Paging in Operating Systems:

Memory Management & Page Replacement Algorithms

By: Josiah Mendez, Freddy Fabian, Sebastian Guevara, and Kevin Sarango

01 What is Paging in OS?

Why is paging useful in operating systems?

02 The FIFO Algorithm

FIFO - First In First Out

03 The LRU Algorithm

LRU - Least Recently Used

04 The RAND Algorithm

RAND - Random

05 The MRU Algorithm

MRU - Most Recently Used

01

What is Paging in OS?

Why is paging useful in operating systems?

Paging Explained

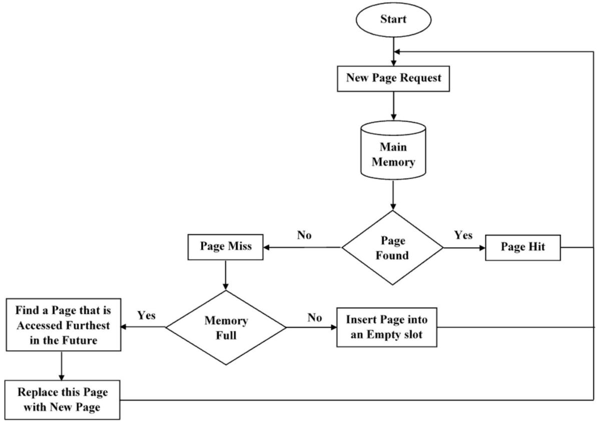
Paging in a nutshell:

- Memory management
- We would have misses until we fill all x pages of space
- Afterwards if we try to save a page that’s already there, we get a hit
- Else we get a miss and we need to find some space to place it

Optimal Page Replacement Algorithm

Optimal Page Replacement (OPT):

- It finds the most efficient way of minimizing misses by making predictions.
- The algorithm is not practical since you need information of what will be added beforehand.
- The algorithm will be used to compare with other paging algorithms.


Start
New Page Request
Main Memory
Page Found
No
Yes
Page Miss
Page Hit
Memory Full
Yes
No
Find a Page that is Accessed Furthest in the Future
Insert Page into an Empty slot
Replace this Page with New Page

02

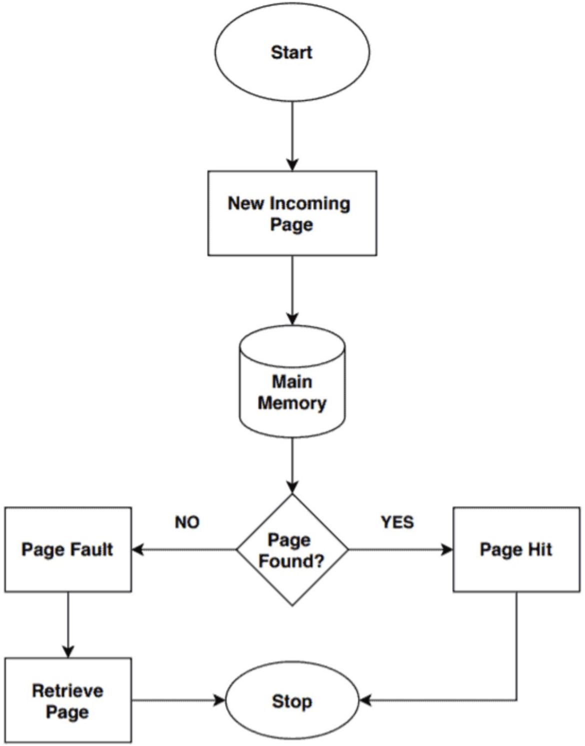
The FIFO Algorithm

FIFO - First In First Out

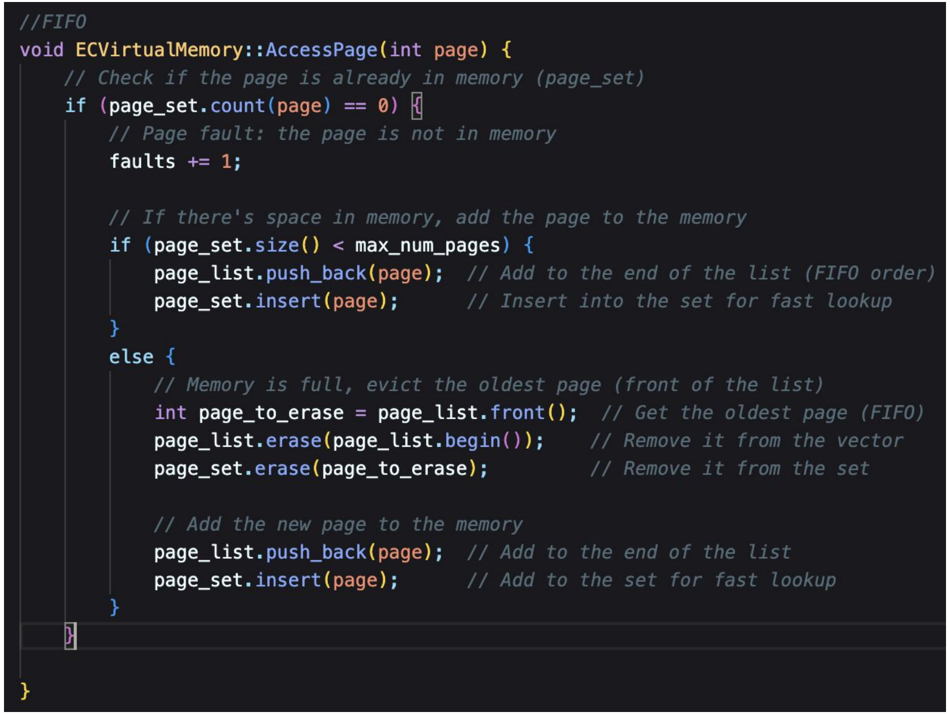
First In First Out (FIFO) Algorithm

The FIFO Algorithm:

