1B(addressbook)

```
#!/bin/bash

create()
{
if [ -e addressbook.txt ]; #-e is checking exitence of addressbook.txt
then
        echo -e "ADDRESSBOOK ALREADY EXISTS!! \n"
else
        touch addressbook.txt #touch creates a new adressbook if not already exiting .
        echo -e "NEW ADDRESSBOOK CREATED SUCCESSFULLY!!!!! \n"
fi
}


insert()
{

echo -e "ENTER THE FIRST NAME:- \n" # -e is used to  make \n available for use
read fname
echo -e "ENTER THE LAST NAME:- \n"
read lname
echo -e "ENTER THE EMAIL-ID:- \n"
read email
echo -e "ENTER THE MOBILE NUMBER:- \n"
read mno
echo -e "ENTER THE ADDRESS:- \n"
read addr

echo "$fname  $lname$email $mno   $addr" >> addressbook.txt
echo -e "RECORD INSERTION SUCCESSFUL !! \n"
}


view()
{
if [ ! -s addressbook.txt ];
then
        echo -e "THE ADDRESSBOOK IS EMPTY !!! \n"
else
        sed -n 'p' addressbook.txt

fi
}


delete()
{
if [ ! -s addressbook.txt ];
then
        echo -e "ADDRESSBOOK EMPTY !!! \n"
else
```

```
	echo -e "ENTER THE FIRST NAME OR EMAIL-ID FOR DELETING RECORD:- \n"
	read fnem
	grep -n "$fnem" addressbook.txt

	echo -e "ENTER THE LINE NUMBER TO DELETE:- \n"
	read lineno

	for line in `grep -n "$fnem" addressbook.txt`
	do

	number=`echo $line|cut -c1`

		if [ "$number" == "$lineno" ];
		then
			sed -i -e "${lineno}d" addressbook.txt #ratta maro
			echo -e "RECORD DELETION SUCCESSFUL!!!! \n"
		fi

	done
fi

}

modify()
{
if [ ! -s addressbook.txt ];
then
	echo -e "ADDRESSBOOK EMPTY !!! \n"
else
	echo -e "ENTER THE NAME(FIRST NAME/LAST NAME) OF THE PERSON TO BE
EDITED:- \N"
	read name
	grep -n "$name" addressbook.txt

			echo -e "ENTER THE LINE NUMBER OF THE RECORD TO BE
EDITED:- \n"
			read lineno

			for line in `grep -n "$name" addressbook.txt`
			do
				number=`echo "$line"|cut -c1`
				if [ "$number" == "$lineno" ]
					then
					echo -e "ENTER THE MODIFICATION DETAILS:- \n"
					insert
					sed -i -e "${lineno}s/.*/$insert/" addressbook.txt # ratta maro
				echo -e "MODIFICATION SUCCESSFUL !!! \n"
				fi

			done
		fi
}
```

```
ch=1;
while [ $ch -lt 6 ]; # less than
do
echo -e "1) CREATE ADDRESSBOOK \n"
echo -e "2) VIEW ADDRESSBOOK \n"
echo -e "3) INSERT RECORD IN ADDRESSBOOK \n"
echo -e "4) DELETE RECORD FROM ADDRESSBOOK \n"
echo -e "5) MODIFY ADRESSBOOK \n"
echo -e "6) EXIT"

read -p "ENTER YOUR CHOICE:-" ch

case $ch in
1)create;;
2)view;;
3)insert;;
4)delete;;
5)modify;;
6)echo "ADDRESSBOOK EXITED!!"
esac

done
```

2A(orphan)

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

// fucntion to pass to qsort for sorting in ascending order
int asc(const void *i, const void *j) {
   return (*(int *) i - *(int *) j);
}

// fucntion to pass to qsort for sorting in descending order
int desc(const void *i, const void *j) {
   return (*(int *) j - *(int *) i);
}

// fork function
void forktest(int arr[], int n) {
   int cpid = fork();

   if (cpid < 0) {
      printf("\nFork unsuccessful.");
   } else if (cpid == 0) {
      printf("\n\nIn Child process.");
```

```
        printf("\n\nChild process ID is : %d", getpid());
        printf("\nParent process ID is : %d", getppid());
        printf("\n\n");

        sleep(5);
        // At this time parent process has finished, so it will show different process id.
        printf("\n\nIn Child process in Orphan State.");

