```c
// SJF.
#include <stdio.h>
#include <stdlib.h>
struct process{
    int pid;
    int AT;
    int BT;
    int WT;
    int TAT;
    int CT;};
int main(){
    int totalTAT, totalWT, currentTime = 0;
    int n;
    printf("Enter the number of process:");
    scanf("%d", &n);
    struct process *p = (struct process *)malloc(n * sizeof(struct process *));
    for (int i = 0; i < n; i++){
        p[i].pid = i + 1;
        printf("Enter arrival time for process %d:", i + 1);
        scanf("%d", &p[i].AT);
        printf("Enter brust time for process %d:", i + 1);
        scanf("%d", &p[i].BT);
        p[i].CT = 0;}
    for (int i = 0; i < n - 1; i++){
        for (int j = 0; j < n - i - 1; j++){
            if (p[j].AT > p[j + 1].AT){
                struct process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;}}}
    for (int i = 0; i < n; i++){
        if (currentTime < p[i].AT){
            currentTime = p[i].AT;}
        p[i].WT = currentTime - p[i].AT;
        p[i].TAT = p[i].WT + p[i].BT;
        currentTime += p[i].BT;
        totalTAT += p[i].TAT;
        totalWT += p[i].WT;}
    printf("Process AT  BT  WT  TAT\n");
    for (int i = 0; i < n; i++) {
        printf("%5d %5d %5d %5d %5d\n", p[i].pid, p[i].AT, p[i].BT, p[i].WT,
p[i].TAT); }
    printf("Avarage waiting time = %.2d\n", totalWT / n);
    printf("Avarage turnaround time =%.2d\n", totalTAT / n);
    free(0);return 0;}
```

```c
#include<stdio.h>
#include<stdlib.h>
typedef struct{
    int PID,AT,BT,ST,TAT,CT,WT;
}process;
int main(){
    int n,i,j,lc=0;
    float twt=0,ttat=0,awt=0,atat=0;
    process p[20],temp;
    printf("Enter the number of the processes: ");
    scanf("%d",&n);
    printf("\n");
    for(i = 0 ; i<n ;i++){
        printf("Enter PID AT BT of the process %d : ",i+1);
        scanf("%d%d%d",&p[i].PID,&p[i].AT,&p[i].BT);
        printf("\n");}
    for(i = 0; i<n-1 ; i++){   ////sorting the AT
        for(j = 0 ; j < n-1 ; j++){
            if(p[j].AT > p[j+1].AT){
            temp = p[j];
            p[j] = p[j+1];
            p[j+1] = temp;}}}    //calculation :
    p[0].WT = 0;
    for(i =0 ; i<n;i++){
        if(p[i].AT > lc)
        p[i].ST = p[i].AT;
        else
        p[i].ST = lc;
        p[i].CT = p[i].ST + p[i].BT;
        p[i].TAT = p[i].CT - p[i].AT;
        p[i].WT = p[i].TAT- p[i].BT;
        lc = p[i].CT;}
    printf("PID\tAT\tBT\tST\tCT\tTAT\tWT\t");
    printf("\n");

for(i=0;i<n;i++){     printf("%d\t%d\t%d\t%d\t%d\t%d\t%d",p[i].PID,p[i].AT,p[i].B
T,p[i].ST,p[i].CT,p[i].TAT,p[i].WT);
        printf("\n");
        ttat += p[i].TAT;
        twt += p[i].WT; }
    atat = ttat/n;
    awt = twt/n;
    printf("AVG TAT : %f\n",atat);
    printf("AVG WT : %f\n",awt);
    return 0;}
#include <stdio.h>
typedef struct {
    int pid; // Process ID
    int arrival_time; // Arrival time
    int burst_time; // Burst time
    int remaining_time; // Remaining burst time
    int completion_time; // Completion time
    int waiting_time; // Waiting time
    int turnaround_time; // Turnaround time
} Process;
void calculateTimes(Process processes[], int n, int quantum) {
    int time = 0;
    int completed = 0;
    int i;
    while (completed != n) {
      for (i = 0; i < n; i++) {
     if (processes[i].arrival_time <= time && processes[i].remaining_time > 0) {
      if (processes[i].remaining_time > quantum) {
                    time += quantum;
                    processes[i].remaining_time -= quantum;
                } else {
                    time += processes[i].remaining_time;
                    processes[i].completion_time = time;
                    processes[i].remaining_time = 0;
                    completed++;}}}}
    for (i = 0; i < n; i++) {
        processes[i].turnaround_time = processes[i].completion_time -
processes[i].arrival_time;
        processes[i].waiting_time = processes[i].turnaround_time -
processes[i].burst_time;}}
void printProcessTable(Process processes[], int n) {
    printf("PID\tArrival Time\tBurst Time\tCompletion Time\tTurnaround
Time\tWaiting Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t%d\t\t%d\t\t%d\t\t%d\t\t%d\n", processes[i].pid,
processes[i].arrival_time, processes[i].burst_time, processes[i].completion_time,
processes[i].turnaround_time, processes[i].waiting_time);}}
int main() {
    int n, quantum;
    printf("Enter the number of processes: ");
    scanf("%d", &n);
    Process processes[n];
    for (int i = 0; i < n; i++) {
        processes[i].pid = i + 1;
        printf("Enter arrival time and burst time for process %d: ", i + 1);
        scanf("%d%d", &processes[i].arrival_time, &processes[i].burst_time);
        processes[i].remaining_time = processes[i].burst_time;}
    printf("Enter the time quantum: ");
    scanf("%d", &quantum);
    calculateTimes(processes, n, quantum);
    printProcessTable(processes, n);
    return 0;}
```

