Virtual Memory Management

A Mini Project Report

Submitted By

Arnav Agarwal	200905188
Prakhar Choudhary	200905380
Prodeep Ghosh	200905384
Aditya Ballabh	200905386

In partial fulfilment for the award of the degree of
Bachelor's in Technology (B. Tech)
IN
Computer Science & Engineering

MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

INSPIRED BY LIFE

Department of Computer Science & Engineering

Department of Computer Science & Engineering

BONAFIDE CERTIFICATE

Certified that this project report "**Virtual Memory Management"** is the bonafide work of:

Arnav Agarwal	200905188
Prakhar Choudhary	200905380
Prodeep Ghosh	200905384
Aditya Ballabh	200905386

who carried out the mini project work under my supervision.

Dr. Ashalatha Nayak
Head of Department, CSE

Sucharitha Shetty
Supervisor

Submitted to the Viva voce Examination held on

____

EXAMINER 1

EXAMINER 2

INTRODUCTION

Virtual memory is a technique that abstracts main memory into a large, uniform array of storage, separating the logical memory perceived by the user from the physical memory. This frees the programmer from concerns of storage limitations and allows the execution of processes that are not completely in memory. It also allows processes to easily share files and implement shared memory.

Paging involves breaking physical memory into fixed-sized blocks called frames. Logical memory is also split into blocks of the same size called pages. When a process is executed, its pages are brought from their source to the available frames. Paging suffers from internal fragmentation but avoids external fragmentation and hence does not require compaction.

A commonly used approach for paging is demand paging, where pages are loaded only when they are needed during execution. The pager swaps in only those pages are likely to be needed and replaces pages from secondary memory in case of page faults, i.e., if the referenced page is not in the main memory.

If there are no free frames when a page fault occurs, a page replacement algorithm is used to choose a frame that is not being used currently and free it. The freed frame can now be used to hold the page for which the process faulted. Global replacement allows the process to select a replacement frame from the set of all frames, whereas local replacement requires that each process select from only its own set of allocated frames.

OVERVIEW

This implementation of virtual memory management uses pure demand paging with an inverted page table, along with global replacement. The page table is represented by a vector containing pairs of the form {pid, page_num}. Several page replacement algorithms such as OPT, LRU, MRU, LIFO and FIFO have been implemented. All of these internally use a list to keep track of the pages that are currently in main memory, along with a map for faster lookup and deletion.