        //quick sort
        qsort(arr, n, sizeof(int), desc);

        printf("\n\nSorted elements in array in descending order using quick sort : ");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }

        printf("\n\nChild process ID is : %d", getpid());
        printf("\nParent process ID is : %d\n", getppid());
    } else {
        printf("\n\nIn Parent process.");

        //quick sort
        qsort(arr, n, sizeof(int), asc);

        printf("\n\nSorted elements in array in ascending order using quick sort : ");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n\nChild process ID is : %d", getpid());
        printf("\nParent process ID is : %d\n", getppid());

    }
}

int main() {
    int n = 0;
    printf("\n\t***Demonstration of Orphan State***\t\n");
    printf("\nEnter size of array : ");
    scanf("%d", &n);

    int arr[n];
    printf("\nEnter numbers in the array : \n");
    for (int i = 0; i < n; i++) {
        scanf(" %d", &arr[i]);
    }

    forktest(arr, n);
    return 0;
}
```

2A(zombie)

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

// fucntion to pass to qsort for sorting in ascending order
int asc(const void *i, const void *j) {
   return (*(int *) i - *(int *) j);
}

// fucntion to pass to qsort for sorting in descending order
int desc(const void *i, const void *j) {
   return (*(int *) j - *(int *) i);
}

// fork function
void forktest(int arr[], int n) {
   int cpid = fork();

   if (cpid < 0) {
      printf("\nFork unsuccessful.");
   } else if (cpid == 0) {
      printf("\n\nIn Child process.");

      //quick sort
      qsort(arr, n, sizeof(int), desc);

      printf("\n\nSorted elements in array in descending order using quick sort : ");
      for (int i = 0; i < n; i++) {
         printf("%d ", arr[i]);
      }

      printf("\n\nChild process ID is : %d", getpid());
      printf("\nParent process ID is : %d", getppid());

      sleep(2);
      printf("\n\nChild process is exiting.\n\n");
   } else {
      printf("\n\nIn Parent process.");

      //quick sort
      qsort(arr, n, sizeof(int), asc);

      printf("\n\nSorted elements in array in ascending order using quick sort : ");
      for (int i = 0; i < n; i++) {
         printf("%d ", arr[i]);
      }
```

```
        printf("\n\n1.Child process ID is : %d", getpid());
        printf("\n1.Parent process ID is : %d\n", getppid());


        sleep(10);
        wait(NULL);
    }
}

int main() {
    int n = 0;
    printf("\n    ***Demonstration of Zombie State***\t\n");
    printf("\nEnter size of array : ");
    scanf("%d", &n);

    int arr[n];
    printf("\nEnter numbers in the array : \n");
    for (int i = 0; i < n; i++) {
        scanf(" %d", &arr[i]);
    }

    forktest(arr, n);
    return 0;
}
```

2B(parent) parent_sort.c

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

void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```
int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);

    int arr[n];
    printf("Enter the elements: ");
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    pid_t pid = fork();

    if (pid < 0) {
        perror("Fork failed");
        exit(1);
    }
    else if (pid == 0) {
        // ---- CHILD PROCESS ----
        bubble_sort(arr, n);

        // Build full path to child_reverse
        char path[PATH_MAX];
        getcwd(path, sizeof(path));
        strcat(path, "/child_reverse");

        // Prepare execve arguments
        char *args[n + 2];
        args[0] = path;
        for (int i = 0; i < n; i++) {
            char *arg = malloc(12);
            snprintf(arg, 12, "%d", arr[i]);
            args[i + 1] = arg;
        }
        args[n + 1] = NULL;

        execve(args[0], args, NULL);

        perror("execve failed");
        for (int i = 1; i <= n; i++) {
            free(args[i]);
        }
        exit(1);
    }
    else {
        // ---- PARENT ----
        wait(NULL);
        printf("Parent process finished.\n");
    }

    return 0;
```

```
}
```

2B(child) child_reverse

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

int main(int argc, char *argv[]) {
    if (argc < 2) {
        fprintf(stderr, "No array elements provided\n");
        return 1;
    }

    printf("\nArray in reverse order: ");
    for (int i = argc - 1; i > 0; i--) {
        printf("%s ", argv[i]);
    }
    printf("\n");

    return 0;
}
```

