5. Page Replacement Algorithms

Aim:

To understand and implement page replacement algorithms - FIFO, LRU, OPR

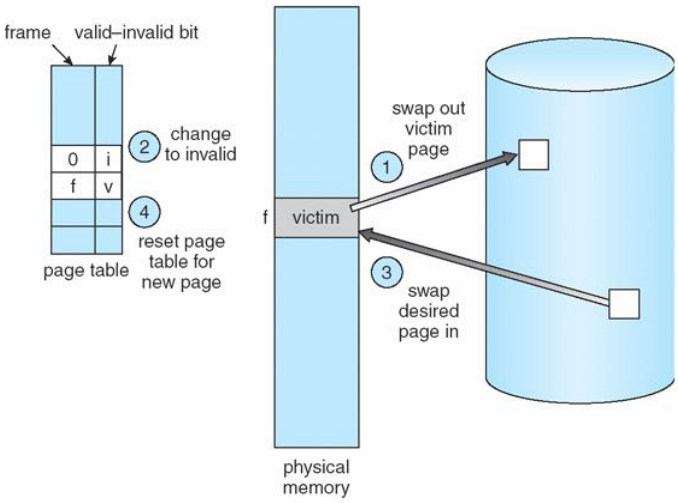
Theory:

In computer operating systems, paging is a memory management scheme by which a computer stores and retrieves data from secondary storage for use in main memory. In this scheme, the operating system retrieves data from secondary storage pages.
The most important terms related to paging are as follows:

1. **Page Hit** - When a process demands a page and that page is already residing in the cache, then this event is referred to as a page hit.
2. **Page Fault** - When a process demands a page and that page is not available in the cache, then this event is called as a page fault. A page fault requires that one of the pages residing in the cache be replaced (using a page replacement algorithm) to accomodate this newly demanded page.
3. **Page Stream** - A page stream denotes the sequence in which a process is to demand the pages from cache.
4. **Page Replacement Algorithms** - In OSs that use paging, page replacement algorithms decide which memory pages to page out, or write to disk, when a page of memory needs to be allocated.

The efficiency of the various algorithms is calculated in terms of hit ratio, which is defined as follows:

Hit Ratio = No. of page hits / size of page stream



1. First In First Out (FIFO) Algorithm

This is the simplest page replacement algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

The FIFO algorithm is said to suffer from Belady's anomaly. Belady's Anomaly is the phenomenon in which increasing the number of page frames results in an increase in the number of page faults for certain memory access patterns. This phenomenon is commonly experienced when using the first-in first-out (FIFO) page replacement algorithm. In FIFO, the page fault may or may not increase as the page frames increase, but in Optimal and stack-based algorithms like LRU, as the page frames increase the page fault decreases.

The basic outline of the FIFO algorithm is as follows:

```
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.
```

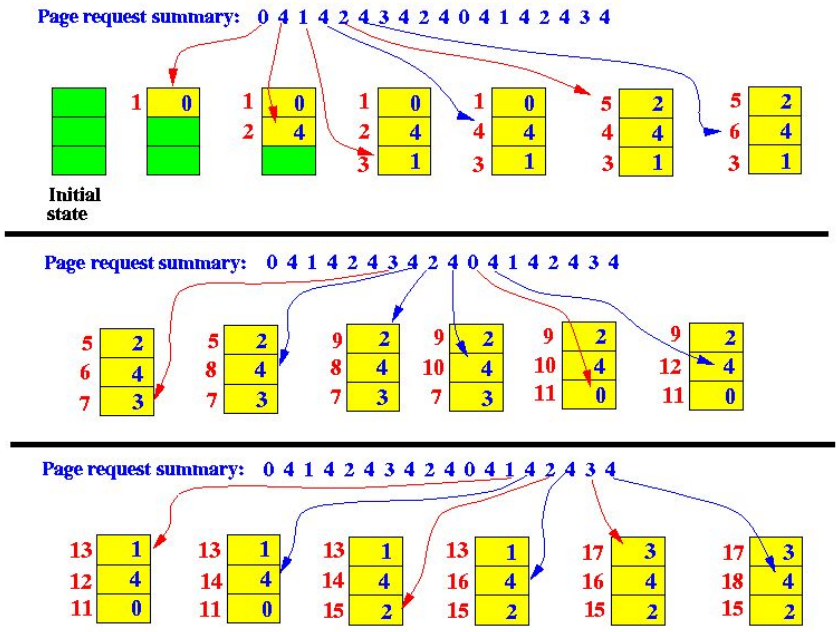
Example:

