Contents

1 Week 1: Demonstration of fork() System Call

Program

```
//SINGLE FORK

//HEADER FILES
#include <stdio.h>
#include <unistd.h>

int main() {
    //CALLING FORK TO CREATE A CHILD PROCESS
    fork();
    printf("LINUX\n");
    return 0;
}
```

Output

```
LINUX
LINUX
```

Program

```
//MULTI TIME FORK

//HEADER FILES
#include <stdio.h>
#include <unistd.h>

int main() {
    //CALLING FORK TO CREATE A CHILD PROCESS
    fork();
    printf("LINUX\n");
    //CALLING FORK TO CREATE A CHILD PROCESS
    fork();
    printf("UNIX\n");
    //CALLING FORK TO CREATE A CHILD PROCESS
    fork();
    printf("RED HAT\n");
    return 0;
}
```

Output

```
LINUX
LINUX
UNIX
UNIX
RED HAT
UNIX
RED HAT
RED HAT
UNIX
RED HAT
RED HAT
RED HAT
RED HAT
RED HAT
```

2 Week 2: Parent Process Computes the Sum Of Even and Child Process Computes the sum of Odd Numbers using fork

Program

```c
// parent -> sum of even
// child -> sum of odd

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

#define max 20

int main() {
    pid_t pid;
    int a[max], n, sum = 0, i, status;

    printf("Enter the no of terms in the array: ");
    scanf("%d", &n);

    printf("Enter values in the array: ");
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }

    pid = fork();
    // wait(&status);

    if (pid == 0) {
        // child process
        for (i = 0; i < n; i++) {
            if (a[i] % 2 != 0) {
                sum = sum + a[i];
            }
        }
        printf("Sum of odd no. = %d\n", sum);
        exit(0);
    } else {
        // parent process
        for (i = 0; i < n; i++) {
            if (a[i] % 2 == 0) {
                sum = sum + a[i];
            }
        }
        printf("Sum of even nos = %d\n", sum);
    }
    return 0;
}
```

Output

3 Week 3: Demonstration of wait() System Call

Program

```c
// wait() syscall

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

int main() {
    int status;
    pid_t pid;
    pid = fork();
    if (pid == 0) {
        // child process
        printf("I m Child\n");
        exit(0);
    } else {
        // parent process
        wait(&status);
        printf("I'm Parent\n");
        printf("The Child PID = %d\n", pid);
    }
    return 0;
}
```

Output

4 Week 4: Implementation of Orphan Process & Zombie Process

Program

```c
// orphan process
// process inherited by the init

#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid;
    pid = fork();

    if (pid == 0) {
        // child
        sleep(6); // wait and let the parent die
        printf("I'm Child. My PID = %d And PPID = %d\n", getpid(), getppid());
    } else {
        // parent
        printf("I'm Parent. My Child PID = %d And my PID = %d\n", pid, getpid());
    }
    printf("Terminating PID = %d\n", getpid());
    return 0;
}
```

Output

Program

```c
// zombie process

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

int main() {
    pid_t pid;
    pid = fork();

    if (pid != 0) {
        //child
        while (1)
            sleep(50);
    } else {
        //parent
        exit(0);
    }
}
```

Output

5 Week 5: Implementation of PIPE

Program

```
// pipe

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

#define SIZE 100

int main() {
    pid_t pid;
    char arr[SIZE], str[SIZE];
    int fd[2];    // store file descriptors
    int nbr, nbw; // no of bytes read and write

    //CREATING A PIPE
    pipe(fd);

    pid = fork();

    if (pid == 0) {
        // child
        printf("Enter a string: ");
        fgets(str, SIZE, stdin);
        nbw = write(fd[1], str, strlen(str));
        printf("Child wrote %d bytes\n", nbw);
        exit(0);
    } else {
        // parent
        nbr = read(fd[0], arr, sizeof(arr));
        arr[nbr] = '\0';
        printf("Parent read %d bytes : %s\n", nbr, arr);
    }
    return 0;
}
```