Bankers algo

```
#include <stdio.h>
#define MAX_PROCESSES 5
#define MAX_RESOURCES 3
void calculate_need(int processes, int resources, int max[][MAX_RESOURCES], int
allocation[][MAX_RESOURCES], int need[][MAX_RESOURCES]) {
    for (int i = 0; i < processes; i++) {
        for (int j = 0; j < resources; j++) {
            need[i][j] = max[i][j] - allocation[i][j];}}}
int is_safe(int processes, int resources, int available[], int
max[][MAX_RESOURCES], int allocation[][MAX_RESOURCES], int need[][MAX_RESOURCES])
{
    int work[MAX_RESOURCES], finish[MAX_PROCESSES];
    int i, j, k;
    // Initialize work and finish arrays
    for (i = 0; i < resources; i++) {
        work[i] = available[i];}
    for (i = 0; i < processes; i++) {
        finish[i] = 0;}
    // Safety algorithm logic
    int count = 0;
    while (count < processes) {
        int found = 0;
        for (i = 0; i < processes; i++) {
            if (finish[i] == 0) {
                for (j = 0; j < resources; j++) {
                    if (need[i][j] > work[j]) {
                        break;}}
                if (j == resources) {
                    // Process i can be allocated resources
                    for (j = 0; j < resources; j++) {
                        work[j] += allocation[i][j];}
                    finish[i] = 1;
                    found = 1;
                    count++;}}}
        if (found == 0) {
            return 0; }}// System is in unsafe state
    return 1; }// System is in safe state
int main() {
    int processes, resources;
    printf("Enter number of processes: ");
    scanf("%d", &processes);
    printf("Enter number of resources: ");
    scanf("%d", &resources);
    if (processes > MAX_PROCESSES || resources > MAX_RESOURCES) {
        printf("Exceeded maximum processes or resources limit.\n");
        return 1;}
    // Initialize matrices
    int available[MAX_RESOURCES], max[MAX_PROCESSES][MAX_RESOURCES],
allocation[MAX_PROCESSES][MAX_RESOURCES], need[MAX_PROCESSES][MAX_RESOURCES];
    // Input values for Available
    printf("Enter available resources: ");
    for (int i = 0; i < resources; i++) {
        scanf("%d", &available[i]);}
    // Input values for Max matrix
    printf("Enter max resources for each process: \n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < resources; j++) {
            scanf("%d", &max[i][j]);}}
    // Input values for Allocation matrix
    printf("Enter allocated resources for each process: \n");
    for (int i = 0; i < processes; i++) {
        printf("Process %d: ", i);
        for (int j = 0; j < resources; j++) {
            scanf("%d", &allocation[i][j]);}}
    // Calculate Need matrix
    calculate_need(processes, resources, max, allocation, need);
    // Check if system is in safe state
    if (is_safe(processes, resources, available, max, allocation, need)) {
        printf("System is in safe state\n");
    } else {
        printf("System is in unsafe state\n");}
    return 0;}
```

