join(a_thread, &result);
  printf("Thread joined\n");
  for(j=20;j<25;j++)
  {
  printf("%d\n",j);
  sleep(1);
  }
  printf("thread returned %s\n",(char *)result);
  }
  void *thread_function(void *arg) {
  int sum=0;
  n=atoi(arg);
for(i=0;i<n;i++)
```

```
 {
 printf("%d\n",i);
 sleep(1);
 }
 pthread_exit("Done"); // Thread returns "Done"
 }
```

Program 3: Program to create a thread. The thread is passed more than one input from the main process. For passing multiple inputs we need to create structure and include all the variables that are to be passed in this structure.

```
#include<stdio.h
#include<pthread.h>
struct arg_struct {     //structure which contains multiple variables that are to passed as input to the thread
        int arg1;
        int arg2;
 };
void *arguments(void *arguments)
{
      struct arg_struct *args=arguments;
      printf("%d\n", args -> arg1);
      printf("%d\n", args -> arg2);
      pthread_exit(NULL);
}
int main()
 {
        pthread_t t;
        struct arg_struct args;
        args.arg1 = 5;
        args.arg2 = 7;
        pthread_create(&t, NULL, arguments, &args);
//structure passed as 4th argument
        pthread_join(t, NULL); /* Wait until thread is finished */
}
```

```
//deadlock
#include<stdio.h>
#include<pthread.h>
#include<unistd.h>
void *function1();
void *function2();
pthread_mutex_t first_mutex; //mutex lock
pthread_mutex_t second_mutex;
int main() {
pthread_mutex_init(&first_mutex,NULL); //initialize the lock
pthread_mutex_init(&second_mutex,NULL);
pthread_t one, two;
pthread_create(&one, NULL, function1, NULL); // create thread
pthread_create(&two, NULL, function2, NULL);
pthread_join(one, NULL);
pthread_join(two, NULL);
printf("Thread joined\n");
}
void *function1( ) {
pthread_mutex_lock(&first_mutex); // to acquire the resource/mutex lock
```

```
printf("Thread ONE acquired first_mutex\n");
sleep(1);
pthread_mutex_lock(&second_mutex);
printf("Thread ONE acquired second_mutex\n");
pthread_mutex_unlock(&second_mutex); // to release the resource
printf("Thread ONE released second_mutex\n");
pthread_mutex_unlock(&first_mutex);
printf("Thread ONE released first_mutex\n");
}
void *function2( ) {
pthread_mutex_lock(&second_mutex);
printf("Thread TWO acquired second_mutex\n");
sleep(1);
pthread_mutex_lock(&first_mutex);
printf("Thread TWO acquired first_mutex\n");
pthread_mutex_unlock(&first_mutex);
printf("Thread TWO released first_mutex\n");
pthread_mutex_unlock(&second_mutex);
printf("Thread TWO released second_mutex\n");
}


//race condition
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
int main()
{
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last updated value of shared
variable
}
void *fun1()
{
int x;
x=shared;//thread one reads value of shared variable
printf("Thread1 reads the value of shared variable as %d\n",x);
x++; //thread one increments its value
printf("Local updation by Thread1: %d\n",x);
sleep(1); //thread one is preempted by thread 2
shared=x; //thread one updates the value of shared variable
printf("Value of shared variable updated by Thread1 is: %d\n",shared);
}
void *fun2()
{
int y;
y=shared;//thread two reads value of shared variable
printf("Thread2 reads the value as %d\n",y);
y--; //thread two increments its value
printf("Local updation by Thread2: %d\n",y);
```

```
sleep(1); //thread two is preempted by thread 1
shared=y; //thread one updates the value of shared variable
printf("Value of shared variable updated by Thread2 is: %d\n",shared);
}


