operating system solved slips (SEM V)

slip 1

Q1.Write the simulation program to implement demand paging and show the
page scheduling and total number of page faults according to the LFU page replacement
algorithm.Assume the memory of n frames.
Reference String : 3,4,5,4,3,4,7,2,4,5,6,7,2,4,6

```
#include<stdio.h>
#define MAX 20
int frames[MAX],ref[MAX],mem[MAX][MAX],faults,
sp,m,n,count[MAX];
void accept()
{
int i;
printf("Enter no.of frames:");
scanf("%d", &n);
printf("Enter no.of references:");
scanf("%d", &m);
printf("Enter reference string:\n");
for(i=0;i<m;i++)
{
printf("[%d]=",i);
scanf("%d",&ref[i]);
}
}
void disp()
{
int i,j;
for(i=0;i<m;i++)
printf("%3d",ref[i]);
printf("\n\n");
for(i=0;i<n;i++)
 {
  for(j=0;j<m;j++)
  {
   if(mem[i][j])
    printf("%3d",mem[i][j]);
   else
    printf("   ");
  }
  printf("\n");
 }
 printf("Total Page Faults: %d\n",faults);
}
int search(int pno)
{
 int i;
 for(i=0;i<n;i++)
 {
  if(frames[i]==pno)
   return i;
 }
 return -1;
}
int get_lfu(int sp)
{
 int i,min_i,min=9999;

 i=sp;
 do
 {
  if(count[i]<min)
  {
   min = count[i];
   min_i = i;
  }
  i=(i+1)%n;
 }while(i!=sp);

 return min_i;
}
void lfu()
{
 int i,j,k;

 for(i=0;i<m && sp<n;i++)
 {
  k=search(ref[i]);
  if(k==-1)
  {
   frames[sp]=ref[i];
   count[sp]++;
   faults++;
   sp++;

   for(j=0;j<n;j++)
    mem[j][i]=frames[j];
  }
  else
   count[k]++;

 }

 sp=0;
 for(;i<m;i++)
 {
  k = search(ref[i]);
  if(k==-1)
  {
   sp = get_lfu(sp);
   frames[sp] = ref[i];
   count[sp]=1;
   faults++;
   sp = (sp+1)%n;

   for(j=0;j<n;j++)
    mem[j][i] = frames[j];
  }
  else
   count[k]++;
 }
}


int main()
{
 accept();
 lfu();
 disp();

 return 0;
}
```

Q2.Write a C program to implement the shell which displays the
command prompt “myshell$”. It accepts the command, tokenize the command
line and execute it by creating the child process. Also implement
the additional command ‘typeline’ as
typeline +n filename :- To print first n lines in the file.
typeline -a filename :- To print all lines in the file.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <unistd.h>
int make_toks(char *s, char *tok[])
{
        int i = 0;
    char *p;
    p = strtok(s, " ");
while(p != NULL)
{
       tok[i++] = p;
    p = strtok(NULL, " ");
}
tok[i] = NULL;
return i;
}
void typeline(char *op, char *fn)
{
        int fh, i, j, n;
    char c;
fh = open(fn, O_RDONLY);
if(fh == -1)
{
printf("File %s not found.\n", fn);
return;
}
if(strcmp(op, "a") == 0)
{
    while(read(fh, &c, 1) > 0)
            printf("%c", c);
        close(fh);
    return;
}
n = atoi(op);
if(n > 0)
{
        i = 0;
        while(read(fh, &c, 1) > 0)
    {
            printf("%c", c);
  if(c == '\n')
```

```
                i++;
        if(i == n)
            break;
}
}
if(n < 0)
{
        i = 0;
        while(read(fh, &c, 1) > 0)
            {
            if(c == '\n') i++;
        }
        lseek(fh, 0, SEEK_SET);
j = 0;
while(read(fh, &c, 1) > 0)
{
    if(c == '\n')
            j++;
    if(j == i+n+1)
            break;
}
while(read(fh, &c, 1) > 0)
{
        printf("%c", c);
}
}
close(fh);
}
int main()
{
        char buff[80], *args[10];
        while(1)
    {
            printf ("\n");
        printf("\nmyshell$ ");
        fgets(buff, 80, stdin);
            buff[strlen(buff)-1] = '\0';
            int n = make_toks(buff, args);
        switch (n)
        {
                case 1: if(strcmp(args[0], "exit") == 0)
exit(1);
if (!fork())
execlp (args [0], args[0], NULL);
break;
            case 2:
if (!fork ())
execlp (args [0], args[0], args[1], NULL);
break;
                case 3: if (strcmp(args[0], "typeline") == 0)
                        typeline (args[1], args[2]);
else
{
        if (!fork ())
execlp (args [0], args[0], args[1], args[2], NULL);
}
break;
case 4:
if (!fork ())
execlp (args [0], args [0], args [1], args [2], args [3], NULL);
```

```
break;
}
}
return 0;
}
```

---

slip 2

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.
Reference String : 3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
   int x;
   for(x=0;x<nf;x++)
   {
      if(sv==frame[x][0])
      {
         return 0;
      }
   }
   return 1;
}
void displayMemory()
{
   int i;
   printf("\n\nFrame Contains |");
   for(i=0;i<nf;i++)
      printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
   int i,min=frame[0][1],ri=0;
   for(i=0;i<nf;i++)
   {
      if(frame[i][1]==-1)
      {
         return i;
      }
   }
   //LRU
   for(i=0;i<nf;i++)
   {
      if(min>frame[i][1])
      {
         min=frame[i][1];
         ri=i;
      }
   }
   return ri;
}
int main()
{
   int rs[]={3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
   int ts=0;
   int n=15,i,j,pf=0,srch,insert_index;
   printf("Enter how many frames");
   scanf("%d",&nf);
   for(i=0;i<nf;i++)
   {
      for(j=0;j<2;j++)
         frame[i][j]=-1;
   }
   displayMemory();
   for(i=0;i<n;i++)
   {
      srch=searchFrames(rs[i]);
      if(srch==1)
      {
         pf++;
         insert_index=findFreeFrame();
         frame[insert_index][0]=rs[i];
         frame[insert_index][1]=ts++;
      }

      displayMemory();
   }
   printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q2.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given following command by creating the child process. Additionally it should interpret the ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
void make_toks(char *s, char *tok[])
{
    int i=0;
    char *p;
    p = strtok(s," ");
    while(p!=NULL)
    {   tok[i++]=p;
       p=strtok(NULL," ");
    }

    tok[i]=NULL;
}

void list(char *dn, char op)
{
    DIR *dp;
    struct dirent *entry;
    int dc=0,fc=0;

    dp = opendir(dn);
    if(dp==NULL)
    {
        printf("Dir %s not found.\n",dn);
        return;
    }
      switch(op)
    {
        case 'f':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\n",entry->d_name);
            }   break;
        case 'n':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_DIR) dc++;
                if(entry->d_type==DT_REG) fc++;
            }

            printf("%d Dir(s)\t%d File(s)\n",dc,fc);
            break;
        case 'i':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\t%lu\n",entry->d_name,entry->d_fileno);
            }
    }

    closedir(dp);
}

int main() {
    char buff[80],*args[10];
    int pid;

    while(1)
    {
        printf("myshell$");
        fflush(stdin);
        fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
        make_toks(buff,args);
        if(strcmp(args[0],"list")==0)
            list(args[2],args[1][0]);
        else
        {
            pid = fork();
            if(pid>0)
                wait();
            else
            {
              if(execvp(args[0],args)==-1)
                    printf("Bad command.\n");
            }
        }
    }
    return 0;
}
```

---

slip 3

Q1. Write the simulation program to implement demand paging and show the page scheduling and total number of page faults according to the LRU (using counter method) page replacement algorithm. Assume the memory of n frames.
Reference String : 3,5,7,2,5,1,2,3,1,3,5,3,1,6,2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
    int x;
    for(x=0;x<nf;x++)
    {
        if(sv==frame[x][0])
        {
            return x;
        }
    }
    return -2;
}
void displayMemory()
{
    int i;
    printf("\n\nFrame Contains |");
    for(i=0;i<nf;i++)
        printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
    int i,min=frame[0][1],ri=0;
    for(i=0;i<nf;i++)
    {
        if(frame[i][1]==-1)
        {
            return i;
        }
    }

    for(i=0;i<nf;i++)
    {
          if(min>frame[i][1])
        {
            min=frame[i][1];
            ri=i;
        }
    }
    return ri;
}
 int main()
```

```
{
   int rs[]={3,5,7,2,5,1,2,3,1,3,5,3,1,6,2};
   int ts=0;
   int n=15,i,j,pf=0,srch,insert_index;
   printf("Enter how many frames");
   scanf("%d",&nf);
   for(i=0;i<nf;i++)
   {
      for(j=0;j<2;j++)
         frame[i][j]=-1;
   }
   displayMemory();
   for(i=0;i<n;i++)
   {
      srch=searchFrames(rs[i]);
      if(srch==-2)
      {
         pf++;
         insert_index=findFreeFrame();
         frame[insert_index][0]=rs[i];
         frame[insert_index][1]=ts++;
      }
      else
      {
         frame[srch][1]=ts++;
      }

      displayMemory();
   }
   printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q2.Write a programto implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
count c filename :- To print number of characters in the file.
count w filename :- To print number of words in the file.
count l filename :- To print number of lines in the file

```
#include <sys/types.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void make_toks(char *s, char *tok[])
{
      int i=0;
   char *p;
   p = strtok(s," ");
while(p!=NULL)
{  tok[i++]=p;
      p=strtok(NULL," ");
}
tok[i]=NULL;
}
void count(char *fn, char op)
```

```
{ int fh,cc=0,wc=0,lc=0;
        char c;
fh = open(fn,O_RDONLY);
if(fh==-1) {
 printf("File %s not found.\n",fn);
 return;
}
while(read(fh,&c,1)>0)
{ if(c==' ')
    wc++;
        else if(c=='\n')
 {
 wc++;
 lc++;
 }  cc++;
} close(fh);
switch(op)
{
        case 'c':
 printf("No.of characters:%d\n",cc-1);
 break;
        case 'w':
 printf("No.of words:%d\n",wc);
 break;
    case 'l':
 printf("No.of lines:%d\n",lc+1);
  break;
}
} int main()
{
    char buff[80],*args[10];
    int pid;
    while(1) { printf("myshell$ ");
        fflush(stdin);
            fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
            make_toks(buff,args);
if(strcmp(args[0],"count")==0)
 count(args[2],args[1][0]);
else
 {  pid = fork();
     if(pid>0)
         wait();
         else
 {
 if(execvp(args[0],args)==-1)
 printf("Bad command.\n");
 }
 }
    }
        return 0;
}
```