Bank appli algo

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdbool.h>
int *balance;
int turn;
bool flag[2];
int i=0,j=1;
int main(){
    int shmid = shmget(12345,sizeof(int), 0666 | IPC_CREAT);
    if(shmid == -1){
        perror("shmget failed");
        exit(1);}
    balance = (int *)shmat(shmid,NULL,0);
    if(balance == (int *)-1){
        perror("shmat failed");
        exit(1);}
     *balance = 1000; //assume initial balance
     int ch,amt;
     while(true){
     printf("enter 1 for deposit and 2 for withdrawing 3 for exit : ");
     scanf("%d",&ch);
     printf("\n");
     switch(ch){
        case 1 : printf("Enter the amount to be deposited : ");
                scanf("%d",&amt);
                printf("\n");
                 flag[i] = true;
                 turn = j;
                 while(flag[j] && turn ==j);
                *balance += amt;
                printf("deposited %d , new balance : %d\n",amt,*balance);
                flag[i] = false;
                break;
        case 2 :  printf("Enter the amount to be withdrawn : ");
                scanf("%d",&amt);
                printf("\n");
                if(amt>*balance){
                    printf("insufficient balance, try again!\n");
                    break;}
                //Entry section
                flag[j] = true;
                turn = i;
                while(flag[i] && turn ==i);
                //Enter critical section
                *balance -= amt;
                printf("withdraw of amt %d is successful , new 
balance : %d\n",amt,*balance);
                // end of Critical Section  | Remainder section
                flag[j] = false;
                break;
        case 3 :   shmdt(balance);
                    exit(0);
        default:
            printf("Invlaid Choice");}}
    return 0;}
```

MEMORY ALLOCATION

```
#include <stdio.h>
#include <stdlib.h>
typedef struct free_block {
    int start_addr;
    int size;
    struct free_block *next;
} free_block;
typedef struct allocated_block {
    int start_addr;
    int size;
    struct allocated_block *next;
} allocated_block;
free_block *free_head = NULL;
allocated_block *allocated_head = NULL;
void insert_free_block(int start_addr, int size) {
    free_block *new_block = (free_block *)malloc(sizeof(free_block));
    new_block->start_addr = start_addr;
    new_block->size = size;
    new_block->next = free_head;
    free_head = new_block;
```

```
}
void insert_allocated_block(int start_addr, int size) {
    allocated_block *new_block = (allocated_block
*)malloc(sizeof(allocated_block));
    new_block->start_addr = start_addr;
    new_block->size = size;
    new_block->next = allocated_head;
    allocated_head = new_block;}

void best_fit(int size) {
    free_block *curr = free_head;
    free_block *best = NULL;
    free_block *best_prev = NULL;
    free_block *prev = NULL;
    int best_size = -1;
    while (curr) {
        if (curr->size >= size && (best_size == -1 || curr->size < best_size)) {
            best = curr;
            best_size = curr->size;
            best_prev = prev;}
        prev = curr;
        curr = curr->next;}
    if (best) {
        insert_allocated_block(best->start_addr, size);
        if (best_size == size) {
            if (best_prev) {
                best_prev->next = best->next;
            } else {
                free_head = best->next;}
            free(best);
        } else {
            best->size -= size;
            best->start_addr += size;}
void first_fit(int size) {
    free_block *curr = free_head;
    free_block *prev = NULL;
    while (curr) {
        if (curr->size >= size) {
            insert_allocated_block(curr->start_addr, size);
            if (curr->size == size) {
                if (prev) {
                    prev->next = curr->next;
                } else {
                    free_head = curr->next;}
```

