CSE 430: Operating Systems

Instructor: Dr. Violet R. Syrotiuk

LRU Cache Replacement Policy

Design and Analysis Report

Project 3: Milestone

Author: Ankit Rathi

ASU ID: 1207543476

Contents

LRU Caching Scheme

In Least Recently Used caching algorithm, we are given possible number of pages that will be referenced and cache size. Cache Size helps to determine number of pages that the cache can hold at any instant of time. This scheme of cache replacement policy is to remove the least recently frame when the cache is full and add a new frame into the cache whenever it is accessed and it is not already present in the cache memory.

Data Structure to implement cache

I have implemented the Queue using a Doubly Linked List. Doubly Linked List is a dynamic data structure that can grow size of the number of pages that can be accommodated completely in the cache. Each node of the queue is implemented with the help of another data structure that is QNode. QNode can contains pointers to point to the previous and the next element in the queue. The maximum size of the queue will be equal to the total number of frames available. One more important thing to be noted is that most recently used page will at the front of the queue and the least recently used pages will be at the rear end of the queue. Initial assumption that has been made is that initially no pages are present in the cache.

```
/*
* A Queue Node (Implementation of queue using Doubly Linked List)
*/
typedef struct QNode
{
	struct QNode *prev, *next;
	unsigned int iPageNumber;  // the page number stored in this QNode
} QNode;
```

Each node of the queue is represented using the QNode structure. This structure contains the pointers to the previous and next nodes in the doubly linked list. It also contains a variable to hold the page number. This stores the values of the page number that has been accessed.

```
/*
*A FIFO Queue using QNodes
*/
typedef struct Queue
{
	unsigned int iCount;  // Number of frames that have been filled.
	unsigned int iNumberOfFrames; // total number of frames
	QNode *front, *rear;
} Queue;
```

The above data structure represents the Queue structure. The queue structure contains a count variable to keep track of the number of frames that have been filled in the cache. It also contains a variable to keep note of the total number of frames or pages that can be loaded into the memory at the same time. It also keeps tracks of the front and rear end of the queue. This is required as we insert the elements in the front end and but remove the elements from the rear end. This is a basic implementation of the double linked list which provide the features of a FIFO queue.

newQNode is a utility function to create a new node. This node will be used to store the pagenumber that has been accessed.

```
/*
*A utility function to create a new Queue Node. The queue Node
* will store the given 'pageNumber'
*/
QNode* newQNode(unsigned int pageNumber)
{
	// Allocate memory and assign 'pageNumber' for the first cache
	QNode* temp = (QNode *)malloc(sizeof(QNode));
	temp->iPageNumber = pageNumber;

	// Initialize prev and next as NULL
	temp->prev = NULL;
	temp->next = NULL;

	return temp;
}
```

createQueue is a utility function to create an empty queue. The queue can at most accommodate the numberofframe variable of the data structure. This function gets called only once during the entire program execution. So for the first time all the initializations needs to be done. Front and the rear pointers are initialized to NULL.

```
/*
* A utility function to create an empty Queue.
* The queue can have at most 'numberOfFrames' nodes
*/
Queue* createQueue(unsigned int numberOfFrames)
{
	Queue* queue = (Queue *)malloc(sizeof(Queue));

	// The queue is empty
	queue->iCount = 0;
	queue->front = NULL;
	queue->rear = NULL;

	// Number of frames that can be stored in memory
	queue->iNumberOfFrames = numberOfFrames;

	return queue;
}
```

isQueueFull function is used to check if the number of pages in the cache has reached its maximum limit.

```
/*
*A function to check if there is slot available to bring
* in a new page into memory.
*/
```

```
int isQueueFull(Queue* queue)
{
	return queue->iCount == queue->iNumberOfFrames;
}
```

isQueueEmpty function is used to check if the queue is empty.

```
/*
*A function to check if queue is empty
*/
int isQueueEmpty(Queue* queue)
{
	return queue->rear == NULL;
}
```

