Experiment 3

Aim : To implement page replacement algorithms and memory allocation techniques

Page Replacement Algorithms

Optimal Page Replacement

Theory:

Optimal page replacement algorithm is a memory management algorithm used in operating systems to manage virtual memory. It works by replacing the page in memory that will not be used for the longest period of time in the future. The optimal page replacement algorithm is theoretically the best page replacement algorithm because it always selects the page that will not be used for the longest period of time in the future. However, in practice, it is not feasible because it requires knowledge of future memory access patterns, which is impossible to predict. The optimal page replacement algorithm is mainly used for benchmarking other page replacement algorithms. It provides a measure of the best possible performance that can be achieved with page replacement algorithms. It also helps in evaluating the effectiveness of other page replacement algorithms by comparing their performance to the optimal algorithm. Overall, the optimal page replacement algorithm is not practical for real-world use but is useful for providing a benchmark for other page replacement algorithms.

Code:

```
#include <bits/stdc++.h>
using namespace std;
bool search(int key, vector<int>& fr)
{
    for (int i = 0; i < fr.size(); i++)
        if (fr[i] == key)
            return true;
    return false;
}
int predict(int pg[], vector<int>& fr, int pn, int index)
{
    int res = -1, farthest = index;
    for (int i = 0; i < fr.size(); i++) {
        int j;
        for (j = index; j < pn; j++) {
            if (fr[i] == pg[j]) {
                if (j > farthest) {
                    farthest = j;
                    res = i;
                }
                break;
            }}
        if (j == pn)
            return i; }
    return (res == -1) ? 0 : res;
}
void optimalPage(int pg[], int pn, int fn)
{
    vector<int> fr;
    int hit = 0;
    for (int i = 0; i < pn; i++) {
        if (search(pg[i], fr)) {
            hit++;
            continue;
```

```
        }
        if (fr.size() < fn)
            fr.push_back(pg[i]);
        else {
            int j = predict(pg, fr, pn, i + 1);
            fr[j] = pg[i]; } }
    cout << "No. of hits = " << hit << endl;
    cout << "No. of misses = " << pn - hit << endl;
}
int main()
{
    int pg[] = { 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 };
    int pn = sizeof(pg) / sizeof(pg[0]);
    int fn = 4;
    optimalPage(pg, pn, fn);
    return 0;
}
```

Output:

```
No. of hits = 7
No. of misses = 6
```

FIFO:

Theory:

FIFO (First-In-First-Out) page replacement algorithm is a memory management algorithm used in operating systems to manage virtual memory. It works by replacing the page that was first brought into memory and has been there the longest. The FIFO page replacement algorithm maintains a queue of the pages that are currently in memory. Whenever a page fault occurs, the page at the front of the queue, which was brought into memory first, is selected for replacement. One of the main advantages of the FIFO page replacement algorithm is its simplicity. It is easy to implement and requires minimal bookkeeping. However, it suffers from the Belady's anomaly, where increasing the number of page frames may actually increase the number of page faults. FIFO page replacement algorithm is mainly used in systems with limited resources, where simplicity and low overhead are more important than performance. It is also used as a baseline for evaluating the performance of other page replacement algorithms.

Code:

```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity)
{
    unordered_set<int> s;
    queue<int> indexes;
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i])==s.end())
            {
                s.insert(pages[i]);
                page_faults++;
                indexes.push(pages[i]);
            }
        }
        else
```

```
        {
            if (s.find(pages[i]) == s.end())
            {
                int val = indexes.front();
                indexes.pop();
                s.erase(val);
                s.insert(pages[i]);
                indexes.push(pages[i]);
                page_faults++;
            }
        }
    }
    return page_faults;
}

int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    int x =pageFaults(pages, n, capacity);
    cout << "No of miss " <<x<<endl;
    cout << "No of hits "<< n-x;
    return 0;
}
```

Output:

```
No of miss 7
No of hits 6
```

LRU:

Theory:

LRU (Least Recently Used) page replacement algorithm is a memory management algorithm used in operating systems to manage virtual memory. It works by replacing the page that has not been accessed for the longest period of time. The LRU page replacement algorithm maintains a list of the pages that are currently in memory, ordered by the time of their last access. Whenever a page fault occurs, the page that has not been accessed for the longest period of time is selected for replacement. One of the main advantages of the LRU page replacement algorithm is its effectiveness. It performs well in most scenarios and is able to adapt to changing access patterns. However, it requires more bookkeeping than simpler algorithms like FIFO. LRU page replacement algorithm is commonly used in modern operating systems that require high performance and efficient use of resources. It is also used in databases and cache systems to manage memory.

