Name: Pratyush Kumar
Reg. No.: 19BCE0506
Slot: L41+L42
Course: CSE2005, Operating Systems

LAB ASSIGNMENT – 4

1.

Aim:

To implement Fitting algorithms (First fit, Best fit and Worst Fit) in Fixed(or Static) Partition mechanism

Algorithm:

Static (or fixed) Partioning

a. Number of partitions are fixed
b. Size of each partition may or may not be the same
c. Contigious allocation, hence, no SPANNING is allowed

Partition Allocation

1. Input memory blocks with size and processes with size.
2. Initialize all memory blocks as free.
3. Start by picking each process and check if it can be assigned to current block.
4. If size-of-process <= size-of-block if yes then assign and check for next process.
5. If not then keep checking the further blocks.
6. Calculate total internal fragmentation by subtracting the block size by process size
7. External fragmentation can be calculated by counting the number of blocks that have not been used.

Code:

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

typedef struct process{
        int psize;
        int pflag;
}process;

typedef struct block{
        int bsize;
        int bflag;
}block;

void accept(process p[],block b[],int *n,int *m){

        int i, j;
```

```
	printf("\n Enter no. of blocks : ");
	scanf("%d", m);
	for(i = 0; i < *m;i++){
		printf(" Enter size of block[%d] : ",i);
		scanf("%d", &b[i].bsize);
	}

	printf("\n Enter no. of processes : ");
	scanf("%d", n);
	for(i = 0; i < *n; i++){
		printf(" Enter size of block[%d] : ",i);
		scanf("%d", &p[i].psize);
	}

}
void re_init(process p[],block b[],int n,int m){
	int i;
	for(i = 0; i < n; i++)
		p[i].pflag = 0;
	for(i = 0;i < m; i++)
		b[i].bflag = 0;
}

void first_fit(process p[],block b[],int n,int m){

	int i, j, in_frag = 0, ex_frag = 0;
	for(i = 0; i < n; i++){
		for(j = 0;j < m; j++){
			if(p[i].psize <= b[j].bsize && b[j].bflag == 0 && p[i].pflag == 0){
				b[j].bflag = p[i].pflag = 1;
				in_frag += b[j].bsize - p[i].psize;
				printf("\n P[%d]\t-\tB[%d]",i,j);
				break;
			}
		}
		if(p[i].pflag == 0)
			printf("\n P[%d]\t-\tUnassigned",i);
	}
	printf("\n\n Total internal fragmentation : %d", in_frag );
	for(j=0;j<m;j++){
		if(b[j].bflag == 0)
			ex_frag += b[j].bsize;
	}
	printf("\n Total external fragmentation : %d", ex_frag );
}

void best_fit(process p[],block b[],int n,int m){
	int i, j, min_frag, in_frag = 0, ex_frag=0, id;
	for(i = 0; i < n; i++){
		min_frag = 9999;
		id = 9999;
```

```
		for(j = 0; j < m; j++){
			if(p[i].psize <= b[j].bsize && b[j].bflag == 0 && min_frag > (b[j].bsize - p[i].psize)){
					min_frag = b[j].bsize - p[i].psize;
					id = j;
			}
		}
		if(min_frag != 9999){
				b[id].bflag = p[i].pflag = 1;
				in_frag += b[id].bsize - p[i].psize;
				printf("\n P[%d]\t-\tB[%d]",i,id);
		}
		if(p[i].pflag == 0)
			printf("\n P[%d]\t-\tUnassigned",i);
	}

	printf("\n\n Total internal fragmentation : %d", in_frag );
	for(j = 0; j < m; j++){
		if(b[j].bflag == 0)
			ex_frag += b[j].bsize;
	}
	printf("\n Total external fragmentation : %d", ex_frag );


}

void worst_fit(process p[],block b[],int n,int m){
	int i, j, max_frag,in_frag=0,ex_frag=0,id;

	for(i=0;i<n;i++){
		max_frag = -1;
		id=9999;
		for(j=0;j<m;j++){
			if(p[i].psize <= b[j].bsize && b[j].bflag == 0 && max_frag < (b[j].bsize - p[i].psize)){
					max_frag = b[j].bsize - p[i].psize;
					id = j;
			}
		}
		if(max_frag != -1){
				b[id].bflag = p[i].pflag = 1;
				in_frag += b[id].bsize - p[i].psize;
				printf("\n P[%d]\t-\tB[%d]",i,id);
		}
		if(p[i].pflag == 0)
			printf("\n P[%d]\t-\tUnassigned",i);
	}

	printf("\n\n Total internal fragmentation : %d",in_frag );
	for(j=0;j<m;j++){
		if(b[j].bflag == 0)
			ex_frag+=b[j].bsize;
```

