**Name: Srinivas Sanjeevkumar Chenna**

**Division: A**

**Roll Number: 48**

**Subject: Operating System (OS) LAB**

**Lab Assignment 6**

**Implement following memory management schemes**

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

**First fit:**

#include <bits/stdc++.h>

using namespace std;

void firstFit(int blockSize[], int m, int processSize[], int n)

{

int allocation[n];

memset(allocation, -1, sizeof(allocation));

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

{

for (int j = 0; j < m; j++)

{

if (blockSize[j] >= processSize[i])

{

allocation[i] = j;

blockSize[j] -= processSize[i];

break;

}

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

firstFit(blockSize, m, processSize, n);

return 0;

}

**Output:**

Text

Description automatically generated

**Nest Fit:**

#include <bits/stdc++.h>

using namespace std;

void NextFit(int blockSize[], int m, int processSize[], int n)

{

int allocation[n], j = 0, t = m - 1;

memset(allocation, -1, sizeof(allocation));

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

{

while (j < m)

{

if (blockSize[j] >= processSize[i])

{

allocation[i] = j;

blockSize[j] -= processSize[i];

t = (j - 1) % m;

break;

}

if (t == j)

{

t = (j - 1) % m;

break;

}

j = (j + 1) % m;

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

// Driver program

int main()

{

int blockSize[] = { 5, 10, 20 };

int processSize[] = { 10, 20, 5 };

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

NextFit(blockSize, m, processSize, n);

return 0;

}

**Output:**

Text

Description automatically generated

**Best fit:**

#include <iostream>

using namespace std;

void bestFit(int blockSize[], int m, int processSize[], int n)

{

int allocation[n];

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

allocation[i] = -1;

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

{

int bestIdx = -1;

for (int j = 0; j < m; j++)

{

if (blockSize[j] >= processSize[i])

{

if (bestIdx == -1)

bestIdx = j;

else if (blockSize[bestIdx] > blockSize[j])

bestIdx = j;

}

}

if (bestIdx != -1)

{

allocation[i] = bestIdx;

blockSize[bestIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i + 1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize) / sizeof(blockSize[0]);

int n = sizeof(processSize) / sizeof(processSize[0]);

bestFit(blockSize, m, processSize, n);

return 0;

}

**Output:**

Text

Description automatically generated

**Worst fit:**

#include<bits/stdc++.h>

using namespace std;

void worstFit(int blockSize[], int m, int processSize[],int n)

{

int allocation[n];

memset(allocation, -1, sizeof(allocation));

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

{

int wstIdx = -1;

for (int j=0; j<m; j++)

{

if (blockSize[j] >= processSize[i])

{

if (wstIdx == -1)

wstIdx = j;

else if (blockSize[wstIdx] < blockSize[j])

wstIdx = j;

}

}

if (wstIdx != -1)

{

allocation[i] = wstIdx;

blockSize[wstIdx] -= processSize[i];

}

}

cout << "\nProcess No.\tProcess Size\tBlock no.\n";

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

{

cout << " " << i+1 << "\t\t" << processSize[i] << "\t\t";

if (allocation[i] != -1)

cout << allocation[i] + 1;

else

cout << "Not Allocated";

cout << endl;

}

}

int main()

{

int blockSize[] = {100, 500, 200, 300, 600};

int processSize[] = {212, 417, 112, 426};

int m = sizeof(blockSize)/sizeof(blockSize[0]);

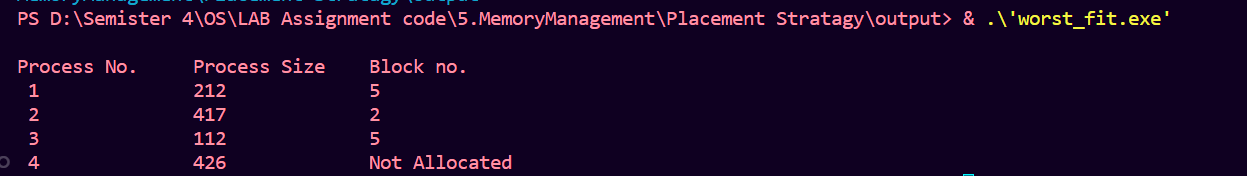
int n = sizeof(processSize)/sizeof(processSize[0]);

worstFit(blockSize, m, processSize, n);

return 0 ;