deQueue function is used to delete a frame from queue. If the queue is empty then the function just returns back. It also checks if there is only one element in the queue, then the front of the queue is set to NULL. Then we need to change the rear end of the queue also and decrement the counter by 1 to keep track of number of elements in the queue.

```
/*
*A utility function to delete a frame from queue
*/
void deQueue(Queue* queue)
{
	if (isQueueEmpty(queue))
	{
		return;
	}

	// If this is the only node in list, then change front
	if (queue->front == queue->rear)
	{
		queue->front = NULL;
	}

	// Change rear and remove the previous rear
	QNode* temp = queue->rear;
	queue->rear = queue->rear->prev;

	if (queue->rear)
	{
		queue->rear->next = NULL;
	}

	free(temp);

	// decrement the number of full frames by 1
	queue->iCount--;
}
```

Enqueue function is used to add the element into the queue. First it checks if all the elements are full, if so, then it dequeues one element from the queue. Then as usual we can create a new node and add an element into the queue and increment the counter by 1.

```
/*
*A function to add a page with given 'pageNumber' to queue.
*/
void Enqueue(Queue* queue, unsigned int pageNumber)
{
	// If all frames are full, remove the page at the rear
	if (isQueueFull(queue))
	{
		deQueue(queue);
	}

	// Create a new node with given page number,
	// And add the new node to the front of queue
	QNode* temp = newQNode(pageNumber);
	temp->next = queue->front;

	// If queue is empty, change both front and rear pointers
	if (isQueueEmpty(queue))
	{
		queue->rear = queue->front = temp;
	}
	else  // Else change the front
	{
		queue->front->prev = temp;
		queue->front = temp;
	}

	// increment number of full frames
	queue->iCount++;
}
```

ReferencePage function is used to reference a page from the cache memory. There are two cases that are handled by the function. First, If the frame is not in memory, then we need to bring that frame into the memory and add it to the front of the queue. Second, if the frame is already in memory then we need to remove the element from the queue and bring it to the front of the queue. This maintains the condition of Least Recently Used caching scheme. Every element that has been accessed now, will be moved to the front of the queue, this means that most recently accessed element is the front of the queue and elements which have not been accessed recently will be moved to the end of the queue. Hence the elements that are present at the end of the queue refer to pages that have been least recently used. Every time an element is found in the memory, the iHitCount variable is incremented and every time a new element is bought into the memory then the iMissCount variable will be incremented.

```
/*
* This function is called when a page with given 'pageNumber' is referenced
* from cache (or memory). There are two cases:
* 1. Frame is not there in memory, then bring it into memory and add to the front
*    of queue
```

```
* 2. Frame is there in memory, then move the frame to front of queue
*/
void ReferencePage(Queue* queue, unsigned pageNumber)
{
	QNode* reqPage = NULL;

	//Check if the page or the frame being referenced is already present in
	//the queue of nodes.
	QNode* tempReqPage = queue->front;
	while (tempReqPage != NULL)
	{
		if (tempReqPage->iPageNumber == pageNumber)
		{
			reqPage = tempReqPage;
			break;
		}
		else
		{
			tempReqPage = tempReqPage->next;
		}
	}

	// the page is not in cache, bring it
	if (reqPage == NULL)
	{
		Enqueue(queue, pageNumber);
		iMissCount++;
	}

	// page is there and not at front, change pointer
	else if (reqPage != queue->front)
	{
		// Unlink rquested page from its current location
		// in queue.
		reqPage->prev->next = reqPage->next;
		if (reqPage->next)
		{
			reqPage->next->prev = reqPage->prev;
		}

		// If the requested page is rear, then change rear
		// as this node will be moved to front
		if (reqPage == queue->rear)
		{
			queue->rear = reqPage->prev;
			queue->rear->next = NULL;
		}

		// Put the requested page before current front
		reqPage->next = queue->front;
		reqPage->prev = NULL;

		// Change prev of current front
		reqPage->next->prev = reqPage;

		// Change front to the requested page
		queue->front = reqPage;
		iHitCount++;
```

```
	}
}
```