//no deadlock & race condition
#include<pthread.h>
#include<stdio.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
pthread_mutex_t l; //mutex lock
int main()
{
pthread_mutex_init(&l, NULL); //initializing mutex locks
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last updated value of shared
variable
}
void *fun1()
{
int x;
printf("Thread1 trying to acquire lock\n");
pthread_mutex_lock(&l); //thread one acquires the lock. Now thread 2 will not be able to
acquire the lock //until it is unlocked by thread 1
printf("Thread1 acquired lock\n");
x=shared;//thread one reads value of shared variable
printf("Thread1 reads the value of shared variable as %d\n",x);
x++; //thread one increments its value
printf("Local updation by Thread1: %d\n",x);
sleep(1); //thread one is pre
shared=x; //thread one updates the value of shared variable
printf("Value of shared variable updated by Thread1 is: %d\n",shared);
pthread_mutex_unlock(&l);
printf("Thread1 released the lock\n");
}
void *fun2()
{
int y;
printf("Thread2 trying to acquire lock\n");
pthread_mutex_lock(&l);
printf("Thread2 acquired lock\n");
y=shared;//thread two reads value of shared variable
printf("Thread2 reads the value as %d\n",y);
y--; //thread two increments its value
printf("Local updation by Thread2: %d\n",y);
sleep(1); //thread two is preempted by thread 1
shared=y; //thread one updates the value of shared variable
printf("Value of shared variable updated by Thread2 is: %d\n",shared);
pthread_mutex_unlock(&l);
printf("Thread2 released the lock\n");
```

```
}


//semaphore
#include<pthread.h>
#include<stdio.h>
#include<semaphore.h>
#include<unistd.h>
void *fun1();
void *fun2();
int shared=1; //shared variable
sem_t s; //semaphore variable
int main()
{
sem_init(&s,0,1); //initialize semaphore variable - 1st argument is address of variable, 2nd is number of processes sharing semaphore, 3rd argument is the initial value of semaphore variable
pthread_t thread1, thread2;
pthread_create(&thread1, NULL, fun1, NULL);
pthread_create(&thread2, NULL, fun2, NULL);
pthread_join(thread1, NULL);
pthread_join(thread2,NULL);
printf("Final value of shared is %d\n",shared); //prints the last updated value of shared variable
}
void *fun1()
{
int x;
sem_wait(&s); //executes wait operation on s
x=shared;//thread1 reads value of shared variable
printf("Thread1 reads the value as %d\n",x);
x++; //thread1 increments its value
printf("Local updation by Thread1: %d\n",x);
sleep(1); //thread1 is preempted by thread 2
shared=x; //thread one updates the value o
printf("Value of shared variable updated by Thread1 is: %d\n",shared);
sem_post(&s);
}
void *fun2()
{
int y;
sem_wait(&s);
y=shared;//thread2 reads value of shared variable
printf("Thread2 reads the value as %d\n",y);
y--; //thread2 increments its value
printf("Local updation by Thread2: %d\n",y);
sleep(1); //thread2 is preempted by thread 1
shared=y; //thread2 updates the value of shared variable
printf("Value of shared variable updated by Thread2 is: %d\n",shared);
sem_post(&s);
}


//dining philosopher problem
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
```

```
#include<semaphore.h>
#include<unistd.h>
sem_t chopstick[5];
void * philos(void *);
void eat(int);
int main()
{
int i,n[5];
pthread_t T[5];
for(i=0;i<5;i++)
sem_init(&chopstick[i],0,1);
for(i=0;i<5;i++){
n[i]=i;
pthread_create(&T[i],NULL,philos,(void *)&n[i]);
}
for(i=0;i<5;i++)
pthread_join(T[i],NULL);
}
void * philos(void * n)
{
int ph=*(int *)n;
printf("Philosopher %d wants to eat\n",ph);
printf("Philosopher %d tries to pick left chopstick\n",ph);
sem_wait(&chopstick[ph]);
printf("Philosopher %d picks the left chopstick\n",ph);
printf("Philosopher %d tries to pick the right chopstick\n",ph);
sem_wait(&chopstick[(ph+1)%5]);
printf("Philosopher %d picks the right chopstick\n",ph);
eat(ph);
sleep(2);
printf("Philosopher %d has finished eating\n",ph);
sem_post(&chopstick[(ph+1)%5]);
printf("Philosopher %d leaves the right chopstick\n",ph);
sem_post(&chopstick[ph]);
printf("Philosopher %d leaves the left chopstick\n",ph);
}
void eat(int ph)
{
printf("Philosopher %d begins to eat\n",ph);
}
```

```
//IPC-inter process communication
#include<stdio.h>
FILE *popen(const char *command, const char *type)

#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main()
{
FILE *rd;
char buffer[50];
sprintf(buffer,"name first");
rd=popen("wc -c","w"); // wc -c -> is the process which counts the number of characters passed. 2nd parameter is "w" which means pipe is opened in writing mode
```

```
fwrite(buffer,sizeof(char),strlen(buffer),rd); // to write the data into the pipe
pclose(rd);
}
```

Program to read from a pipe i.e. to receive data from another process

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<string.h>
int main()
{
FILE *rd;
char buffer[50];
rd=popen("ls","r"); //pipe opened in reading mode
fread(buffer, 1, 40, rd); //will read only 50 characters
printf("%s\n", buffer);
pclose(rd);
}
```

