CSE312
Operating Systems

Homework 2 | 2021 - 2022

Ahmet Tuğkan Ayhan

1901042692

Global Variables

```c
#define TABLE_SIZE 5
// Page list to test with the algorithm
char pages[] = "dgjhj";
int  size = sizeof(pages) - 1;
// Disk & Page Table Represantation
char page_table[TABLE_SIZE];
// Other variables
int first_empty_page = 0;            // Shows index of the first empty place in page table
// Result variables
int hit          = 0;
int page_fault   = 0;
int pages_loaded = 0;
int disk_written = 0;
```

I will start my report by explaining global variables and define values.

- **TABLE_SIZE =** Table size shows the how many pages can fit into page table. An upper limit let's say.
- **pages[] =** Pages are actually char values need to be sorted. Name 'pages' a bit misleading but, these values are stored in disk. This is our disk.
- **size =** sizeof(pages) value takes into account '\0' character which indicates end of the array. But since we are concerned about the size of the disk, we will calculate it by removing 1 from sizeof(pages).
- **page_table =** Page table is our table which we store pages. This page_table stored in our RAM. When we use page algorithms, we actually move pages from pages[] array to page_table array(from disk to array) or vice versa.
- **first_empty_page =** First empty page, as it name implies, shows first index that is empty on the page_table. So we can use it while there are empty spaces on page_table. Once every place is occupied, then this value is set to -1 and so we can understand that, our page_table is full so we need to get rid of one of the pages.
- **hit =** Stores number of hits performed during the sorting algorithms. Hit value incremented when a page is exist in the page_table.
- **page_fault =** Stores number of page faults performed during the sorting algorithms. Page fault occurs when a page doesn't exist in the page_table
- **pages_loaded =** Number of pages loaded. It is incremented every time we call a page replacement algorithm. So it is actually the sum of both hit + page_fault

- **disk_written =** Number of pages written back to the disk. This value incremented every time we get rid of a page to replace it with the new page.

Sorting Algorithms

Bubble Sort

```
171   void bubbleSort() {
172       printArray("Before Bubble Sort : ");
173       int temp;
174       for(int i=0; i<size; i++) {
175           for(int j=0; j<size-(i+1); j++) {
176               lru           (pages[j]);       pages_loaded++;
177               lru           (pages[j+1]);     pages_loaded++;
178               //fifo          (pages[j]);       pages_loaded++;
179               //fifo          (pages[j+1]);     pages_loaded++;
180               //secondChance (pages[j]);       pages_loaded++;
181               //secondChance (pages[j+1]);     pages_loaded++;
182               if(pages[j] > pages[j+1]) {
183                   temp        = pages[j];
184                   pages[j]    = pages[j+1];
185                   pages[j+1] = temp;
186               }
187           }
188       }
189       printArray("After Bubble Sort  : ");
190       printResult();
191   }
```

Bubble sort algorithm is easy to implement. All you have to do is control elements adjacent to each other and if element with the big index is smaller than the low index, than swap them. Do this until every element with big index is bigger than the low index. Bubble sort function doesn't get any parameter because I assigned these functions to keyboard. And when you assign them to keyboard, you can't use any parameters on that function.

After explaining the bubble sort, lets talk about lines between 176-181. These lines shows which paging algorithm will be called. pages array indicates the disk like a said in the Global Variables section. We are trying to access disk value at pages[j] and pages[j+1]. So we have to put these two values into our page table. Everytime one of these functions are called I also incremented the pages_loaded value by 1. After placing pages[j] and pages[j+1] to page_table (to memory) we are ready to swap them. We will not write these swapped values back to disk until new page takes the place of one of these pages.

Insertion Sort

```
void insertionSort() {
    printArray("Before Insertion Sort : ");
    int page, j;
    for(int i=1; i<size; i++) {
        lru           (pages[i]);      pages_loaded++;
        //fifo          (pages[i]);      pages_loaded++;
        //secondChance (pages[i]);      pages_loaded++;
        page = pages[i];
        j = i - 1;
        while(page < pages[j] && j >= 0) {
            lru           (pages[j]);      pages_loaded++;
            lru           (pages[j+1]);    pages_loaded++;
            //fifo          (pages[j]);      pages_loaded++;
            //fifo          (pages[j+1]);    pages_loaded++;
            //secondChance (pages[j]);      pages_loaded++;
            //secondChance (pages[j+1]);    pages_loaded++;
            pages[j+1] = pages[j];
            j = j-1;
        }
        lru           (pages[i]);      pages_loaded++;
        //fifo          (pages[i]);      pages_loaded++;
        //secondChance (pages[i]);      pages_loaded++;
        pages[j+1] = page;
    }
    printArray("After Insertion Sort  : ");
    printResult();
}
```

Insertion sort says that, start from index 0, take an element and move it to the left until there is no element that is lower than this element and repeat this until everything is sorted.