0	2	1	6	4	0	1	0	3	1	2	1
0	0	0	0	4	4			4	4	2	
	2	2	2	2	0		hit	0	0	0	
		1	1	1	1	hit		3	3	3	
			6	6	6			6	1	1	hit

2. Least Recently Used (LRU) Algorithm

Least **R**ecently **U**sed (LRU) algorithm is a Greedy algorithm where the page to be replaced is least recently used. The idea is based on locality of reference, the least recently used page is not likely to be reused again. The general outline of the LRU algorithm is as follows:

```
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.
```



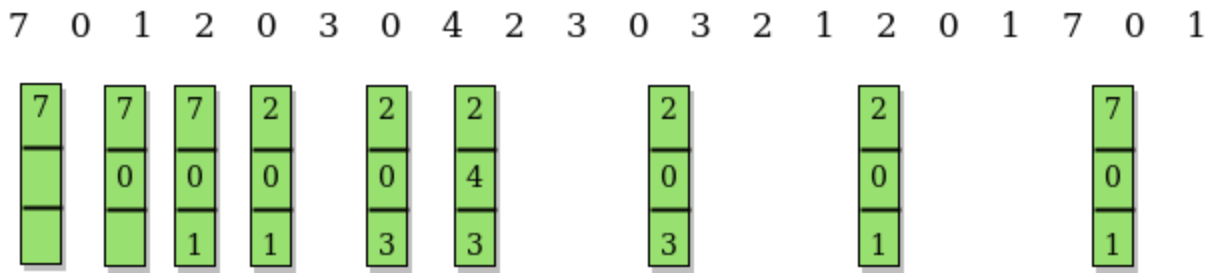
3. Optimal Page Replacement (OPR) Algorithm

In operating systems, whenever a new page is referred and not present in memory, page fault occurs and Operating System replaces one of the existing pages with newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce number of page faults.

In this algorithm, OS replaces the page that will not be used for the longest period of time in future.

The idea is simple, for every reference we do following :

- If referred page is already present, increment hit count.
- If not present, find if a page that is never referenced in future. If such a page exists, replace this page with new page. If no such page exists, find a page that is referenced farthest in future. Replace this page with new page.



Program:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

/**
 * A Java program to demonstrate the various
 * page replacement algorithms
 *
 * @author jimil
 */
public class Paging {

	public static final int MAX_FRAME_SIZE = 3;
	public static int tick;
	public static List<Page> mainMemory = new ArrayList<Page>();
	public static List<Page> availablePages = new ArrayList<Page>();
	public static List<Integer> pageStream = new ArrayList<Integer>();
	public static int streamSize;
	public static int numberOfPages;
	public static int hitCount;
	public static int faultCount;

	/**
	* Driver function for program
	*
	* @param args
	*/
	public static void main(String[] args) throws Exception {
	tick = 0;
	BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
	System.out.println("Enter the size of the page stream: ");
	streamSize = Integer.parseInt(br.readLine());
	System.out.println("Enter the number of pages: ");
	numberOfPages = Integer.parseInt(br.readLine());

	for(int i = 0; i < numberOfPages; i++) {
		Page page = new Page();
		page.setId(i);
```

```
            availablePages.add(page);
      }

      System.out.println("Enter the page stream: ");
      for(int i = 0; i < streamSize; i++) {
            pageStream.add(Integer.parseInt(br.readLine()));
      }

      System.out.println("Select the page replacement policy to be
implemented: ");
      System.out.println("1.    FIFO");
      System.out.println("2.    LRU");
      System.out.println("3.    Optimal Page Replacement");
      int choice = Integer.parseInt(br.readLine());
      switch(choice) {
            case 1: fifo();
                  break;
            case 2: lru();
                  break;
            case 3: optimalPageReplacement();
                  break;
            default: System.out.println("Enter a valid choice!");
                  break;
      }
      }

      /**
      * Function to demonstrate the action of
      * processor demanding a page from the cache.
      *
      * Can also be utilized to search for page
      * in main memory.
      *
      * @param pageNo
      * @return
      */
      public static boolean demand(int pageNo) {
      for(int i = 0; i < MAX_FRAME_SIZE; i++) {
            if(!mainMemory.isEmpty() && i < mainMemory.size()) {
                  Page p = mainMemory.get(i);
                  if(p != null && p.getId() == pageNo) {
                  return true;
```