Output

6 Week 6: Implementation of FIFO

Program (Writer)

```c
// fifo
// writer

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

#define SIZE 100

int main() {
    int fd;
    int nbw; // no of bytes written
    char str[SIZE];

    // make fifo -> myfifo
    mknod("myfifo", S_IFIFO | 0666, 0);

    printf("Writing for reader Process:\n");

    // open the fifo for write operation
    // O_WRONLY -> write only operation
    fd = open("myfifo", O_WRONLY);

    while (fgets(str, SIZE, stdin)) {
        nbw = write(fd, str, strlen(str));
        printf("Writer process write %d bytes: %s\n", nbw, str);
    }
    return 0;
}
```

Program (Reader)

```c
// fifo
// reader

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

#define SIZE 100

int main() {
    int fd;
    int nbr; // no of bytes read
    char arr[SIZE];

    // make fifo -> myfifo
    mknod("myfifo", S_IFIFO | 0666, 0);

    // open a fifo for read
    // O_RDONLY -> read only permission
    fd = open("myfifo", O_RDONLY);
```

```
    printf("If you got a writer process then type some data\n");

    do {
        nbr = read(fd, arr, sizeof(arr));
        arr[nbr] = '\0';
        printf("Reader process read %d bytes: %s\n", nbr, arr);
    } while (nbr > 0);

    return 0;
}
```

Output

7 Week 7: Implementation of Message Queue

Program (Writer)

```
// message queue
// writer

#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>

#define SIZE 100

struct msgbuf {
    long mtype;
    char mtext[SIZE];
}svarname;

int main() {
    key_t key;
    int msgid, c;

    // create a key
    key = ftok("progfile", 'A');

    // get a message queue
    msgid = msgget(key, 0666 | IPC_CREAT);

    //
    svarname.mtype = 1;

    printf("Enter a string : ");
    fgets(svarname.mtext, SIZE, stdin);

    // sending msg to message queue
    // msgid, msgp, msg_size, flags
    c = msgsnd(msgid, &svarname, strlen(svarname.mtext), 0);

    printf("Sender wrote the text :\t %s \n", svarname.mtext);

    return 0;
}
```

Program (Reader)

```
// message queue
// reader

#include <stdio.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>

#define SIZE 100

struct msgbuf {
    long mtype;
    char mtext[SIZE];
```

```
} svarname;

int main() {
    key_t key;
    int msgid, c;

    // create a key
    key = ftok("progfile", 'A');

    // get a message queue
    msgid = msgget(key, 0666 | IPC_CREAT);

    // receive a message from message queue
    // msgid, msgp, msg_size, msg_type, flags
    msgrcv(msgid, &svarname, sizeof(svarname), 1, 0);

    printf("Data Received isL %s\n", svarname.mtext);

    // message queue control operation
    msgctl(msgid, IPC_RMID, NULL);

    return 0;
}
```

Output

8 Week 8: Implementation of Shared Memory

Program (Writer)

```c
// shared memory
// writer

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

#define SIZE 100

int main() {
    key_t key;
    int shmid; // shared memory id
    char *ptr; // pointer to the shared memory location

    // generate a unique key
    key = ftok("shmfile", 'A');

    // get shared memory segment
    // pass key, size, flag
    shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    // attach shared memory segment
    ptr = shmat(shmid, (void *)0, 0);

    printf("Input Data : ");
    fgets(ptr, SIZE, stdin);

    // detach shared memory segment
    shmdt(ptr);

    return 0;
}
```

Program (Reader)

```c
// shared memory
// reader

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

int main() {
    key_t key;
    int shmid; // shared memory id
    char *ptr; // pointer to shared memory location

    // generate a unique key
    key = ftok("shmfile", 'A');

    // get shared memory segment
    // pass key, size, flag
    shmid = shmget(key, 1024, 0666 | IPC_CREAT);
```

```
    // attach shared memory segment
    ptr = shmat(shmid, (void *)0, 0);

    printf("The Data stored : %s\n", ptr);

    // detach shared memory segment
    shmdt(ptr);

    // shared memory control operation
    // remove id
    shmctl(shmid, IPC_RMID, NULL);

    return (0);
}
```