```c
        free(curr);
      } else {
        curr->size -= size;
        curr->start_addr += size; }
      return;}
    prev = curr;
    curr = curr->next;}}
void worst_fit(int size) {
  free_block *curr = free_head;
  free_block *worst = NULL;
  free_block *worst_prev = NULL;
  free_block *prev = NULL;
  int worst_size = -1;
  while (curr) {
   if (curr->size >= size && (worst_size == -1 || curr->size > worst_size))
     { worst = curr;
      worst_size = curr->size;
      worst_prev = prev;}
    prev = curr;
    curr = curr->next;}
  if (worst) {
    insert_allocated_block(worst->start_addr, size);
    if (worst_size == size) {
      if (worst_prev) {
        worst_prev->next = worst->next;
      } else {
        free_head = worst->next;}
      free(worst);
    } else {
      worst->size -= size;
      worst->start_addr += size;}}}
void print_free_list() {
  free_block *curr = free_head;
  printf("Free List: \n");
  while (curr) {
    printf("[%d-%d] \n", curr->start_addr, curr->start_addr + curr->size -
1);
    curr = curr->next; }
  printf("NULL\n");}
void print_allocated_list() {
  allocated_block *curr = allocated_head;
  printf("Allocated List: \n");
  while (curr) {
        printf("[%d-%d] \n", curr->start_addr, curr->start_addr + curr->size - 
1);
        curr = curr->next;}
    printf("NULL\n");}
int main() {
    int ch, b, n, ba, s, i;
    printf("Enter number of blocks:\n");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        printf("Enter the base address and size: ");
        scanf("%d%d", &ba, &s);
        insert_free_block(ba, s);}
    printf("Initial Free List:\n");
    print_free_list();
    while (1) {
        printf("Enter choice (1: Best Fit, 2: First Fit, 3: Worst Fit, 4: Exit): 
");
        scanf("%d", &ch);
        switch (ch) {
            case 1:printf("Enter the size of process to be allocated: ");
                scanf("%d", &b);
                best_fit(b);
                printf("After Best Fit Allocation:\n");
                print_free_list();
                print_allocated_list();
                break;
            case 2: printf("Enter the size of process to be allocated: ");
                scanf("%d", &b);
                first_fit(b);
                printf("After First Fit Allocation:\n");
                print_free_list();
                print_allocated_list();
                break;
            case 3:   printf("Enter the size of process to be allocated: ");
                scanf("%d", &b);
                worst_fit(b);
                printf("After Worst Fit Allocation:\n");
                print_free_list();
                print_allocated_list();
                break;
            case 4:
                return 0;
            default:
                printf("Invalid choice. Please enter 1, 2, 3, or 4.\n");
                break;}}
    return 0;}
```

PRODUCER CONSUMER

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#define BUFFSIZE 6;
int buffer[BUFFSIZE];
int in = 0, out = 0;
sem_t empty, full, mutex;
void *producer(){
   int item;
   while (1){
      item = rand() % 100;
      sem_wait(&empty);
      sem_wait(&mutex);
      buffer[in] = item;
      printf("Producer : placed item %d into the buffer\n", item);
      in = (in + 1) % BUFFSIZE;
      sem_post(&mutex);
      sem_post(&full);
      sleep(1);}}
void *consumer(){
   int item;
   while (1){
      // decreament full and mutex
      sem_wait(&full);
      sem_wait(&mutex);
      // retrieve item from the buffer using out index
      item = buffer[out];
      printf("Consumer consumed item %d\n", item);
      out = (out + 1) % BUFFSIZE;
      // increament mutex and empty
      sem_post(&mutex);
      sem_post(&empty);
      sleep(1);}}
int main(){
   sem_init(&empty, 0, BUFFSIZE);
   sem_init(&full, 0, 0);
   sem_init(&mutex, 0, 1);
   pthread_t prod, cons;
   if (pthread_create(&prod, NULL, producer, NULL) != 0){
      perror("Thread creation failed");
      return 1;}
   if (pthread_create(&cons, NULL, consumer, NULL) != 0){
      perror("Thread creation failed");
      return 1;}
   if (pthread_join(prod, NULL) != 0){
      perror("Couldn't join threads");
      return 1;}
   if (pthread_join(cons, NULL) != 0){
      perror("Couldn't join threads");
      return 1;}
   sem_destroy(&mutex);
   sem_destroy(&full);
   sem_destroy(&empty);
   return 0;}
```

