Ryan Fazal
Prof. Jinwoo Kim
CSCI 375
Project #3

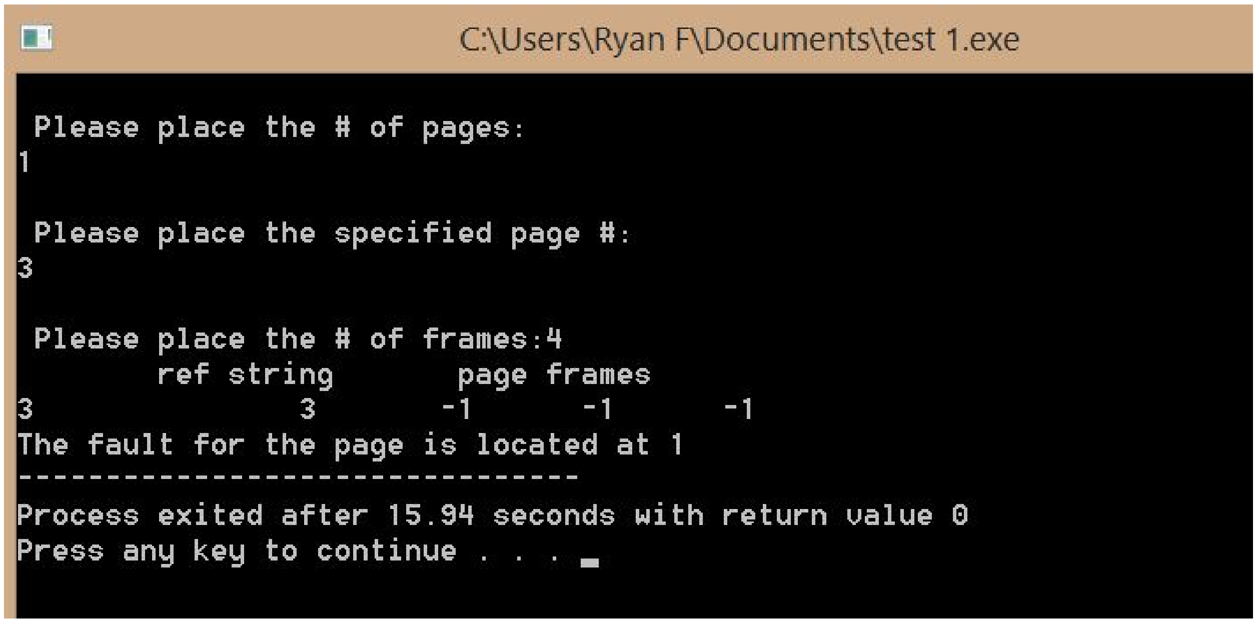
Subproject 1:

Important Notes:
I used a C++ Program

For this project, I decided to separate the FIFO and LRU. This was in order to show both of their potentials equally. For FIFO, I decided to make the count up to 20. This would allow any numbers between 0-20 to be inputted. However only 0-9 were recommended for this project. Next was the frame [5]. This was the physical entity of the page. This frame is referring to the number of pages within that particular memory. This means that the main memory is broken down into a form of blocks. These blocks are what are known as the frames. Every frame can hold a page. In the case of the program below, it can hold 5 frames. This program has three particular write outs. For "\n Please place the # of pages:\n" and "\n Please place the specified page #:\n" they must input a number between 0-9. Because if the number goes above a 9, the program would stop working, or even cancel the process. Lastly, the print "\n Please place the # of frames:" will give us the recorded number of faults that may of occurred upon this algorithm.This, as shown below, is a number we must input ourselves. However, since the number of faults is set to 0-5, we must only use the numbers in that range. Just like the print previously, using a higher number would stop the programming, as it won't be able to process that number.When the fault has been successfully located, the program will print "The fault for the page is located at %d. This will tell us where the fault entered, has occurred. For FIFO , I am using a queue.