}

**Output:**



**2. Buddy System**

**Code:**

#include <iostream>

#include <vector>

#include <cmath>

using namespace std;

class Buddy {

public:

class Pair {

public:

int lb, ub;

Pair(int a = 0, int b = 0) : lb(a), ub(b) {}

};

int size;

vector<Pair> arr[100];

Buddy(int s) {

size = s;

int x = (int)ceil(log2(s));

for (int i = 0; i <= x; i++) {

arr[i].clear();

}

arr[x].push\_back(Pair(0, size - 1));

}

void allocate(int s) {

int x = (int)ceil(log2(s));

int i;

Pair temp;

if (!arr[x].empty()) {

temp = arr[x][0];

arr[x].erase(arr[x].begin());

cout << "Memory from " << temp.lb << " to " << temp.ub << " allocated" << endl;

return;

}

for (i = x + 1; i < 100; i++) {

if (arr[i].empty()) {

continue;

}

break;

}

if (i == 100) {

cout << "Sorry, failed to allocate memory" << endl;

return;

}

temp = arr[i][0];

i--;

for (; i >= x; i--) {

Pair newPair = Pair(temp.lb, temp.lb + (temp.ub - temp.lb) / 2);

Pair newPair2 = Pair(temp.lb + (temp.ub - temp.lb + 1) / 2, temp.ub);

arr[i].push\_back(newPair);

arr[i].push\_back(newPair2);

temp = arr[i][0];

arr[i].erase(arr[i].begin());

}

cout << "Memory from " << temp.lb << " to " << temp.ub << " allocated" << endl;

}

};

int main() {

int initialMemory = 0, val = 0;

cout << "Enter size of hard disk KB : ";

cin >> initialMemory;

Buddy obj(initialMemory);

while (true) {

cout << "Enter size of process in KB : ";

cin >> val;

if (val <= 0) {

break;

}

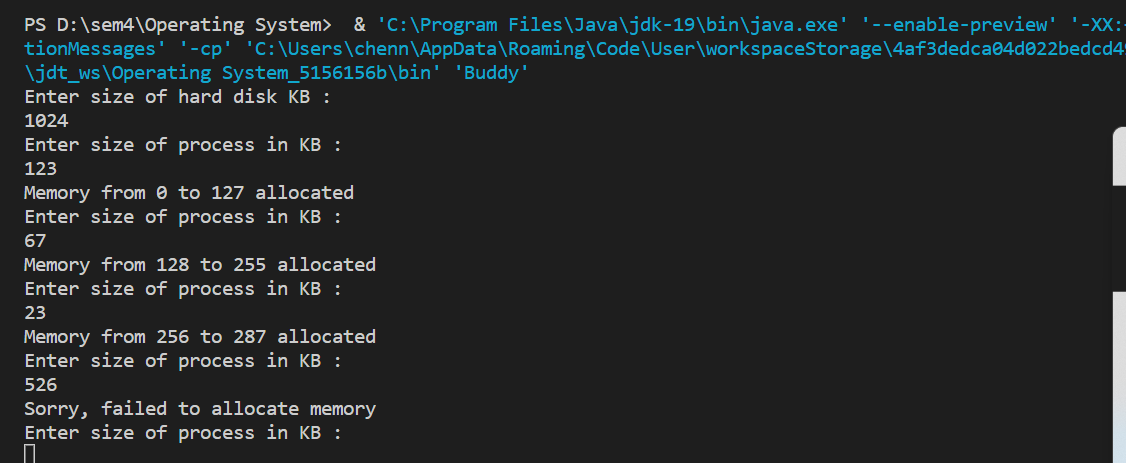
obj.allocate(val);

}

return 0;

}

Output:



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

**Code:**

#include <bits/stdc++.h>

using namespace std;

int ADDRESSMAP(int C\_VA, int arr[], int page\_size, int n)

{

int pte = C\_VA / page\_size;

string temp = "";

if (pte >= n)

{

cout << "Page Fault" << endl;

return -1;