---

slip 4

Q1.Write the simulation program for demand paging and show the page
scheduling and total number of page faults according the MFU page

replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
#define MAX 20

int frames[MAX],ref[MAX],mem[MAX][MAX],faults,
 sp,m,n,count[MAX];

void accept()
{
 int i;

 printf("Enter no.of frames:");
 scanf("%d", &n);
 printf("Enter no.of references:");
 scanf("%d", &m);

 printf("Enter reference string:\n");
 for(i=0;i<m;i++)
 {
  printf("[%d]=",i);
  scanf("%d",&ref[i]);
 }
}

void disp()
{
 int i,j;

 for(i=0;i<m;i++)
  printf("%3d",ref[i]);

 printf("\n\n");

 for(i=0;i<n;i++)
 {
  for(j=0;j<m;j++)
  {
   if(mem[i][j])
    printf("%3d",mem[i][j]);
   else
    printf("   ");
  }
  printf("\n");
 }

 printf("Total Page Faults: %d\n",faults);
}

int search(int pno)
{
 int i;

 for(i=0;i<n;i++)
 {
  if(frames[i]==pno)
   return i;
 }
 return -1;
}

int get_mfu(int sp)
{
 int i,max_i,max=-9999;

 i=sp;
 do
 {
  if(count[i]>max)
  {
   max = count[i];
   max_i = i;
  }
  i=(i+1)%n;
 }while(i!=sp);

 return max_i;
}

void mfu()
{
 int i,j,k;

 for(i=0;i<m && sp<n;i++)
 {
  k=search(ref[i]);
  if(k==-1)
  {
   frames[sp]=ref[i];
   count[sp]++;
   faults++;
   sp++;

   for(j=0;j<n;j++)
    mem[j][i]=frames[j];
  }
  else
   count[k]++;

 }

 sp=0;
 for(;i<m;i++)
 {
  k = search(ref[i]);
  if(k==-1)
  {
   sp = get_mfu(sp);
 frames[sp] = ref[i];
   count[sp]=1;
   faults++;
   sp = (sp+1)%n;

   for(j=0;j<n;j++)
    mem[j][i] = frames[j];
  }
  else
   count[k]++;
 }
}
int main()
{
 accept();
 mfu();
 disp();

 return 0;
}
```

Q2.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search a filename pattern :- To search all the occurrence of
 pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence
 of pattern in the file.

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

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void execute_command(char **args) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {

        if (execvp(args[0], args) < 0) {
            perror("Execution failed");
            exit(EXIT_FAILURE);
        }
    } else {
        // Parent process
        wait(NULL);
    }
}

void tokenize_input(char *input, char **args) {
    int index = 0;
    char *token = strtok(input, " \n");
    while (token != NULL) {
        args[index++] = token;
        token = strtok(NULL, " \n");
    }
    args[index] = NULL; // Null-terminate the array of arguments
}

int main() {
    char input[MAX_CMD_LEN];
    char *args[MAX_ARGS];

    while (1) {
        printf("myshell$ "); // Display the prompt
        if (fgets(input, sizeof(input), stdin) == NULL) {
            perror("fgets failed");
            continue;
        }


        if (strlen(input) == 1) continue;


        tokenize_input(input, args);


        if (strcmp(args[0], "exit") == 0) {
            exit(0);
        }

        else if (strcmp(args[0], "cd") == 0) {
            if (args[1] == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
        } else {

            execute_command(args);
        }
    }

    return 0;
}
```

---

slip 5

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int main()
{
  int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1,
flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");
```

```
for(i = 0; i < no_of_pages; ++i){
    scanf("%d", &pages[i]);
}

for(i = 0; i < no_of_frames; ++i){
    frames[i] = -1;
}

for(i = 0; i < no_of_pages; ++i){
    flag1 = flag2 = 0;

    for(j = 0; j < no_of_frames; ++j){
        if(frames[j] == pages[i]){
               flag1 = flag2 = 1;
               break;
           }
    }

    if(flag1 == 0){
        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == -1){
                faults++;
                frames[j] = pages[i];
                flag2 = 1;
                break;
            }
        }
    }

    if(flag2 == 0){
     flag3 =0;

        for(j = 0; j < no_of_frames; ++j){
         temp[j] = -1;

         for(k = i + 1; k < no_of_pages; ++k){
         if(frames[j] == pages[k]){
         temp[j] = k;
         break;
         }
         }
        }

        for(j = 0; j < no_of_frames; ++j){
         if(temp[j] == -1){
         pos = j;
         flag3 = 1;
          break;
         }
        }

        if(flag3 ==0){
         max = temp[0];
         pos = 0;

         for(j = 1; j < no_of_frames; ++j){
         if(temp[j] > max){
         max = temp[j];
         pos = j;
         }
```

```
            }
           }
frames[pos] = pages[i];
faults++;
       }
          printf("\n");
              for(j = 0; j < no_of_frames; ++j){
           printf("%d\t", frames[j]);
       }
   }
   printf("\n\nTotal Page Faults = %d", faults);

   return 0;
}
```

Q2.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence of pattern in the file.

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

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void execute_command(char **args) {
   pid_t pid = fork();
   if (pid < 0) {
      perror("Fork failed");
      exit(EXIT_FAILURE);
   } else if (pid == 0) {

      if (execvp(args[0], args) < 0) {
         perror("Execution failed");
         exit(EXIT_FAILURE);
      }
   } else {
      // Parent process
      wait(NULL);
   }
}

void tokenize_input(char *input, char **args) {
   int index = 0;
   char *token = strtok(input, " \n");
   while (token != NULL) {
      args[index++] = token;
      token = strtok(NULL, " \n");
   }
   args[index] = NULL; // Null-terminate the array of arguments
}

int main() {
    char input[MAX_CMD_LEN];
    char *args[MAX_ARGS];

    while (1) {
        printf("myshell$ "); // Display the prompt
        if (fgets(input, sizeof(input), stdin) == NULL) {
            perror("fgets failed");
            continue;
        }


        if (strlen(input) == 1) continue;


        tokenize_input(input, args);


        if (strcmp(args[0], "exit") == 0) {
            exit(0);
        }

        else if (strcmp(args[0], "cd") == 0) {
            if (args[1] == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
        } else {

            execute_command(args);
        }
    }

    return 0;
}
```

---

slip 6

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the MRU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
    int x;
    for(x=0;x<nf;x++)
    {
        if(sv==frame[x][0])
         {
            return x;
        }
    }
    return -2;
}
void displayMemory()
{
    int i;
    printf("\n\nFrame Contains |");
    for(i=0;i<nf;i++)
        printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
    int i,max=frame[0][1],ri=0;
    for(i=0;i<nf;i++)
    {
        if(frame[i][1]==-1)
        {
            return i;
        }
    }

    for(i=0;i<nf;i++)
    {
        if(max<frame[i][1])
        {
            max=frame[i][1];
            ri=i;
        }
    }
    return ri;
}
 int main()
{
    int rs[]={12,15,12,18,6,8,11,12,19,12,6,8,12,15,19,8};
    int ts=0;
    int n=16,i,j,pf=0,srch,insert_index;
    printf("Enter how many frames");
    scanf("%d",&nf);
    for(i=0;i<nf;i++)
    {
        for(j=0;j<2;j++)
            frame[i][j]=-1;
    }
    displayMemory();
    for(i=0;i<n;i++)
     {
        srch=searchFrames(rs[i]);
        if(srch==-2)
        {
            pf++;
            insert_index=findFreeFrame();
            frame[insert_index][0]=rs[i];
            frame[insert_index][1]=ts++;
        }
        else
        {
            frame[srch][1]=ts++;
        }
        displayMemory();
    }
    printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q2.Write a programto implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of
                                      pattern in the file.
myshell$ search a filename pattern :- To search all the occurrence of
                                      pattern in the file

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

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void execute_command(char **args) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {

        if (execvp(args[0], args) < 0) {
            perror("Execution failed");
            exit(EXIT_FAILURE);
        }
    } else {
        // Parent process
        wait(NULL);
    }
}

void tokenize_input(char *input, char **args) {
    int index = 0;
    char *token = strtok(input, " \n");
    while (token != NULL) {
        args[index++] = token;
        token = strtok(NULL, " \n");
    }
    args[index] = NULL; // Null-terminate the array of arguments
}

int main() {
    char input[MAX_CMD_LEN];
    char *args[MAX_ARGS];

    while (1) {
        printf("myshell$ "); // Display the prompt
        if (fgets(input, sizeof(input), stdin) == NULL) {
            perror("fgets failed");
            continue;
        }


        if (strlen(input) == 1) continue;


        tokenize_input(input, args);


        if (strcmp(args[0], "exit") == 0) {
            exit(0);
        }

        else if (strcmp(args[0], "cd") == 0) {
            if (args[1] == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
        } else {

            execute_command(args);
        }
    }

    return 0;
}
```

---

slip 7

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the Optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int main()
{
  int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1,
flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                    flag1 = flag2 = 1;
                    break;
                }
        }

        if(flag1 == 0){
            for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == -1){
                    faults++;
                    frames[j] = pages[i];
                    flag2 = 1;
                    break;
                }
            }
        }

        if(flag2 == 0){
         flag3 =0;

            for(j = 0; j < no_of_frames; ++j){
             temp[j] = -1;

             for(k = i + 1; k < no_of_pages; ++k){
             if(frames[j] == pages[k]){
             temp[j] = k;
             break;
             }
             }
            }

            for(j = 0; j < no_of_frames; ++j){
             if(temp[j] == -1){
             pos = j;
             flag3 = 1;
              break;
             }
            }

            if(flag3 ==0){
             max = temp[0];
             pos = 0;

             for(j = 1; j < no_of_frames; ++j){
             if(temp[j] > max){
             max = temp[j];
             pos = j;
             }
             }
            }
frames[pos] = pages[i];
faults++;
        }
```

```
          printf("\n");
              for(j = 0; j < no_of_frames; ++j){
           printf("%d\t", frames[j]);
        }
    }
    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

Q2.Write a program to implement shell. It should display the command prompt "myshell$". Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search a filename pattern :- To search all the occurrence of
                                  pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence
                                  of pattern in the file.

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

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void execute_command(char **args) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {

        if (execvp(args[0], args) < 0) {
            perror("Execution failed");
            exit(EXIT_FAILURE);
        }
    } else {
        // Parent process
        wait(NULL);
    }
}

void tokenize_input(char *input, char **args) {
    int index = 0;
    char *token = strtok(input, " \n");
    while (token != NULL) {
        args[index++] = token;
        token = strtok(NULL, " \n");
    }
    args[index] = NULL; // Null-terminate the array of arguments
}

int main() {
    char input[MAX_CMD_LEN];
    char *args[MAX_ARGS];
    while (1) {
        printf("myshell$ "); // Display the prompt
        if (fgets(input, sizeof(input), stdin) == NULL) {
            perror("fgets failed");
            continue;
        }


        if (strlen(input) == 1) continue;


        tokenize_input(input, args);


        if (strcmp(args[0], "exit") == 0) {
            exit(0);
        }

        else if (strcmp(args[0], "cd") == 0) {
            if (args[1] == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
        } else {

            execute_command(args);
        }
    }

    return 0;
}
```