```
                }
            }
        }
        return false;
        }

        /**
        * Method that implements page replacement using fifo.
        */
        public static void fifo() {
        for(int pageNo: pageStream) {
            tick++;
            boolean result = demand(pageNo);
            if(result) {
                hitCount++;
                System.out.println("HIT! Page " + pageNo + " residing in
cache!");
            } else {
                if(mainMemory.size() < MAX_FRAME_SIZE) {
                faultCount++;
                availablePages.get(pageNo).setStartTick(tick);
                System.out.println("Main Memory is empty! Adding page: "
+ pageNo);
                mainMemory.add(availablePages.get(pageNo));
                } else {
                faultCount++;
                int toReplacePage = findFirstIn();
                System.out.println("PAGE FAULT! Replacing page: " +
mainMemory.get(toReplacePage).getId());
                mainMemory.remove(toReplacePage);
                availablePages.get(pageNo).setStartTick(tick);
                mainMemory.add(toReplacePage,
availablePages.get(pageNo));
                }
            }
            System.out.println("Current Main Memory Status: ");
            for(Page p: mainMemory) {
                System.out.print(p.getId() + ", ");
            }
            System.out.println();
        }
        System.out.println("OUTPUT STATS: ");
```

```
	System.out.println("Hit Ratio = " +
((float)hitCount/(float)streamSize));
	System.out.println("Fault Count = " + faultCount);
	System.out.println("Hit Count = " + hitCount);
	}

	/**
	* Method that implements page replacement using the
	* LRU algorithm
	*/
	public static void lru() {
	int currentIndex = -1;
	for(int pageNo: pageStream) {
		currentIndex++;
		boolean result = demand(pageNo);
		if(result) {
			hitCount++;
			System.out.println("HIT! Page " + pageNo + " residing in
cache!");
		} else {
			if(mainMemory.size() < MAX_FRAME_SIZE) {
			faultCount++;
			availablePages.get(pageNo).setStartTick(tick);
			System.out.println("Main Memory is empty! Adding page: "
+ pageNo);
			mainMemory.add(availablePages.get(pageNo));
			} else {
			faultCount++;
			int toReplacePage = findLeastRecentlyUsed(currentIndex);
			if(toReplacePage >= 0) {
				System.out.println("PAGE FAULT! Replacing page: " +
mainMemory.get(toReplacePage).getId());
				mainMemory.remove(toReplacePage);
				availablePages.get(pageNo).setStartTick(tick);
				mainMemory.add(toReplacePage,
availablePages.get(pageNo));
			}
			}
		}
		System.out.println("Current Main Memory Status: ");
		for(Page p: mainMemory) {
			System.out.print(p.getId() + ", ");
```

```
            }
            System.out.println();
      }
      System.out.println("OUTPUT STATS: ");
      System.out.println("Hit Ratio = " +
((float)hitCount/(float)streamSize));
      System.out.println("Fault Count = " + faultCount);
      System.out.println("Hit Count = " + hitCount);
      }

      /**
      * Method that implements paging and uses optimal
      * page replacement strategy to handle page faults.
      */
      public static void optimalPageReplacement() {
      int currentIndex = -1;
      for(int pageNo: pageStream) {
            currentIndex++;
            boolean result = demand(pageNo);
            if(result) {
                  hitCount++;
                  System.out.println("HIT! Page " + pageNo + " residing in
cache!");
            } else {
                  if(mainMemory.size() < MAX_FRAME_SIZE) {
                  faultCount++;
                  System.out.println("Main Memory is empty! Adding page: "
+ pageNo);
                  mainMemory.add(availablePages.get(pageNo));
                  } else {
                  faultCount++;
                  int toReplacePage =
toReplaceUsingOptSolution(currentIndex);
                  if(toReplacePage >= 0) {
                        System.out.println("PAGE FAULT! Replacing page: " +
mainMemory.get(toReplacePage).getId());
                        mainMemory.remove(toReplacePage);
                        mainMemory.add(toReplacePage,
availablePages.get(pageNo));
                  }
                  }
            }
```

