Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology

(An Autonomous Institute affiliated to Savitribai Phule Pune University)

**Assignment-7**

|  |  |
| --- | --- |
| **Subject** | Operating System |
| **Name** | Ajinkya Walunj |
| **Class** | CS-A |
| **Roll No.** | 05 |

**Implement following memory management schemes**

**1. Placement Strategies - First fir, Next Fit, Best Fit & Worst fit.**

**2. Buddy System**

**3. Conversion of logical address in to physical address using paging.**

**4. Page replacement algorithms (FIFO, LRU, Optimal)**

1. **Placement strategies**

*Code :-*

#include <stdlib.h>

typedef struct Process {

    int pid;

    int size;

} Process;

typedef struct Partition {

    int pid;

    int size;

    struct Partition\* prev;

    struct Partition\* next;

} Partition;

Partition\* create\_partition\_list(int\* partition\_sizes, int n) {

    Partition\* head = NULL;

    Partition\* tail = NULL;

    for (int i = 0; i < n; i++) {

        Partition\* partition = (Partition\*)malloc(sizeof(Partition));

        partition->pid = 0;

        partition->size = partition\_sizes[i];

        partition->prev = tail;

        partition->next = NULL;

        if (tail != NULL)

            tail->next = partition;

        tail = partition;

        if (head == NULL)

            head = partition;

    }

    return head;

}

void print\_partitions(Partition\* head) {

    printf("Partitions: ");

    Partition\* current = head;

    while (current != NULL) {

        printf("[%d:%d] ", current->pid, current->size);

        current = current->next;

    }

    printf("\n");

}

void first\_fit(Partition\* head, Process\* processes, int m) {

    for (int i = 0; i < m; i++) {

        int pid = i + 1;

        Partition\* current = head;

        while (current != NULL) {

            if (current->pid == 0 && current->size >= processes[i].size) {

                current->pid = pid;

                if (current->size > processes[i].size) {

                    Partition\* new\_partition = (Partition\*)malloc(sizeof(Partition));

                    new\_partition->pid = 0;

                    new\_partition->size = current->size - processes[i].size;

                    new\_partition->prev = current;

                    new\_partition->next = current->next;

                    if (current->next != NULL)

                        current->next->prev = new\_partition;

                    current->next = new\_partition;

                    current->size = processes[i].size;

                }

                break;

            }

            current = current->next;

        }

        if (current == NULL) {

            printf("Process %d cannot be allocated\n", pid);

        }

    }

}

void next\_fit(Partition\* head, Process\* processes, int m) {

    Partition\* current = head;

    for (int i = 0; i < m; i++) {

        int pid = i + 1;

        int flag = 0;

        while (current != NULL) {

            if (current->pid == 0 && current->size >= processes[i].size) {

                current->pid = pid;

                if (current->size > processes[i].size) {

                    Partition\* new\_partition = (Partition\*)malloc(sizeof(Partition));

                    new\_partition->pid = 0;

                    new\_partition->size = current->size - processes[i].size;

                    new\_partition->prev = current;

                    new\_partition->next = current->next;

                    if (current->next != NULL)

                        current->next->prev = new\_partition;

                    current->next = new\_partition;

                    current->size = processes[i].size;

                }

                flag = 1;

                break;

            }

            current = current->next;

            if (current == NULL)

                current = head;

        }

        if (flag == 0) {

            printf("Process %d cannot be allocated\n", pid);

        }

    }

}

void worst\_fit(Partition\* head, Process\* processes, int m) {

    for (int i = 0; i < m; i++) {

        int pid = i + 1;

        Partition\* current = head;

        Partition\* worst = NULL;

        while (current != NULL) {

            if (current->pid == 0 && current->size >= processes[i].size) {

                if (worst == NULL || worst->size < current->size)

                    worst = current;

            }

            current = current->next;

        }

        if (worst != NULL) {

            worst->pid = pid;

            if (worst->size > processes[i].size) {

                Partition\* new\_partition = (Partition\*)malloc(sizeof(Partition));

                new\_partition->pid = 0;

                new\_partition->size = worst->size - processes[i].size;

                new\_partition->prev = worst;

                new\_partition->next = worst->next;

                if (worst->next != NULL)

                    worst->next->prev = new\_partition;

                else

                    worst->next = new\_partition;

                worst->next = new\_partition;

                worst->size = processes[i].size;

            }

        } else {

            printf("Process %d cannot be allocated\n", pid);

        }

    }

}

void best\_fit(Partition\* head, Process\* processes, int m) {

    for (int i = 0; i < m; i++) {

        int pid = i + 1;

        Partition\* current = head;

        Partition\* best = NULL;

        while (current != NULL) {

            if (current->pid == 0 && current->size >= processes[i].size) {

                if (best == NULL || best->size > current->size)

                    best = current;