Code :

```
#include<bits/stdc++.h>
using namespace std;
int pageFaults(int pages[], int n, int capacity)
{
    unordered_set<int> s;
    unordered_map<int, int> indexes;
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
        if (s.size() < capacity)
        {
            if (s.find(pages[i])==s.end())
```

```
            {
                s.insert(pages[i]);
                page_faults++;
            }
            indexes[pages[i]] = i;
        }
        else
        {
            if (s.find(pages[i]) == s.end())
            {
                int lru = INT_MAX, val;
                for (auto it=s.begin(); it!=s.end(); it++)
                {
                    if (indexes[*it] < lru)
                    {
                        lru = indexes[*it];
                        val = *it;
                    }
                }
                s.erase(val);
                s.insert(pages[i]);
                page_faults++;
            }
            indexes[pages[i]] = i;
        }
    }
    return page_faults;
}
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    int x =pageFaults(pages, n, capacity);
    cout << "No of miss " <<x<<endl;
cout << "No of hits "<< n-x;
    return 0;
}
```

Output:

```
No of miss 6
No of hits 7
```

Memory allocation techniques

First Fit:

Theory:

First fit is a memory allocation algorithm used in computer operating systems to allocate memory to processes. It works by allocating the first available block of memory that is large enough to hold the process. The algorithm starts at the beginning of the memory space and searches for the first available block of memory that is large enough to accommodate the process. If a suitable block is found, the process is allocated to that block, and the remaining space is marked as used. The advantage of the first fit algorithm is its simplicity and efficiency. It requires only one pass through the available memory blocks, making it faster than other algorithms. However, it may not be the most optimal algorithm in terms of memory utilization since it can leave small unused memory blocks between allocated processes, which can lead to memory fragmentation. It is used in operating systems, especially in systems with limited memory resources. However, it may not be the best choice

for systems that require optimal memory utilization.

Code:

```
#include<stdio.h>
void firstFit(int blockSize[], int m, int processSize[], int n)
{
    int i, j;
    int allocation[n];
    for(i = 0; i < n; i++)
    {
        allocation[i] = -1;
    }
    for (i = 0; i < n; i++)
    {
        for (j = 0; j < m; j++)
        {
            if (blockSize[j] >= processSize[i])
            {
                allocation[i] = j;
                blockSize[j] -= processSize[i];
                break;
            }
        }
    }
    printf("\nProcess No.\tProcess Size\tBlock no.\n");
    for (int i = 0; i < n; i++)
    {
        printf(" %i\t\t\t", i+1);
        printf("%i\t\t\t", processSize[i]);
        if (allocation[i] != -1)
            printf("%i", allocation[i] + 1);
        else
            printf("Not Allocated");
        printf("\n");
    }
}

int main()
{
    int m;
    int n;
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    m = sizeof(blockSize) / sizeof(blockSize[0]);
    n = sizeof(processSize) / sizeof(processSize[0]);
    firstFit(blockSize, m, processSize, n);
    return 0 ;
}
```

Output:

```
 Process No. Process Size    Block no.
  1          212             2
  2          417             5
  3          112             2
  4          426             Not Allocated
```

Next Fit:

Theory:

Next fit is a memory allocation algorithm used in computer operating systems to allocate memory to processes. It works by starting at the last block of memory that was allocated and searching for the next available block that is large enough to hold the process. If a suitable block is found, the process is allocated to that block, and the remaining space is marked as used. If no suitable block is found, the algorithm starts the search from the beginning of the memory space. The advantage of the next fit algorithm is that it reduces memory fragmentation by avoiding small unused memory blocks between allocated processes. Additionally, it is more efficient than the first fit algorithm because it starts searching from the last allocated block, which is closer to the current position. However, the next fit algorithm can still lead to memory fragmentation, and it is not as efficient as other algorithms, such as the best fit algorithm, which allocates the smallest available block that is large enough to hold the process. Overall, the next fit algorithm is a good compromise between efficiency and memory utilization and is commonly used in modern operating systems.