PROJECT LINK

https://drive.google.com/drive/folders/1qOwmEcm0fTu17Ftzj1VcUvucnUdvKYcW

CODE SNIPPETS

```
void OPT::refer(vector<pair<int,int>>&page_table, pair<int,int> page_ref, int req_num,
vector<pair<int,int>>&page_refs){
  if(mem_page_locs.find(page_ref)==mem_page_locs.end()){
    page_fault_cnt++;
    if(mem_pages.size()==page_table_size){
      int n=page_refs.size();
      set<pair<int,int>>cur_pages;
      for(const pair<int,int> &key:mem_pages)
        cur_pages.insert(key);

      if(cur_pages.size()!=1){
        for(int i=req_num+1;i<n;i++){
          if(cur_pages.find(page_refs[i])!=cur_pages.end())
            cur_pages.erase(page_refs[i]);
          if(cur_pages.size()==1)
            break;
        }
      }
      for(int i=0;i<page_table_size;i++)
        if(page_table[i]==*cur_pages.begin()){
          page_table[i]=page_ref;
          break;
        }

      mem_pages.erase(mem_page_locs[*cur_pages.begin()]);
      mem_page_locs.erase(*cur_pages.begin());
    }
    else{
      int free_frame=findFreeFrame(page_table);
      page_table[free_frame]=page_ref;
    }
    mem_pages.push_front(page_ref);
    mem_page_locs.insert({page_ref, mem_pages.begin()});
  }
}
```

```
void LRU::refer(vector<pair<int,int>>&page_table, pair<int,int> page_ref){
  if(mem_page_locs.find(page_ref)!=mem_page_locs.end())
     mem_pages.erase(mem_page_locs[page_ref]);
  else{
     page_fault_cnt++;

     if(mem_pages.size()==page_table_size){
        for(int i=0;i<page_table_size;i++)
           if(page_table[i]==mem_pages.back()){
              page_table[i]=page_ref;
              break;
           }
        mem_page_locs.erase(mem_pages.back());
        mem_pages.pop_back();
     }
     else{
        int free_frame=findFreeFrame(page_table);
        page_table[free_frame]=page_ref;
     }
  }
  mem_pages.push_front(page_ref);
  mem_page_locs[page_ref]=mem_pages.begin();
}
```

```
void MRU::refer(vector<pair<int,int>>&page_table, pair<int,int> page_ref){
  if(mem_page_locs.find(page_ref)!=mem_page_locs.end())
     mem_pages.erase(mem_page_locs[page_ref]);
  else{
     page_fault_cnt++;
     if(mem_pages.size()==page_table_size){
        for(int i=0;i<page_table_size;i++)
           if(page_table[i]==mem_pages.front()){
              page_table[i]=page_ref;
              break;
           }
        mem_page_locs.erase(mem_pages.front());
        mem_pages.pop_front();
     }
```

```
    else{
        int free_frame=findFreeFrame(page_table);
        page_table[free_frame]=page_ref;
    }
  }
  mem_pages.push_front(page_ref);
  mem_page_locs[page_ref]=mem_pages.begin();
}
}
```

```
void LIFO::refer(vector<pair<int,int>>&page_table, pair<int,int>page_ref){
  if (mem_pages_set.find(page_ref)==mem_pages_set.end()){
    page_fault_cnt++;
    if (mem_pages_set.size() == page_table_size){
        pair<int,int> del_ref = mem_pages.front();
        mem_pages.pop_front();
        mem_pages_set.erase(del_ref);

        for(int i=0;i<page_table_size;i++)
            if(page_table[i]==del_ref){
                page_table[i]=page_ref;
                break;
            }
    }
    else{
        int free_frame=findFreeFrame(page_table);
        page_table[free_frame]=page_ref;
    }
    mem_pages_set.insert(page_ref);
    mem_pages.push_front(page_ref);
  }
```

```
void FIFO::refer(vector<pair<int,int>>&page_table, pair<int,int>page_ref){
  if (mem_pages_set.find(page_ref)==mem_pages_set.end()){
    page_fault_cnt++;
    if (mem_pages_set.size() == page_table_size){
        pair<int,int> del_ref = mem_pages.front();
        mem_pages.pop_front();
```

```
        mem_pages_set.erase(del_ref);

        for(int i=0;i<page_table_size;i++)
          if(page_table[i]==del_ref){
            page_table[i]=page_ref;
            break;
          }
    }
    else{
      int free_frame=findFreeFrame(page_table);
      page_table[free_frame]=page_ref;
    }
    mem_pages_set.insert(page_ref);
    mem_pages.push_back(page_ref);
  }
}
```

```
int findFreeFrame(vector<pair<int,int>>page_table){
  for(int i=0;i<page_table.size();i++)
    if(page_table[i].first==-1)
      return i;
  return -1;
}
```

SAMPLE INPUT

```
Enter the number of frames
2
Enter the number of page references
5
Enter the pid and page number for each reference
0 1
0 0
0 3
0 1
0 3
```

RESULT

1. OPT

```
Page Reference #1:
0,1

List:
(0,1) -> NULL

Page Table:

       pid       page_num
        0               1
       -1              -1

---------------

Page Reference #2:
0,0

List:
(0,0) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               1
        0               0

---------------

Page Reference #3:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               1
        0               3
```

```
Page Reference #4:
0,1

List:
(0,3) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               1
        0               3

---------------

Page Reference #5:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               1
        0               3

---------------

OPT Stats:

Page Faults:    3
Page Hits:      2
Miss Ratio:     0.6
```

2. LRU

```
Page Reference #1:
0,1

List:
(0,1) -> NULL

Page Table:

       pid       page_num
        0               1
       -1              -1

---------------

Page Reference #2:
0,0

List:
(0,0) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               1
        0               0

---------------

Page Reference #3:
0,3

List:
(0,3) -> (0,0) -> NULL

Page Table:

       pid       page_num
        0               3
        0               0
```

```
Page Reference #4:
0,1

List:
(0,1) -> (0,3) -> NULL

Page Table:

       pid       page_num
        0               3
        0               1

---------------

Page Reference #5:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               3
        0               1

---------------

LRU Stats:

Page Faults:    4
Page Hits:      1
Miss Ratio:     0.8
```

3. MRU

```
Page Reference #1:
0,1

List:
(0,1) -> NULL

Page Table:

        pid         page_num
          0                1
         -1               -1

---------------

Page Reference #2:
0,0

List:
(0,0) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                0

---------------

Page Reference #3:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                3
```

```
Page Reference #4:
0,1

List:
(0,1) -> (0,3) -> NULL

Page Table:

        pid         page_num
          0                1
          0                3

---------------

Page Reference #5:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                3

---------------

MRU Stats:

Page Faults:    3
Page Hits:      2
Miss Ratio:     0.6
```