Output

Program (Combined)

```
// shared memory
// both reader and writer

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

#define SIZE 100

int main() {
    key_t key;
    int shmid; // shared memory id
    void *ptr; // pointer to the shared memory location

    // generate a unique key
    key = ftok("srfile", 'A');

    // get shared memory segment
    shmid = shmget(key, 1024, 0666 | IPC_CREAT);

    // attach shared memory segment
    ptr = shmat(shmid, (void *)0, 0);

    printf("\nInput Data:");
    fgets(ptr, SIZE, stdin);

    printf("The Data stored : %s\n", (char *)ptr);

    // detach the shared memory segment
    shmdt(ptr);

    // remove id of the shared memory
    shmctl(shmid, IPC_RMID, NULL);

    return (0);
}
```

9 Week 9: First Come First Served Scheduling Algorithm

Program (Pointers)

```
// fcfs

#include <malloc.h>
#include <stdio.h>
#include <string.h>

typedef struct node {
    char pname[3];
    int burst;
    int arrival;
    struct node *next;
} node;

typedef struct queue {
    node *front;
    node *rear;
} queue;

void insert(queue *q) {
    node *p;
    int bt;       // burst time
    int at;       // arrival time
    char str[3]; // process name

    p = (node *)malloc(sizeof(node));

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

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

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

    p->next = NULL;

    if (q->front == NULL) {
        q->front = p;
        q->rear = p;
    } else {
        q->rear->next = p;
        q->rear = p;
    }
}

void display(queue *q, int n) {
    node *temp = q->front;
    int wttime = 0; // wait time
    int ct = 0;     // completion time
    float turn = 0.0;

    // if queue is not empty
    if (q->front != NULL) {

        // Make Gantt chart
```

```
        printf("\n\n");
        while (temp != NULL) {
            printf("\t%s\t", temp->pname);
            temp = temp->next;
        }

        temp = q->front;
        printf("\n");
        while (temp != NULL) {
            printf("\t(%d)\t", temp->burst);
            temp = temp->next;
        }

        temp = q->front;
        printf("\n(0)\t-");
        while (temp != NULL) {
            wttime += ct;              // calculating total wait time
            turn += ct + temp->burst; // calculating turnaround time
            ct = ct + temp->burst;
            printf("-\t(%d)\t-", ct);
            temp = temp->next;
        }

        printf("\n\n");
        printf("Average wait time = %d\n", wttime / n);
        printf("Turn around time = %f\n", turn / n);
    }
}

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

    queue *q = (queue *)malloc(sizeof(queue));

    for (i = 0; i < n; i++)
        insert(q);

    printf("Executing processes: \n");
    display(q, n);

    return 0;
}
```

Output

```
Enter number of processes: 3
Enter the process name : p1
Enter Burst time : 24
Enter arrival time : 0
Enter the process name : p2
Enter Burst time : 3
Enter arrival time : 0
Enter the process name : p3
Enter Burst time : 3
Enter arrival time : 0
Executing processes:
```

```
        p1              p2              p3
        (24)            (3)             (3)
(0)     --      (24)    --      (27)    --      (30)    -

Average wait time = 17
Turn around time = 27.000000
```

Program (Array)

```c
// fcfs

#include <malloc.h>
#include <stdio.h>
#include <string.h>

#define SIZE 100

int main() {
    char p[SIZE][5]; // process name
    int pt[SIZE];    // process time

    int c = 0, i, j, n;
    float at = 0.0, turn = 0.0;

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

    for (i = 0; i < n; i++) {
        printf("Enter process %d name: ", i + 1);
        scanf("%s", &p[i][0]);

        printf("Enter process time: ");
        scanf("%d", &pt[i]);
    }

    // Make Gantt chart

    printf("\n");
    for (i = 0; i < n; i++) {
        // print process name
        printf("\t%s\t", p[i]);
    }

    printf("\n");
    for (i = 0; i < n; i++) {
        // print process time
        printf("\t(%d)\t", pt[i]);
    }

    printf("\n0\t-");
    for (i = 0; i < n; i++) {
        at += c;
        turn += c + pt[i];
        c = c + pt[i];
        printf("-\t(%d)\t-", c);
    }

    printf("\n");
    printf("Average time: %f\n", at / n);
    printf("Turn around time: %f\n", turn / n);
    return 0;
```

```
}
```