            }

            current = current->next;

        }

        if (best != NULL) {

            best->pid = pid;

            if (best->size > processes[i].size) {

                Partition\* new\_partition = (Partition\*)malloc(sizeof(Partition));

                new\_partition->pid = 0;

                new\_partition->size = best->size - processes[i].size;

                new\_partition->prev = best;

                new\_partition->next = best->next;

                if (best->next != NULL)

                    best->next->prev = new\_partition;

                else

                    best->next = new\_partition;

                best->next = new\_partition;

                best->size = processes[i].size;

            }

        } else {

            printf("Process %d cannot be allocated\n", pid);

        }

    }

}

int main() {

    int n, m;

    printf("Enter the number of memory partitions: ");

    scanf("%d", &n);

    int partition\_sizes[n];

    printf("Enter the sizes of the memory partitions: ");

    for (int i = 0; i < n; i++)

        scanf("%d", &partition\_sizes[i]);

    Partition\* partition\_head = create\_partition\_list(partition\_sizes, n);

    printf("Enter the number of processes: ");

    scanf("%d", &m);

    Process processes[m];

    printf("Enter the sizes of the processes: ");

    for (int i = 0; i < m; i++) {

        int size;

        scanf("%d", &size);

        processes[i].pid = i + 1;

        processes[i].size = size;

    }

    int choice;

    while(choice!=5)

    printf("Choose the memory allocation algorithm:\n");

    printf("1. First fit\n");

    printf("2. Next fit\n");

    printf("3. Worst fit\n");

    printf("4. Best fit\n");

    printf("Enter your choice: ");

    scanf("%d", &choice);

   switch (choice) {

        case 1:

            first\_fit(partition\_head, processes, m);

            break;

        case 2:

            next\_fit(partition\_head, processes, m);

            break;

        case 3:

            worst\_fit(partition\_head, processes, m);

            break;

        case 4:

            best\_fit(partition\_head, processes, m);

            break;

        default:

            printf("Invalid choice\n");

            return 0;

    }

    printf("\nPartition\tSize\tProcess\n");

    Partition\* current = partition\_head;

    while (current != NULL) {

        if (current->pid == 0)

            printf("%d\t\t%d\t%s\n", current->pid, current->size, "Free");

        else

            printf("%d\t\t%d\t%d\n", current->pid, current->size, current->pid);

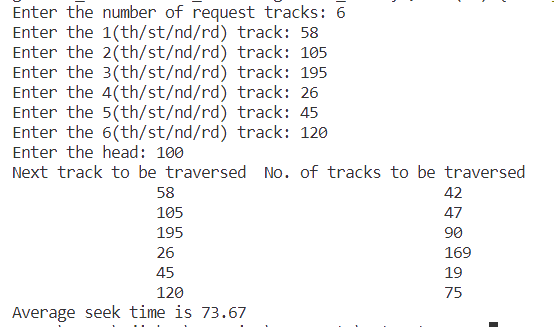
        current = current->next;

    }

    return 0;

}

*Output :-*



1. **Buddy System**

The buddy system is a memory management technique used to allocate memory blocks of varying sizes from a larger memory pool. The pool is divided into blocks of a fixed size, and when a program requests a block of memory, the system searches for a block of the appropriate size. If the block is larger than needed, the system can split it into two smaller blocks, and if the block is smaller than needed, the system can combine it with an adjacent block to make a larger one. The buddy system is efficient in managing memory because it reduces fragmentation and waste by allocating memory in the smallest possible block size necessary. It is also an example of a recursive algorithm, as the system continues to split blocks into smaller and smaller pieces until it finds a block of the right size. Overall, the buddy system is a commonly used memory management technique in operating systems, and is used in systems where multiple programs may compete for access to a limited amount of memory.

*Code :-*

#include<bits/stdc++.h>

using namespace std;

int siz;

vector<pair<int, int>> free\_list[100000];

map<int, int> mp;

void initialize(int sz)

{

    int n = ceil(log(sz) / log(2));

    siz = n + 1;

    for(int i = 0; i <= n; i++)

        free\_list[i].clear();

    free\_list[n].push\_back(make\_pair(0, sz - 1));

}

void allocate(int sz)

{

    int n = ceil(log(sz) / log(2));

    if (free\_list[n].size() > 0)

    {

        pair<int, int> temp = free\_list[n][0];

        free\_list[n].erase(free\_list[n].begin());

        cout << "Memory from " << temp.first

            << " to " << temp.second << " allocated"

            << "\n";

        mp[temp.first] = temp.second -

                        temp.first + 1;

    }

    else

    {

        int i;

        for(i = n + 1; i < siz; i++)