Different from bubble sort, we have much more call for a page replacement algorithm. Rule is simple: if any line contains any pages[] value, before using it, call a page replacement algorithm for that page. This is what I did here and in other sorting algorithms. printArray function prints the current status of the disk and printResult function prints the hit/miss/ values, etc.

Quick Sort

```
void insertionSort() {⋯

void quickSort() {
    printArray("Before Quick Sort : ");
    int low = 0;
    int high = size - 1;
    quickSortStart(low, high);
    printArray("After Quick Sort  : ");
    printResult();
}

void quickSortStart(int low, int high) {
    if(low < high) {
        int pivot = partition(low, high);

        quickSortStart(low, pivot - 1);
        quickSortStart(pivot + 1, high);
    }
}

int partition(int low, int high) {⋯
```

Partition function is very long so I didn’t put the whole code of it here but all it does partitioning the pages according to pivot value. Different from other sorting algorithms, this sorting algorithm calls a page replacement algorithm A LOT. Bubble sort calls 2 times, Insertion Sort calls 4 times and Quick sort calls 6 times. 3 times more than Bubble sort.

Page Replacement Algorithms

FIFO (First In First Out)

```c
323 int nextIndex = 0;
324 void fifo(char page) {
325     printTable(page);
326     int index = 0;
327     // First loop all the page table to find given page
328     for(int i=0; i<TABLE_SIZE; i++) {
329         // Increment hit and reference count by one if the page is present and return
330         if(page_table[i] == page) {
331             hit++;
332             reference_count[i]++;
333             printTable(page);
334             return;
335         }
336     }
337     // If the page is not present in page_table, look if there is an empty place
338     if(first_empty_page != -1) {
339         page_table[first_empty_page] = page;    // Insert in the first empty space
340         page_fault++;                           // Increment the page fault by 1
341         if(first_empty_page == (TABLE_SIZE-1))  // If this is the last empty space at the page table
342             first_empty_page = -1;              // Assign first empty page to -1
343         else
344             first_empty_page++;                 // Else, increment the first_empty_page by 1
345     }
346     else {
347         index = nextIndex % TABLE_SIZE;
348         page_table[index] = page;
349         nextIndex++;
350         disk_written++;
351         page_fault++;                 // Incrementing the page fault by 1
352     }
353     printTable(page);
354 }
355
```

There are 3 situations in FIFO page replacement algorithm. These are:

1. Finding the page in the page_table before retrieving it from disk : Lines : 328-336
2. If page is not in the page_table, then looking for empty space in page_table to put new page from the disk to this place(Lines: 338-345). There could be 2 options now:
    a. **There are more than one empty space in the page table (Lines: 341-342)**
    b. **There is only one space left in the page table (Lines: 343-344)**
3. If page table is full, then according to “nextIndex” value defined in line 323 remov,ng the page with the longest staying time. (Lines: 346-352)

Second Chance

```
356 int chance_table[TABLE_SIZE] = {0};
357 void secondChance(char page) {
358     // First try to find page in the page table
359     for(int i=0; i<TABLE_SIZE; i++) {
360         // Increment hit and reference count by one if the page is present and return
361         if(page_table[i] == page) {
362             hit++;
363             chance_table[i] = 1;        // Set reference value to 1 to give it a second chance
364             printTable(page);
365             return;
366         }
367     }
368     // If page not found in the page table, then check if the table is empty or not
369     if(first_empty_page != -1) {
370         page_table[first_empty_page] = page;     // Insert in the first empty space
371         page_fault++;                            // Increment the page fault by 1
372         chance_table[first_empty_page] = 0;      // Set page with 0 reference point (yet)
373         if(first_empty_page == (TABLE_SIZE-1))   // If this is the last empty space at the page table
374             first_empty_page = -1;               // Assign first empty page to -1
375         else
376             first_empty_page++;                  // Else, increment the first_empty_page by 1
377     }
378     // Else, if page table is not empty and page is not in the page table, then traverse the page table
379     // To find a page with no reference point
380     else {
381         for(int i=0; i<TABLE_SIZE*2; i++) {
382             if(chance_table[i] == 0) {
383                 page_table[i] = page;
384                 printTable(page);
385                 return;
386             }
387             i = i % TABLE_SIZE;
388             chance_table[i] = 0;
389         }
390         page_fault++;
391         disk_written++;
392     }
393     printTable(page);
394 }
395
```