- Looks for the oldest page in memory to replace it.
- Similar to waiting in line, the oldest page is located in the front, which is the first element in the buffer.
- When a page replacement happens, the first page is removed and the remaining pages are pushed forward.
- The new page is added to the end of the buffer

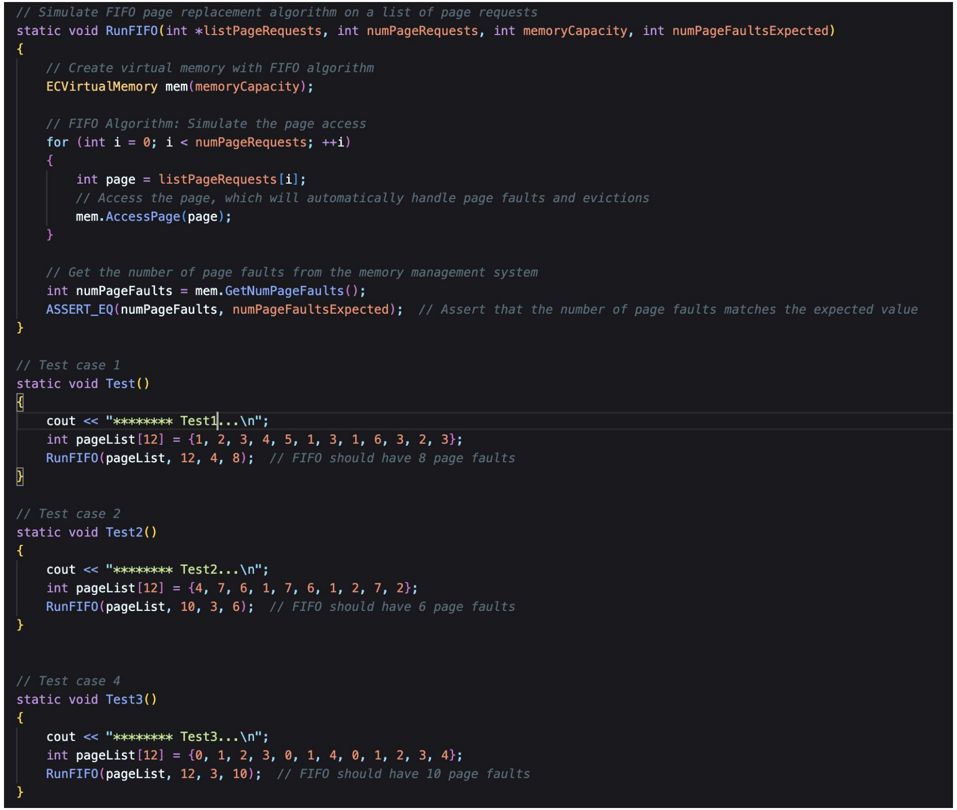


FIFO Code

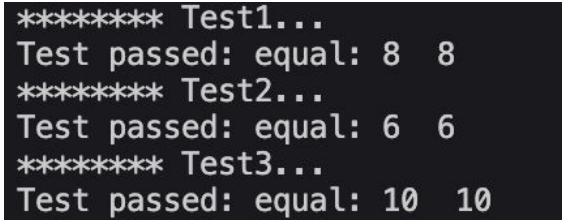


```cpp
//FIFO
void ECVirtualMemory::AccessPage(int page) {
    // Check if the page is already in memory (page_set)
    if (page_set.count(page) == 0) {
        // Page fault: the page is not in memory
        faults += 1;

        // If there's space in memory, add the page to the memory
        if (page_set.size() < max_num_pages) {
            page_list.push_back(page);  // Add to the end of the list (FIFO order)
            page_set.insert(page);      // Insert into the set for fast lookup
        }
        else {
            // Memory is full, evict the oldest page (front of the list)
            int page_to_erase = page_list.front();  // Get the oldest page (FIFO)
            page_list.erase(page_list.begin());    // Remove it from the vector
            page_set.erase(page_to_erase);         // Remove it from the set

            // Add the new page to the memory
            page_list.push_back(page);  // Add to the end of the list
            page_set.insert(page);      // Add to the set for fast lookup
        }
    }

}
```