        {

            if(free\_list[i].size() != 0)

                break;

        }

        if (i == siz)

        {

            cout << "Sorry, failed to allocate memory \n";

        }

        // If found

        else

        {

            pair<int, int> temp;

            temp = free\_list[i][0];

            // Remove first block to split it into halves

            free\_list[i].erase(free\_list[i].begin());

            i--;

            for(; i >= n; i--)

            {

                // Divide block into two halves

                pair<int, int> pair1, pair2;

                pair1 = make\_pair(temp.first,

                                temp.first +

                                (temp.second -

                                temp.first) / 2);

                pair2 = make\_pair(temp.first +

                                (temp.second -

                                temp.first + 1) / 2,

                                temp.second);

                free\_list[i].push\_back(pair1);

                // Push them in free list

                free\_list[i].push\_back(pair2);

                temp = free\_list[i][0];

                // Remove first free block to

                // further split

                free\_list[i].erase(free\_list[i].begin());

            }

            cout << "Memory from " << temp.first

                << " to " << temp.second

                << " allocated" << "\n";

            mp[temp.first] = temp.second -

                            temp.first + 1;

        }

    }

}

// Driver code

int main()

{

    int total,c,req;

    printf("Enter the total size of memory: ");

    cin>>total;

    initialize(total);

    printf("Enter the no. of processes: ");

    cin>>c;

    while(c>0)

    {

        printf("Enter the size of process: ");

        cin>>req;

        if(req < 0)

            break;

        allocate(req);

        c-=1;

    }

    // initialize(128);

    // allocate(32);

    // allocate(7);

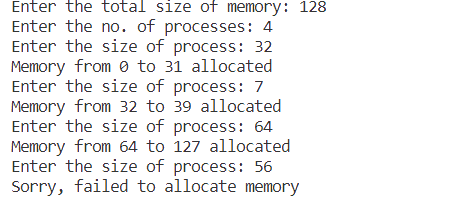
    // allocate(64);

    // allocate(56);

    return 0;

}

*Output :-*



1. **Conversion of logical address to physical address using paging**

*Code:-*

#include <iostream>

#include <math.h>

using namespace std;

int main()

{

    int page\_size, page\_table\_size, num\_pages, page\_number, page\_offset, logical\_address, physical\_address;

    cout << "Enter the page size: ";

    cin >> page\_size;

    cout << "Enter the page table size: ";

    cin >> page\_table\_size;

    num\_pages = page\_table\_size / page\_size;

    int page\_table[num\_pages];

    for(int i=0; i<num\_pages; i++)

    {

        cout << "Enter the frame number for page " << i << ": ";

        cin >> page\_table[i];

    }

    cout << "Enter the logical address: ";

    cin >> logical\_address;

    page\_number = logical\_address / page\_size;

    page\_offset = logical\_address % page\_size;

    if(page\_number >= num\_pages)

    {

        cout << "Page number is out of range" << endl;

        return 0;

    }

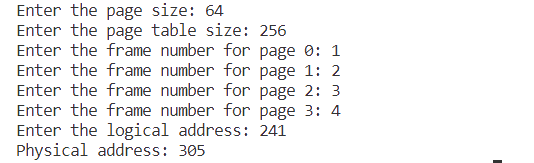
    physical\_address = (page\_table[page\_number] \* page\_size) + page\_offset;

    cout << "Physical address: " << physical\_address << endl;

    return 0;

}

*Output:-*

**

1. **Page replacement algorithms**

*Code:-*

1. FIFO

#include<stdio.h>

int main(){

int frames, pages, i, j, hit=0, fault=0, counter=0;

int reference\_string[100], mem\_layout[100][100];

printf("\nEnter the number of frames: ");

scanf("%d",&frames);

printf("\nEnter the number of pages: ");

scanf("%d",&pages);

printf("\nEnter the reference string: ");

for(i=0;i<pages;i++){

scanf("%d",&reference\_string[i]);

}

for(i=0;i<frames;i++){

mem\_layout[i][0]=-1;

}

for(i=0;i<pages;i++){

hit=0;

for(j=0;j<frames;j++){

if(mem\_layout[j][0]==reference\_string[i]){

hit=1;

mem\_layout[j][i+1]=1;

break;

}

}

if(hit==0){

mem\_layout[counter][0]=reference\_string[i];

fault++;

for(j=0;j<frames;j++){

mem\_layout[counter][j+1]=0;

}

counter++;

if(counter==frames){

counter=0;

}

}

printf("\n");

for(j=0;j<frames;j++){

printf("%d\t",mem\_layout[j][i]);

}

}

printf("\nTotal Page Faults: %d",fault);

return 0;