Code:

```
#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\t\t" << processSize[i]


            << "\t\t\t\t";
        if (allocation[i] != -1)
            cout << allocation[i] + 1;
        else
            cout << "Not Allocated";
        cout << endl;
    }
}
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:

```
 Process No. Process Size    Block no.
  1           10            2
  2           20            3
  3           5             1
```

Best Fit:

Theory:

Best fit is a memory allocation algorithm used in computer operating systems to allocate memory to processes. It works by searching through all the available memory blocks and selecting the smallest block that is large enough to hold the process. This algorithm helps to reduce memory fragmentation and makes better use of the available memory than other algorithms. The advantage of the best fit algorithm is that it allocates the smallest available block that is large enough to hold the process, which reduces memory waste and improves memory utilization. Additionally, it reduces memory fragmentation by using smaller blocks of memory, which are easier to fill. However, the best fit algorithm can be slower than other algorithms because it requires searching through all the available memory blocks to find the best fit. Additionally, it can lead to external fragmentation if small unused memory blocks are left between allocated processes. Overall, the best fit algorithm is a good choice for systems with large amounts of memory, as it makes better use of the available memory and reduces memory waste.

Code:

```
#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:

```
 Process No. Process Size    Block no.
  1       212     4
  2       417     2
  3       112     3
  4       426     5
```

Worst Fit:

Theory:

Worst fit is a memory allocation algorithm used in computer operating systems to allocate memory to processes. It works by searching through all the available memory blocks and selecting the largest block of memory that is not sufficient to hold the process. This algorithm results in the largest amount of external fragmentation and reduces memory utilization. The worst fit algorithm is generally not used in modern operating systems because of its inefficient memory management. This algorithm leads to large amounts of external fragmentation, where there are many small unused memory blocks scattered throughout the memory. It also results in low memory utilization because it leaves large blocks of memory unused. Overall, worst fit is not a good choice for systems with limited memory resources because it wastes too much memory and leads to fragmentation. Instead, other algorithms like first fit, best fit, or next fit are preferred for their more efficient memory management techniques.

Code:

```
#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:

```
 Process No. Process Size    Block no.
  1      212     5
  2      417     2
  3      112     5
  4      426     Not Allocated
```

Conclusion :

Page Replacement

Optimal algorithm selects the page that will not be used for the longest time in future, while FIFO algorithm selects the page that arrived first in memory. LRU algorithm selects the page that has not been accessed for the longest period of time. Optimal algorithm has the highest accuracy, while FIFO algorithm is the simplest to implement. LRU algorithm performs well in most scenarios and is able to adapt to changing access patterns. These algorithms are used in modern operating systems to manage virtual memory and improve system performance. Optimal algorithm is commonly used in research studies to evaluate the performance of other algorithms. FIFO algorithm is used in embedded systems with limited memory resources. LRU algorithm is used in high-performance systems like databases and caches.

Memory Allocation

Best fit searches through all the available memory blocks and selects the smallest block that is large enough to hold the process. This algorithm reduces memory waste and improves memory utilization. First fit searches for the first available block that is large enough to hold the process. This algorithm is fast and efficient but can lead to memory fragmentation. Next fit is similar to first fit, but it starts searching from the last allocated block instead of the beginning. This algorithm reduces fragmentation but can still leave small unused blocks. Worst fit selects the largest available block of memory that is not sufficient to hold the process. This algorithm leads to high fragmentation and low memory utilization. The choice of algorithm depends on the specific needs of the system. Best fit is preferred for systems with large amounts of memory, while first fit and next fit are good for systems with limited memory resources. Worst fit is generally not used in modern operating systems because of its inefficient memory management.