---

slip 8

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
    int x;
    for(x=0;x<nf;x++)
    {
        if(sv==frame[x][0])
        {
            return x;
        }
    }
    return -2;
}
```

```
void displayMemory()
{
    int i;
    printf("\n\nFrame Contains |");
    for(i=0;i<nf;i++)
        printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
    int i,min=frame[0][1],ri=0;
    for(i=0;i<nf;i++)
    {
        if(frame[i][1]==-1)
        {
            return i;
        }
    }

    for(i=0;i<nf;i++)
    {
           if(min>frame[i][1])
        {
            min=frame[i][1];
            ri=i;
        }
    }
    return ri;
}
 int main()
{
    int rs[]={3,5,7,2,5,1,2,3,1,3,5,3,1,6,2};
    int ts=0;
    int n=15,i,j,pf=0,srch,insert_index;
    printf("Enter how many frames");
    scanf("%d",&nf);
    for(i=0;i<nf;i++)
    {
        for(j=0;j<2;j++)
            frame[i][j]=-1;
    }
    displayMemory();
    for(i=0;i<n;i++)
    {
        srch=searchFrames(rs[i]);
        if(srch==-2)
        {
            pf++;
            insert_index=findFreeFrame();
            frame[insert_index][0]=rs[i];
            frame[insert_index][1]=ts++;
        }
        else
        {
            frame[srch][1]=ts++;
        }

        displayMemory();
    }
    printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q2.Write a programto implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of
                                 pattern in the file.
myshell$ search c filename pattern :- To count the number of occurrence
                                 of pattern in the file.

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

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void execute_command(char **args) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {

        if (execvp(args[0], args) < 0) {
            perror("Execution failed");
            exit(EXIT_FAILURE);
        }
    } else {
        // Parent process
        wait(NULL);
    }
}

void tokenize_input(char *input, char **args) {
    int index = 0;
    char *token = strtok(input, " \n");
    while (token != NULL) {
        args[index++] = token;
        token = strtok(NULL, " \n");
    }
    args[index] = NULL; // Null-terminate the array of arguments
}

int main() {
    char input[MAX_CMD_LEN];
    char *args[MAX_ARGS];

    while (1) {
        printf("myshell$ "); // Display the prompt
        if (fgets(input, sizeof(input), stdin) == NULL) {
            perror("fgets failed");
            continue;
        }
        if (strlen(input) == 1) continue;


        tokenize_input(input, args);


        if (strcmp(args[0], "exit") == 0) {
            exit(0);
        }

        else if (strcmp(args[0], "cd") == 0) {
            if (args[1] == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
        } else {

            execute_command(args);
        }
    }

    return 0;
}
```

---

slip 9

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
    int x;
    for(x=0;x<nf;x++)
    {
        if(sv==frame[x][0])
        {
            return 0;
        }
    }
    return 1;
}
void displayMemory()
{
    int i;
    printf("\n\nFrame Contains |");
    for(i=0;i<nf;i++)
        printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
    int i,min=frame[0][1],ri=0;
    for(i=0;i<nf;i++)
    {
        if(frame[i][1]==-1)
        {
            return i;
        }
    }
    //LRU
    for(i=0;i<nf;i++)
    {
        if(min>frame[i][1])
        {
            min=frame[i][1];
            ri=i;
        }
    }
    return ri;
}
int main()
{
    int rs[]={3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int ts=0;
    int n=15,i,j,pf=0,srch,insert_index;
    printf("Enter how many frames");
    scanf("%d",&nf);
    for(i=0;i<nf;i++)
    {
        for(j=0;j<2;j++)
            frame[i][j]=-1;
    }
    displayMemory();
    for(i=0;i<n;i++)
    {
        srch=searchFrames(rs[i]);
        if(srch==1)
        {
            pf++;
            insert_index=findFreeFrame();
            frame[insert_index][0]=rs[i];
            frame[insert_index][1]=ts++;
        }

        displayMemory();
    }
    printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q.2 Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.

myshell$ search f filename pattern :- To display first occurrence of pattern in the file.

myshell$ search a filename pattern :- To search all the occurrence of pattern in the file.

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

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

#define MAX_CMD_LEN 1024
#define MAX_ARGS 100

void execute_command(char **args) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {

        if (execvp(args[0], args) < 0) {
            perror("Execution failed");
            exit(EXIT_FAILURE);
        }
    } else {
        // Parent process
        wait(NULL);
    }
}

void tokenize_input(char *input, char **args) {
    int index = 0;
    char *token = strtok(input, " \n");
    while (token != NULL) {
        args[index++] = token;
        token = strtok(NULL, " \n");
    }
    args[index] = NULL; // Null-terminate the array of arguments
}

int main() {
    char input[MAX_CMD_LEN];
    char *args[MAX_ARGS];

    while (1) {
        printf("myshell$ "); // Display the prompt
        if (fgets(input, sizeof(input), stdin) == NULL) {
            perror("fgets failed");
            continue;
        }


        if (strlen(input) == 1) continue;


        tokenize_input(input, args);


        if (strcmp(args[0], "exit") == 0) {
            exit(0);
        }

        else if (strcmp(args[0], "cd") == 0) {
            if (args[1] == NULL) {
                fprintf(stderr, "cd: missing argument\n");
            } else {
                if (chdir(args[1]) != 0) {
                    perror("cd failed");
                }
            }
        } else {

            execute_command(args);
        }
    }

    return 0;
}
```

---

slip 10

Q.1 Write the simulation program for demand paging and show the page scheduling and total number of page faults according the FIFO page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
    int x;
    for(x=0;x<nf;x++)
    {
        if(sv==frame[x][0])
        {
            return 0;
        }
    }
    return 1;
}
void displayMemory()
{
    int i;
    printf("\n\nFrame Contains |");
    for(i=0;i<nf;i++)
        printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
    int i,min=frame[0][1],ri=0;
    for(i=0;i<nf;i++)
    {
        if(frame[i][1]==-1)
        {
            return i;
        }
    }
    //LRU
    for(i=0;i<nf;i++)
    {
        if(min>frame[i][1])
        {
            min=frame[i][1];
            ri=i;
        }
    }
    return ri;
}
int main()
{
    int rs[]={3, 4, 5, 6, 3, 4, 7, 3, 4, 5, 6, 7, 2, 4, 6};
    int ts=0;
    int n=15,i,j,pf=0,srch,insert_index;
    printf("Enter how many frames");
    scanf("%d",&nf);
    for(i=0;i<nf;i++)
    {
        for(j=0;j<2;j++)
            frame[i][j]=-1;
    }
    displayMemory();
    for(i=0;i<n;i++)
    {
        srch=searchFrames(rs[i]);
        if(srch==1)
        {
            pf++;
            insert_index=findFreeFrame();
            frame[insert_index][0]=rs[i];
            frame[insert_index][1]=ts++;
        }

        displayMemory();
    }
    printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q2.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given following command by creating the child process. Additionally it should interpret the ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
void make_toks(char *s, char *tok[])
{
    int i=0;
    char *p;
    p = strtok(s," ");
    while(p!=NULL)
    {   tok[i++]=p;
```

```
        p=strtok(NULL," ");
    }

    tok[i]=NULL;
}

void list(char *dn, char op)
{
    DIR *dp;
    struct dirent *entry;
    int dc=0,fc=0;

    dp = opendir(dn);
    if(dp==NULL)
    {
        printf("Dir %s not found.\n",dn);
        return;
    }
      switch(op)
    {
        case 'f':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\n",entry->d_name);
            }   break;
        case 'n':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_DIR) dc++;
                if(entry->d_type==DT_REG) fc++;
            }

            printf("%d Dir(s)\t%d File(s)\n",dc,fc);
            break;
        case 'i':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\t%lu\n",entry->d_name,entry->d_fileno);
            }
    }

    closedir(dp);
}

int main() {
    char buff[80],*args[10];
    int pid;

    while(1)
    {
        printf("myshell$");
        fflush(stdin);
        fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
        make_toks(buff,args);
        if(strcmp(args[0],"list")==0)
            list(args[2],args[1][0]);
        else
        {
            pid = fork();
            if(pid>0)
                wait();
            else
            {
               if(execvp(args[0],args)==-1)
                    printf("Bad command.\n");
            }
        }
    }
    return 0;
}
```

---

slip 11

Q1.Write the simulation program to implement demand paging and show the page scheduling and total number of page faults according to the LFU page replacement algorithm.Assume the memory of n frames.
Reference String : 3,4,5,4,3,4,7,2,4,5,6,7,2,4,6

```
#include<stdio.h>
#define MAX 20
int frames[MAX],ref[MAX],mem[MAX][MAX],faults,
sp,m,n,count[MAX];
void accept()
{
int i;
printf("Enter no.of frames:");
scanf("%d", &n);
printf("Enter no.of references:");
scanf("%d", &m);
printf("Enter reference string:\n");
for(i=0;i<m;i++)
{
printf("[%d]=",i);
scanf("%d",&ref[i]);
}
}
void disp()
{
int i,j;
for(i=0;i<m;i++)
printf("%3d",ref[i]);
printf("\n\n");
for(i=0;i<n;i++)
 {
  for(j=0;j<m;j++)
  {
   if(mem[i][j])
    printf("%3d",mem[i][j]);
   else
    printf("   ");
  }
  printf("\n");
 }
 printf("Total Page Faults: %d\n",faults);
```

```
}
int search(int pno)
{
 int i;
 for(i=0;i<n;i++)
 {
  if(frames[i]==pno)
   return i;
 }
 return -1;
}
int get_lfu(int sp)
{
 int i,min_i,min=9999;

 i=sp;
 do
 {
  if(count[i]<min)
  {
   min = count[i];
   min_i = i;
  }
  i=(i+1)%n;
 }while(i!=sp);

 return min_i;
}
void lfu()
{
 int i,j,k;

 for(i=0;i<m && sp<n;i++)
 {
  k=search(ref[i]);
  if(k==-1)
  {
   frames[sp]=ref[i];
   count[sp]++;
   faults++;
   sp++;

   for(j=0;j<n;j++)
    mem[j][i]=frames[j];
  }
  else
   count[k]++;