3(round robin)

```
#include<stdio.h>

struct process {
    int id, AT, BT, WT, TAT;
};

struct process a[10];

// declaration of the ready queue
int queue[100];
int front = -1;
int rear = -1;

// function for insert the element
// into queue
void insert(int n) {
    if (front == -1)
        front = 0;
```

```
    rear = rear + 1;
    queue[rear] = n;
}

// function for delete the
// element from queue
int delete() {
    int n;
    n = queue[front];
    front = front + 1;
    return n;
}

int main() {
    int n, TQ, p, TIME = 0;
    int temp[10], exist[10] = {0};
    float total_wt = 0, total_tat = 0, Avg_WT, Avg_TAT;
    printf("Enter the number of the process\n");
    scanf("%d", &n);
    printf("Enter the arrival time and burst time of the process\n");
    printf("AT BT\n");
    for (int i = 0; i < n; i++) {
        scanf("%d%d", &a[i].AT, &a[i].BT);
        a[i].id = i;
        temp[i] = a[i].BT;
    }
    printf("Enter the time quantum\n");
    scanf("%d", &TQ);
    // logic for round robin scheduling

    // insert first process
    // into ready queue
    insert(0);
    exist[0] = 1;
    // until ready queue is empty
    while (front <= rear) {
        p = delete();
        if (a[p].BT >= TQ) {
            a[p].BT = a[p].BT - TQ;
            TIME = TIME + TQ;
        } else {
            TIME = TIME + a[p].BT;
            a[p].BT = 0;
        }


        //if process is not exist
        // in the ready queue even a single
        // time then insert it if it arrive
        // at time 'TIME'
        for (int i = 0; i < n; i++) {
            if (exist[i] == 0 && a[i].AT <= TIME) {
```

```
            insert(i);
            exist[i] = 1;
         }
      }
      // if process is completed
      if (a[p].BT == 0) {
         a[p].TAT = TIME - a[p].AT;
         a[p].WT = a[p].TAT - temp[p];
         total_tat = total_tat + a[p].TAT;
         total_wt = total_wt + a[p].WT;
      } else {
         insert(p);
      }
   }

   Avg_TAT = total_tat / n;
   Avg_WT = total_wt / n;

   // printing of the answer
   printf("ID WT TAT\n");
   for (int i = 0; i < n; i++) {
      printf("%d  %d  %d\n", a[i].id, a[i].WT, a[i].TAT);
   }
   printf("Average waiting time of the processes is : %f\n", Avg_WT);
   printf("Average turn around time of the processes is : %f\n", Avg_TAT);
   return 0;
}
```

3(SJF)

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

int n;
struct process {
   int bt, at, wt, tat, pid;
   int count;
};
typedef struct process process;
process p[10];

void printt() {

   printf("Process\tAT\tBT\tCounter\tWT\tTAT\n");
   for (int i = 0; i < n; i++) {
      printf("p%d\t%d\t%d\t%d\t%d\t%d\n", p[i].pid, p[i].at, p[i].bt, p[i].count, p[i].wt, p[i].tat);
   }
}

void sort() {
```

```
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].at > p[j + 1].at) {
                process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

int minimum(int tot) {
    int min = INT_MAX;
    int index = -1;
    int i = 0;
    while ((p[i].at <= tot) && i < n) {
        if (p[i].count == 0) {
            i++;
            continue;
        }

        if (p[i].count < min) {
            min = p[i].count;
            index = i;
        }

        i++;
    }
    return index;
}

void sjf() {
    int tot = 0, fin = 0;

    process gantt[100];
    while (fin < n) {
        int min = minimum(tot);
        gantt[tot].count = p[min].pid;
        tot++;

        if (--p[min].count == 0) {
            p[min].tat = tot - p[min].at;
            p[min].wt = p[min].tat - p[min].bt;
            fin++;
        }
    }
    printf("\nTotal time spent :- %d\n", tot);
    printt();
    int avgtat = 0, avgwt = 0;
    for (int i = 0; i < n; i++) {
        avgwt += p[i].wt;
```