FIFO Tests



```cpp
// Simulate FIFO page replacement algorithm on a list of page requests
static void RunFIFO(int *listPageRequests, int numPageRequests, int memoryCapacity, int numPageFaultsExpected)
{
    // Create virtual memory with FIFO algorithm
    ECVirtualMemory mem(memoryCapacity);

    // FIFO Algorithm: Simulate the page access
    for (int i = 0; i < numPageRequests; ++i)
    {
        int page = listPageRequests[i];
        // Access the page, which will automatically handle page faults and evictions
        mem.AccessPage(page);
    }

    // Get the number of page faults from the memory management system
    int numPageFaults = mem.GetNumPageFaults();
    ASSERT_EQ(numPageFaults, numPageFaultsExpected);  // Assert that the number of page faults matches the expected value
}

// Test case 1
static void Test()
{
    cout << "******** Test1...\n";
    int pageList[12] = {1, 2, 3, 4, 5, 1, 3, 1, 6, 3, 2, 3};
    RunFIFO(pageList, 12, 4, 8);  // FIFO should have 8 page faults
}

// Test case 2
static void Test2()
{
    cout << "******** Test2...\n";
    int pageList[12] = {4, 7, 6, 1, 7, 6, 1, 2, 7, 2};
    RunFIFO(pageList, 10, 3, 6);  // FIFO should have 6 page faults
}


// Test case 4
static void Test3()
{
    cout << "******** Test3...\n";
    int pageList[12] = {0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4};
    RunFIFO(pageList, 12, 3, 10);  // FIFO should have 10 page faults
}
```



```
******** Test1...
Test passed: equal: 8  8
******** Test2...
Test passed: equal: 6  6
******** Test3...
Test passed: equal: 10  10
```

03

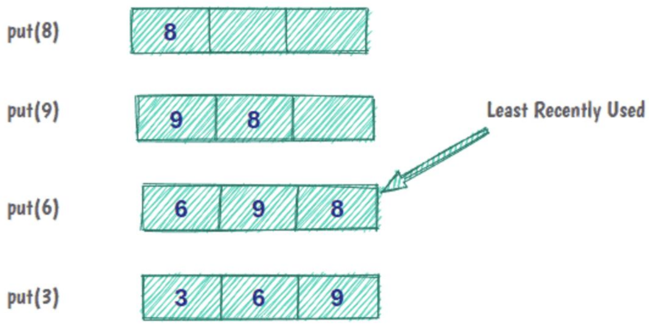
The LRU Algorithm

LRU – Least Recently Used

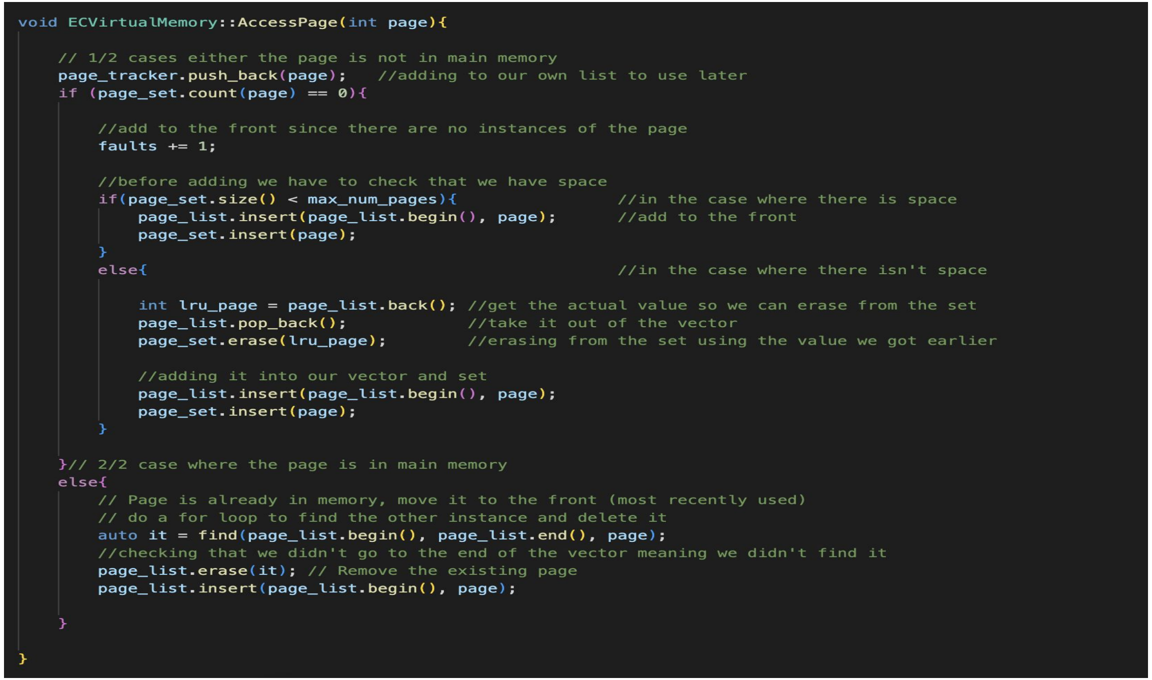
Least Recently Used (LRU) Algorithm

The LRU Algorithm:

- It adds new pages to the front, pushing old pages to the back.
- When replacing a page, it pops the element in the back and adds the new element.
- When we get a hit, we shift the element from whatever position it is in to the front, push all other elements to the right.