Like FIFO, there are 3 situations in FIFO page replacement algorithm. These are:

1. Finding the page in the page_table before retrieving it from disk : Lines : 359-367
2. If page is not in the page_table, then looking for empty space in page_table to put new page from the disk to this place(Lines: 369-377). There could be 2 options now:
    a. **There are more than one empty space in the page table** (Lines: 373-374)
    b. **There is only one space left in the page table** (Lines: 375-376)
3. If page is not in the page_table and page_table is full, then according to chance_table, find a page with the reference value 0. Each time you came across with a page with reference value 1, then make it value back to 0 and continue

searching. Start over again if you can't find any page with reference 0(this time there will be because we set their reference values back to 0)

LRU (Least Recently Used)

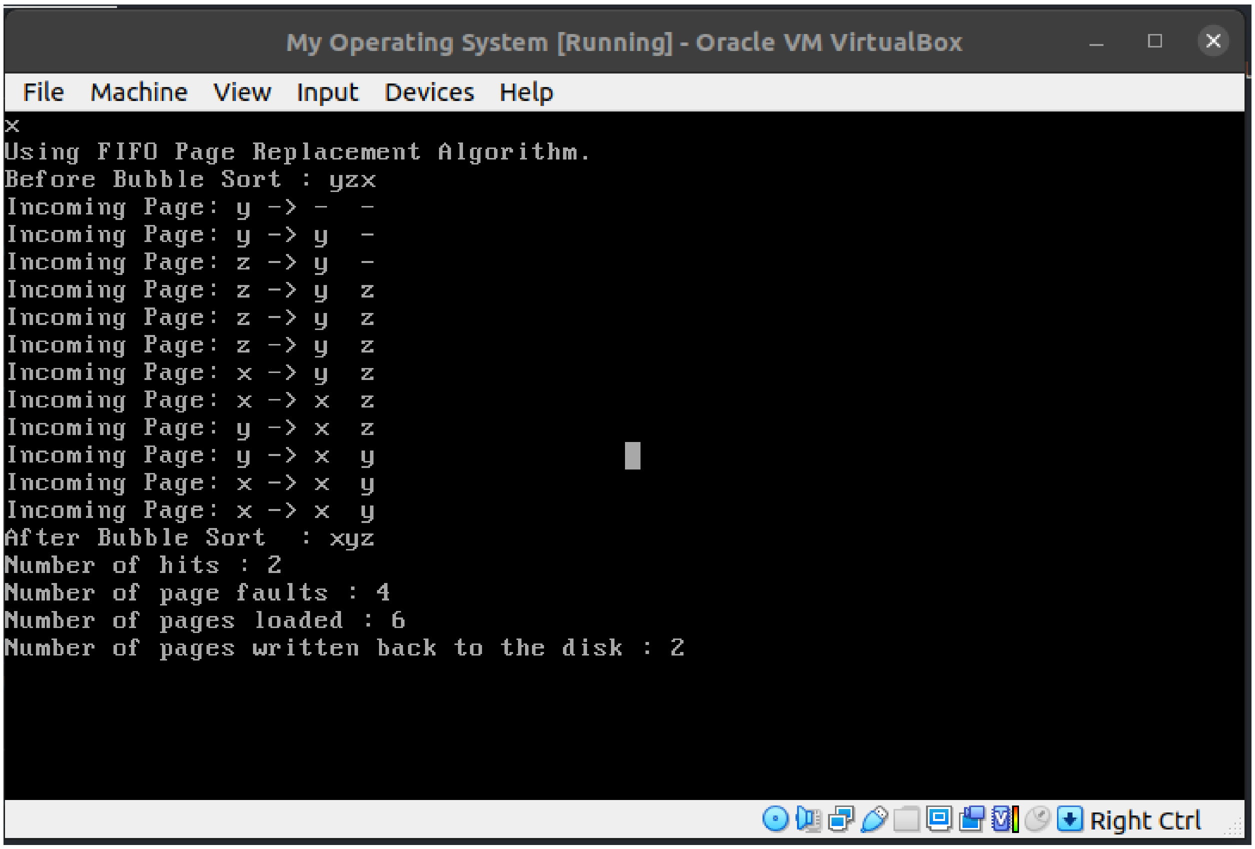
```
int reference_count[TABLE_SIZE] = {0};  // How many times a page is referenced
void lru(char page) {
    // First loop all the page table to find given page
    for(int i=0; i<TABLE_SIZE; i++) {
        // Increment hit and reference count by one if the page is present and return
        if(page_table[i] == page) {
            hit++;
            reference_count[i]++;
            printTable(page);
            return;
        }
    }
    // If the page is not present in page_table, look if there is an empty place
    if(first_empty_page != -1) {
        page_table[first_empty_page] = page;    // Insert in the first empty space
        page_fault++;                           // Increment the page fault by 1
        if(first_empty_page == (TABLE_SIZE-1))  // If this is the last empty space at the page table
            first_empty_page = -1;              // Assign first empty page to -1
        else
            first_empty_page++;                 // Else, increment the first_empty_page by 1
    }
    // If there is no empty space in the page table, then find the page with the least reference
    else {
        // Finding index of the least referenced table
        int maxReference = 10000;
        int index = 0;
        for(int i=0; i<TABLE_SIZE; i++) {
            if(reference_count[i] < maxReference) {
                maxReference = reference_count[i];
                index = i;
            }
        }
        // After finding the index of the least referenced page, we replace it with the new page
        page_table[index] = page;
        page_fault++;               // Incrementing the page fault by 1
        disk_written++;
    }
    printTable(page);
}
```