```
        avgtat += p[i].tat;
    }

    printf("Average wt = %f \nAverage tat = %f\n", (float) avgwt / n, (float) avgtat / n);

    //print gantt chart
    printf("Gantt Chart :- \n");
    int num = 0;
    int prev = 0;
    for (int i = 0; i < tot; i++) {

        if (gantt[i].count != prev) {
            printf(" | P%d", gantt[i].count);
        }
        prev = gantt[i].count;

    }
    printf(" |");

    printf("\n");
    // printf(" 0   ");
    for (int i = 0; i < tot; i++) {


        if (gantt[i].count != prev) {
            printf(" %d   ", num);


        }
        prev = gantt[i].count;
        num = num + 1;
    }
    printf(" %d   ", tot);
    printf("\n");

}

int main() {
    printf("\nEnter total processes(<10)\n");
    scanf("%d", &n);

    //Input and initialize
    printf("\nEnter space separated values for at and bt sequentially\n");
    for (int i = 0; i < n; i++) {
        scanf("%d", &p[i].at);
        scanf("%d", &p[i].bt);
        p[i].wt = p[i].tat = -1;
        p[i].count = p[i].bt;
        p[i].pid = i + 1;
    }
    sort();
    sjf();
```

```
}
```

4a(producer consumer)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define SIZE 3

void *producer(void *);
void *consumer(void *);

struct Shared
{
   int buff[SIZE];
   sem_t full, empty;
};

struct Shared Sh;
pthread_mutex_t mut = PTHREAD_MUTEX_INITIALIZER;
int front = 0, rear = 0;

int main()
{
   int prod, cons;
   printf("\nEnter number of Producers: ");
   scanf("%d", &prod);
   printf("Enter number of Consumers: ");
   scanf("%d", &cons);

   pthread_t p[prod], c[cons];

   sem_init(&Sh.empty, 0, SIZE); // buffer initially empty
   sem_init(&Sh.full, 0, 0);     // nothing produced yet

   for (int i = 0; i < prod; i++)
      pthread_create(&p[i], NULL, producer, NULL);

   for (int i = 0; i < cons; i++)
      pthread_create(&c[i], NULL, consumer, NULL);

   for (int i = 0; i < prod; i++)
      pthread_join(p[i], NULL);
```

```
    for (int i = 0; i < cons; i++)
        pthread_join(c[i], NULL);

    return 0;
}

void *producer(void *arg)
{
    int item;
    while (1)
    {
        printf("\nEnter item to produce: ");
        scanf("%d", &item);

        sem_wait(&Sh.empty);
        pthread_mutex_lock(&mut);

        Sh.buff[rear] = item;
        printf("\nProducer %ld produced: %d", pthread_self(), item);
        rear = (rear + 1) % SIZE;

        pthread_mutex_unlock(&mut);
        sem_post(&Sh.full);
    }
}

void *consumer(void *arg)
{
    int item;
    while (1)
    {
        sem_wait(&Sh.full);
        pthread_mutex_lock(&mut);

        item = Sh.buff[front];
        printf("\nConsumer %ld consumed: %d", pthread_self(), item);
        front = (front + 1) % SIZE;

        pthread_mutex_unlock(&mut);
        sem_post(&Sh.empty);
    }
}
```

4b(reader writer)

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
```

```
#define TOTAL_READERS 10
#define TOTAL_WRITERS 5

sem_t wrt;  // binary semaphore used for both for mutual exclusion and signalling
pthread_mutex_t mutex;  // Provides mutual exclusion when read_count is being modified
int reader_count = 0; // To keep count of the total readers
int count = 1;

void *writer(void *wno) {
   sem_wait(&wrt);
   count = count * 2;
   printf("Writer %d modified count to %d\n", *(int *) wno, count);
   sem_post(&wrt);
}

void *reader(void *rno) {
   // reader acquires lock before modifying read_count
   pthread_mutex_lock(&mutex);
   reader_count++;
   if (reader_count == 1) {
      sem_wait(&wrt); // if this is the first reader then it will block the writer
   }
   pthread_mutex_unlock(&mutex);
   // reading section
   printf("Reader %d modified count to %d\n", *(int *) rno, count);
   // reader acquires the lock before modifying reader_count
   pthread_mutex_lock(&mutex);
   reader_count--;
   if (reader_count == 0) {
      sem_post(&wrt);     // If this is the last reader, it will wake up the writer.
   }
   pthread_mutex_unlock(&mutex);
}

int main() {
   pthread_t read[TOTAL_READERS], write[TOTAL_WRITERS]; // creating reader and writer
threads
   pthread_mutex_init(&mutex, NULL);
   sem_init(&wrt, 0, 1);

   int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // used for numbering the producer and consumer