}

else

{

return ((arr[pte] \* page\_size) + (C\_VA % page\_size));

}

}

int convert(string VA)

{

int n = VA.length();

int a = 1;

int res = 0;

for (int i = n - 1; i >= 0; i--)

{

if (VA[i] == '1')

{

res += a \* 1;

}

a = a \* 2;

}

return res;

}

int main()

{

int ptr;

int page\_size;

string VA;

int C\_VA;

int arr[100];

char M[1000][4];

int VA\_SPACE = 100;

int READ\_SPACE = 300;

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

cin >> page\_size;

cout << endl;

int n;

cout << "Enter the number of entries in the page table:" << endl;

cin >> n;

cout << "Enter the contents of the page table";

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

{

cin >> arr[i];

}

cout << "Enter binary virtual address: ";

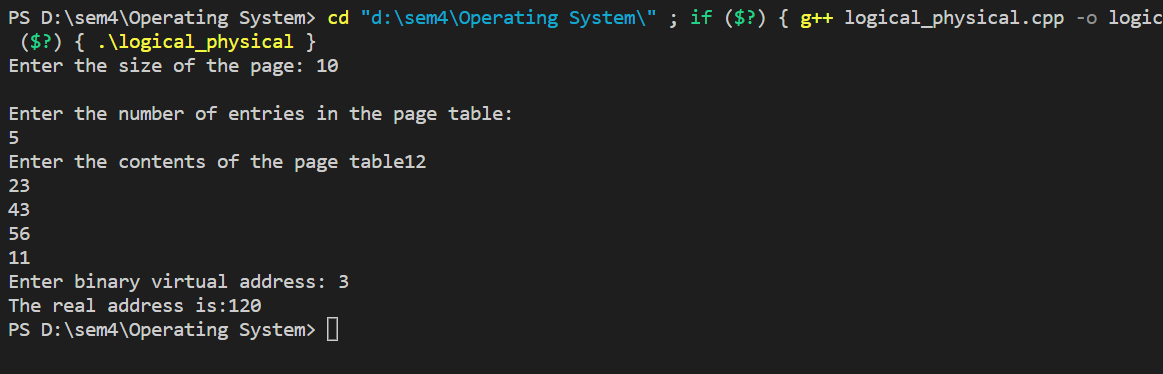
cin >> VA;

C\_VA = convert(VA);

cout << "The real address is:" << ADDRESSMAP(C\_VA, arr, page\_size, n) << endl;

}

**Output:**



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

**FIFO**

**Code:**

#include <iostream>

#include <queue>

using namespace std;

int main() {

int n\_frames, n\_pages, page\_faults = 0;

cout << "Enter the number of frames: ";

cin >> n\_frames;

cout << "Enter the number of pages: ";

cin >> n\_pages;

int pages[n\_pages];

cout << "Enter the page references: ";

for (int i = 0; i < n\_pages; i++) {

cin >> pages[i];

}

queue<int> frame;

bool is\_page\_present[n\_pages] = { false };

cout << "Reference String\tFrames\t\tPage Fault\n";

cout << "-------------------------------------------------------\n";

for (int i = 0; i < n\_pages; i++) {

cout << pages[i] << "\t\t\t";

if (!is\_page\_present[pages[i]]) {

if (frame.size() == n\_frames) {

int removed\_page = frame.front();

frame.pop();

is\_page\_present[removed\_page] = false;

}

frame.push(pages[i]);

is\_page\_present[pages[i]] = true;

for (int j = 0; j < frame.size(); j++) {

int temp = frame.front();

frame.pop();

cout << temp << " ";

frame.push(temp);

}

page\_faults++;

cout << "\t\tMiss";

} else {

for (int j = 0; j < frame.size(); j++) {

int temp = frame.front();

frame.pop();

cout << temp << " ";

frame.push(temp);

}

cout << "\t\tHit";

}

cout << endl;