```
        }
        printf("\n Total external fragmentation : %d",ex_frag );


}


int main(){

        int ch, n = 0,m = 0;
        process p[10];
        block b[10];
        do{
                printf("\n\n SIMULATION OF FIXED PARTITIONING MEMORY
MANAGEMENT SCHEMES");
                printf("\n\n Options:");
                printf("\n 0. Accept");
                printf("\n 1. First Fit");
                printf("\n 2. Best Fit");
                printf("\n 3. Worst Fit");
                printf("\n 4. Best Algo");
                printf("\n Select : ");
                scanf("%d",&ch);
                switch(ch){
                        case 0:
                                accept(p,b,&n,&m);
                                break;
                        case 1:
                                re_init(p,b,n,m);
                                first_fit(p,b,n,m);
                                break;
                        case 2:
                                re_init(p,b,n,m);
                                best_fit(p,b,n,m);
                                break;
                        case 3:
                                re_init(p,b,n,m);
                                next_fit(p,b,n,m);
                                break;
                        case 4:
                                printf("Best algorithm is Best Fit\n");
            break;
                        default:
                                printf("\n Invalid selection.");
                }
        }while(ch < 5);
   return 0;
}
```

Output:

```
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ gcc mem_mng.c
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ ./mem_mng


 SIMULATION OF FIXED PARTITIONING MEMORY MANAGEMENT SCHEMES

 Options:
 0. Accept
 1. First Fit
 2. Best Fit
 3. Worst Fit
 4. Best Algo
 Select : 0

 Enter no. of blocks : 5
 Enter size of block[0] : 5
 Enter size of block[1] : 4
 Enter size of block[2] : 3
 Enter size of block[3] : 6
 Enter size of block[4] : 7

 Enter no. of processes : 4
 Enter size of block[0] : 1
 Enter size of block[1] : 3
 Enter size of block[2] : 5
 Enter size of block[3] : 3


 SIMULATION OF FIXED PARTITIONING MEMORY MANAGEMENT SCHEMES

 Options:
 0. Accept
 1. First Fit
 2. Best Fit
 3. Worst Fit
 4. Best Algo
 Select : 1

 P[0]   -       B[0]
 P[1]   -       B[1]
 P[2]   -       B[3]
 P[3]   -       B[2]

 Total internal fragmentation : 6
 Total external fragmentation : 7

 SIMULATION OF FIXED PARTITIONING MEMORY MANAGEMENT SCHEMES

 Options:
 0. Accept
 1. First Fit
 2. Best Fit
 3. Worst Fit
 4. Best Algo
 Select : 2

 P[0]   -       B[2]
 P[1]   -       B[1]
```

```
P[2]   -       B[0]
P[3]   -       B[3]

Total internal fragmentation : 6
Total external fragmentation : 7

SIMULATION OF FIXED PARTITIONING MEMORY MANAGEMENT SCHEMES

Options:
0. Accept
1. First Fit
2. Best Fit
3. Worst Fit
4. Best Algo
Select : 3

P[0]   -       B[4]
P[1]   -       B[3]
P[2]   -       B[0]
P[3]   -       B[1]

Total internal fragmentation : 10
Total external fragmentation : 3

SIMULATION OF FIXED PARTITIONING MEMORY MANAGEMENT SCHEMES

Options:
0. Accept
1. First Fit
2. Best Fit
3. Worst Fit
4. Best Algo
Select : 4
Best algorithm is Best Fit


SIMULATION OF FIXED PARTITIONING MEMORY MANAGEMENT SCHEMES

Options:
0. Accept
1. First Fit
2. Best Fit
3. Worst Fit
4. Best Algo
Select : 5

Invalid selection.pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$
```

2.

Aim:

To implement Page Replacement algorithms (FIFO, LRU, Optimal), find number of page faults for each page replacement algorithm and find the best algorithm for a given reference string.

Algorithm:

1. **FIFO**

Here, capacity = n (number of pages)

Implementation – Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.
i) If set holds less pages than capacity.
a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
b) Simultaneously maintain the pages in the queue to perform FIFO.
c) Increment page fault
ii) Else
If current page is present in set, do nothing.
Else
a) Remove the first page from the queue as it was the first to be entered in the memory
b) Replace the first page in the queue with the current page in the string.
c) Store current page in the queue.
d) Increment page faults.

2. Return page faults.

2. LRU

Here, capacity = n (number of pages)

Let capacity be the number of pages that memory can hold. Let set be the current set of pages in memory.

1- Start traversing the pages.
i) If set holds less pages than capacity.
a) Insert page into the set one by one until the size of set reaches capacity or all page requests are processed.
b) Simultaneously maintain the recent occurred

index of each page in a map called indexes.
c) Increment page fault
ii) Else
If current page is present in set, do nothing.
Else
a) Find the page in the set that was least
recently used. We find it using index array.
We basically need to replace the page with
minimum index.
b) Replace the found page with current page.
c) Increment page faults.
d) Update index of current page.

2. Return page faults.

3. OPTIMAL Algo