   // main() function can exit before producer and consumer threads are executed
   // this can be prevented by using join operation
   // pthread_join() function provides a simple mechanism allowing an application to wait for a
thread to terminate.
   for (int i = 0; i < 10; i++) {
      pthread_create(&read[i], NULL, (void *) reader, (void *) &a[i]);
   }
   for (int i = 0; i < 5; i++) {
      pthread_create(&write[i], NULL, (void *) writer, (void *) &a[i]);
   }
```

```
    for (int i = 0; i < 10; i++) {
        pthread_join(read[i], NULL);
    }
    for (int i = 0; i < 5; i++) {
        pthread_join(write[i], NULL);
    }

    pthread_mutex_destroy(&mutex);  // destroy the mutex to avoid memory leak
    sem_destroy(&wrt);

    return 0;
}
```

5(banker)

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

bool issafe(int p, int r, int pc, int c, int allocated[p][r],
            int available[r], int need[p][r], int safeseq[p], int com[p]) {
    while (pc != p) {
        for (int i = 0; i < p; i++) {
            c += 1;
            if (com[i] != 1) {
                int ar = 0;
                for (int j = 0; j < r; j++) {
                    if (available[j] < need[i][j]) {
                        break;
                    } else {
                        ar += 1;
                    }
                }
                if (ar == r) {
                    safeseq[pc] = i + 1;
                    com[i] = 1;
                    pc += 1;
                    for (int j = 0; j < r; j++) {
                        available[j] += allocated[i][j];
                    }
                    c = 0;
                }
            }
        }
        if (c == p) {
            break;
        }
    }
```

```
    if (pc != p && c == p) {
        return false;
    } else {
        return true;
    }
}

int main() {
    int p, r;

    printf("\n     Banker's Algorithm \n");

    printf("\nEnter the number of processes : ");
    scanf("%d", &p);

    printf("\nEnter the number of resources : ");
    scanf("%d", &r);

    int allocated[p][r];
    int max[p][r];
    int need[p][r];
    int available[r];
    int com[p];
    int safeseq[p];
    int pc = 0;
    int c = 0;

    for (int i = 0; i < p; i++) {
        com[i] = 0;
    }

    printf("\n\nEnter the allocated matrix for each process : \n");
    for (int i = 0; i < p; i++) {
        printf("Enter the allocated matrix for process P%d : ", (i + 1));
        for (int j = 0; j < r; j++) {
            scanf("%d", &allocated[i][j]);
        }
    }

    printf("\n\nEnter the max matrix for each process : \n");
    for (int i = 0; i < p; i++) {
        printf("Enter the max matrix for process P%d : ", (i + 1));
        for (int j = 0; j < r; j++) {
            scanf("%d", &max[i][j]);
            need[i][j] = max[i][j] - allocated[i][j];
        }
    }

    printf("\n\nEnter the availble resources : ");
    for (int i = 0; i < r; i++) {
        scanf("%d", &available[i]);
    }
```

```
printf("\n\nProcess\t Allocation\tMax\t   Need \tAvailable\n");
for (int i = 0; i < p; i++) {
   printf(" P%d\t ", (i + 1));

   for (int j = 0; j < r; j++) {
      printf("%d  ", allocated[i][j]);
   }
   printf("     ");
   for (int j = 0; j < r; j++) {
      printf("%d  ", max[i][j]);
   }
   printf("    ");
   for (int j = 0; j < r; j++) {
      printf("%d  ", need[i][j]);
   }
   printf("\t");

   if (i == 0) {
      for (int j = 0; j < r; j++) {
         printf("%d  ", available[j]);
      }
   }
   printf("\n");
}

if (issafe(p, r, pc, c, allocated, available, need, safeseq, com)) {
   printf("\n\nThe system is safe. The safe sequence is :");
   for (int i = 0; i < p; i++) {
      printf(" P%d", safeseq[i]);
      if (i != p - 1) {
         printf(" > ");
      }
   }
   printf("\n");
} else {
   printf("\n\nThe system is not safe.\n\n");
   printf("\n\nProgram exited successfully.\n\n");
   exit(0);
}

int ch;
printf("\n\nDo you want to make an additional request for resources ? (1=Yes|0=No)");
scanf("%d", &ch);

if (ch == 0) {
   printf("\n\nProgram exited successfully.\n\n");
   exit(0);
}

int req[r];
int pno;
```

```
    printf("\nEnter the number of the process that needs more resources : P");
    scanf("%d", &pno);

    printf("\n\nEnter the addititonal request of resources for process P%d : ", pno);
    for (int i = 0; i < r; i++) {
        scanf("%d", &req[i]);
    }

    for (int i = 0; i < r; i++) {
        if (req[i] > need[pno][i]) {
            printf("\n\nExceeding the max allocation for process P%d !", pno);
            printf("\nNot a good request.");
            printf("\n\nProgram exited successfully.\n\n");
            exit(0);
        }
    }

    for (int i = 0; i < r; i++) {
        if (req[i] > available[i]) {
            printf("\n\nExceeding the available resources !");
            printf("\nNot a good request.");
            printf("\n\nProgram exited successfully.\n\n");
            exit(0);
        }
    }

    for (int i = 0; i < r; i++) {
        available[i] -= req[i];
        allocated[pno][i] += req[i];
        need[pno][i] -= req[i];
    }

    if (issafe(p, r, pc, c, allocated, available, need, safeseq, com)) {
        printf("\n\nThe requested resources can be allocated to the process.\nThe system is safe. The
safe sequence is :");
        for (int i = 0; i < p; i++) {
            printf(" P%d", safeseq[i]);
            if (i != p - 1) {
                printf(" > ");
            }
        }
    } else {
        printf("\n\nThe requested resources cannot be allocated to the process.\nThe system is not
safe.");
    }

    printf("\n\nProgram exited successfully.\n\n");
    return 0;
}
```