Main functions first reads the arguments supplied along with the file. If less than 2 arguments are supplied then an error will be generated. After that I have created and initialized the queue. Then I have opened the file using the fopen command and read line by line the contents of the file. After reading each line I reference that page in the memory. This is done in infinite loop till the end of the file is reached. Then the hit ratio is calculated.

```
/*
*Driver program to test above functions
*/
int main(int argc, char **argv)
{
	FILE *fp;
	time_t start, stop;

	if (argc == 3)
	{
		//argv[1] is the cache size
		//argv[2] is the Trace File Name
		iCacheSize = strtol(argv[1], NULL, 10);
		fileName = argv[2];
	}
	else
	{
		printf("Cache Size and Trace File Name not supplied properly.\nProgram
Terminating\n");
		getchar();
		exit(-1);
	}

	printf("Cache Size = %d\n",iCacheSize);
	printf("TraceFile Name = %s\n",fileName);

	//Create cache can hold c pages
	Queue* q = createQueue(iCacheSize);

	fp = fopen(fileName, "r");
	if (fp == NULL)
	{
		printf("Error while opening the file: %s\n", fileName);
		exit(EXIT_FAILURE);
	}

	unsigned int iStartingBlock = 0, iNumberOfBlocks = 0, iIgnore = 0, iRequestNumber
= 0;
	unsigned int i = 0;

	time(&start);

	while (1)
	{
		if (-1 != fscanf(fp, "%u %u %u %u", &iStartingBlock, &iNumberOfBlocks,
&iIgnore, &iRequestNumber))
		{
```

```
                        //printf("%u\n", iRequestNumber);
                        for (i = iStartingBlock; i < (iStartingBlock + iNumberOfBlocks);
i++)
                        {
                                ReferencePage(q, i);
                        }
                }
                else
                {
                        break;
                }
        }

        time(&stop);
        printf("Miss Count = %u\n",iMissCount);
        printf("Hit Count = %u\n", iHitCount);
        //printf("Hit Ratio = %5.4f %%\n", ((float)(iHitCount * 100) / (iHitCount +
iMissCount)));
        printf("Rounded Hit Ratio = %5.2f %%\n", floor(((float)(iHitCount * 100) /
(iHitCount + iMissCount)) * 100 + 0.5) / 100);
        printf("Finished in about %.0f seconds. \n", difftime(stop, start));

        //getchar();
        return 0;
}
```

Representation of important variables

I have used unsigned int for counter variables to keep count of number of hits and number of misses. The cache size that I receive as an input from the user is also stored in an unsigned int.

```
/*
*Global Variable Declarations
*/
unsigned int iCacheSize = 0;
unsigned int iMissCount = 0;
unsigned int iHitCount = 0;
```

The page number that is being referenced is stored as part of the QNode structure and it is also stored as an unsigned int, since the page number can never be negative.

```
unsigned int iPageNumber;  // the page number stored in this QNode

unsigned int iCount;  // Number of frames that have been filled.
unsigned int iNumberOfFrames; // total number of frames
```

Hit Ratio

The Hit ratio is calculated as the ratio of number of hits to the total number of cache references. The below equation gives the hit ratio in percentage. The numerator is converted to float before division to auto typecasting happens and the result is displayed in flat form.

$$Hit\ Ratio = \frac{(float)(iHitCount * 100)}{iHitCount + iMissCount}$$

To Round of the Hit Ratio to the hundredth decimal, I have used the math.h library function floor. The floor function rounds of the value to the next correct decimal value by using the equation shown below.

$$Hit\ Ratio = floor\left(\frac{\left(\left(\frac{(float)(iHitCount * 100)}{iHitCount + iMissCount}\right) * 100 + 0.5\right)}{100}\right)$$

Time Complexity

The program makes use of doubly linked list to implement the cache. Hence to search for an element in the queue it takes time of O(cache size). And the total number of pages that we reference be equal to the say NumberOfPageReference, then the complexity of the algorithm implemented is

O(Cache Size * NumberOfPageReference).