Lab9: Memory Management and Page Replacement Algorithms

1. Objective

- Understand Memory Management and Page Replacement Algorithm and read its code in C++, try to write other kinds of algorithms.

2. Syllabus

- Understand Memory Management and Page Replacement algorithms;
- Implement these algorithms with C++.

3. Prerequisite

- C++ language
- Computer which run Linux system (e.g. Ubuntu)

4. Concepts and Principles of this lab

Please refer to the chapter 7. The main contents have been lectured in the eighth week, 10/30/2017 and 11/1/2017. The slide of this chapter can be found in the course website: http://www.thinkmesh.net/~jiangxl/teaching/106F14A/.

5. Experimental Contents

5.1 Program for First Fit algorithm

Description:

In the first fit, partition is allocated which is first sufficient from the top of Main Memory.

- Its advantage is that it is the fastest search as it searches only the first block i.e. enough to assign a process.
- It may problems of not allowing processes to take space even if it was possible to allocate. Consider the above example, process number 4 (of size 426) does not get memory. However it was possible to allocate memory if we had allocated using best fit allocation [block number 4 (of size 300) to process 1, block number 2 to process 2, block number 3 to process 3 and block number 5 to process 4].

Note:

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.

```
// C++ implementation of First - Fit algorithm
#include<bits/stdc++.h>
using namespace std;
```

```
// Function to allocate memory to blocks as per First fit
// algorithm
void firstFit(int blockSize[], int m, int processSize[], int n)
{
	// Stores block id of the block allocated to a
	// process
	int allocation[n];

	// Initially no block is assigned to any process
	memset(allocation, -1, sizeof(allocation));

	// pick each process and find suitable blocks
	// according to its size ad assign to it
	for (int i=0; i<n; i++)
	{
		for (int j=0; j<m; j++)
		{
			if (blockSize[j] >= processSize[i])
			{
				// allocate block j to p[i] process
				allocation[i] = j;

				// Reduce available memory in this block.
				blockSize[j] -= processSize[i];

				break;
			}
		}
	}

	cout << "\nProcess No.\tProcess Size\tBlock no.\n";
	for (int i = 0; i < n; i++)
	{
		cout << "	" << i+1 << "\t\t" << processSize[i] << "\t\t";
		if (allocation[i] != -1)
			cout << allocation[i] + 1;
		else
			cout << "Not Allocated";
		cout << endl;
	}
}

// Driver code
int main()
```

```
{
    int blockSize[] = {100, 500, 200, 300, 600};
    int processSize[] = {212, 417, 112, 426};
    int m = sizeof(blockSize)/sizeof(blockSize[0]);
    int n = sizeof(processSize)/sizeof(processSize[0]);

    firstFit(blockSize, m, processSize, n);

    return 0 ;
}
```

File name: FirstFit.cpp

How to run: g++ -o FirstFit FirstFit.cpp
./FirstFit

5.2 Program for Next Fit algorithm

Description: Next fit is a modified version of 'first fit'. It begins as first fit to find a free partition but when called next time it starts searching from where it left off, not from the beginning. This policy makes use of a roving pointer. The pointer roves along the memory chain to search for a next fit. This helps in, to avoid the usage of memory always from the head (beginning) of the free block chain.

Note:

1. Input the number of memory blocks and their sizes and initializes all the blocks as free.
2. Input the number of processes and their sizes.
3. Start by picking each process and check if it can be assigned to current block, if yes, allocate it the required memory and check for next process but from the block where we left not from starting.
4. If current block size is smaller then keep checking the further blocks.

```
// C/C++ program for next fit
// memory management algorithm
#include <bits/stdc++.h>
using namespace std;

// Function to allocate memory to blocks as per Next fit
// algorithm
void NextFit(int blockSize[], int m, int processSize[], int n)
{
    // Stores block id of the block allocated to a
    // process
    int allocation[n], j = 0;

    // Initially no block is assigned to any process
    memset(allocation, -1, sizeof(allocation));
```

```
        // pick each process and find suitable blocks
        // according to its size ad assign to it
        for (int i = 0; i < n; i++) {

            // Do not start from beginning
            while (j < m) {

                if (blockSize[j] >= processSize[i]) {

                    // allocate block j to p[i] process
                    allocation[i] = j;

                    // Reduce available memory in this block.
                    blockSize[j] -= processSize[i];

                    break;
                }
                j++;
            }
        }

        cout << "\nProcess No.\tProcess Size\tBlock no.\n";
        for (int i = 0; i < n; i++) {
            cout << " " << i + 1 << "\t\t" << processSize[i]
                 << "\t\t";
            if (allocation[i] != -1)
                cout << allocation[i] + 1;
            else
                cout << "Not Allocated";
            cout << endl;
        }
    }

    // Driver program
    int main()
    {
        int blockSize[] = { 5, 10, 20 };
        int processSize[] = { 10, 20, 30 };
        int m = sizeof(blockSize) / sizeof(blockSize[0]);
        int n = sizeof(processSize) / sizeof(processSize[0]);

        NextFit(blockSize, m, processSize, n);

        return 0;
```

```
}
```

File name: NextFit.cpp

How to run: g++ -o NextFit NextFit.cpp
./NextFit

5.3 Program for First In First Out (FIFO) algorithm

Description: In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. 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.

Note: 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.

For example, consider page reference string 1, 3, 0, 3, 5, 6 and 3 page slots. Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots —> 3 Page Faults. When 3 comes, it is already in memory so —> 0 Page Faults. Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>1 Page Fault. Finally 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>1 Page Fault. So total page faults = 6.

```
// C++ implementation of FIFO page replacement
// in Operating Systems.
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using FIFO
int pageFaults(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())
            {
                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())
            {
                //Pop the first page from the queue
                int val = indexes.front();

                indexes.pop();

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

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

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

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

```
    return page_faults;
}

// Driver code
int main()
{
    int pages[] = {7, 0, 1, 2, 0, 3, 0, 4,
                   2, 3, 0, 3, 2};
    int n = sizeof(pages)/sizeof(pages[0]);
    int capacity = 4;
    cout << pageFaults(pages, n, capacity);
    return 0;
}
```

File name: FIFO.cpp

How to run: g++ -o FIFO FIFO.cpp
./FIFO

5.4 Program for Least Recently Used (LRU) algorithm

Description: In operating systems that use paging for memory management, page replacement algorithm are needed to decide which page needed to be replaced when new page comes in. 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.

Note: In Least Recently Used (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.

Let say the page reference string 7 0 1 2 0 3 0 4 2 3 0 3 2 . Initially we have 4 page slots empty.
Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> 4 Page faults
0 is already their so —> 0 Page fault.
when 3 came it will take the place of 7 because it is least recently used —>1 Page fault
0 is already in memory so —> 0 Page fault.
4 will takes place of 1 —> 1 Page Fault
Now for the further page reference string —> 0 Page fault because they are already available in the memory.

```
//C++ implementation of above algorithm
#include<bits/stdc++.h>
using namespace std;

// Function to find page faults using indexes
int pageFaults(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;
}

// Driver code
int main()
{
  int pages[] = {7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2};
  int n = sizeof(pages)/sizeof(pages[0]);
  int capacity = 4;
  cout << pageFaults(pages, n, capacity);
  return 0;
}
```

File name: LRU.cpp

How to run: g++ -o LRU LRU.cpp
./LRU

6. Conclusion:

In this chapter, four experiments have been completed. Now let's try to search and read some other algorithms, such as Best-fit, LFU, OPT.