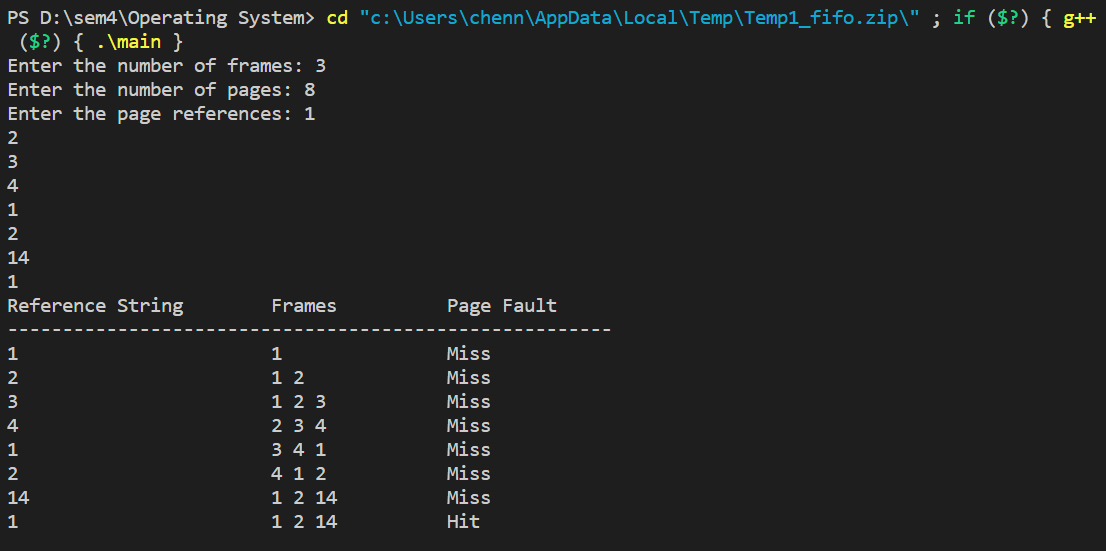
}

cout << "\nNumber of page faults: " << page\_faults << endl;

return 0;

}

**Output:**

****

**LRU**

**Code:**

#include <iostream>

#include <algorithm>

#include <deque>

using namespace std;

int main() {

int num\_frames, num\_pages, page\_faults = 0;

cout << "Enter the number of frames: ";

cin >> num\_frames;

cout << "Enter the number of pages: ";

cin >> num\_pages;

int pages[num\_pages];

cout << "Enter the page references: ";

for (int i = 0; i < num\_pages; i++) {

cin >> pages[i];

}

deque<int> frame\_deque;

bool is\_page\_present[num\_pages] = { false };

cout << "Reference String\tFrames\t\tPage Fault\n";

for (int i = 0; i < num\_pages; i++) {

cout << pages[i] << "\t\t\t";

if (!is\_page\_present[pages[i]]) {

if (frame\_deque.size() == num\_frames) {

int removed\_page = frame\_deque.back();

frame\_deque.pop\_back();

is\_page\_present[removed\_page] = false;

}

frame\_deque.push\_front(pages[i]);

is\_page\_present[pages[i]] = true;

page\_faults++;

for (int j = 0; j < num\_frames; j++) {

if (j < frame\_deque.size()) {

cout << frame\_deque[j] << " ";

} else {

cout << " ";

}

}

cout << "\t\t";

cout << "Fault";

cout << endl;

} else {

frame\_deque.erase(find(frame\_deque.begin(), frame\_deque.end(), pages[i]));

frame\_deque.push\_front(pages[i]);

for (int j = 0; j < num\_frames; j++) {

if (j < frame\_deque.size()) {

cout << frame\_deque[j] << " ";

} else {

cout << " ";

}

}

cout << "\t\t";

cout << "Hit";

cout << endl;