First 2 steps are same with the other paging algorithms. But last step is different in each of them. For LRU (Least Recently Used) algorithm I can explain it like this:

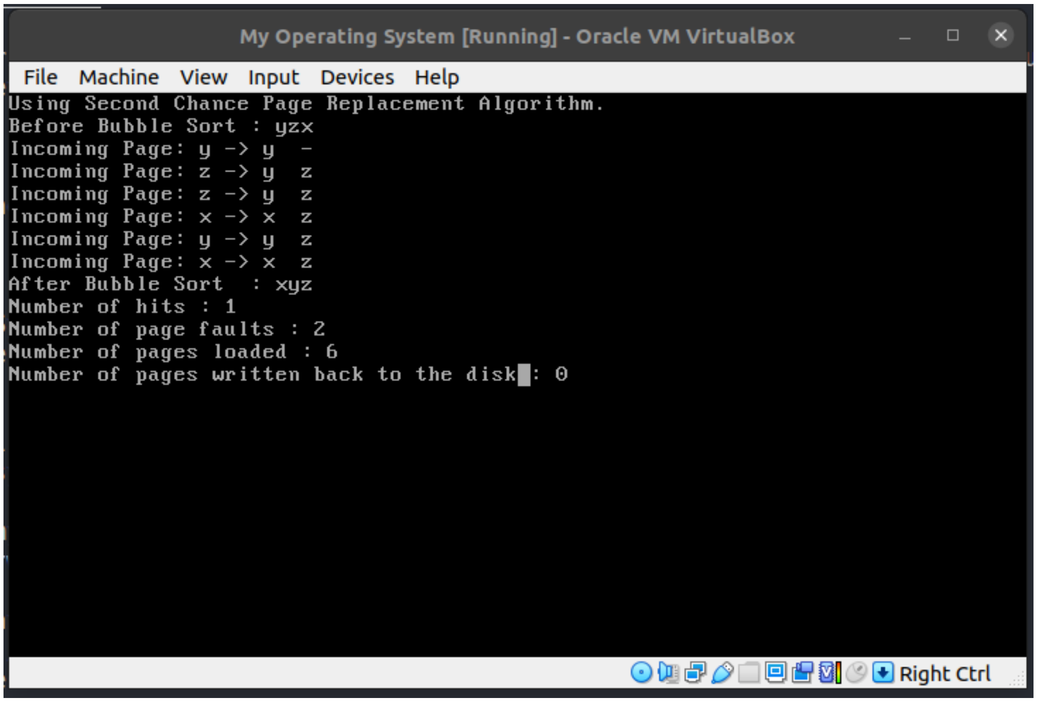
1. First create a value with max reference possible (there is no INF so i used 1000)
2. Then starting from beginning of the page table, looking every page and changing maxReference value if there is any page with reference value less than **maxReference.**
3. After finding page with the least reference value, we can insert our new page into this page's place and increment page_fault and disk_written by 1.