6(FCFS,LRU,OPTIMAL)

```
#include<stdio.h>

int n, nf;
int in[100];
int p[50];
int hit = 0;
int i, j, k;
int pgfaultcnt = 0;

void getData() {
   printf("\nEnter length of page reference sequence:");
   scanf("%d", &n);
   printf("\nEnter the page reference sequence:");
   for (i = 0; i < n; i++)
      scanf("%d", &in[i]);
   printf("\nEnter no of frames:");
   scanf("%d", &nf);
}

void initialize() {
   pgfaultcnt = 0;
   for (i = 0; i < nf; i++)
      p[i] = 9999;
}

int isHit(int data) {
   hit = 0;
   for (j = 0; j < nf; j++) {
      if (p[j] == data) {
         hit = 1;
         break;
      }
   }
   return hit;
}

int getHitIndex(int data) {
   int hitind;
   for (k = 0; k < nf; k++) {
      if (p[k] == data) {
         hitind = k;
         break;
      }
   }
   return hitind;
}

void dispPages() {
   for (k = 0; k < nf; k++) {
      if (p[k] != 9999)
```

```
            printf(" %d", p[k]);
        }
    }

    void dispPgFaultCnt() {
        printf("\nTotal no of page faults:%d", pgfaultcnt);
    }

    void fifo() {
        initialize();
        for (i = 0; i < n; i++) {
            printf("\nFor %d :", in[i]);
            if (isHit(in[i]) == 0) {
                for (k = 0; k < nf - 1; k++)
                    p[k] = p[k + 1];
                p[k] = in[i];
                pgfaultcnt++;
                dispPages();
            } else
                printf("No page fault");
        }
        dispPgFaultCnt();
    }

    void optimal() {
        initialize();
        int near[50];
        for (i = 0; i < n; i++) {
            printf("\nFor %d :", in[i]);
            if (isHit(in[i]) == 0) {
                for (j = 0; j < nf; j++) {
                    int pg = p[j];
                    int found = 0;
                    for (k = i; k < n; k++) {
                        if (pg == in[k]) {
                            near[j] = k;
                            found = 1;
                            break;
                        } else
                            found = 0;
                    }
                    if (!found)
                        near[j] = 9999;
                }
                int max = -9999;
                int repindex;
                for (j = 0; j < nf; j++) {
                    if (near[j] > max) {
                        max = near[j];
                        repindex = j;
                    }
                }
```

```
            p[repindex] = in[i];
            pgfaultcnt++;
            dispPages();
        } else
            printf("No page fault");
    }
    dispPgFaultCnt();
}

void lru() {
    initialize();
    int least[50];
    for (i = 0; i < n; i++) {
        printf("\nFor %d :", in[i]);
        if (isHit(in[i]) == 0) {
            for (j = 0; j < nf; j++) {
                int pg = p[j];
                int found = 0;
                for (k = i - 1; k >= 0; k--) {
                    if (pg == in[k]) {
                        least[j] = k;
                        found = 1;
                        break;
                    } else
                        found = 0;
                }
                if (!found)
                    least[j] = -9999;
            }
            int min = 9999;
            int repindex;
            for (j = 0; j < nf; j++) {
                if (least[j] < min) {
                    min = least[j];
                    repindex = j;
                }
            }
            p[repindex] = in[i];
            pgfaultcnt++;
            dispPages();
        } else
            printf("No page fault!");
    }
    dispPgFaultCnt();
}

int main() {
    int choice;
    do {
        printf("\n\nPage Replacement Algorithms"
              "\n1.Enterdata\n2.FIFO\n3.Optimal\n4.LRU\n0.Exit\nEnter your choice:");
        scanf("%d", &choice);
```

```
        switch (choice) {
            case 1:
                getData();
                break;
            case 2:
                fifo();
                break;
            case 3:
                optimal();
                break;
            case 4:
                lru();
                break;
            case 0:
                return 0;
                break;
            default:
                printf("Please Enter the valid option.");
                break;
        }
    } while (choice != 0);
}
```