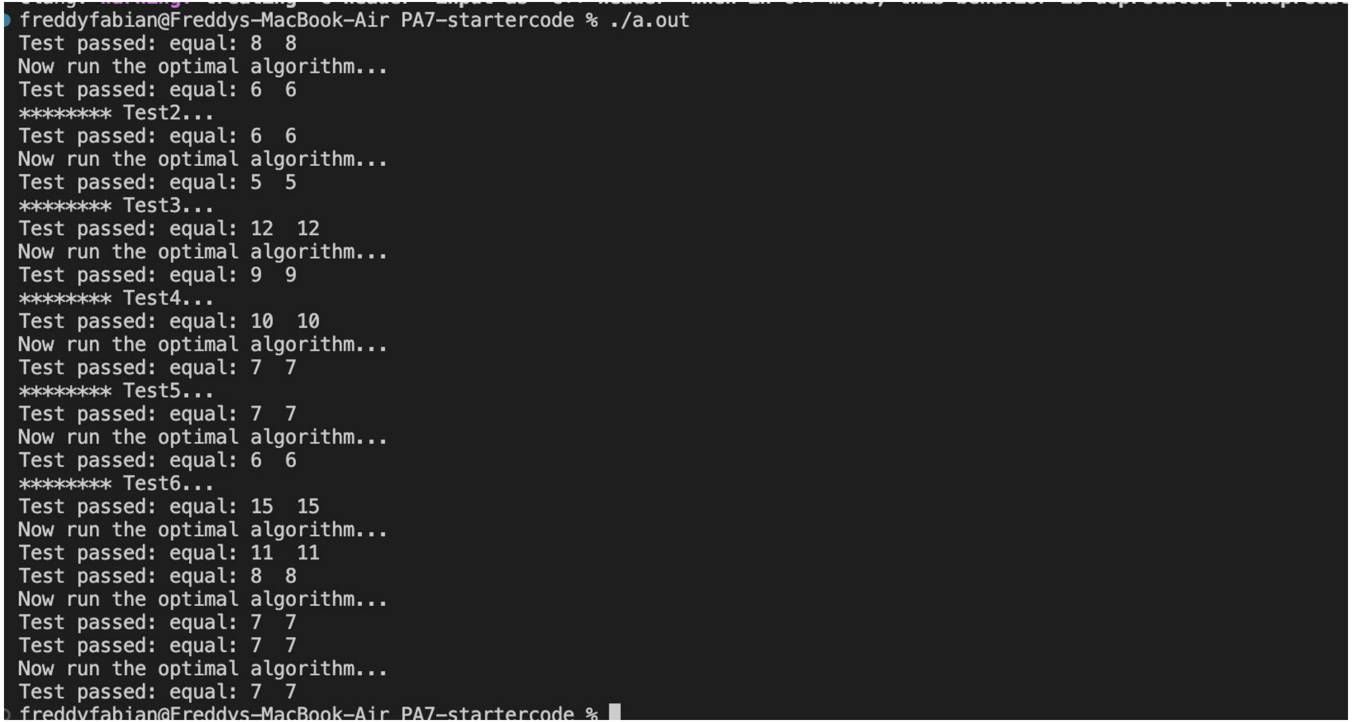


LRU Code



```
void ECVirtualMemory::AccessPage(int page){

    // 1/2 cases either the page is not in main memory
    page_tracker.push_back(page);   //adding to our own list to use later
    if (page_set.count(page) == 0){

        //add to the front since there are no instances of the page
        faults += 1;

        //before adding we have to check that we have space
        if(page_set.size() < max_num_pages){                //in the case where there is space
            page_list.insert(page_list.begin(), page);      //add to the front
            page_set.insert(page);
        }
        else{                                               //in the case where there isn't space

            int lru_page = page_list.back(); //get the actual value so we can erase from the set
            page_list.pop_back();            //take it out of the vector
            page_set.erase(lru_page);        //erasing from the set using the value we got earlier

            //adding it into our vector and set
            page_list.insert(page_list.begin(), page);
            page_set.insert(page);
        }

    }// 2/2 case where the page is in main memory
    else{
        // Page is already in memory, move it to the front (most recently used)
        // do a for loop to find the other instance and delete it
        auto it = find(page_list.begin(), page_list.end(), page);
        //checking that we didn't go to the end of the vector meaning we didn't find it
        page_list.erase(it); // Remove the existing page
        page_list.insert(page_list.begin(), page);

    }

}
```

LRU Results



```
clang: warning: treating 'c-header' input as 'c++-header' when in c++ mode, this behavior is deprecated [-Wdeprecat
freddyfabian@Freddys-MacBook-Air PA7-startercode % ./a.out
Test passed: equal: 8  8
Now run the optimal algorithm...
Test passed: equal: 6  6
******** Test2...
Test passed: equal: 6  6
Now run the optimal algorithm...
Test passed: equal: 5  5
******** Test3...
Test passed: equal: 12  12
Now run the optimal algorithm...
Test passed: equal: 9  9
******** Test4...
Test passed: equal: 10  10
Now run the optimal algorithm...
Test passed: equal: 7  7
******** Test5...
Test passed: equal: 7  7
Now run the optimal algorithm...
Test passed: equal: 6  6
******** Test6...
Test passed: equal: 15  15
Now run the optimal algorithm...
Test passed: equal: 11  11
Test passed: equal: 8  8
Now run the optimal algorithm...
Test passed: equal: 7  7
Test passed: equal: 7  7
Now run the optimal algorithm...
Test passed: equal: 7  7
freddyfabian@Freddys-MacBook-Air PA7-startercode %
```