Example Outputs

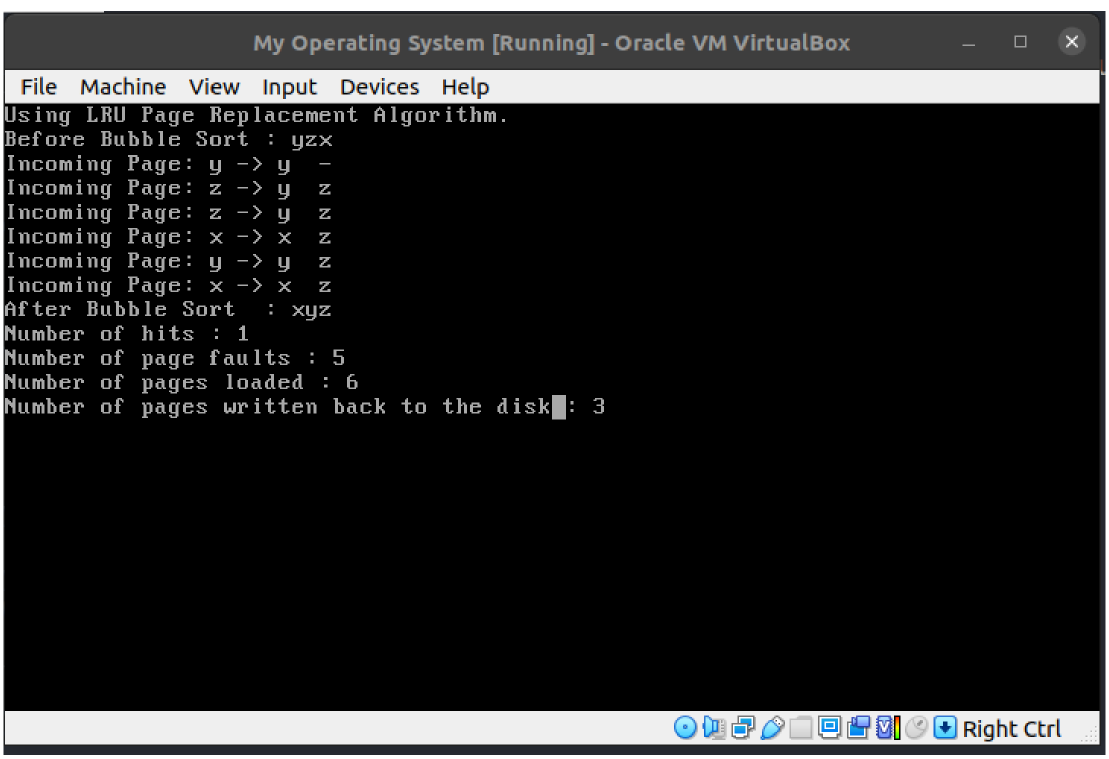
Bubble Sort - FIFO



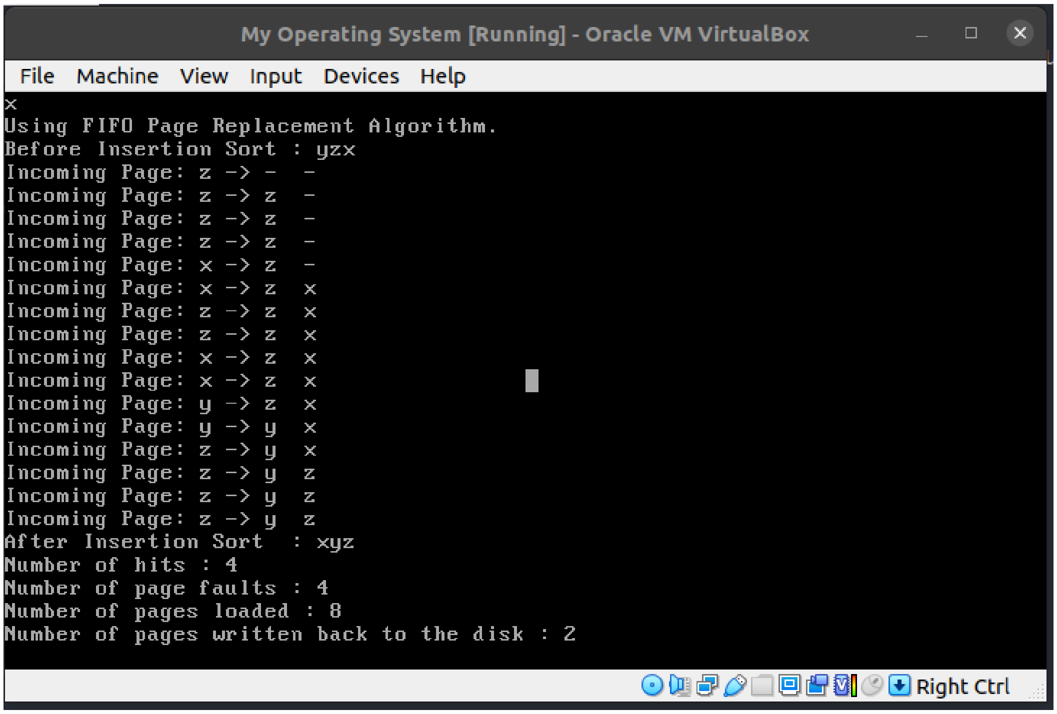