7A(FIFO)

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

int main() {
    int fd, fd1;
    char *myfifo = "myfifo";
    char *myfifo1 = "myfifo1";
    char buf[1024];
          char ch[1024];
    int words = 0, characters = 0, lines = 0;
    FILE *fp;

    // Create FIFO for writing the result
    mkfifo(myfifo1, 0777);

    // Open the FIFO for reading from A_FIFO
    fd = open(myfifo, O_RDONLY);

    // Read data from FIFO
```

```
read(fd, buf, sizeof(buf));

// Print the received message
printf("\nFirst message received: \n\n%s\n\n", buf);

// Count words, characters, and lines
int i = 0;
int in_word = 0; // Flag to track whether we are inside a word

while (buf[i] != '\0') {
   characters++;  // Every character counts

   // If the character is space, tab, or newline, consider it a word boundary
   if (buf[i] == ' ' || buf[i] == '\t' || buf[i] == '\n') {
      if (in_word) {
         words++; // Count a word when we reach a word boundary
         in_word = 0; // Reset the word flag
      }

      if (buf[i] == '\n') {
         lines++; // Count a line when we encounter a newline
      }
   } else {
      in_word = 1; // We are inside a word
   }

   i++;
}

// If the last character was part of a word, count it as a word
if (in_word) {
   words++;
}

// If there are no newlines in the text, count at least one line
if (lines == 0 && characters > 0) {
   lines = 1;
}

// Display the results
printf("\nTotal Words: %d\n", words);
printf("Total Lines: %d\n", lines);
printf("Total Characters: %d\n", characters);

// Write the result to a file
fp = fopen("test.txt", "w+");
fprintf(fp, "\nTotal Words: %d\n", words);
fprintf(fp, "Total Lines: %d\n", lines);
fprintf(fp, "Total Characters: %d\n", characters);
fclose(fp);

// Read the file back and send it through FIFO1
```

```
    fp = fopen("test.txt", "r");
    int j = 0;
    while ((ch[j] = fgetc(fp)) != EOF) {
        j++;
    }
    fclose(fp);

    // Open FIFO1 to send the result
    fd1 = open(myfifo1, O_WRONLY);
    write(fd1, ch, strlen(ch) + 1); // Write the result content to FIFO1
    close(fd1);