 }

 sp=0;
 for(;i<m;i++)
 {
  k = search(ref[i]);
  if(k==-1)
  {
   sp = get_lfu(sp);
   frames[sp] = ref[i];
   count[sp]=1;
   faults++;
   sp = (sp+1)%n;

   for(j=0;j<n;j++)
    mem[j][i] = frames[j];
  }
  else
   count[k]++;
 }
}


int main()
{
 accept();
 lfu();
 disp();

 return 0;
}
```

Q2.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given following command by creating the child process. Additionally it should interpret the ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current directory.
myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
void make_toks(char *s, char *tok[])
{
    int i=0;
    char *p;
    p = strtok(s," ");
    while(p!=NULL)
    {   tok[i++]=p;
        p=strtok(NULL," ");
    }

    tok[i]=NULL;
}

void list(char *dn, char op)
{
    DIR *dp;
    struct dirent *entry;
    int dc=0,fc=0;

    dp = opendir(dn);
    if(dp==NULL)
    {
        printf("Dir %s not found.\n",dn);
        return;
    }
      switch(op)
    {
        case 'f':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\n",entry->d_name);
            }   break;
        case 'n':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_DIR) dc++;
                if(entry->d_type==DT_REG) fc++;
            }

            printf("%d Dir(s)\t%d File(s)\n",dc,fc);
            break;
        case 'i':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\t%lu\n",entry->d_name,entry->d_fileno);
            }
    }

    closedir(dp);
}

int main() {
    char buff[80],*args[10];
    int pid;

    while(1)
    {
        printf("myshell$");
        fflush(stdin);
        fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
        make_toks(buff,args);
        if(strcmp(args[0],"list")==0)
            list(args[2],args[1][0]);
        else
        {
            pid = fork();
            if(pid>0)
                wait();
            else
            {
               if(execvp(args[0],args)==-1)
                    printf("Bad command.\n");
            }
        }
    }
    return 0;
}
```

slip 12

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
   int x;
   for(x=0;x<nf;x++)
   {
      if(sv==frame[x][0])
      {
         return x;
      }
   }
   return -2;
}
void displayMemory()
{
   int i;
   printf("\n\nFrame Contains |");
   for(i=0;i<nf;i++)
      printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
   int i,min=frame[0][1],ri=0;
   for(i=0;i<nf;i++)
   {
      if(frame[i][1]==-1)
      {
         return i;
      }
   }

   for(i=0;i<nf;i++)
   {
        if(min>frame[i][1])
      {
         min=frame[i][1];
         ri=i;
      }
   }
   return ri;
}
 int main()
{
   int rs[]={3,5,7,2,5,1,2,3,1,3,5,3,1,6,2};
   int ts=0;
   int n=15,i,j,pf=0,srch,insert_index;
   printf("Enter how many frames");
   scanf("%d",&nf);
   for(i=0;i<nf;i++)
   {
        for(j=0;j<2;j++)
            frame[i][j]=-1;
    }
    displayMemory();
    for(i=0;i<n;i++)
    {
        srch=searchFrames(rs[i]);
        if(srch==-2)
        {
            pf++;
            insert_index=findFreeFrame();
            frame[insert_index][0]=rs[i];
            frame[insert_index][1]=ts++;
        }
        else
        {
            frame[srch][1]=ts++;
        }

        displayMemory();
    }
    printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q2.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given following command by creating the child process. Additionally it should interpret the ‘list’ commands as

myshell$ list f dirname :- To print names of all the files in current directory.

myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
void make_toks(char *s, char *tok[])
{
    int i=0;
    char *p;
    p = strtok(s," ");
    while(p!=NULL)
    {   tok[i++]=p;
        p=strtok(NULL," ");
    }

    tok[i]=NULL;
}

void list(char *dn, char op)
{
    DIR *dp;
    struct dirent *entry;
    int dc=0,fc=0;
    dp = opendir(dn);
    if(dp==NULL)
    {
        printf("Dir %s not found.\n",dn);
        return;
    }
      switch(op)
    {
        case 'f':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\n",entry->d_name);
            }   break;
        case 'n':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_DIR) dc++;
                if(entry->d_type==DT_REG) fc++;
            }

            printf("%d Dir(s)\t%d File(s)\n",dc,fc);
            break;
        case 'i':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\t%lu\n",entry->d_name,entry->d_fileno);
            }
    }

    closedir(dp);
}

int main() {
    char buff[80],*args[10];
    int pid;

    while(1)
    {
        printf("myshell$");
        fflush(stdin);
        fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
        make_toks(buff,args);
        if(strcmp(args[0],"list")==0)
            list(args[2],args[1][0]);
        else
        {
            pid = fork();
            if(pid>0)
                wait();
            else
            {
               if(execvp(args[0],args)==-1)
                    printf("Bad command.\n");
            }
        }
    }
```

```
   return 0;
}
```

---

slip 13

Q1.Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as
typeline +n filename :- To print first n lines in the file.
typeline -a filename :- To print all lines in the file.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <unistd.h>
int make_toks(char *s, char *tok[])
{
      int i = 0;
   char *p;
   p = strtok(s, " ");
while(p != NULL)
{
      tok[i++] = p;
   p = strtok(NULL, " ");
}
tok[i] = NULL;
return i;
}
void typeline(char *op, char *fn)
{
      int fh, i, j, n;
   char c;
fh = open(fn, O_RDONLY);
if(fh == -1)
{
printf("File %s not found.\n", fn);
return;
}
if(strcmp(op, "a") == 0)
{
   while(read(fh, &c, 1) > 0)
         printf("%c", c);
      close(fh);
   return;
}
n = atoi(op);
if(n > 0)
{
      i = 0;
      while(read(fh, &c, 1) > 0)
   {
         printf("%c", c);
```

```
  if(c == '\n')
              i++;
       if(i == n)
           break;
}
}
if(n < 0)
{
       i = 0;
       while(read(fh, &c, 1) > 0)
           {
           if(c == '\n') i++;
       }
       lseek(fh, 0, SEEK_SET);
j = 0;
while(read(fh, &c, 1) > 0)
{
    if(c == '\n')
           j++;
    if(j == i+n+1)
           break;
}
while(read(fh, &c, 1) > 0)
{
       printf("%c", c);
}
}
close(fh);
}
int main()
{
       char buff[80], *args[10];
       while(1)
    {
           printf ("\n");
       printf("\nmyshell$ ");
       fgets(buff, 80, stdin);
           buff[strlen(buff)-1] = '\0';
           int n = make_toks(buff, args);
       switch (n)
       {
              case 1: if(strcmp(args[0], "exit") == 0)
exit(1);
if (!fork())
execlp (args [0], args[0], NULL);
break;
           case 2:
if (!fork ())
execlp (args [0], args[0], args[1], NULL);
break;
              case 3: if (strcmp(args[0], "typeline") == 0)
                     typeline (args[1], args[2]);
else
{
       if (!fork ())
execlp (args [0], args[0], args[1], args[2], NULL);
}
break;
case 4:
if (!fork ())
```

```
execlp (args [0], args [0], args [1], args [2], args [3], NULL);
break;
}
}
return 0;
}
```

Q.2 Write the simulation program for Round Robin scheduling for given time quantum. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give the Gantt chart, turnaround time and waiting time for each process. Also display the average turnaround time and average waiting time.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct process_info
{
    char pname[20];
    int at,bt,ct,bt1;
    struct process_info *next;
}NODE;
int n,ts;
NODE *first,*last;
void accept_info()
{
    NODE *p;
    int i;
    printf("Enter no.of process:");
 scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        p = (NODE*)malloc(sizeof(NODE));
        printf("Enter process name:");
        scanf("%s",p->pname);
        printf("Enter arrival time:");
        scanf("%d",&p->at);
        printf("Enter first CPU burst time:");
        scanf("%d",&p->bt);
        p->bt1 = p->bt;

        p->next = NULL;
        if(first==NULL)
            first=p;
        else
            last->next=p;
        last = p;
    }
    printf("Enter time slice:");
    scanf("%d",&ts);
}
void print_output()
{
    NODE *p;
    float avg_tat=0,avg_wt=0;
    printf("pname\tat\tbt\tct\ttat\twt\n");
    p = first;
    while(p!=NULL)
    {
        int tat = p->ct-p->at;
        int wt = tat-p->bt;

        avg_tat+=tat;
        avg_wt+=wt;
        printf("%s\t%d\t%d\t%d\t%d\t%d\n",
                p->pname,p->at,p->bt,p->ct,tat,wt);

        p=p->next;
    }
    printf("Avg TAT=%f\tAvg WT=%f\n",
            avg_tat/n,avg_wt/n);
}
void print_input()
{
    NODE *p;
    p = first;
    printf("pname\tat\tbt\n");
 while(p!=NULL)
    {
        printf("%s\t%d\t%d\n",
                p->pname,p->at,p->bt1);
        p = p->next;
    }
}
void sort()
{
    NODE *p,*q;
    int t;
    char name[20];
    p = first;
    while(p->next!=NULL)
    {
        q=p->next;
        while(q!=NULL)
        {
            if(p->at > q->at)
            {
                strcpy(name,p->pname);
                strcpy(p->pname,q->pname);
                strcpy(q->pname,name);
                t = p->at;
                p->at = q->at;
                q->at = t;

                t = p->bt;
                p->bt = q->bt;
                q->bt = t;
                t = p->ct;
                p->ct = q->ct;
                q->ct = t;
                t = p->bt1;
                p->bt1 = q->bt1;
                q->bt1 = t;
            }
            q=q->next;
        }
        p=p->next;
    }
}
int time;
int is_arrived()
{
    NODE *p;
    p = first;
    while(p!=NULL)
    {
        if(p->at<=time && p->bt1!=0)
            return 1;
        p=p->next;
    }
    return 0;
}
NODE * delq()
{
    NODE *t;
    t = first;
    first = first->next;
    t->next=NULL;
    return t;
}
void addq(NODE *t)
{
    last->next = t;
    last = t;
}
struct gantt_chart
{
    int start;
    char pname[30];
    int end;
}s[100],s1[100];
int k;
void rr()
{
    int prev=0,n1=0;
    NODE *p;
    while(n1!=n)
    {
        if(!is_arrived())
        {
            time++;
            s[k].start = prev;
            strcpy(s[k].pname,"*");
            s[k].end = time;
            k++;
            prev=time;
        }
        else
        {
            p = first;
            while(1)
            {
                if(p->at<=time && p->bt1!=0)
                     break;
                p = delq();
                addq(p);
                p = first;
            }
            if(p->bt1<=ts)
            {
```