04

The RAND Algorithm

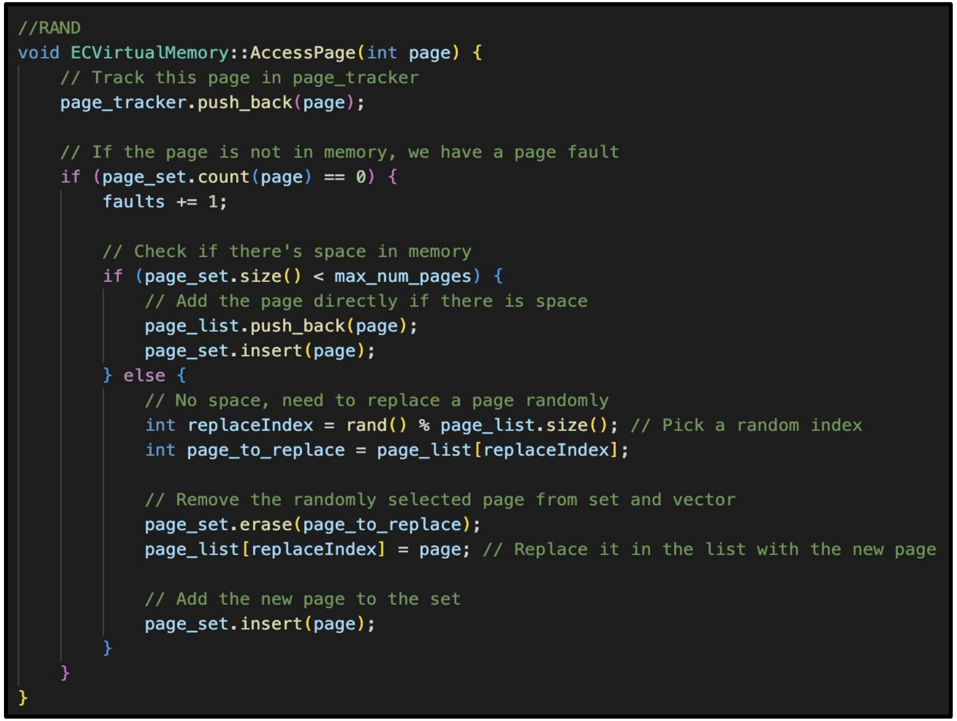
RAND - Random

Random (RAND) Algorithm

The RAND Algorithm:

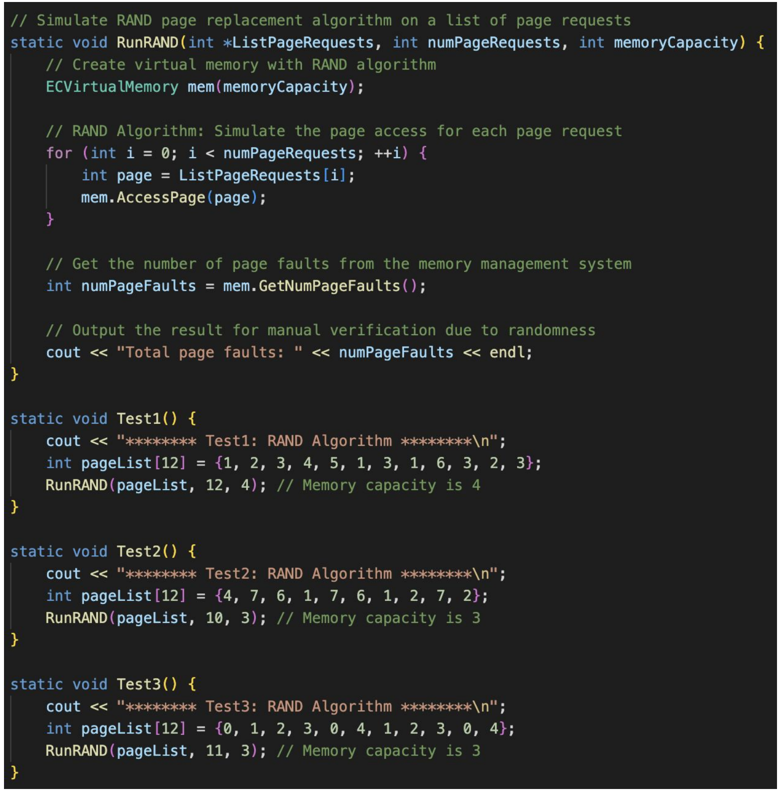
- **Purpose:** RAND chooses a page for replacement randomly rather than using a specific strategy.
- **Advantage:** It's simple to implement and avoids the computational overhead of more sophisticated algorithms.
- **Disadvantage:** It's usually less efficient than algorithms like LRU or FIFO since it doesn't consider page access patterns or future references.

RAND Code

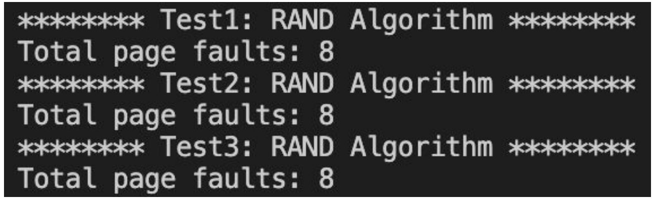


```
//RAND
void ECVirtualMemory::AccessPage(int page) {
    // Track this page in page_tracker
    page_tracker.push_back(page);

    // If the page is not in memory, we have a page fault
    if (page_set.count(page) == 0) {
        faults += 1;

        // Check if there's space in memory
        if (page_set.size() < max_num_pages) {
            // Add the page directly if there is space
            page_list.push_back(page);
            page_set.insert(page);
        } else {
            // No space, need to replace a page randomly
            int replaceIndex = rand() % page_list.size(); // Pick a random index
            int page_to_replace = page_list[replaceIndex];

            // Remove the randomly selected page from set and vector
            page_set.erase(page_to_replace);
            page_list[replaceIndex] = page; // Replace it in the list with the new page

            // Add the new page to the set
            page_set.insert(page);
        }
    }
}
```