}

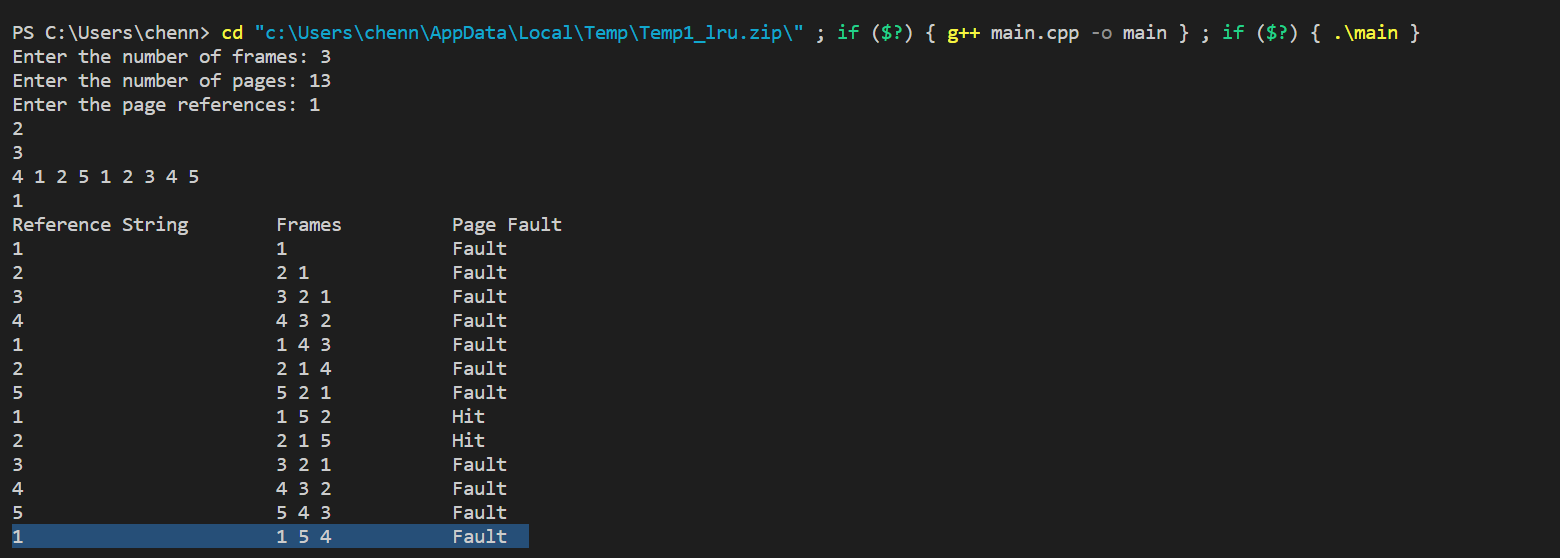
}

cout << "\nNumber of page faults: " << page\_faults << endl;

return 0;

}

**Output:**

****

**Optimal**

**Code:**

#include <iostream>

#include <algorithm>

using namespace std;

int main() {

int num\_frames, num\_pages, page\_faults = 0;

cout << "Enter the number of frames: ";

cin >> num\_frames;

cout << "Enter the number of pages: ";

cin >> num\_pages;

int pages[num\_pages];

cout << "Enter the page references: ";

for (int i = 0; i < num\_pages; i++) {

cin >> pages[i];

}

int frames[num\_frames];

bool is\_page\_present[num\_pages] = { false };

int next\_use[num\_frames];

cout << "Reference String\tFrames\t\tPage Fault\n";

for (int i = 0; i < num\_pages; i++) {

cout << pages[i] << "\t\t\t";

if (!is\_page\_present[pages[i]]) {

if (page\_faults < num\_frames) {

frames[page\_faults] = pages[i];

next\_use[page\_faults] = find(pages + i + 1, pages + num\_pages, pages[i]) - pages;

} else {

int index = 0;

for (int j = 1; j < num\_frames; j++) {

if (next\_use[j] > next\_use[index]) {

index = j;

}

}

//cout<<index<<endl;

is\_page\_present[frames[index]] = false;

frames[index] = pages[i];

next\_use[index] = find(pages + i + 1, pages + num\_pages, pages[i]) - pages;

}

page\_faults++;

is\_page\_present[pages[i]] = true;

for (int j = 0; j < num\_frames; j++) {

if (j < page\_faults) {

cout << frames[j] << " ";

} else {

cout << " ";

}

}

cout << "\t\tMiss";

} else {

for (int j = 0; j < num\_frames; j++) {

if (j < page\_faults) {

cout << frames[j] << " ";

} else {

cout << " ";

}

}

cout << "\t\tHit";

}

cout << endl;

}

cout << "Page faults: " << page\_faults << endl;

return 0;

}

**Output:**

**Text

Description automatically generated**