```
            time+=p->bt1;
            p->bt1=0;
         }
         else
         {
            time+=ts;
            p->bt1-=ts;
         }
         p->ct = time;
         s[k].start = prev;
         strcpy(s[k].pname,p->pname);
         s[k].end = time;
         k++;
         prev = time;

         if(p->bt1==0) n1++;
         p = delq();
         addq(p);
      }
      print_input();
   }
}
void print_gantt_chart()
{
   int i,j,m;
   s1[0] = s[0];
   for(i=1,j=0;i<k;i++)
   {
      if(strcmp(s[i].pname,s1[j].pname)==0)
         s1[j].end = s[i].end;
      else
         s1[++j] = s[i];
   }
   printf("%d",s1[0].start);
   for(i=0;i<=j;i++)
   {
      m = (s1[i].end - s1[i].start);
      for(k=0;k<m/2;k++)
         printf("-");
      printf("%s",s1[i].pname);
      for(k=0;k<(m+1)/2;k++)
  printf("-");
      printf("%d",s1[i].end);
   }
}
int main()
{
   accept_info();
   sort();
   rr();
   print_output();
   print_gantt_chart();
      return 0;
}
```

---

slip 14

Q1.Write a C program to implement the shell which displays the
command prompt “myshell$”. It accepts the command, tokenize the command
line and execute it by creating the child process. Also implement
the additional command ‘typeline’ as
typeline +n filename :- To print first n lines in the file.
typeline -a filename :- To print all lines in the file.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <unistd.h>
int make_toks(char *s, char *tok[])
{
        int i = 0;
    char *p;
    p = strtok(s, " ");
while(p != NULL)
{
       tok[i++] = p;
    p = strtok(NULL, " ");
}
tok[i] = NULL;
return i;
}
void typeline(char *op, char *fn)
{
        int fh, i, j, n;
    char c;
fh = open(fn, O_RDONLY);
if(fh == -1)
{
printf("File %s not found.\n", fn);
return;
}
if(strcmp(op, "a") == 0)
{
    while(read(fh, &c, 1) > 0)
            printf("%c", c);
        close(fh);
    return;
}
n = atoi(op);
if(n > 0)
{
        i = 0;
        while(read(fh, &c, 1) > 0)
    {
            printf("%c", c);
  if(c == '\n')
                i++;
        if(i == n)
            break;
}
}
if(n < 0)
{
        i = 0;
        while(read(fh, &c, 1) > 0)
            {
            if(c == '\n') i++;
        }
        lseek(fh, 0, SEEK_SET);
j = 0;
while(read(fh, &c, 1) > 0)
{
    if(c == '\n')
            j++;
    if(j == i+n+1)
            break;
}
while(read(fh, &c, 1) > 0)
{
        printf("%c", c);
}
}
close(fh);
}
int main()
{
        char buff[80], *args[10];
        while(1)
    {
            printf ("\n");
        printf("\nmyshell$ ");
        fgets(buff, 80, stdin);
            buff[strlen(buff)-1] = '\0';
            int n = make_toks(buff, args);
        switch (n)
        {
                case 1: if(strcmp(args[0], "exit") == 0)
exit(1);
if (!fork())
execlp (args [0], args[0], NULL);
break;
            case 2:
if (!fork ())
execlp (args [0], args[0], args[1], NULL);
break;
                case 3: if (strcmp(args[0], "typeline") == 0)
                        typeline (args[1], args[2]);
else
{
        if (!fork ())
execlp (args [0], args[0], args[1], args[2], NULL);
}
break;
case 4:
if (!fork ())
execlp (args [0], args [0], args [1], args [2], args [3], NULL);
break;
}
}
return 0;
}
```

Q.2 Write a C program to simulate Non-preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1;
 struct process_info *next;
}NODE;

int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);

 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));

  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);


  p->bt1 = p->bt;
  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tct\ttat\twt\n");
 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->ct,tat,wt);

  p=p->next;
 }

 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;

 p = first;

 printf("pname\tat\tbt\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\n",
   p->pname,p->at,p->bt1);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;
    t = p->ct;
    p->ct = q->ct;
    q->ct = t;

    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;

   }

   q=q->next;
  }

  p=p->next;
 }
}

int time;

NODE * get_sjf()
{
 NODE *p,*min_p=NULL;
 int min=9999;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0 &&
   p->bt1<min)
  {
   min = p->bt1;
   min_p = p;
  }
  p=p->next;
 }

 return min_p;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void sjfnp()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_sjf();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;

   prev = time;
   k++;
  }
  else
  {
   time+=p->bt1;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1 = 0;

   n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}
```

```
int main()
{
 accept_info();
 sort();
 sjfnp();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 15

Q1.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given following command by creating the child process. Additionally it should interpret the ‘list’ commands as

myshell$ list f dirname :- To print names of all the files in current directory.

myshell$ list n dirname :- To print the number of all entries in the current directory

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
void make_toks(char *s, char *tok[])
{
   int i=0;
   char *p;
   p = strtok(s," ");
   while(p!=NULL)
   {   tok[i++]=p;
      p=strtok(NULL," ");
   }

   tok[i]=NULL;
}

void list(char *dn, char op)
{
   DIR *dp;
   struct dirent *entry;
   int dc=0,fc=0;

   dp = opendir(dn);
   if(dp==NULL)
   {
      printf("Dir %s not found.\n",dn);
      return;
   }
     switch(op)
   {
        case 'f':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\n",entry->d_name);
            }   break;
        case 'n':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_DIR) dc++;
                if(entry->d_type==DT_REG) fc++;
            }

            printf("%d Dir(s)\t%d File(s)\n",dc,fc);
            break;
        case 'i':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\t%lu\n",entry->d_name,entry->d_fileno);
            }
    }

    closedir(dp);
}

int main() {
    char buff[80],*args[10];
    int pid;

    while(1)
    {
        printf("myshell$");
        fflush(stdin);
        fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
        make_toks(buff,args);
        if(strcmp(args[0],"list")==0)
            list(args[2],args[1][0]);
        else
        {
            pid = fork();
            if(pid>0)
                wait();
            else
            {
               if(execvp(args[0],args)==-1)
                    printf("Bad command.\n");
            }
        }
    }
    return 0;
}
```

Q.2 Write the program to simulate preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1;
 struct process_info *next;
}NODE;

int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);

 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));

  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);

  p->bt1 = p->bt;

  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tct\ttat\twt\n");

 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->ct,tat,wt);

  p=p->next;
 }

 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;

 p = first;

 printf("pname\tat\tbt\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\n",
   p->pname,p->at,p->bt1);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;

    t = p->ct;
    p->ct = q->ct;
    q->ct = t;
    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;
   }

   q=q->next;
  }

  p=p->next;
 }
}

int time;

NODE * get_sjf()
{
 NODE *p,*min_p=NULL;
 int min=9999;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0 &&
   p->bt1<min)
  {
   min = p->bt1;
   min_p = p;
  }
  p=p->next;
 }

 return min_p;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void sjfp()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_sjf();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;

   prev = time;
   k++;
  }
  else
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1--;

   if(p->bt1==0)
    n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}

int main()
{
 accept_info();
 sort();
 sjfp();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 16

Q1.Write a programto implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
count c filename :- To print number of characters in the file.
count w filename :- To print number of words in the file.
count l filename :- To print number of lines in the file

```
#include <sys/types.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void make_toks(char *s, char *tok[])
{
        int i=0;
    char *p;
    p = strtok(s," ");
while(p!=NULL)
{  tok[i++]=p;
        p=strtok(NULL," ");
}
tok[i]=NULL;
}
void count(char *fn, char op)
{ int fh,cc=0,wc=0,lc=0;
        char c;
fh = open(fn,O_RDONLY);
if(fh==-1) {
 printf("File %s not found.\n",fn);
 return;
}
while(read(fh,&c,1)>0)
{ if(c==' ')
    wc++;
        else if(c=='\n')
 {
 wc++;
 lc++;
 }  cc++;
} close(fh);
switch(op)
{
        case 'c':
 printf("No.of characters:%d\n",cc-1);
 break;
        case 'w':
```

```
 printf("No.of words:%d\n",wc);
 break;
    case 'l':
 printf("No.of lines:%d\n",lc+1);
  break;
}
} int main()
{
   char buff[80],*args[10];
   int pid;
   while(1) { printf("myshell$ ");
       fflush(stdin);
          fgets(buff,80,stdin);
       buff[strlen(buff)-1]='\0';
          make_toks(buff,args);
if(strcmp(args[0],"count")==0)
 count(args[2],args[1][0]);
else
 {  pid = fork();
     if(pid>0)
        wait();
        else
 {
 if(execvp(args[0],args)==-1)
 printf("Bad command.\n");
 }
 }
   }
       return 0;
}
```

Q.2 Write the program to simulate Non preemptive priority scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1,p;
 struct process_info *next;
}NODE;

int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);
 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));

  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);

  printf("Enter priority:");
  scanf("%d",&p->p);

  p->bt1 = p->bt;

  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tp\ttct\ttat\twt\n");

 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->p,p->ct,tat,wt);

  p=p->next;
 }

 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;

 p = first;
 printf("pname\tat\tbt\tp\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt1,p->p);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;

    t = p->ct;
    p->ct = q->ct;
    q->ct = t;

    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;

    t = p->p;
    p->p = q->p;
    q->p = t;
   }

   q=q->next;
  }

  p=p->next;
 }
}

int time;

NODE * get_p()
{
 NODE *p,*min_p=NULL;
 int min=9999;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0 &&
   p->p<min)
  {
   min = p->p;
   min_p = p;
  }
  p=p->next;
 }

 return min_p;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void pnp()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_p();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;

   prev = time;
   k++;
  }
  else
  {
   time+=p->bt1;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1 = 0;

   n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}

int main()
{
 accept_info();
 sort();
 pnp();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 17

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the Optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6, 2

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

```
int main()
{
  int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1,
flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;

        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                   flag1 = flag2 = 1;
                   break;
               }
        }

        if(flag1 == 0){
            for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == -1){
                    faults++;
                    frames[j] = pages[i];
                    flag2 = 1;
                    break;
                }
            }
        }

        if(flag2 == 0){
         flag3 =0;

            for(j = 0; j < no_of_frames; ++j){
             temp[j] = -1;

             for(k = i + 1; k < no_of_pages; ++k){
             if(frames[j] == pages[k]){
             temp[j] = k;
             break;
             }
             }
            }

            for(j = 0; j < no_of_frames; ++j){
             if(temp[j] == -1){
             pos = j;
             flag3 = 1;
              break;
             }
            }

            if(flag3 ==0){
             max = temp[0];
             pos = 0;

             for(j = 1; j < no_of_frames; ++j){
             if(temp[j] > max){
             max = temp[j];
             pos = j;
             }
             }
            }
frames[pos] = pages[i];
faults++;
        }
          printf("\n");
                for(j = 0; j < no_of_frames; ++j){
            printf("%d\t", frames[j]);
        }
    }
    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