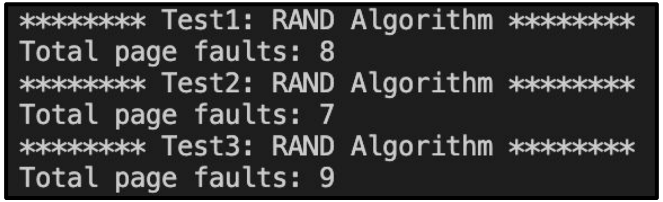
RAND Tests



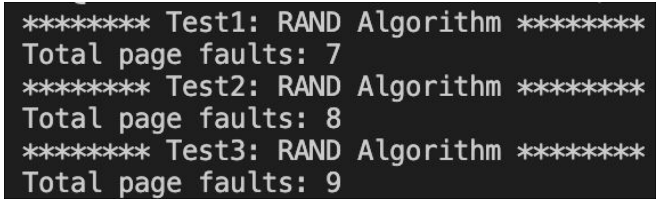
```
// Simulate RAND page replacement algorithm on a list of page requests
static void RunRAND(int *ListPageRequests, int numPageRequests, int memoryCapacity) {
    // Create virtual memory with RAND algorithm
    ECVirtualMemory mem(memoryCapacity);

    // RAND Algorithm: Simulate the page access for each page request
    for (int i = 0; i < numPageRequests; ++i) {
        int page = ListPageRequests[i];
        mem.AccessPage(page);
    }

    // Get the number of page faults from the memory management system
    int numPageFaults = mem.GetNumPageFaults();

    // Output the result for manual verification due to randomness
    cout << "Total page faults: " << numPageFaults << endl;
}

static void Test1() {
    cout << "******** Test1: RAND Algorithm ********\n";
    int pageList[12] = {1, 2, 3, 4, 5, 1, 3, 1, 6, 3, 2, 3};
    RunRAND(pageList, 12, 4); // Memory capacity is 4
}

static void Test2() {
    cout << "******** Test2: RAND Algorithm ********\n";
    int pageList[12] = {4, 7, 6, 1, 7, 6, 1, 2, 7, 2};
    RunRAND(pageList, 10, 3); // Memory capacity is 3
}

static void Test3() {
    cout << "******** Test3: RAND Algorithm ********\n";
    int pageList[12] = {0, 1, 2, 3, 0, 4, 1, 2, 3, 0, 4};
    RunRAND(pageList, 11, 3); // Memory capacity is 3
}
```



```
******** Test1: RAND Algorithm ********
Total page faults: 8
******** Test2: RAND Algorithm ********
Total page faults: 8
******** Test3: RAND Algorithm ********
Total page faults: 8
```



```
******** Test1: RAND Algorithm ********
Total page faults: 8
******** Test2: RAND Algorithm ********
Total page faults: 7
******** Test3: RAND Algorithm ********
Total page faults: 9
```



```
******** Test1: RAND Algorithm ********
Total page faults: 7
******** Test2: RAND Algorithm ********
Total page faults: 8
******** Test3: RAND Algorithm ********
Total page faults: 9
```

05

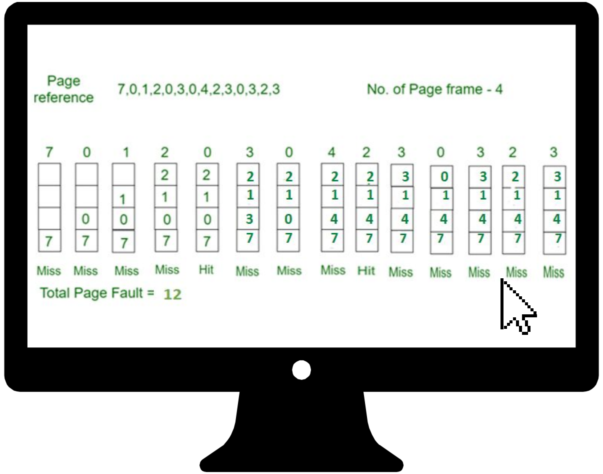
The MRU Algorithm

MRU - Most Recently Used

Most Recently Used (MRU) Algorithm

The MRU Algorithm:

- Replaces the page from the previous call/iteration.
- Before entering the next iteration, we save the page's index and save itself as the MRU page, whether it is a **hit** or **miss**.