Output

```
Enter number of processes: 3
Enter the process name : p1
Enter Burst time : 24
Enter arrival time : 0
Enter the process name : p2
Enter Burst time : 3
Enter arrival time : 0
Enter the process name : p3
Enter Burst time : 3
Enter arrival time : 0
Executing processes:


        p1              p2              p3
        (24)            (3)             (3)
(0)     --      (24)    --      (27)    --      (30)    -

Average wait time = 17
Turn around time = 27.000000
```

10 Week 10: Shortest Job First Scheduling Algorithm

Program (Pointers)

```
// sjf using pointers

#include <malloc.h>
#include <stdio.h>
#include <string.h>

typedef struct node {
    char name[3];
    int burst;
    struct node *next;
} node;

typedef struct queue {
    node *front;
    node *rear;
} queue;

void insert(queue *q) {
    node *p, *temp;

    p = (node *)malloc(sizeof(node));

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

    printf("Enter Burst time: ");
    scanf("%d", &(p->burst));
    p->next = NULL;

    if (q->front == NULL) {
        // first element
        q->front = p;
        q->rear = p;
    } else if (p->burst < q->front->burst) {
        // insert in front
        p->next = q->front;
        q->front = p;
    } else if (p->burst > q->rear->burst) {
        // insert at last
        q->rear->next = p;
        q->rear = p;
    } else {
        // insert in between
        temp = q->front;
        while (p->burst > (temp->next)->burst)
            temp = temp->next;
        p->next = temp->next;
        temp->next = p;
    }
}

void display(queue *q, int n) {
    node *temp = q->front;
    int c = 0;
    float turn = 0.0, wttime = 0.0;
    if (q->front != NULL) {
```

```
        // Make Gantt chart

        printf("\n\n");
        while (temp != NULL) {
            printf("\t%s\t", temp->name);
            temp = temp->next;
        }

        temp = q->front;
        printf("\n");
        while (temp != NULL) {
            printf("\t(%d)\t ", temp->burst);
            temp = temp->next;
        }

        temp = q->front;
        printf("\n(0)\t-");
        while (temp != NULL) {
            wttime += c;
            turn += c + temp->burst;
            c = c + temp->burst;
            printf("\t(%d)\t ", c);
            temp = temp->next;
        }
        printf("\n");
        printf("Average waiting time: %f\n", wttime / n);
        printf("Turn around time: %f\n", turn / n);
    }
}

int main() {
    int i, n;
    queue *q = (queue *)malloc(sizeof(queue));

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

    for (i = 0; i < n; i++)
        insert(q);

    printf("Executing processes: \n");
    display(q, n);
    return 0;
}
```

Output

```
Enter number of processes: 3
Enter the process name: p1
Enter Burst time: 24
Enter the process name: p2
Enter Burst time: 2
Enter the process name: p3
Enter Burst time: 3
Executing processes:


        p2              p3              p1
        (2)             (3)             (24)
(0)     -       (2)             (5)             (29)
```

```
Average waiting time: 2.333333
Turn around time: 12.000000
```

Program (Array)

```c
// sjf using arrays

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

#define SIZE 100

int main() {
    char p[SIZE][5]; // process names
    int pt[SIZE];    // process interval

    int c = 0, i, j, n, temp1;
    float bst = 0.0, turn = 0.0;

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

    for (i = 0; i < n; i++) {
        printf("Enter process %d name: ", i + 1);
        scanf("%s", &p[i][0]);
        printf("Enter process time: ");
        scanf("%d", &pt[i]);
    }

    // sorting according to the process time
    // using bubble sort
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (pt[i] > pt[j]) {
                // swap
                char temp[5];
                temp1 = pt[i];
                pt[i] = pt[j];
                pt[j] = temp1;
                strcpy(temp, p[i]);
                strcpy(p[i], p[j]);
                strcpy(p[j], temp);
            }
        }
    }

    // Make Gantt chart

    printf("\n\n");
    for (i = 0; i < n; i++) {
        printf("\t%s\t", p[i]);
    }

    printf("\n");
    for (i = 0; i < n; i++) {
        printf("\t(%d)\t", pt[i]);
    }

    printf("\n(0)\t-");
    for (i = 0; i < n; i++) {
        bst += c;
```

```
        turn += c + pt[i];
        c = c + pt[i];
        printf("-\t%d\t-", c);
    }

    printf("\n\n");
    printf("Average time: %f\n", bst / n);
    printf("Turn around time: %f\n", turn / n);

    return 0;
}
```