Bubble Sort - Second Chance



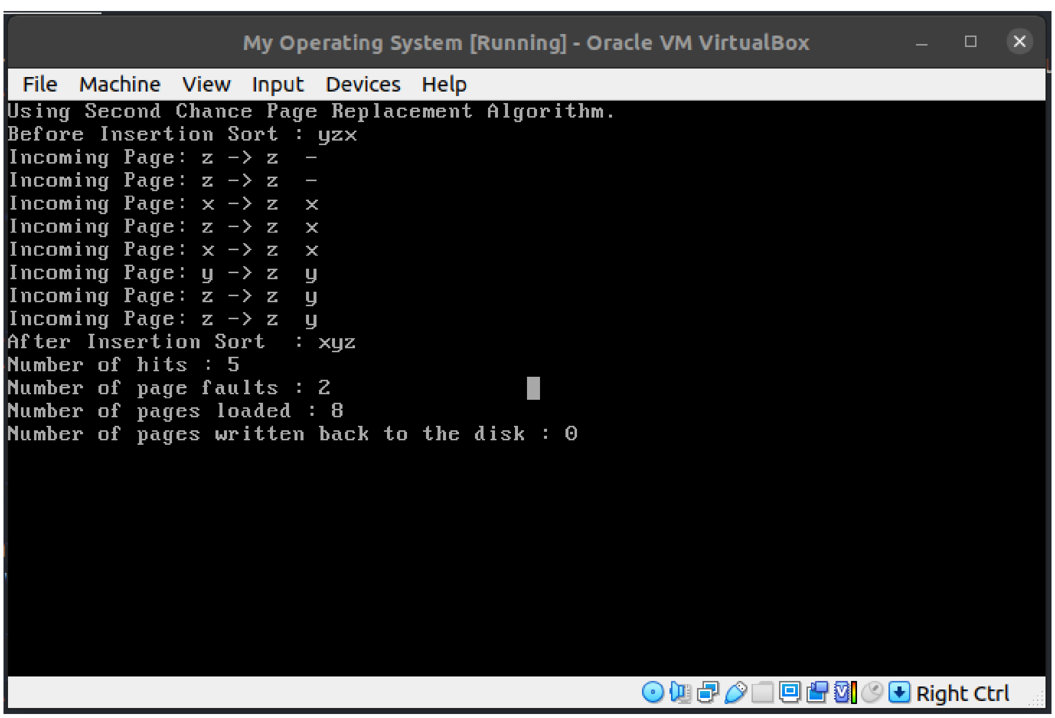
Bubble Sort - LRU



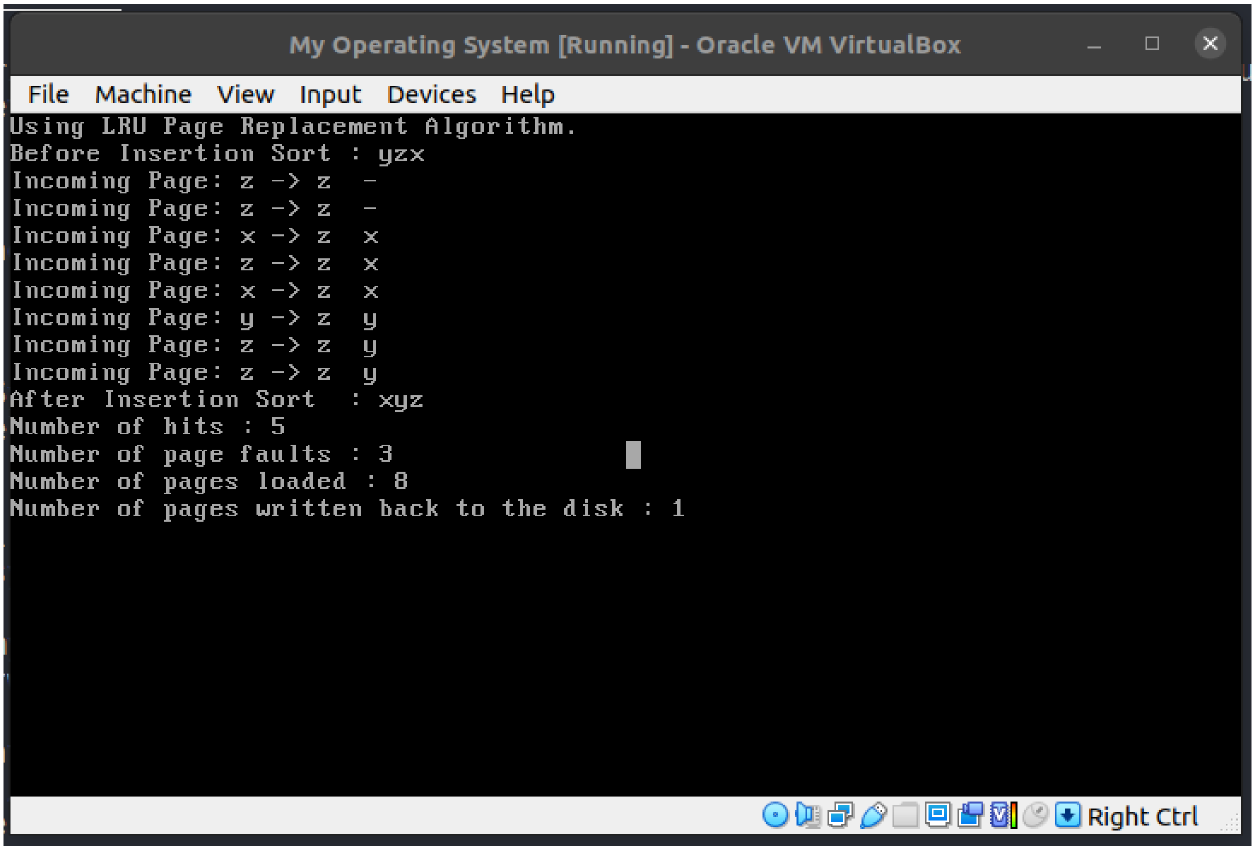