Q.2 Write the program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1;
 struct process_info *next;
}NODE;

int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);

 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));
  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);

  p->bt1 = p->bt;

  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tct\ttat\twt\n");

 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->ct,tat,wt);

  p=p->next;
 }

 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;

 p = first;

 printf("pname\tat\tbt\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\n",
   p->pname,p->at,p->bt1);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;

    t = p->ct;
    p->ct = q->ct;
    q->ct = t;

    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;
   }

   q=q->next;
  }

  p=p->next;
 }
}

int time;

NODE * get_fcfs()
{
 NODE *p;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0)
   return p;

  p=p->next;
 }
 return NULL;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void fcfs()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_fcfs();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;

   prev = time;
   k++;
  }
  else
  {
   time+=p->bt1;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1 = 0;

   n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}

int main()
{
 accept_info();
 sort();
 fcfs();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 18

Q1.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the LRU page replacement algorithm. Assume the memory of n frames.
Reference String : 8, 5, 7, 8, 5, 7, 2, 3, 7, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int frame[5][2],nf;
int searchFrames(int sv)
{
    int x;
    for(x=0;x<nf;x++)
    {
        if(sv==frame[x][0])
        {
            return x;
        }
    }
    return -2;
}
void displayMemory()
{
    int i;
    printf("\n\nFrame Contains |");
    for(i=0;i<nf;i++)
        printf(" %d  | ",frame[i][0]);
}
int findFreeFrame()
{
    int i,min=frame[0][1],ri=0;
    for(i=0;i<nf;i++)
    {
        if(frame[i][1]==-1)
        {
            return i;
        }
    }

    for(i=0;i<nf;i++)
    {
          if(min>frame[i][1])
        {
            min=frame[i][1];
            ri=i;
        }
    }
    return ri;
}
 int main()
{
    int rs[]={3,5,7,2,5,1,2,3,1,3,5,3,1,6,2};
    int ts=0;
    int n=15,i,j,pf=0,srch,insert_index;
    printf("Enter how many frames");
    scanf("%d",&nf);
    for(i=0;i<nf;i++)
    {
        for(j=0;j<2;j++)
            frame[i][j]=-1;
    }
    displayMemory();
    for(i=0;i<n;i++)
    {
        srch=searchFrames(rs[i]);
        if(srch==-2)
        {
            pf++;
            insert_index=findFreeFrame();
            frame[insert_index][0]=rs[i];
            frame[insert_index][1]=ts++;
        }
        else
        {
            frame[srch][1]=ts++;
        }

        displayMemory();
    }
    printf("\n\nTotal Page Faults Occured is %d\n",pf);
}
```

Q.2 Write the program to simulate FCFS CPU-scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1;
 struct process_info *next;
}NODE;

int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);

 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));

  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);

  p->bt1 = p->bt;

  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tct\ttat\twt\n");
 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->ct,tat,wt);

  p=p->next;
 }

 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;

 p = first;

 printf("pname\tat\tbt\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\n",
   p->pname,p->at,p->bt1);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;
    t = p->ct;
    p->ct = q->ct;
    q->ct = t;

    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;
   }

   q=q->next;
  }

  p=p->next;
 }
}

int time;

NODE * get_fcfs()
{
 NODE *p;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0)
   return p;

  p=p->next;
 }

 return NULL;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void fcfs()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_fcfs();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;
   prev = time;
   k++;
  }
  else
  {
   time+=p->bt1;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1 = 0;

   n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}

int main()
{
 accept_info();
 sort();
 fcfs();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 19

Q1.Write a program to implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given following command by creating the child process. Additionally it should interpret the ‘list’ commands as
myshell$ list f dirname :- To print names of all the files in current
directory.
myshell$ list n dirname :- To print the number of all entries in the current
directory

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <dirent.h>
#include <unistd.h>
void make_toks(char *s, char *tok[])
{
   int i=0;
   char *p;
   p = strtok(s," ");
   while(p!=NULL)
   {   tok[i++]=p;
      p=strtok(NULL," ");
   }

   tok[i]=NULL;
}

void list(char *dn, char op)
{
   DIR *dp;
   struct dirent *entry;
   int dc=0,fc=0;

   dp = opendir(dn);
   if(dp==NULL)
   {
      printf("Dir %s not found.\n",dn);
      return;
   }
    switch(op)
   {
      case 'f':
         while(entry=readdir(dp))
         {
            if(entry->d_type==DT_REG)
               printf("%s\n",entry->d_name);
            }   break;
        case 'n':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_DIR) dc++;
                if(entry->d_type==DT_REG) fc++;
            }

            printf("%d Dir(s)\t%d File(s)\n",dc,fc);
            break;
        case 'i':
            while(entry=readdir(dp))
            {
                if(entry->d_type==DT_REG)
                    printf("%s\t%lu\n",entry->d_name,entry->d_fileno);
            }
    }

    closedir(dp);
}

int main() {
    char buff[80],*args[10];
    int pid;

    while(1)
    {
        printf("myshell$");
        fflush(stdin);
        fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
        make_toks(buff,args);
        if(strcmp(args[0],"list")==0)
            list(args[2],args[1][0]);
        else
        {
            pid = fork();
            if(pid>0)
                wait();
            else
            {
               if(execvp(args[0],args)==-1)
                    printf("Bad command.\n");
            }
        }
    }
    return 0;
}
```

Q.2 Write the simulation program for Round Robin scheduling for given time quantum. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give the Gantt chart, turnaround time and waiting time for each process. Also display the average turnaround time and average waiting time.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct process_info
```

```
{
    char pname[20];
    int at,bt,ct,bt1;
    struct process_info *next;
}NODE;
int n,ts;
NODE *first,*last;
void accept_info()
{
    NODE *p;
    int i;
    printf("Enter no.of process:");
 scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        p = (NODE*)malloc(sizeof(NODE));
        printf("Enter process name:");
        scanf("%s",p->pname);
        printf("Enter arrival time:");
        scanf("%d",&p->at);
        printf("Enter first CPU burst time:");
        scanf("%d",&p->bt);
        p->bt1 = p->bt;

        p->next = NULL;
        if(first==NULL)
            first=p;
        else
            last->next=p;
        last = p;
    }
    printf("Enter time slice:");
    scanf("%d",&ts);
}
void print_output()
{
    NODE *p;
    float avg_tat=0,avg_wt=0;
    printf("pname\tat\tbt\tct\ttat\twt\n");
    p = first;
    while(p!=NULL)
    {
        int tat = p->ct-p->at;
        int wt = tat-p->bt;

        avg_tat+=tat;
        avg_wt+=wt;
        printf("%s\t%d\t%d\t%d\t%d\t%d\n",
                p->pname,p->at,p->bt,p->ct,tat,wt);

        p=p->next;
    }
    printf("Avg TAT=%f\tAvg WT=%f\n",
            avg_tat/n,avg_wt/n);
}
void print_input()
{
    NODE *p;
    p = first;
    printf("pname\tat\tbt\n");
 while(p!=NULL)
    {
        printf("%s\t%d\t%d\n",
                p->pname,p->at,p->bt1);
        p = p->next;
    }
}
void sort()
{
    NODE *p,*q;
    int t;
    char name[20];
    p = first;
    while(p->next!=NULL)
    {
        q=p->next;
        while(q!=NULL)
        {
            if(p->at > q->at)
            {
                strcpy(name,p->pname);
                strcpy(p->pname,q->pname);
                strcpy(q->pname,name);
                t = p->at;
                p->at = q->at;
                q->at = t;

                t = p->bt;
                p->bt = q->bt;
                q->bt = t;
                t = p->ct;
                p->ct = q->ct;
                q->ct = t;
                t = p->bt1;
                p->bt1 = q->bt1;
                q->bt1 = t;
            }
            q=q->next;
        }
        p=p->next;
    }
}
int time;
int is_arrived()
{
    NODE *p;
    p = first;
    while(p!=NULL)
    {
        if(p->at<=time && p->bt1!=0)
            return 1;
        p=p->next;
    }
    return 0;
}
NODE * delq()
{
    NODE *t;
    t = first;
    first = first->next;
    t->next=NULL;
    return t;
}
void addq(NODE *t)
{
    last->next = t;
    last = t;
}
struct gantt_chart
{
    int start;
    char pname[30];
    int end;
}s[100],s1[100];
int k;
void rr()
{
    int prev=0,n1=0;
    NODE *p;
    while(n1!=n)
    {
        if(!is_arrived())
        {
            time++;
            s[k].start = prev;
            strcpy(s[k].pname,"*");
            s[k].end = time;
            k++;
            prev=time;
        }
        else
        {
            p = first;
            while(1)
            {
                if(p->at<=time && p->bt1!=0)
                     break;
                p = delq();
                addq(p);
                p = first;
            }
            if(p->bt1<=ts)
            {
                time+=p->bt1;
                p->bt1=0;
            }
            else
            {
                time+=ts;
                p->bt1-=ts;
            }
            p->ct = time;
            s[k].start = prev;
            strcpy(s[k].pname,p->pname);
            s[k].end = time;
            k++;
            prev = time;