    // Close the original FIFO
    close(fd);

    return 0;
}
```

7B(server)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

#define SHM_KEY  ftok("shmfile",65)

int main() {
    key_t key = SHM_KEY;
    int shmid;
    char *shared_memory;

    // Create shared memory segment
    shmid = shmget(key, 1024, 0666 | IPC_CREAT);
    shared_memory = (char*) shmat(shmid, NULL, 0);

    printf("Enter message to store in shared memory: ");
    fgets(shared_memory, 1024, stdin);

    printf("Message written to shared memory.\n");

    shmdt(shared_memory);
    return 0;
}

//how to run
//gcc server.c -o server
//gcc client.c -o client
//./server
```

```
//Hello from Shared Memory!
//./client
//Data read from shared memory:
//Hello from Shared Memory!
```

7B(client)

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_KEY  ftok("shmfile",65)

int main() {
   key_t key = SHM_KEY;
   int shmid;
   char *shared_memory;

   // Access the shared memory segment
   shmid = shmget(key, 1024, 0666);
   shared_memory = (char*) shmat(shmid, NULL, 0);

   printf("Data read from shared memory:\n%s\n", shared_memory);

   shmdt(shared_memory);

   // Destroy shared memory after reading
   shmctl(shmid, IPC_RMID, NULL);

   return 0;
}
```

8(disk scheduling)

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

void sort(int arr[], int n) {
   for(int i=0;i<n-1;i++)
      for(int j=i+1;j<n;j++)
         if(arr[i]>arr[j]) {
            int temp=arr[i];
            arr[i]=arr[j];
            arr[j]=temp;
         }
}
```

```
int main() {
    int req[50], n, head, i, j, total = 0;
    int left[50], right[50], l = 0, r = 0;

    printf("Enter number of disk requests: ");
    scanf("%d", &n);

    printf("Enter disk requests (track numbers): ");
    for(i = 0; i < n; i++)
        scanf("%d", &req[i]);

    printf("Enter initial head position: ");
    scanf("%d", &head);

    printf("\n--- SSTF (Shortest Seek Time First) ---\n");
    int visited[50]={0}, min, pos, diff;
    int head_sstf = head, total_sstf = 0;

    for(i=0;i<n;i++) {
        min = 9999;
        for(j=0;j<n;j++) {
            if(!visited[j]) {
                diff = abs(req[j] - head_sstf);
                if(diff < min) {
                    min = diff;
                    pos = j;
                }
            }
        }
        total_sstf += abs(req[pos] - head_sstf);
        head_sstf = req[pos];
        visited[pos] = 1;
    }
    printf("Total Head Movement (SSTF): %d\n", total_sstf);

    printf("\n--- SCAN (Head moving outward → increasing direction) ---\n");
    int max = 199; // Assuming disk size: 0 to 199
    for(i=0;i<n;i++) {
        if(req[i] < head) left[l++] = req[i];
        else right[r++] = req[i];
    }
    sort(left, l);
    sort(right, r);

    int head_scan = head, total_scan = 0;

    // Move towards higher tracks first (away from spindle)
    for(i = 0; i < r; i++) {
        total_scan += abs(right[i] - head_scan);
        head_scan = right[i];
    }
```

```
    total_scan += abs(max - head_scan);
    head_scan = max;

    for(i = l-1; i >= 0; i--) {
        total_scan += abs(left[i] - head_scan);
        head_scan = left[i];
    }
    printf("Total Head Movement (SCAN): %d\n", total_scan);

    printf("\n--- C-LOOK (Circular look, head moves upward only) ---\n");
    int head_clook = head, total_clook = 0;

    for(i = 0; i < r; i++) {
        total_clook += abs(right[i] - head_clook);
        head_clook = right[i];
    }

    if(l > 0) {
        total_clook += abs(head_clook - left[0]);
        head_clook = left[0];
    }

    for(i = 1; i < l; i++) {
        total_clook += abs(left[i] - head_clook);
        head_clook = left[i];
    }

    printf("Total Head Movement (C-LOOK): %d\n", total_clook);

    return 0;
}

//how to run
//gcc disk.c -o disk
//./disk
//Enter number of disk requests: 8
//Enter disk requests: 98 183 37 122 14 124 65 67
//Enter initial head position: 53
```