PART B

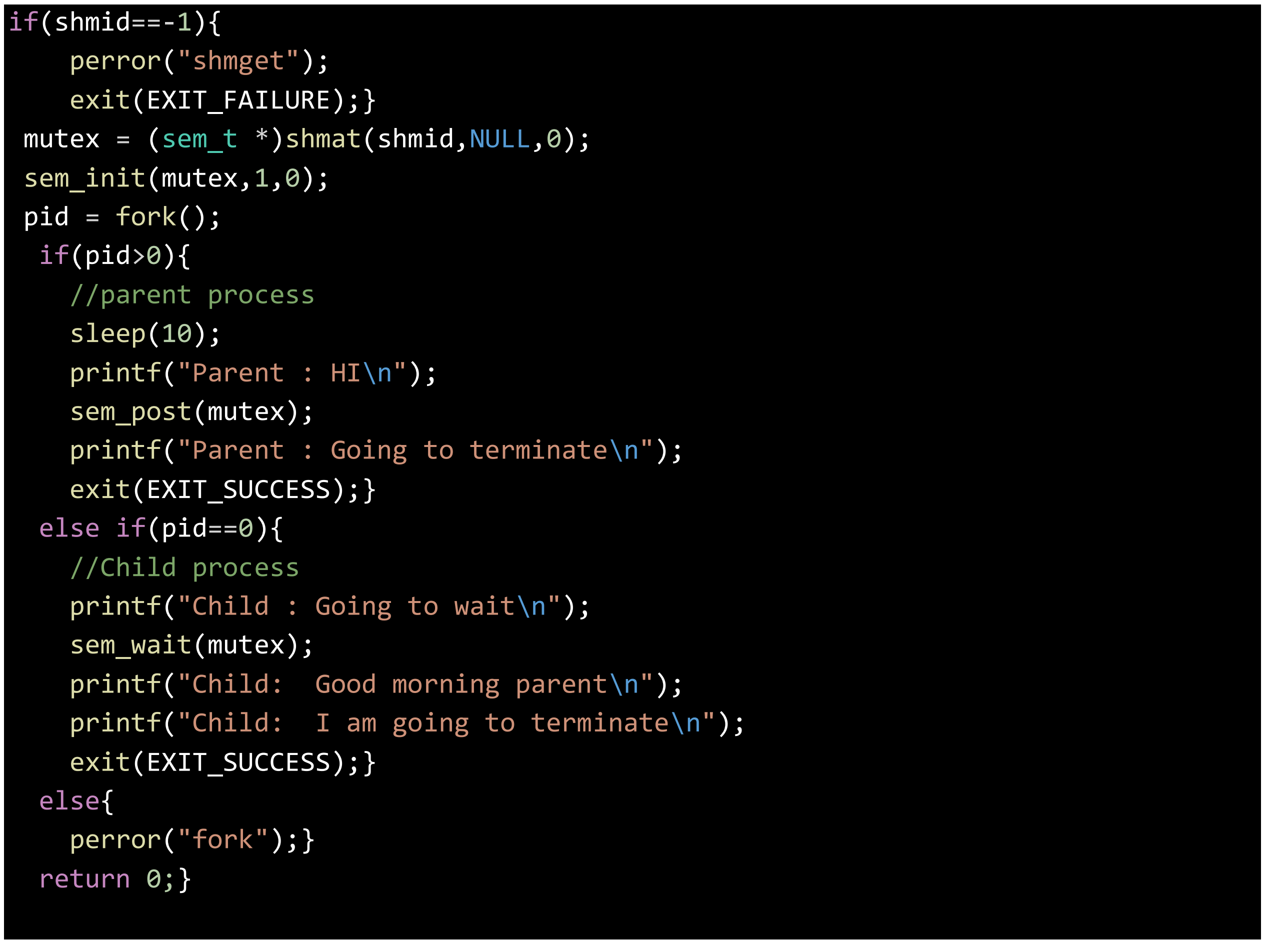
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#define buffSize 1024
int main(int argc, char *argv[]){
    if(argc < 2){
        printf("usage : %s <file1> <file2>..<filen>\n",argv[0]);
        exit(1);}
    int fd;
    int buffer[buffSize];
    for(int i = 1 ; i < argc ; i++){
        fd = open(argv[i],O_RDONLY);
        if(fd==-1){
            perror("Error opening file!");
            exit(1);    }
        int bytes;
        while((bytes = read(fd,buffer,buffSize))>0){
            write(1,buffer,bytes);}
        close(fd);}
    return 0;}

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

```c
#define buffSize 1024
int main(int argc, char *argv[]){
    if(argc < 2){
        printf("usage : %s <file1> <file2>..<filen>\n",argv[0]);
        exit(1);}
    int fd;
    int buffer[buffSize];
    char *destf = argv[argc-1];
    int wfd = open(destf,O_WRONLY,O_CREAT,O_TRUNC,0644);
    for(int i = 1 ; i < argc ; i++){
        fd = open(argv[i],O_RDONLY);
        if(fd==-1){
            perror("Error opening file!");
            continue; }
        int bytes;
        while((bytes = read(fd,buffer,buffSize))>0){
            write(wfd,buffer,bytes);}
        close(fd); close(wfd);} return 0;}
```