Insertion Sort - FIFO



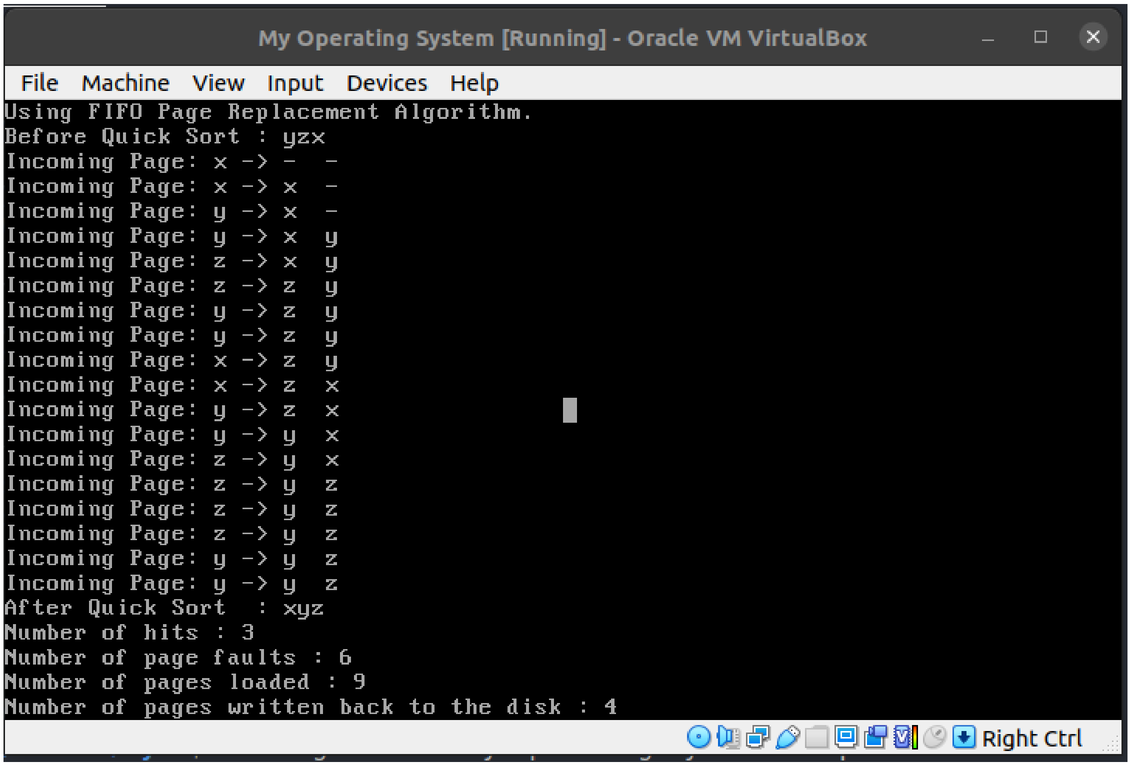
Insertion Sort - Second Chance