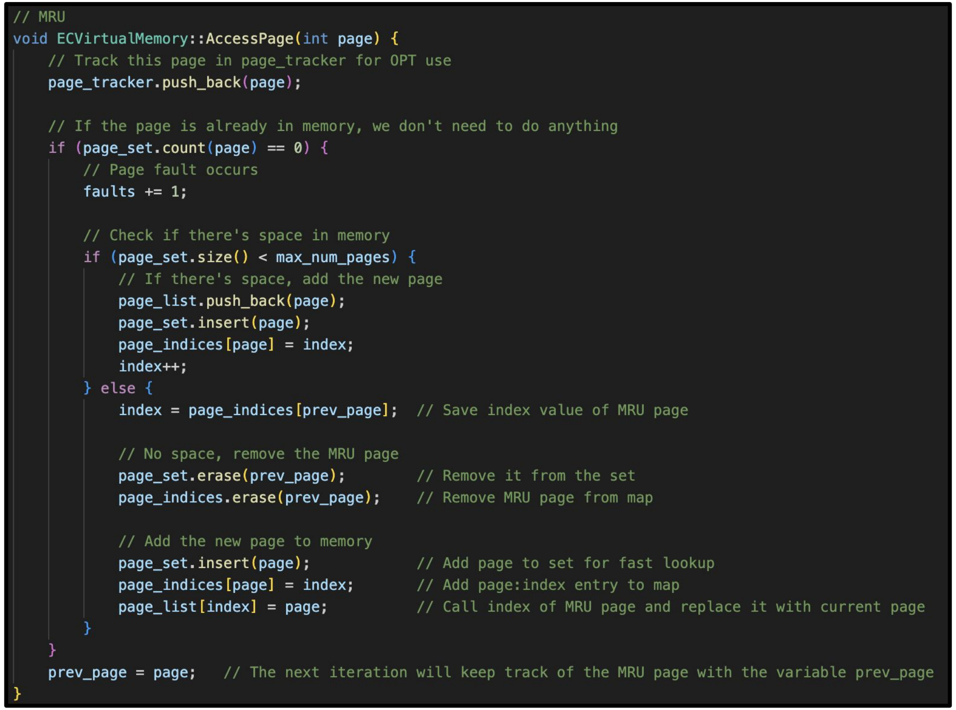
Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3

No. of Page frame - 4

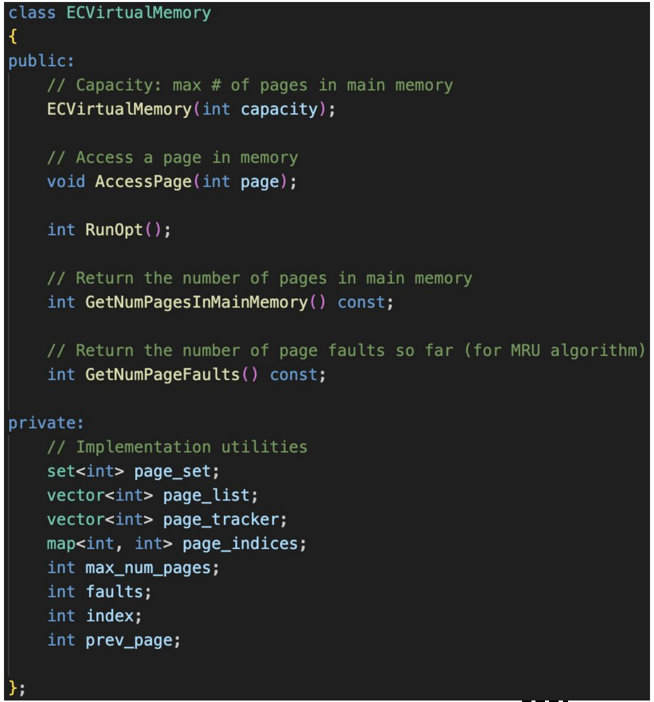
7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	3	0	3	2	3
		1	1	1	1	1	1	1	1	1	1	1	1
	0	0	0	0	3	0	4	4	4	4	4	4	4
7	7	7	7	7	7	7	7	7	7	7	7	7	7
Miss	Miss	Miss	Miss	Hit	Miss	Miss	Miss	Hit	Miss	Miss	Miss	Miss	Miss

Total Page Fault = 12

MRU Code

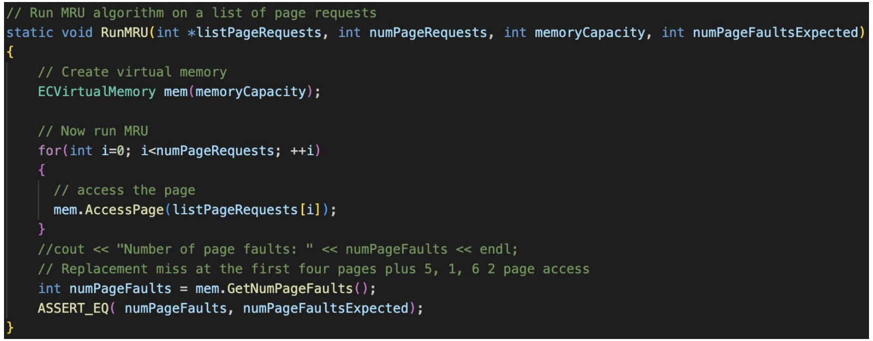


```
// MRU
void ECVirtualMemory::AccessPage(int page) {
    // Track this page in page_tracker for OPT use
    page_tracker.push_back(page);

    // If the page is already in memory, we don't need to do anything
    if (page_set.count(page) == 0) {
        // Page fault occurs
        faults += 1;

        // Check if there's space in memory
        if (page_set.size() < max_num_pages) {
            // If there's space, add the new page
            page_list.push_back(page);
            page_set.insert(page);
            page_indices[page] = index;
            index++;
        } else {
            index = page_indices[prev_page];  // Save index value of MRU page

            // No space, remove the MRU page
            page_set.erase(prev_page);        // Remove it from the set
            page_indices.erase(prev_page);    // Remove MRU page from map

            // Add the new page to memory
            page_set.insert(page);            // Add page to set for fast lookup
            page_indices[page] = index;       // Add page:index entry to map
            page_list[index] = page;          // Call index of MRU page and replace it with current page
        }
    }
    prev_page = page;   // The next iteration will keep track of the MRU page with the variable prev_page
}
```