Output

```
Enter no of processes: 3
Enter process 1 name: p1
Enter process time: 24
Enter process 2 name: p2
Enter process time: 2
Enter process 3 name: p3
Enter process time: 3


        p2              p3              p1
        (2)             (3)             (24)
(0)     --      2       --      5       --      29      -

Average time: 2.333333
Turn around time: 12.000000
```

11 Week 11: Priority Scheduling Algorithm

Program (Pointers)

```
// priority scheduling

#include <malloc.h>
#include <stdio.h>
#include <string.h>

typedef struct node {
    char process[3];
    int burst;
    int priority;
    struct node *next;
} node;

typedef struct queue {
    node *front;
    node *rear;
} queue;

void insert(queue *q) {
    node *p, *temp;
    int b, pri;

    p = (node *)malloc(sizeof(node));

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

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

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

    /*
        inserting the new node so
        it is sorted according to
        priority
    */

    if (q->front == NULL) {
        // first element
        q->front = p;
        q->rear = p;
    } else if (p->priority < q->front->priority) {
        // at start
        p->next = q->front;
        q->front = p;
    } else if (p->priority > q->rear->priority) {
        // at end
        q->rear->next = p;
        q->rear = p;
    } else {
        // in between
        temp = q->front;
        while (p->priority > (temp->next)->priority)
```

```
            temp = temp->next;
        p->next = temp->next;
        temp->next = p;
    }
}
void display(queue *q, int n) {
    node *temp;
    int c = 0;
    float turn = 0.0, wttime = 0.0;

    if (q->front != NULL) {

        // Make Gantt chart

        temp = q->front;
        printf("\n\n");
        while (temp != NULL) {
            printf("\t%s\t", temp->process);
            temp = temp->next;
        }

        temp = q->front;
        printf("\n");
        while (temp != NULL) {
            printf("\t(%d)\t ", temp->burst);
            temp = temp->next;
        }

        temp = q->front;
        printf("\n(0)\t-");
        while (temp != NULL) {
            wttime += c;
            turn += c + temp->burst;
            c = c + temp->burst;
            printf("-\t(%d)\t-", c);
            temp = temp->next;
        }
        printf("\n\n");
        printf("Average wait time = %f\n", wttime / n);
        printf("Turn around time = %f\n", turn / n);
    }
}

int main() {
    int i, n;
    queue *q = (queue *)malloc(sizeof(queue));
    printf("Enter number of processes: ");
    scanf("%d", &n);

    for (i = 0; i < n; i++)
        insert(q);

    printf("Executing processes: \n");
    display(q, n);
    return 0;
}
```

Output

```
Enter number of processes: 3
Enter the process name: p1
```

```
Enter Burst time: 24
Enter Priority: 3
Enter the process name: p2
Enter Burst time: 3
Enter Priority: 1
Enter the process name: p3
Enter Burst time: 2
Enter Priority: 2
Executing processes:


        p2              p3              p1
        (3)             (2)             (24)
(0)     --      (3)     --      (5)     --      (29)    -

Average wait time = 2.666667
Turn around time = 12.333333
```