Program for IPC using named pipes (mkfifo())

```
#include<sys/types.h>
#include<sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Program1: Creating fifo/named pipe

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/stat.h>
int main()
{
int res;
res = mkfifo("fifo1",0777); //creates a named pipe with the name fifo1
printf("named pipe created\n");
}
```

Program2: Writing to a fifo/named pipe (2.c)

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
int res,n;
res=open("fifo1",O_WRONLY);
write(res,"Message",7);
printf("Sender Process %d sent the data\n",getpid());
}
```

Program 3: Reading from the named pipe (3.c)

```
#include<unistd.h>
#include<stdio.h>
#include<fcntl.h>
int main()
{
int res,n;
char buffer[100];
```

```
res=open("fifo1",O_RDONLY);
n=read(res,buffer,100);
printf("Reader process %d started\n",getpid());
printf("Data received by receiver %d is: %s\n",getpid(), buffer);
}
```

This program creates a shared memory segment, attaches itself to it and then writes some content into the shared memory segment.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666|IPC_CREAT); //creates shared memory segment with key 2345, having size 1024 bytes. IPC_CREAT is used to create the shared segment if it does not exist. 0666 are the permisions on the shared segment
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory); //this prints the address where the segment is attached with this process
printf("Enter some data to write to shared memory\n");
read(0,buff,100); //get some input from user
strcpy(shared_memory,buff); //data written to shared memory
printf("You wrote : %s\n",(char *)shared_memory);
}
```

This program attaches itself to the shared memory segment created in Program 1. Finally, it reads the content of the shared memory

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/shm.h>
#include<string.h>
int main()
{
int i;
void *shared_memory;
char buff[100];
int shmid;
shmid=shmget((key_t)2345, 1024, 0666);
printf("Key of shared memory is %d\n",shmid);
shared_memory=shmat(shmid,NULL,0); //process attached to shared memory segment
printf("Process attached at %p\n",shared_memory);
printf("Data read from shared memory is : %s\n",(char *)shared_memory);
}
```

Program for IPC using message queues

```
int msgget(key_t key, int msgflg);
int msgsnd(int msqid, const void *msg_ptr, size_t msg_sz, int msgflg);
int msgrcv(int msqid, void *msg_ptr, size_t msg_sz, long int msgtype, int msgflg);
int msgctl(int msqid, int command, struct msqid_ds *buf);
```

Command Description
IPC_STAT Sets the data in the msqid_ds structure to reflect the values associated with the message queue.
IPC_SET If the process has permission to do so, this sets the values associated with the message queue to those provided in the msqid_ds data structure.
IPC_RMID Deletes the message queue.

Program 1: Program for IPC using Message Queues To send data to a message queue

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
#define MAX_TEXT 512 //maximum length of the message that can be sent allowed
struct my_msg{
long int msg_type;
char some_text[MAX_TEXT];
};
int main()
{
int running=1;
int msgid;
struct my_msg some_data;
char buffer[50]; //array to store user input
msgid=msgget((key_t)14534,0666|IPC_CREAT);
if (msgid == -1) // -1 means the message queue is not created
{
printf("Error in creating queue\n");
exit(0);
}

while(running)
{
printf("Enter some text:\n");
fgets(buffer,50,stdin);
some_data.msg_type=1;
strcpy(some_data.some_text,buffer);
if(msgsnd(msgid,(void *)&some_d
MAX_TEXT,0)==-1) // msgsnd returns -1 if the message is not sent
{
printf("Msg not sent\n");
}
if(strncmp(buffer,"end",3)==0)
{
running=0;
}
}
}
```

Program 2: Program for IPC using Message Queues To receive/read message from the above -created message queue

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/msg.h>
struct my_msg{
long int msg_type;
char some_text[BUFSIZ];
};
int main()
{
int running=1;
int msgid;
struct my_msg some_data;
long int msg_to_rec=0;
msgid=msgget((key_t)12345,0666|IPC_CREAT);
while(running)
{
msgrcv(msgid,(void *)&some_data,BUFSIZ,msg_to_rec,0);
printf("Data received: %s\n",some_data.some_text);
if(strncmp(some_data.some_text,"end",3)==0)
{
running=0;
}
}
msgctl(msgid,IPC_RMID,0);
}
```