            if(p->bt1==0) n1++;
            p = delq();
            addq(p);
        }
        print_input();
    }
}
void print_gantt_chart()
{
    int i,j,m;
    s1[0] = s[0];
    for(i=1,j=0;i<k;i++)
    {
        if(strcmp(s[i].pname,s1[j].pname)==0)
            s1[j].end = s[i].end;
        else
            s1[++j] = s[i];
    }
    printf("%d",s1[0].start);
    for(i=0;i<=j;i++)
    {
        m = (s1[i].end - s1[i].start);
        for(k=0;k<m/2;k++)
            printf("-");
        printf("%s",s1[i].pname);
        for(k=0;k<(m+1)/2;k++)
  printf("-");
        printf("%d",s1[i].end);
    }
}
int main()
{
    accept_info();
    sort();
    rr();
    print_output();
    print_gantt_chart();
        return 0;
}
```

---

slip 20

Q1.Write a C program to implement the shell which displays the command prompt “myshell$”. It accepts the command, tokenize the command line and execute it by creating the child process. Also implement the additional command ‘typeline’ as
typeline +n filename :- To print first n lines in the file.
typeline -a filename :- To print all lines in the file.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <dirent.h>
#include <unistd.h>
int make_toks(char *s, char *tok[])
```

```
{
        int i = 0;
    char *p;
    p = strtok(s, " ");
while(p != NULL)
{
       tok[i++] = p;
    p = strtok(NULL, " ");
}
tok[i] = NULL;
return i;
}
void typeline(char *op, char *fn)
{
        int fh, i, j, n;
    char c;
fh = open(fn, O_RDONLY);
if(fh == -1)
{
printf("File %s not found.\n", fn);
return;
}
if(strcmp(op, "a") == 0)
{
    while(read(fh, &c, 1) > 0)
            printf("%c", c);
        close(fh);
    return;
}
n = atoi(op);
if(n > 0)
{
        i = 0;
        while(read(fh, &c, 1) > 0)
    {
            printf("%c", c);
  if(c == '\n')
                i++;
        if(i == n)
            break;
}
}
if(n < 0)
{
        i = 0;
        while(read(fh, &c, 1) > 0)
            {
            if(c == '\n') i++;
        }
        lseek(fh, 0, SEEK_SET);
j = 0;
while(read(fh, &c, 1) > 0)
{
    if(c == '\n')
            j++;
    if(j == i+n+1)
            break;
}
while(read(fh, &c, 1) > 0)
{
        printf("%c", c);
}
}
close(fh);
}
int main()
{
        char buff[80], *args[10];
        while(1)
    {
            printf ("\n");
        printf("\nmyshell$ ");
        fgets(buff, 80, stdin);
            buff[strlen(buff)-1] = '\0';
            int n = make_toks(buff, args);
        switch (n)
        {
                case 1: if(strcmp(args[0], "exit") == 0)
exit(1);
if (!fork())
execlp (args [0], args[0], NULL);
break;
            case 2:
if (!fork ())
execlp (args [0], args[0], args[1], NULL);
break;
                case 3: if (strcmp(args[0], "typeline") == 0)
                        typeline (args[1], args[2]);
else
{
        if (!fork ())
execlp (args [0], args[0], args[1], args[2], NULL);
}
break;
case 4:
if (!fork ())
execlp (args [0], args [0], args [1], args [2], args [3], NULL);
break;
}
}
return 0;
}
```

Q.2 Write a C program to simulate Non-preemptive Shortest Job First (SJF) – scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1;
 struct process_info *next;
}NODE;
```

```
int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);

 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));

  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);


  p->bt1 = p->bt;
  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tct\ttat\twt\n");

 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->ct,tat,wt);

  p=p->next;
 }
 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;

 p = first;

 printf("pname\tat\tbt\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\n",
   p->pname,p->at,p->bt1);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;

    t = p->ct;
    p->ct = q->ct;
    q->ct = t;

    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;

   }

   q=q->next;
  }

  p=p->next;
 }
```

```
}

int time;

NODE * get_sjf()
{
 NODE *p,*min_p=NULL;
 int min=9999;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0 &&
   p->bt1<min)
  {
   min = p->bt1;
   min_p = p;
  }
  p=p->next;
 }

 return min_p;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void sjfnp()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_sjf();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;

   prev = time;
   k++;
  }
  else
  {
   time+=p->bt1;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1 = 0;

   n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}

int main()
{
 accept_info();
 sort();
 sjfnp();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 21

Q.1 Write a C Program to create a child process using fork (), display parent and child process id. Child process will display the message “I am Child Process” and the parent process should display “I am Parent Process”

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
     int pid = fork();
    if (pid > 0) {
        printf("I am Parent process\n");
        printf("ID : %d\n\n", getpid());
    }
    else if (pid == 0) {
        printf("I am Child process\n");
        printf("ID: %d\n", getpid());
  }
    else {
        printf("Failed to create child process");
    }
    return 0;
}
```

Q.2 Write a C program to simulate Preemptive Priority scheduling. The arrival time and first CPU-burst and priority for different n number of processes should be input to the algorithm. Assume the fixed IO waiting time (2 units). The next CPU-burst should be generated randomly. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

#define MAX_PROCESSES 10
typedef struct {
    int arrival_time;
    int burst_time;
    int priority;
    int waiting_time;
    int turnaround_time;
} Process;

void generate_next_burst(Process *p) {
    p->burst_time = rand() % 10 + 1;
}

void preemptive_priority_scheduling(Process p[], int n) {
    int current_time = 0;
    int i, j;
    Process temp;


for (i = 0; i < n; i++) {
for (j = i + 1; j < n; j++) {
if (p[i].arrival_time > p[j].arrival_time || (p[i].arrival_time == p[j].arrival_time && p[i].priority < p[j].priority)) {
     temp = p[i];
     p[i] = p[j];
     p[j] = temp;
    }
   }
}

    printf("Gantt Chart:\n");
    for (i = 0; i < n; i++) {
        printf("P%d | ", i);
        while (p[i].burst_time > 0) {
            current_time++;
            p[i].burst_time--;
            printf("%d ", current_time);
            if (p[i].burst_time == 0) {
                p[i].turnaround_time = current_time - p[i].arrival_time;
                p[i].waiting_time = p[i].turnaround_time - p[i].burst_time;
                printf("| ");
            }
            for (j = i + 1; j < n; j++) {
                if (p[j].arrival_time <= current_time && p[j].priority > p[i].priority) {
                    temp = p[i];
                    p[i] = p[j];
                    p[j] = temp;
                    i--;
                    break;
                }
            }
        }
        printf("\n");
    }

    printf("Process\tTurnaround Time\tWaiting Time\n");
    for (i = 0; i < n; i++) {
        printf("P%d\t%d\t\t%d\n", i, p[i].turnaround_time, p[i].waiting_time);
    }

    int total_turnaround_time = 0, total_waiting_time = 0;
    for (i = 0; i < n; i++) {
        total_turnaround_time += p[i].turnaround_time;
        total_waiting_time += p[i].waiting_time;
    }

    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
    printf("Average Waiting Time: %.2f\n", (float)total_waiting_time / n);
}

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

    Process p[n];
    for (int i = 0; i < n; i++) {
        printf("Enter arrival time, burst time, and priority for process P%d: ", i);
        scanf("%d %d %d", &p[i].arrival_time, &p[i].burst_time, &p[i].priority);
    }

    srand(time(NULL));

    preemptive_priority_scheduling(p, n);
    return 0;
}
```

---

slip 22

Q.1 Write a C program that demonstrates the use of nice() system call. After a child   Process is started using fork (), assign higher priority to the child using nice () system call.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
pid_t pid;
pid = fork();
if (pid == 0)
{
printf("\nI am child process, id=%d\n",getpid());
printf("\nPriority :%d,id=%d\n",nice (-7),getpid());
}
else
{
printf("\nI am parent process, id=%d\n",getpid());
nice(1);
printf("\nPriority :%d,id=%d\n",nice (15),getpid());
}
return 0;
}
```

Q.2 Write the program to simulate Non preemptive priority scheduling. The arrival time and first CPU-burst of different jobs should be input to the system. Accept no. of Processes, arrival time and burst time. The output should give Gantt chart, turnaround time and waiting time for each process. Also find the average waiting time and turnaround time.

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

typedef struct process_info
{
 char pname[20];
 int at,bt,ct,bt1,p;
 struct process_info *next;
}NODE;

int n;
NODE *first,*last;

void accept_info()
{
 NODE *p;
 int i;

 printf("Enter no.of process:");
 scanf("%d",&n);

 for(i=0;i<n;i++)
 {
  p = (NODE*)malloc(sizeof(NODE));

  printf("Enter process name:");
  scanf("%s",p->pname);

  printf("Enter arrival time:");
  scanf("%d",&p->at);

  printf("Enter first CPU burst time:");
  scanf("%d",&p->bt);

  printf("Enter priority:");
  scanf("%d",&p->p);

  p->bt1 = p->bt;

  p->next = NULL;

  if(first==NULL)
   first=p;
  else
   last->next=p;

  last = p;
 }
}

void print_output()
{
 NODE *p;
 float avg_tat=0,avg_wt=0;

 printf("pname\tat\tbt\tp\ttct\ttat\twt\n");

 p = first;
 while(p!=NULL)
 {
  int tat = p->ct-p->at;
  int wt = tat-p->bt;

  avg_tat+=tat;
  avg_wt+=wt;

  printf("%s\t%d\t%d\t%d\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt,p->p,p->ct,tat,wt);

  p=p->next;
 }

 printf("Avg TAT=%f\tAvg WT=%f\n",
   avg_tat/n,avg_wt/n);
}

void print_input()
{
 NODE *p;
 p = first;

 printf("pname\tat\tbt\tp\n");
 while(p!=NULL)
 {
  printf("%s\t%d\t%d\t%d\n",
   p->pname,p->at,p->bt1,p->p);
  p = p->next;
 }
}

void sort()
{
 NODE *p,*q;
 int t;
 char name[20];

 p = first;
 while(p->next!=NULL)
 {
  q=p->next;
  while(q!=NULL)
  {
   if(p->at > q->at)
   {
    strcpy(name,p->pname);
    strcpy(p->pname,q->pname);
    strcpy(q->pname,name);

    t = p->at;
    p->at = q->at;
    q->at = t;

    t = p->bt;
    p->bt = q->bt;
    q->bt = t;

    t = p->ct;
    p->ct = q->ct;
    q->ct = t;

    t = p->bt1;
    p->bt1 = q->bt1;
    q->bt1 = t;

    t = p->p;
    p->p = q->p;
    q->p = t;
   }

   q=q->next;
  }

  p=p->next;
 }
}

int time;
```

```
NODE * get_p()
{
 NODE *p,*min_p=NULL;
 int min=9999;

 p = first;
 while(p!=NULL)
 {
  if(p->at<=time && p->bt1!=0 &&
   p->p<min)
  {
   min = p->p;
   min_p = p;
  }
  p=p->next;
 }

 return min_p;
}

struct gantt_chart
{
 int start;
 char pname[30];
 int end;
}s[100],s1[100];

int k;

void pnp()
{
 int prev=0,n1=0;
 NODE *p;

 while(n1!=n)
 {
  p = get_p();

  if(p==NULL)
  {
   time++;
   s[k].start = prev;
   strcpy(s[k].pname,"*");
   s[k].end = time;

   prev = time;
   k++;
  }
  else
  {
   time+=p->bt1;
   s[k].start = prev;
   strcpy(s[k].pname, p->pname);
   s[k].end = time;

   prev = time;
   k++;

   p->ct = time;
   p->bt1 = 0;
   n1++;
  }

  print_input();
  sort();
 }
}

void print_gantt_chart()
{
 int i,j,m;

 s1[0] = s[0];

 for(i=1,j=0;i<k;i++)
 {
  if(strcmp(s[i].pname,s1[j].pname)==0)
   s1[j].end = s[i].end;
  else
   s1[++j] = s[i];
 }

 printf("%d",s1[0].start);
 for(i=0;i<=j;i++)
 {
  m = (s1[i].end - s1[i].start);

  for(k=0;k<m/2;k++)
   printf("-");

  printf("%s",s1[i].pname);

  for(k=0;k<(m+1)/2;k++)
   printf("-");

  printf("%d",s1[i].end);
 }
}

int main()
{
 accept_info();
 sort();
 pnp();
 print_output();
 print_gantt_chart();

 return 0;
}
```

---

slip 23

Q.1 Write a C program to illustrate the concept of orphan process. Parent process creates a child and terminates before child has finished its task. So child process becomes orphan process. (Use fork(), sleep(), getpid(), getppid()).