}

*Output:-*



1. LRU

#include <stdio.h>

#define MAX\_PAGES 100

int findLRU(int time[], int n){

int i, min = time[0], pos = 0;

for(i=1;i<n;i++){

if(time[i]<min){

min = time[i];

pos = i;

}

}

return pos;

}

int main(){

int pages[MAX\_PAGES], frames, n, i, j, k, faults = 0, pos;

printf("Enter the number of frames: ");

scanf("%d", &frames);

printf("Enter the number of pages: ");

scanf("%d", &n);

printf("Enter the reference string: ");

for(i=0;i<n;i++){

scanf("%d", &pages[i]);

}

int mem[frames], time[frames];

for(i=0;i<frames;i++){

mem[i] = -1;

time[i] = 0;

}

for(i=0;i<n;i++){

for(j=0;j<frames;j++){

if(mem[j]==pages[i]){

time[j] = i+1;

break;

}

}

if(j==frames){

pos = findLRU(time, frames);

mem[pos] = pages[i];

time[pos] = i+1;

faults++;

}

printf("\n");

for(k=0;k<frames;k++){

printf("%d\t", mem[k]);

}

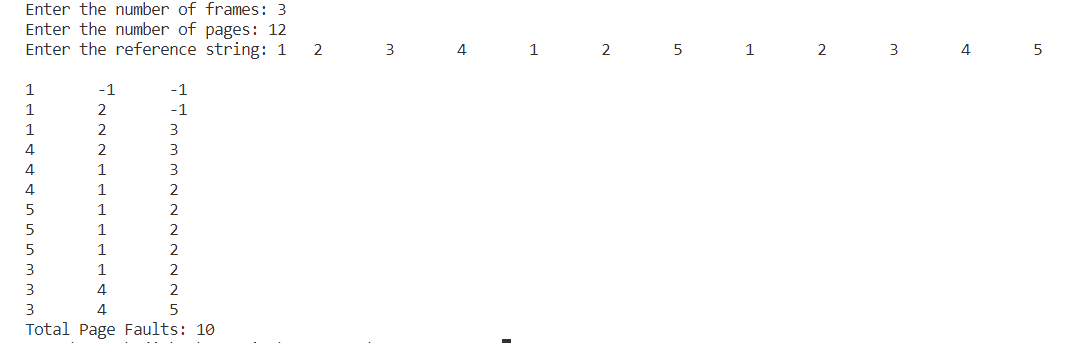
}

printf("\nTotal Page Faults: %d", faults);

return 0;

}

*Output:-*



1. OPTIMAL

#include<stdio.h>

#include<limits.h>

int main(){

int frames, pages, i, j, k, l, hit=0, fault=0, max\_dist, max\_frame, flag;

int reference\_string[100], mem\_layout[100][100], distance[100];

printf("\nEnter the number of frames: ");

scanf("%d",&frames);

printf("\nEnter the number of pages: ");

scanf("%d",&pages);

printf("\nEnter the reference string: ");

for(i=0;i<pages;i++){

scanf("%d",&reference\_string[i]);

}

for(i=0;i<frames;i++){

mem\_layout[i][0]=-1;

}

for(i=0;i<pages;i++){

hit=0;

for(j=0;j<frames;j++){

if(mem\_layout[j][0]==reference\_string[i]){

hit=1;

break;

}

}

if(hit==0){

fault++;

flag=0;

for(j=0;j<frames;j++){

if(mem\_layout[j][0]==-1){

mem\_layout[j][0]=reference\_string[i];

flag=1;

break;

}

}

if(flag==0){

for(j=0;j<frames;j++){

distance[j]=INT\_MAX;

for(k=i+1;k<pages;k++){

if(reference\_string[k]==mem\_layout[j][0]){

distance[j]=k-i;

break;

}

}

}

max\_dist=-1;

for(j=0;j<frames;j++){

if(distance[j]>max\_dist){

max\_dist=distance[j];

max\_frame=j;

}

}

mem\_layout[max\_frame][0]=reference\_string[i];

}

}

for(j=0;j<frames;j++){

mem\_layout[j][i+1]=mem\_layout[j][i];

}

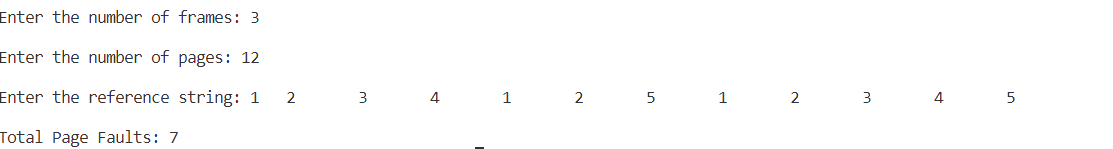
}

printf("\nTotal Page Faults: %d",fault);

return 0;

}

*Output:-*

**