```
Start
Step 1-> In function int predict(int page[], vector<int>& fr, int pn, int index)
  Declare and initialize res = -1, farthest = index
  Loop For i = 0 and  i < fr.size() and i++
    Loop For j = index and j < pn and j++
      If fr[i] == page[j] then,
        If j > farthest
          Set farthest = j
        End If
        Set res = i
        break
      If j == pn then,
      Return i
  Return (res == -1) ? 0 : res
Step 2-> In function bool search(int key, vector<int>& fr)
  Loop For i = 0 and i < fr.size() and i++
  If fr[i] == key then,
    Return true
    Return false
Step 3-> In function void opr(int page[], int pn, int fn)
  Declare vector<int> fr
  Set hit = 0
  Loop For i = 0 and i < pn and i++
  If search(page[i], fr) then,
    Increment hit by 1
    continue
  If fr.size() < fn then,
    fr.push_back(page[i])
  Else
    Set j = predict(page, fr, pn, i + 1)
    Set fr[j] = page[i]
    Print the number of hits
    Print the number of misses
Step 4-> In function int main()
  Declare and assign page[] = { 1, 7, 8, 3, 0, 2, 0, 3, 5, 4, 0, 6, 1 }
```

```
    Set pn = sizeof(page) / sizeof(page[0])
    Set fn = 3
    opr(page, pn, fn)
  Stop
```

CODE:

```
#include <iostream>
#include <vector>
#include <queue>
#include <unordered_set>
#include <unordered_map>
#include <limits.h>


using namespace std;


int FIFO_pageFaults(vector<int> pages, int n, int capacity)
{
   // To represent set of current pages. We use
   // an unordered_set so that we quickly check
   // if a page is present in set or not
   unordered_set<int> s;

   // To store the pages in FIFO manner
   queue<int> indexes;

   // Start from initial page
   int page_faults = 0;
   for (int i=0; i<n; i++)
   {
      // Check if the set can hold more pages
      if (s.size() < capacity)
      {
         // Insert it into set if not present
         // already which represents page fault
         if (s.find(pages[i])==s.end())
         {
            // Insert the current page into the set
            s.insert(pages[i]);

            // increment page fault
            page_faults++;

            // Push the current page into the queue
            indexes.push(pages[i]);
         }
      }

      // If the set is full then need to perform FIFO
      // i.e. remove the first page of the queue from
```

```
        // set and queue both and insert the current page
        else
        {
            // Check if current page is not already
            // present in the set
            if (s.find(pages[i]) == s.end())
            {
                // Store the first page in the
                // queue to be used to find and
                // erase the page from the set
                int val = indexes.front();

                // Pop the first page from the queue
                indexes.pop();

                // Remove the indexes page from the set
                s.erase(val);

                // insert the current page in the set
                s.insert(pages[i]);

                // push the current page into
                // the queue
                indexes.push(pages[i]);

                // Increment page faults
                page_faults++;
            }
        }
    }

    return page_faults;
}

int LRU_pageFaults(vector<int> pages, int n, int capacity)
{
    // To represent set of current pages. We use
    // an unordered_set so that we quickly check
    // if a page is present in set or not
    unordered_set<int> s;

    // To store least recently used indexes
    // of pages.
    unordered_map<int, int> indexes;

    // Start from initial page
    int page_faults = 0;
    for (int i=0; i<n; i++)
    {
        // Check if the set can hold more pages
        if (s.size() < capacity)
        {
```

```
        // Insert it into set if not present
        // already which represents page fault
        if (s.find(pages[i])==s.end())
        {
            s.insert(pages[i]);

            // increment page fault
            page_faults++;
        }

        // Store the recently used index of
        // each page
        indexes[pages[i]] = i;
    }

    // If the set is full then need to perform lru
    // i.e. remove the least recently used page
    // and insert the current page
    else
    {
        // Check if current page is not already
        // present in the set
        if (s.find(pages[i]) == s.end())
        {
            // Find the least recently used pages
            // that is present in the set
            int lru = INT_MAX, val;
            for (auto it=s.begin(); it!=s.end(); it++)
            {
                if (indexes[*it] < lru)
                {
                    lru = indexes[*it];
                    val = *it;
                }
            }

            // Remove the indexes page
            s.erase(val);