Program (Array)

```c
// priority scheduling

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

#define SIZE 100

int main() {
    char p[10][5]; // process name
    int pt[SIZE];  // process time
    int pr[SIZE];  // process priority

    int c = 0, i, j, n;
    char temp[5];
    float bst = 0.0, turn = 0.0;

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

    for (i = 0; i < n; i++) {
        printf("Enter process %d name: ", i + 1);
        scanf("%s", &p[i][0]);
        printf("Enter process time: ");
        scanf("%d", &pt[i]);
        printf("Enter the priority of process: ");
        scanf("%d", &pr[i]);
    }

    // sort by priority
    // bubble sort
    for (i = 0; i < n - 1; i++) {
        for (j = i + 1; j < n; j++) {
            if (pr[i] > pr[j]) {
                // swap
                int temp1 = pt[i];
                pt[i] = pt[j];
                pt[j] = temp1;
                int t = pr[i];
                pr[i] = pr[j];
                pr[j] = t;
                strcpy(temp, p[i]);
```

```
                strcpy(p[i], p[j]);
                strcpy(p[j], temp);
            }
        }
    }

    // Make Gantt chart

    printf("\n\n");
    for (i = 0; i < n; i++) {
        printf("\t%s\t", p[i]);
    }
    printf("\n");
    for (i = 0; i < n; i++) {
        printf("\t(%d)\t", pt[i]);
    }
    printf("\n(0)-\t");
    for (i = 0; i < n; i++) {
        bst += c;
        turn += c + pt[i];
        c = c + pt[i];
        printf("-\t%d\t-", c);
    }

    printf("\n\n");
    printf("Average time: %f\n", bst / n);
    printf("Turn around time: %f\n", turn / n);

    return 0;
}
```

Output

```
Enter no of processes: 3
Enter process 1 name: p1
Enter process time: 24
Enter the priority of process: 3
Enter process 2 name: p2
Enter process time: 3
Enter the priority of process: 1
Enter process 3 name: p3
Enter process time: 2
Enter the priority of process: 2


        p2              p3              p1
        (3)             (2)             (24)
(0)-    -       3       --      5       --      29      -

Average time: 2.666667
Turn around time: 12.333333
```

12 Week 12: First In First Out Page Replacement Policy

Program (Pointers)

```c
// fifo page replacement

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

typedef struct list {
    int size; // current size of list
    int cs;   // counter at which to insert new page
    int nf;   // no of free pages
    int *f;   // array to store page
} list;

list *newlist(int nf) {
    list *l = (list *)malloc(sizeof(list));
    l->cs = 0;
    l->f = (int *)malloc(sizeof(int) * nf);
    l->nf = nf;
    return l;
}

int find(list *l, int x) {
    for (int i = 0; i < l->size; i++)
        if (l->f[i] == x)
            return 1;
    return 0;
}

/*
    insert the page x
    if full replace it with oldest page
*/
void insert(list *l, int x) {
    if (l->size < l->nf) l->size++;
    if (l->cs == l->nf)
        // list full
        l->cs = 0;
    l->f[l->cs] = x;
    l->cs++;
}

void display(list *l) {
    int i;
    for (i = 0; i < l->size; i++)
        printf("%d ", l->f[i]);
    for (i = l->size; i < l->nf; i++)
        printf("_ ");
    // printf("\n");
}

int main() {
    int pf = 0; // no of page faults
    int rfs;    // reference string length
    int *rf;    // reference string
    int i, nf;

    printf("FIFO page replacement\n");
```

```
    printf("Enter the size of reference string: ");
    scanf("%d", &rfs);

    rf = (int *)malloc(sizeof(int) * rfs);
    printf("Enter the reference string: ");
    for (i = 0; i < rfs; i++) {
        scanf("%d", &rf[i]);
    }

    printf("Enter the number of free frames: ");
    scanf("%d", &nf);

    // make a list with number of pages equal to nf
    list *l = newlist(nf);

    insert(l, rf[0]);
    display(l);
    printf("\tMiss! %d\n", rf[0]);
    pf = 1; // first page fault will always occur

    for (i = 1; i < rfs; i++) {
        if (!find(l, rf[i])) {
            // element not found
            pf++; // pagefault
            insert(l, rf[i]);
            display(l);
            printf("\tMiss! %d\n", rf[i]);
        } else {
            display(l);
            printf("\tHit!! %d\n", rf[i]);
        }
    }

    printf("No of page faults: %d\n", pf);

    return 0;
}
```