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
 int pid;
 pid=getpid();
 printf("Current Process ID is : %d\n",pid);
 printf("\n[Forking Child Process ... ] \n");
 pid=fork();
 if(pid < 0)
 {
  printf("\nProcess can not be created ");
 }
 else
 {
        if(pid==0)
  {
   printf("\nChild Process is Sleeping ...");
   sleep(5);
   printf("\nOrphan Child's Parent ID : %d",getppid());
  }

  else
  { /* Parent Process */
   printf("\nParent Process Completed ...");
  }
 }
 return 0;
}
```

Q2.Write the simulation program for demand paging and show the page scheduling and total number of page faults according the Optimal page replacement algorithm. Assume the memory of n frames.
Reference String : 7, 5, 4, 8, 5, 7, 2, 3, 1, 3, 5, 9, 4, 6, 2

```
#include<stdio.h>
int main()
{
  int no_of_frames, no_of_pages, frames[10], pages[30], temp[10], flag1,
flag2, flag3, i, j, k, pos, max, faults = 0;
    printf("Enter number of frames: ");
    scanf("%d", &no_of_frames);

    printf("Enter number of pages: ");
    scanf("%d", &no_of_pages);

    printf("Enter page reference string: ");

    for(i = 0; i < no_of_pages; ++i){
        scanf("%d", &pages[i]);
    }

    for(i = 0; i < no_of_frames; ++i){
        frames[i] = -1;
    }

    for(i = 0; i < no_of_pages; ++i){
        flag1 = flag2 = 0;
        for(j = 0; j < no_of_frames; ++j){
            if(frames[j] == pages[i]){
                   flag1 = flag2 = 1;
                   break;
               }
        }

        if(flag1 == 0){
            for(j = 0; j < no_of_frames; ++j){
                if(frames[j] == -1){
                    faults++;
                    frames[j] = pages[i];
                    flag2 = 1;
                    break;
                }
            }
        }

        if(flag2 == 0){
         flag3 =0;

            for(j = 0; j < no_of_frames; ++j){
             temp[j] = -1;

             for(k = i + 1; k < no_of_pages; ++k){
             if(frames[j] == pages[k]){
             temp[j] = k;
             break;
             }
             }
            }

            for(j = 0; j < no_of_frames; ++j){
             if(temp[j] == -1){
             pos = j;
             flag3 = 1;
              break;
             }
            }

            if(flag3 ==0){
             max = temp[0];
             pos = 0;

             for(j = 1; j < no_of_frames; ++j){
             if(temp[j] > max){
             max = temp[j];
             pos = j;
             }
             }
            }
frames[pos] = pages[i];
faults++;
        }
           printf("\n");
                for(j = 0; j < no_of_frames; ++j){
            printf("%d\t", frames[j]);
        }
    }
    printf("\n\nTotal Page Faults = %d", faults);

    return 0;
}
```

---

slip 24

Q.1 Write a C program to accept n integers to be sorted. Main function creates child process using fork system call. Parent process sorts the integers using bubble sort and waits for child process using wait system call. Child process sorts the integers using insertion sort.

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

void bubblesort(int arr[30],int n)
{
	int i,j,temp;
	for(i=0;i<n;i++)
	{
		for(j=0;j<n-1;j++)
		{
			if(arr[j]>arr[j+1])
			{
				temp=arr[j];
				arr[j]=arr[j+1];
				arr[j+1]=temp;
			}
		}
	}
}

void insertionsort(int arr[30], int n)
{
    int i, j, temp;
    for (i = 1; i < n; i++) {
        temp = arr[i];
        j = i - 1;

        while(j>=0 && temp <= arr[j])
        {
            arr[j+1] = arr[j];
            j = j-1;
        }
        arr[j+1] = temp;
    }
}
void fork1()
{
 	int arr[25],arr1[25],n,i,status;
 	printf("\nEnter the no of values in array :");
 	scanf("%d",&n);
 	printf("\nEnter the array elements :");
 	for(i=0;i<n;i++)
    		scanf("%d",&arr[i]);
  int pid=fork();
   if(pid==0)
    {
       sleep(10);
       printf("\nchild process\n");
       printf("child process id=%d\n",getpid());
       insertionsort(arr,n);
        printf("\nElements Sorted Using insertionsort:");
      printf("\n");
        for(i=0;i<n;i++)
      printf("%d,",arr[i]);
      printf("\b");
      printf("\nparent process id=%d\n",getppid());
      system("ps -x");
       }
      else
       {
      printf("\nparent process\n");
      printf("\nparent process id=%d\n",getppid());
   bubblesort(arr,n);
  printf("Elements Sorted Using bubblesort:");
      printf("\n");
       for(i=0;i<n;i++)
      printf("%d,",arr[i]);
      printf("\n\n\n");
      }
 }
 int main()
 {
   fork1();
   return 0;
 }
```

Q2.Write a programto implement the toy shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
count c filename :- To print number of characters in the file.
count w filename :- To print number of words in the file.
count l filename :- To print number of lines in the file

```
#include <sys/types.h>
#include<unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void make_toks(char *s, char *tok[])
{
        int i=0;
    char *p;
    p = strtok(s," ");
while(p!=NULL)
{  tok[i++]=p;
        p=strtok(NULL," ");
}
tok[i]=NULL;
}
```

```
void count(char *fn, char op)
{ int fh,cc=0,wc=0,lc=0;
        char c;
fh = open(fn,O_RDONLY);
if(fh==-1) {
 printf("File %s not found.\n",fn);
 return;
}
while(read(fh,&c,1)>0)
{ if(c==' ')
    wc++;
        else if(c=='\n')
 {
 wc++;
 lc++;
 }  cc++;
} close(fh);
switch(op)
{
        case 'c':
 printf("No.of characters:%d\n",cc-1);
 break;
        case 'w':
 printf("No.of words:%d\n",wc);
 break;
    case 'l':
 printf("No.of lines:%d\n",lc+1);
  break;
}
} int main()
{
    char buff[80],*args[10];
    int pid;
    while(1) { printf("myshell$ ");
        fflush(stdin);
            fgets(buff,80,stdin);
        buff[strlen(buff)-1]='\0';
            make_toks(buff,args);
if(strcmp(args[0],"count")==0)
 count(args[2],args[1][0]);
else
 {  pid = fork();
     if(pid>0)
         wait();
         else
 {
 if(execvp(args[0],args)==-1)
 printf("Bad command.\n");
 }
 }
    }
        return 0;
}
```

---

slip 25

Q.1 Write a C program that accepts an integer array. Main function forks child process. Parent process sorts an integer array and passes the sorted array to child process through the command line arguments of execve() system call. The child process uses execve() system call to load new program that uses this sorted array for performing the binary search to search the particular item in the array.

parent.c

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

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

int main() {
    int arr[] = {5, 2, 8, 3, 1, 6, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) { // Child process
        execl("./child", "child", (char *)NULL);
        perror("execl");
        exit(1);
    } else { // Parent process
        sort_array(arr, n);

        char str[10];
        char *argv[] = {"child"};
        for (int i = 0; i < n; i++) {
            sprintf(str, "%d", arr[i]);
            argv[i + 1] = str;
        }
        argv[n + 1] = NULL;

        wait(NULL);

        printf("Parent process: Sorted array is ");
        for (int i = 0; i < n; i++) {
            printf("%d ", arr[i]);
        }
        printf("\n");
    }

    return 0;
}
```

child.c

```
#include <stdio.h>
#include <stdlib.h>
int binary_search(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = (low + high) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1;
}

int main(int argc, char *argv[]) {
    int n = argc - 1;
    int arr[n];
    for (int i = 0; i < n; i++) {
        arr[i] = atoi(argv[i + 1]);
    }

    int target;
    printf("Enter the target element: ");
    scanf("%d", &target);

    int result = binary_search(arr, n, target);
    if (result != -1) {
        printf("Element found at index %d\n", result);
    } else {
        printf("Element not found\n");
    }

    return 0;
}
```

Q.2 Write a programto implement the shell. It should display the command prompt “myshell$”. Tokenize the command line and execute the given command by creating the child process. Additionally it should interpret the following commands.
myshell$ search f filename pattern :- To display first occurrence of pattern in the file.

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

```
#include <sys/wait.h>

#define MAX_LINE 1024
#define MAX_TOKENS 64

void tokenize(char *line, char *tokens[]) {
    int i = 0;
    char *token = strtok(line, " \t\n");
    while (token != NULL && i < MAX_TOKENS) {
        tokens[i++] = token;
        token = strtok(NULL, " \t\n");
    }
    tokens[i] = NULL;
}

void execute_command(char *tokens[]) {
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork");
        exit(1);
    }

    if (pid == 0) { // Child process
        execvp(tokens[0], tokens);
        perror("execvp");
        exit(1);
    } else { // Parent process
        wait(NULL);
    }
}

void search_command(char *filename, char *pattern) {
    FILE *fp = fopen(filename, "r");
    if (fp == NULL) {
        perror("fopen");
        return;
    }

    char line[MAX_LINE];
    while (fgets(line, MAX_LINE, fp) != NULL) {
        if (strstr(line, pattern) != NULL) {
            printf("%s", line);
            break;
        }
    }

    fclose(fp);
}

int main() {
    char line[MAX_LINE];
    char *tokens[MAX_TOKENS];

    while (1) {
        printf("myshell$ ");
        fflush(stdout);
        fgets(line, MAX_LINE, stdin);

        tokenize(line, tokens);
        if (strcmp(tokens[0], "search") == 0) {
            if (tokens[1] != NULL && tokens[2] != NULL) {
                search_command(tokens[1], tokens[2]);
            } else {
                printf("Usage: search <filename> <pattern>\n");
            }
        } else {
            execute_command(tokens);
        }
    }

    return 0;
}
```