```
            System.out.println("Current Main Memory Status: ");
            for(Page p: mainMemory) {
                  System.out.print(p.getId() + ", ");
            }
            System.out.println();
      }
      System.out.println("OUTPUT STATS: ");
      System.out.println("Hit Ratio = " +
((float)hitCount/(float)streamSize));
      System.out.println("Fault Count = " + faultCount);
      System.out.println("Hit Count = " + hitCount);
      }

      /**
      * Returns the index of the page in main memory
      * that arrived first.
      *
      * @return
      */
      public static int findFirstIn() {
      int index = 0;
      int min = mainMemory.get(index).getStartTick();
      for(int i = 1; i < MAX_FRAME_SIZE; i++) {
            if(mainMemory.get(i).getStartTick() < min) {
                  index = i;
                  min = mainMemory.get(i).getStartTick();
            }
      }
      return index;
      }

      /**
      * Returns the index of the page in main memory
      * that is least recently used.
      *
      * Backtracks the page stream upto MAX_FRAME_SIZE
      * steps and returns the index of that page in main
      * memory.
      *
      * @param index
      * @return
      */
```

```
	public static int findLeastRecentlyUsed(int index) {
	List<Integer> stack = new ArrayList<Integer>();
	int count = 0;
	for(int i = index-1; i >= 0; i--) {
		if(!stack.contains(pageStream.get(i))) {
			stack.add(pageStream.get(i));
			count++;
		}
		if(count == MAX_FRAME_SIZE) {
			break;
		}
	}
	int pageId = (int)stack.get(stack.size() - 1);
	int returnIndex = -10;
	for(Page p: mainMemory) {
		if(p.getId() == pageId) {
			returnIndex = mainMemory.indexOf(p);
			break;
		}
	}
	return returnIndex;
	}

	/**
	* Returns the index of the page that is not likely to
	* be demanded in the near future.
	*
	* This is done by traversing the input page stream
	* from current index upto MAX_FRAME_SIZE steps and
	* comparing with available pages in memory to find out
	* the ones that are not going to be demanded in
	* the near future.
	*
	* @param index
	* @return
	*/
	public static int toReplaceUsingOptSolution(int index) {
	List<Integer> list = new ArrayList<>();
	List<Integer> pageIds = new ArrayList<>();
	int count = 0;
	for(int i = index+1; i < pageStream.size(); i++) {
		if(!list.contains(pageStream.get(i))) {
```

```
                int pageId = pageStream.get(i);
                for(Page p: mainMemory) {
                if(!pageIds.contains(p.getId())) {
                      pageIds.add(p.getId());
                }
                if (p.getId() == pageId) {
                      list.add(p.getId());
                      count++;
                      break;
                }
                }
                if(count == MAX_FRAME_SIZE) {
                break;
                }
          }
      }
      pageIds.removeAll(list);
      int returnIndex = -10;
      if(pageIds.size() >= 1) {
            int pageId = pageIds.get(pageIds.size() - 1);
            for(Page p: mainMemory) {
                  if(p.getId() == pageId) {
                  returnIndex = mainMemory.indexOf(p);
                  break;
                  }
            }
      } else {
            returnIndex = MAX_FRAME_SIZE-1;
      }
      return returnIndex;
      }
}

/**
 * A POJO that denotes a simple page in the
 * system.
 *
 * @author jimil
 */
class Page {
      private int id;
      private int useCount;
```

```
	private int startTick;
	private int frequency;
	public int getId() {
	return id;
	}
	public void setId(int id) {
	this.id = id;
	}
	public int getUseCount() {
	return useCount;
	}
	public void setUseCount(int useCount) {
	this.useCount = useCount;
	}
	public int getStartTick() {
	return startTick;
	}
	public void setStartTick(int startTick) {
	this.startTick = startTick;
	}
	public int getFrequency() {
	return frequency;
	}
	public void setFrequency(int frequency) {
	this.frequency = frequency;
	}
}
```

Output:

FIFO Algorithm:

```
Enter the size of the page stream: 12
Enter the number of pages: 6
Enter the page stream: 0 1 2 3 1 0 4 5 1 0 1 2
Select the page replacement policy to be implemented:
1.    FIFO
2.    LRU
3.    Optimal Page Replacement
1
Main Memory is empty! Adding page: 0
Current Main Memory Status:
0,
Main Memory is empty! Adding page: 1
Current Main Memory Status:
0, 1,
Main Memory is empty! Adding page: 2
Current Main Memory Status:
0, 1, 2,
PAGE FAULT! Replacing page: 0
Current Main Memory Status:
3, 1, 2,
HIT! Page 1 residing in cache!
Current Main Memory Status:
3, 1, 2,
PAGE FAULT! Replacing page: 1
Current Main Memory Status:
3, 0, 2,
PAGE FAULT! Replacing page: 2
Current Main Memory Status:
3, 0, 4,
PAGE FAULT! Replacing page: 3
Current Main Memory Status:
5, 0, 4,
PAGE FAULT! Replacing page: 0
Current Main Memory Status:
5, 1, 4,
PAGE FAULT! Replacing page: 4
Current Main Memory Status:
5, 1, 0,
HIT! Page 1 residing in cache!
```

```
Current Main Memory Status:
5, 1, 0,
PAGE FAULT! Replacing page: 5
Current Main Memory Status:
2, 1, 0,
OUTPUT STATS:
Hit Ratio = 0.16666667
Fault Count = 10
Hit Count = 2
```

LRU Algorithm:

```
Enter the size of the page stream: 12
Enter the number of pages: 6
Enter the page stream: 0 1 2 3 1 0 4 5 1 0 1 2
Select the page replacement policy to be implemented:
1.    FIFO
2.    LRU
3.    Optimal Page Replacement
2
Main Memory is empty! Adding page: 0
Current Main Memory Status:
0,
Main Memory is empty! Adding page: 1
Current Main Memory Status:
0, 1,
Main Memory is empty! Adding page: 2
Current Main Memory Status:
0, 1, 2,
PAGE FAULT! Replacing page: 0
Current Main Memory Status:
3, 1, 2,
HIT! Page 1 residing in cache!
Current Main Memory Status:
3, 1, 2,
PAGE FAULT! Replacing page: 2
Current Main Memory Status:
3, 1, 0,
PAGE FAULT! Replacing page: 3
Current Main Memory Status:
4, 1, 0,
PAGE FAULT! Replacing page: 1
```

```
Current Main Memory Status:
4, 5, 0,
PAGE FAULT! Replacing page: 0
Current Main Memory Status:
4, 5, 1,
PAGE FAULT! Replacing page: 4
Current Main Memory Status:
0, 5, 1,
HIT! Page 1 residing in cache!
Current Main Memory Status:
0, 5, 1,
PAGE FAULT! Replacing page: 5
Current Main Memory Status:
0, 2, 1,
OUTPUT STATS:
Hit Ratio = 0.16666667
Fault Count = 10
Hit Count = 2
```

Optimal Page Replacement:

```
Enter the size of the page stream: 12
Enter the number of pages: 6
Enter the page stream: 0 1 2 3 1 0 4 5 1 0 1 2
Select the page replacement policy to be implemented:
1.    FIFO
2.    LRU
3.    Optimal Page Replacement
3
Main Memory is empty! Adding page: 0
Current Main Memory Status:
0,
Main Memory is empty! Adding page: 1
Current Main Memory Status:
0, 1,
Main Memory is empty! Adding page: 2
Current Main Memory Status:
0, 1, 2,
PAGE FAULT! Replacing page: 2
Current Main Memory Status:
0, 1, 3,
HIT! Page 1 residing in cache!
```

```
Current Main Memory Status:
0, 1, 3,
HIT! Page 0 residing in cache!
Current Main Memory Status:
0, 1, 3,
PAGE FAULT! Replacing page: 3
Current Main Memory Status:
0, 1, 4,
PAGE FAULT! Replacing page: 4
Current Main Memory Status:
0, 1, 5,
HIT! Page 1 residing in cache!
Current Main Memory Status:
0, 1, 5,
HIT! Page 0 residing in cache!
Current Main Memory Status:
0, 1, 5,
HIT! Page 1 residing in cache!
Current Main Memory Status:
0, 1, 5,
PAGE FAULT! Replacing page: 5
Current Main Memory Status:
0, 1, 2,
OUTPUT STATS:
Hit Ratio = 0.41666666
Fault Count = 7
Hit Count = 5
```

Conclusion:

Thus, I studied and understood the various concepts of paging and implemented three paging replacement algorithms -FIFO, LRU, and OPR. It was observed that FIFO suffered from Belady's anomaly, but optimal algorithms like LRU and OPR gave better hit ratios. Also, for the given page sequence and frame size, it was observed that OPR gave the best hit ratio.