Output

```
FIFO page replacement
Enter the size of reference string: 12
Enter the reference string: 0 2 1 6 4 0 1 0 3 1 2 1
Enter the number of free frames: 4
0 _ _ _         Miss! 0
0 2 _ _         Miss! 2
0 2 1 _         Miss! 1
0 2 1 6         Miss! 6
4 2 1 6         Miss! 4
4 0 1 6         Miss! 0
4 0 1 6         Hit!! 1
4 0 1 6         Hit!! 0
4 0 3 6         Miss! 3
4 0 3 1         Miss! 1
2 0 3 1         Miss! 2
2 0 3 1         Hit!! 1
No of page faults: 9
```

13 Week 13: LRU Page Replacement Policy

Program (Pointers)

```c
// least recently use
// (lru) page replacement

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

int fsize;       // frame size
int ssize;       // reference string size
int rstring[30]; // reference string
int frame[10];   // list to store the pages
int arrive[30];  // arrive time for the pages

/* return 1 if page is found in the list */
int pagefound(int x) {
    for (int i = 0; i < fsize; i++) {
        if (x == frame[i]) {
            return 1;
        }
    }
    return 0;
}

/* display the list */
void display() {
    int i;
    for (i = 0; i < fsize; i++) {
        if (frame[i] >= 0) {
            printf("%d ", frame[i]);
        } else
            printf("_ ");
    }
}

/* returns the index of page with least arrival time */
int leastused() {
    int i, min = 0;
    for (i = 0; i < fsize; i++) {
        if (arrive[i] < arrive[min]) {
            min = i;
        }
    }
    return min;
}

/* return the index at which pageno is located */
int pagelocation(int pageno) {
    int i;
    for (i = 0; i < fsize; i++) {
        if (frame[i] == pageno) {
            return i;
        }
    }
    return i;
}

int main() {
```

```
    int pf = 0; // no of page faults
    int cs = 0; // current size
    int lfi;    // last recently used page index
    int i, idx;
    int f, ls = 0;
    int j = 0, y, k, z = 0, time = 0;

    printf("LRU Page Replacement\n");

    printf("Enter the frame size: ");
    scanf("%d", &fsize);

    printf("Enter the reference string size: ");
    scanf("%d", &ssize);

    printf("Enter the reference string: ");
    for (i = 0; i < ssize; i++)
        scanf("%d", &rstring[i]);

    // initilise time and frame for page
    for (k = 0; k < fsize; k++) {
        frame[k] = -3;
        arrive[k] = 0;
    }

    for (i = 0; i < ssize; i++) {
        y = pagefound(rstring[i]);
        if (y == 0) {
            // page fault
            pf++;
            if (cs >= fsize) {
                // replace with lru page
                lfi = leastused();
                frame[lfi] = rstring[i];
                arrive[lfi] = ++time;
            } else if (cs < fsize) {
                // if list still have some space
                frame[cs] = rstring[i];
                arrive[cs] = ++time;
            }
            display();
            printf("\tMiss! %d\n", rstring[i]);
        } else {
            // page found
            idx = pagelocation(rstring[i]);
            arrive[idx] = ++time;
            display();
            printf("\tHit!! %d\n", rstring[i]);
        }
        cs++;
    }

    printf("Page fault=%d\n", pf);

    return 0;
}
```

Output

```
LRU Page Replacement
Enter the frame size: 4
```

```
Enter the reference string size: 13
Enter the reference string: 7 0 1 2 0 3 0 4 2 3 0 3 2 3
7 _ _ _         Miss! 7
7 0 _ _         Miss! 0
7 0 1 _         Miss! 1
7 0 1 2         Miss! 2
7 0 1 2         Hit!! 0
3 0 1 2         Miss! 3
3 0 1 2         Hit!! 0
3 0 4 2         Miss! 4
3 0 4 2         Hit!! 2
3 0 4 2         Hit!! 3
3 0 4 2         Hit!! 0
3 0 4 2         Hit!! 3
3 0 4 2         Hit!! 2
Page fault=6
```