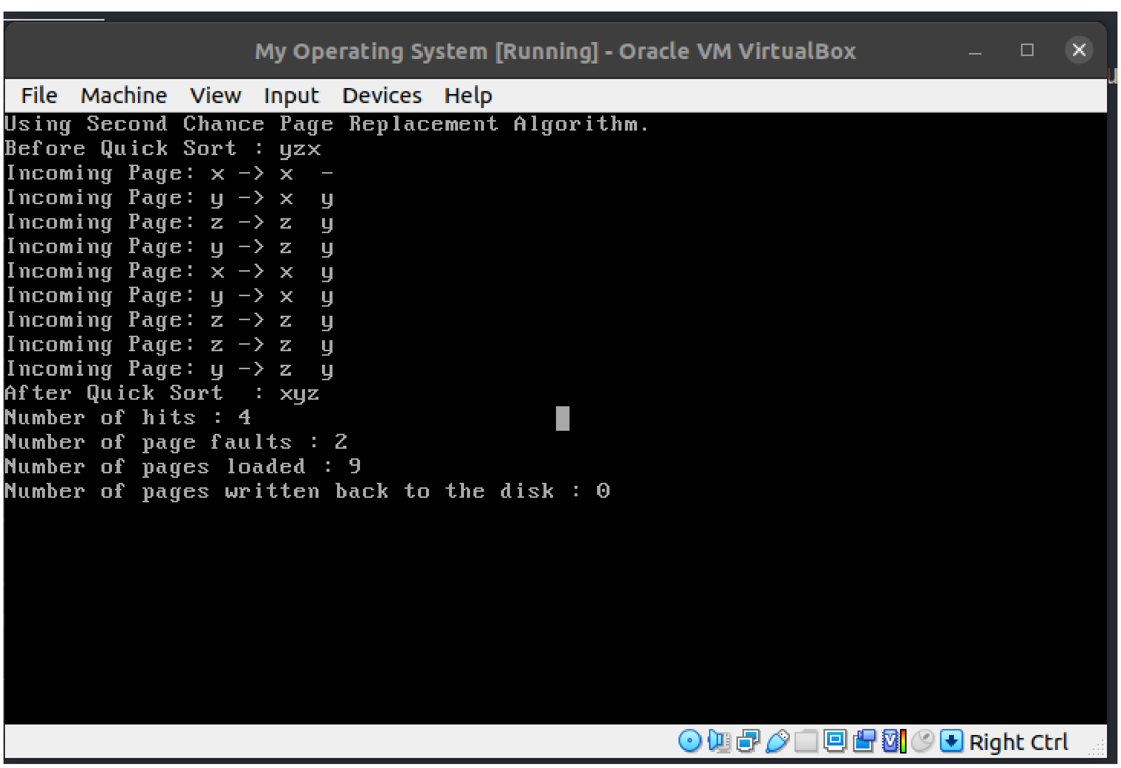
Insertion Sort - LRU



Quick Sort - FIFO



Quick Sort - Second Chance



Quick Sort - LRU