PIPE USING

```c
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#define BUFFSIZE 7
void sortArr(int arr[],int n){
    for(int i =0 ; i<n-1;i++){
        for(int j =0 ;j<n-1;j++){
            if(arr[j]>arr[j+1]){
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;}}}}
int main(){
    int pipeFd[2];
    pid_t pid;
    int buffer[BUFFSIZE] = {1,6,9,5,2,3,4};
    int n = sizeof(buffer)/sizeof(buffer[0]);
    if(pipe(pipeFd)==-1){
        perror("pipe");
        exit(EXIT_FAILURE);   }
    pid = fork();
    if(pid<0){
        perror("fork");
        exit(EXIT_FAILURE);}
    else if(pid > 0){
        // parent process
        close(pipeFd[0]);
        write(pipeFd[1],buffer,sizeof(buffer));
        printf("Parent : Sent the numbers to child\n");
        close(pipeFd[1]);
        wait(NULL);}
    else if(pid ==0){
        //child process
        int recievedArr[BUFFSIZE] = {};
        close(pipeFd[1]);
        read(pipeFd[0],recievedArr,sizeof(recievedArr));
        sortArr(recievedArr,n);
        printf("Child : sorted numbers are : \n");
        for(int i =0 ; i<n ; i++){
            printf("%d\t",recievedArr[i]);}
        printf("\n");
        close(pipeFd[0]);}
    return 0;}
SIGNAL
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
void sigfn1(int signo){
    printf(" parent work is over, child -you can proceed\n");}
void sigfn2(int signo){
    printf(" Child  work is over, parent -you can proceed\n");}
int main(){
    signal(SIGUSR1, sigfn1);
    signal(SIGUSR2, sigfn2);
    int temp;
    key_t key = ftok("consumer1.c", 165);
    int shmid = shmget(key, 10 * sizeof(int), IPC_CREAT | 0666);
    if (shmid > 0)
        printf("it is okshm id=%d\n", shmid);
    int *ptr = (int *)(shmat(shmid, 0, 0));
    int pid = fork();
    if (pid == 0){
        printf(" child:child is going to wait for data \n");
        pause();
        printf(" Child: got signal from parent\n");
        for (int i = 0; i < 10; i++)
            printf("       child:%d\n", *(ptr + i));
        printf("child: I am going to sort \n");
        for (int i = 0; i < 9; i++)
            for (int j = i + 1; j < 10; ++j)
                if (*(ptr + i) > *(ptr + j)){
                    temp = *(ptr + i);
                    *(ptr + i) = *(ptr + j);
                    *(ptr + j) = temp;}
        printf("      Child : sorted data\n");
        for (int i = 0; i < 10; i++)
            printf(" child:%d\n", *(ptr + i));
        printf("       Child : I am signalling parent\n");
        kill(getppid(), SIGUSR2);
        printf("child: I am going to terminate\n"); } // end of child process
    else // parent process{
        printf("parent is writing data\n");
        printf("Parent: enter 10 data : \n");
        for (int i = 0; i < 10; i++)
            scanf("%d", ptr + i);
        printf(" Parent:Data written in memory: \n");
        for (int i = 0; i < 10; i++)
            printf("Parent: %d\n", *(ptr + i));
        kill(pid,SIGUSR1);
        wait(NULL);
        shmdt(ptr);}
    return 0;}
POINT TO POINT
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<semaphore.h>
#include<sys/ipc.h>
#include<sys/shm.h>
#include<unistd.h>
#include<pthread.h>
#include<fcntl.h>
#define KEY 69;
int main(){
sem_t *mutex;
int shmid,pid;
key_t key = KEY;
shmid = shmget(key,sizeof(int), 0666 | IPC_CREAT);
```



```c
if(shmid==-1){
    perror("shmget");
    exit(EXIT_FAILURE);}
 mutex = (sem_t *)shmat(shmid,NULL,0);
 sem_init(mutex,1,0);
 pid = fork();
  if(pid>0){
    //parent process
    sleep(10);
    printf("Parent : HI\n");
    sem_post(mutex);
    printf("Parent : Going to terminate\n");
    exit(EXIT_SUCCESS);}
  else if(pid==0){
    //Child process
    printf("Child : Going to wait\n");
    sem_wait(mutex);
    printf("Child:  Good morning parent\n");
    printf("Child:  I am going to terminate\n");
    exit(EXIT_SUCCESS);}
  else{
    perror("fork");}
  return 0;}
```