            // insert the current page
            s.insert(pages[i]);

            // Increment page faults
            page_faults++;
        }

        // Update the current page index
        indexes[pages[i]] = i;
    }
}

return page_faults;
```

```
}

int OPTIMAL_pagefaults(vector<int> pages, int n,int frame_size)
{
   vector<int> fr;
   int page_faults = 0;
   for (int i = 0; i < n; i++)
   {
      int k;
      for (k = 0; k < fr.size(); k++)
         if (fr[k] == pages[i])
            break;
      if (k==fr.size())
      {
         if (fr.size() < frame_size)
            fr.push_back(pages[i]);

         else
         {
            int index=i+1;
            int res = -1, farthest = index;
            for (int l = 0; l < fr.size(); l++)
            {
               int j;
               for (j = index; j < n; j++)
               {
                  if (fr[l] == pages[j])
                  {
                     if (j > farthest)
                     {
                        farthest = j;
                        res = l;
                     }
                     break;
                  }
               }
               if (j == n)
               {
                  res=l;
                  break;
               }
            }
            fr[res] = pages[i];
         }
         page_faults++;
      }
   }
   return page_faults;
}
```

```
int main() {
   int choice, input, n, nof; //nof = number of frames
   vector<int> ref_string;
   int page_faults = 0;
   int FIFO_page_faults = 0, LRU_page_faults = 0, OPTIMAL_page_faults = 0;
   cout << "Enter number of elements in the reference string";
   cin >> n;

   cout << "Enter reference string";
   for(int i = 0; i < n; i++) {
      cin >> input;
      ref_string.push_back(input);
   }

   cout << "Enter number of free frames available";
   cin >> nof;

   do{
      cout << "Choose one Page Replacement algorithm program\n";
      cout << "1. FIFO (First In First Out)\n";
      cout << "2. LRU (Least Recently Used)\n";
      cout << "3. Optimal Algortihm\n";
      cout << "4. Best Algorithm";
      cin >> choice;
      switch (choice)
      {
      case 1:
         page_faults = FIFO_pageFaults(ref_string, n, nof);
         cout << "Number of page faults in FIFO is: " << page_faults << "\n";
         break;

      case 2:
         page_faults = LRU_pageFaults(ref_string, n ,nof);
         cout << "Number of page faults in LRU is: " << page_faults << "\n";
         break;

      case 3:
         page_faults = OPTIMAL_pagefaults(ref_string, n, nof);
         cout << "Number of page faults in Optimal algo is: " << page_faults << "\n";
         break;

      case 4:
         cout << "Best algorithm\n";
         FIFO_page_faults = FIFO_pageFaults(ref_string, n, nof);
         LRU_page_faults = LRU_pageFaults(ref_string, n ,nof);
         OPTIMAL_page_faults = OPTIMAL_pagefaults(ref_string, n, nof);
         if(FIFO_page_faults < LRU_page_faults && FIFO_page_faults < OPTIMAL_page_faults)
            cout << "FIFO\n";
                     else if (LRU_page_faults < FIFO_page_faults && LRU_page_faults <
OPTIMAL_page_faults)
            cout << "LRU\n";
         else
```

```
            cout << "Optimal algorithm\n";
            break;

        default:
            cout << "Invalid selection\n";
            break;
        }
    }while(choice < 5);


    return 0;
}
```

OUTPUT:

```
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ g++ page_replacement.cpp
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ ./page_replacement
Enter number of elements in the reference string
20
Enter reference string
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
Enter number of free frames available
4
Choose one Page Replacement algorithm program
1. FIFO (First In First Out)
2. LRU (Least Recently Used)
3. Optimal Algortihm
4. Best Algorithm
1
Number of page faults in FIFO is: 10
Choose one Page Replacement algorithm program
1. FIFO (First In First Out)
2. LRU (Least Recently Used)
3. Optimal Algortihm
4. Best Algorithm
2
Number of page faults in LRU is: 8
Choose one Page Replacement algorithm program
1. FIFO (First In First Out)
2. LRU (Least Recently Used)
3. Optimal Algortihm
4. Best Algorithm
3
Number of page faults in Optimal algo is: 8
Choose one Page Replacement algorithm program
1. FIFO (First In First Out)
2. LRU (Least Recently Used)
3. Optimal Algortihm
4. Best Algorithm
4
Best algorithm
Optimal algorithm
Choose one Page Replacement algorithm program
1. FIFO (First In First Out)
2. LRU (Least Recently Used)
3. Optimal Algortihm
4. Best Algorithm
5
Invalid selection
pratyush@pratyush-Inspiron-5570:~/Desktop/Pratyush$ flameshot gui
```