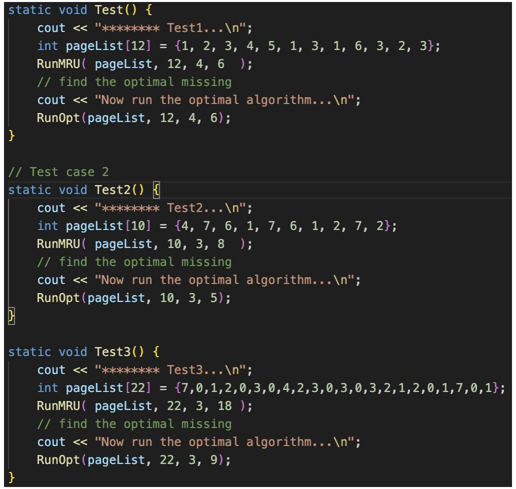


```
class ECVirtualMemory
{
public:
    // Capacity: max # of pages in main memory
    ECVirtualMemory(int capacity);

    // Access a page in memory
    void AccessPage(int page);

    int RunOpt();

    // Return the number of pages in main memory
    int GetNumPagesInMainMemory() const;

    // Return the number of page faults so far (for MRU algorithm)
    int GetNumPageFaults() const;

private:
    // Implementation utilities
    set<int> page_set;
    vector<int> page_list;
    vector<int> page_tracker;
    map<int, int> page_indices;
    int max_num_pages;
    int faults;
    int index;
    int prev_page;

};
```

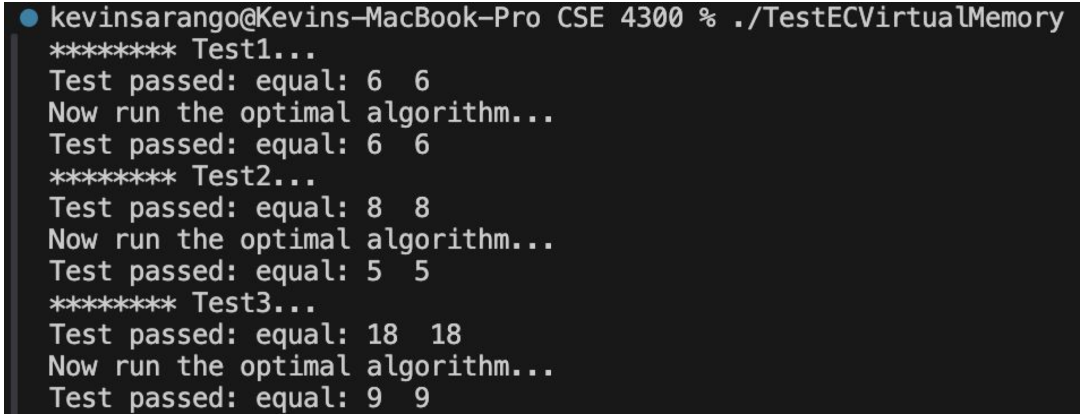
MRU Tests



```
// Run MRU algorithm on a list of page requests
static void RunMRU(int *listPageRequests, int numPageRequests, int memoryCapacity, int numPageFaultsExpected)
{
    // Create virtual memory
    ECVirtualMemory mem(memoryCapacity);

    // Now run MRU
    for(int i=0; i<numPageRequests; ++i)
    {
      // access the page
      mem.AccessPage(listPageRequests[i]);
    }
    //cout << "Number of page faults: " << numPageFaults << endl;
    // Replacement miss at the first four pages plus 5, 1, 6 2 page access
    int numPageFaults = mem.GetNumPageFaults();
    ASSERT_EQ( numPageFaults, numPageFaultsExpected);
}
```



```
static void Test() {
    cout << "******** Test1...\n";
    int pageList[12] = {1, 2, 3, 4, 5, 1, 3, 1, 6, 3, 2, 3};
    RunMRU( pageList, 12, 4, 6  );
    // find the optimal missing
    cout << "Now run the optimal algorithm...\n";
    RunOpt(pageList, 12, 4, 6);
}

// Test case 2
static void Test2() {
    cout << "******** Test2...\n";
    int pageList[10] = {4, 7, 6, 1, 7, 6, 1, 2, 7, 2};
    RunMRU( pageList, 10, 3, 8  );
    // find the optimal missing
    cout << "Now run the optimal algorithm...\n";
    RunOpt(pageList, 10, 3, 5);
}

static void Test3() {
    cout << "******** Test3...\n";
    int pageList[22] = {7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1};
    RunMRU( pageList, 22, 3, 18 );
    // find the optimal missing
    cout << "Now run the optimal algorithm...\n";
    RunOpt(pageList, 22, 3, 9);
}
```



```
kevinsarango@Kevins-MacBook-Pro CSE 4300 % ./TestECVirtualMemory
******** Test1...
Test passed: equal: 6  6
Now run the optimal algorithm...
Test passed: equal: 6  6
******** Test2...
Test passed: equal: 8  8
Now run the optimal algorithm...
Test passed: equal: 5  5
******** Test3...
Test passed: equal: 18  18
Now run the optimal algorithm...
Test passed: equal: 9  9
```