4. LIFO

```
Page Reference #1:
0,1

List:
(0,1) -> NULL

Page Table:

        pid         page_num
          0                1
         -1               -1

---------------

Page Reference #2:
0,0

List:
(0,0) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                0

---------------

Page Reference #3:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                3
```

```
Page Reference #4:
0,1

List:
(0,3) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                3

---------------

Page Reference #5:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

        pid         page_num
          0                1
          0                3

---------------

LIFO Stats:

Page Faults:    3
Page Hits:      2
Miss Ratio:     0.6
```

5. FIFO

```
Page Reference #1:
0,1

List:
(0,1) -> NULL

Page Table:

       pid       page_num
        0               1
       -1              -1

---------------

Page Reference #2:
0,0

List:
(0,1) -> (0,0) -> NULL

Page Table:

       pid       page_num
        0               1
        0               0

---------------

Page Reference #3:
0,3

List:
(0,0) -> (0,3) -> NULL

Page Table:

       pid       page_num
        0               3
        0               0
```

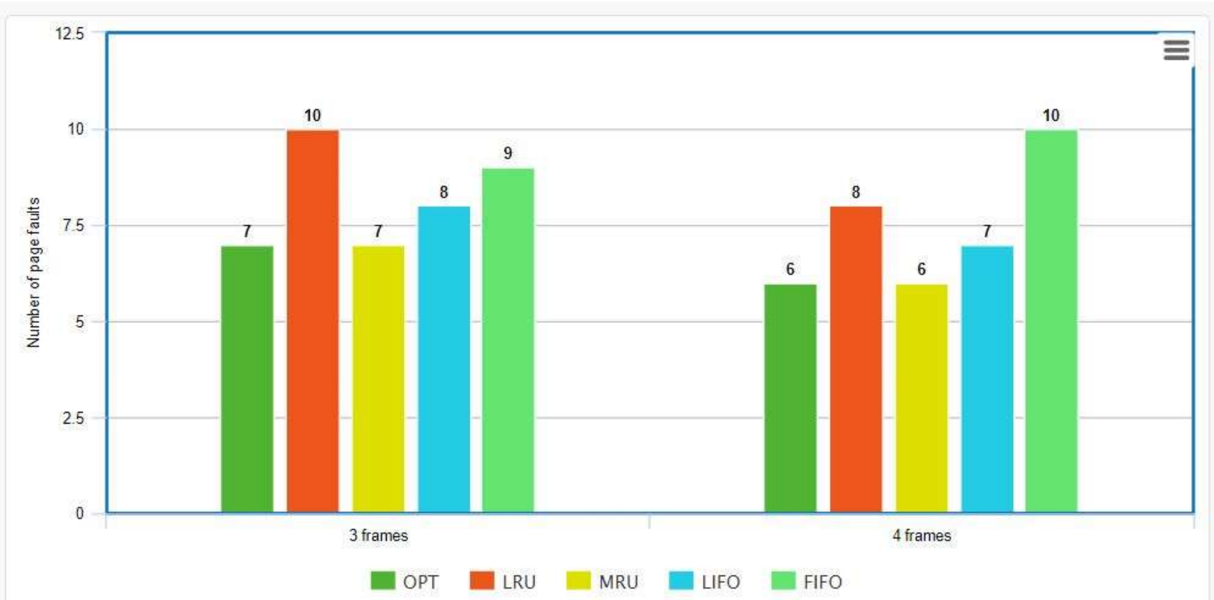
```
Page Reference #4:
0,1

List:
(0,3) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               3
        0               1

---------------

Page Reference #5:
0,3

List:
(0,3) -> (0,1) -> NULL

Page Table:

       pid       page_num
        0               3
        0               1

---------------

FIFO Stats:

Page Faults:    4
Page Hits:      1
Miss Ratio:     0.8
```

CONCLUSION

OPT consistently produces the lowest page fault rate possible, since it knows in advance which page will be requested last. This limitation prevents it from being used in practice, but it still serves as a benchmark against which other replacement algorithms can be compared.

FIFO exhibits Belady's anomaly on some reference strings i.e., an increase in the number of frames can lead to an increase in the number of page faults. OPT, LRU, MRU and LIFO do not exhibit Belady's anomaly on any reference string since they belong to the class of stack algorithms i.e., the set of pages in memory for n frames will always be a subset of the pages that would be in memory if there were n+1 frames.

Additionally, OPT and LRU produce the same page fault rate if the reference string is reversed.



Reference String: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

REFERENCES

1. Operating System Concepts (9e) - Silberschatz, Galvin, Gagne
2. https://www.geeksforgeeks.org/page-replacement-algorithms-in-operating-systems/