Fifo
```
#include<stdio.h>
int main()
{
int i,j,n,a[20],frame[5],no,k,avail,count=0;
printf("\n Please place the # of pages:\n");
scanf("%d",&n);  //used to check that the proper value has been used. This is to avoid an error.
printf("\n Please place the specified page #:\n");
 for(i=1;i<=n;i++)
scanf("%d",&a[i]);  //error avoidance
printf("\n Please place the # of frames:");
scanf("%d",&no);  //error avoidance
for(i=0;i<no;i++)
 frame[i]= -1;  // this will indicated an error has occurred within the frame section.
j=0;
 printf("\tref string\t page frames\n");  // this will indicate the write out for the page frames
for(i=1;i<=n;i++)
{
 printf("%d\t\t",a[i]);
avail=0;
 for(k=0;k<no;k++)  // this is used for a creation of a brand new process. This can change the formatting of the parent/child
if(frame[k]==a[i])
 avail=1;
if (avail==0)
{
frame[j]=a[i];
 j=(j+1)%no;
Count++;  //this is known as a post increment. It indicates the number of values that are running within the writeout.
 for(k=0;k<no;k++)
("%d\t",frame[k]); // This is the number specified for the count
}
printf("\n");
}
printf("The fault for the page is located at %d",count); // this creates the print for where the fault is located within the FIFO
return 0;
}
```

Output


C:\Users\Ryan F\Documents\test 1.exe

```

 Please place the # of pages:
1

 Please place the specified page #:
3

 Please place the # of frames:4
        ref string       page frames
3                3       -1      -1      -1
The fault for the page is located at 1
--------------------------------
Process exited after 15.94 seconds with return value 0
Press any key to continue . . .
```

LRU

Important Notes:
I used a C++ Program

Just like FIFO, LRU for this algorithm, is using a queue. The queue for this algorithm is done using a doubly linked list. Here, the size of the queue, to its fullest, and the number of frames within the algorithm are equivalent to one another. This means that pages that are used the most, will be directed towards the front of the queue. This differs for the pages that are rarely used. They are the ones directed to the back of the queue. This is apart of the caching scheme. If the cache were to be full, the frame that isn't used the most will be removed to make space. With this caching, it allows a quick direct access to the memory. For this algorithm, I use what is known as a hash. With a hash, it can map out any form of data and or number that we would want. It takes our input number, which in this case is 0-7. And then it creates a fixed size for that designated value that we have entered. This is done by converting one value to another size.

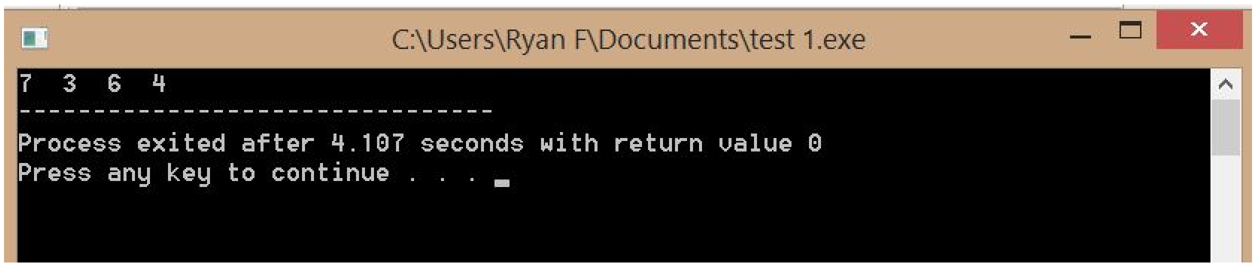
```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;


typedef struct QNode  //represents the parent and child relationship.This identifies that a Queue
node is to be used. In this case its the doubly linked list.

{
   QNode *prev, *next;
   unsigned pageNumber;  //this is the pagenumber represented in a binary format
} QNode;


typedef struct Queue  //This also the implementation of FIFO. This will be within the notation of
the Queue nodes.
{
unsigned count;
unsigned numberOfFrames; //This also gets a representation in binary
QNode *front, *rear;  // This designates which page is dropped, or brought to the front of the line
} Queue;
```

```
typedef struct Hash  //this identifies that hash will be used as a pointer for the node of Queue
{
int capacity;
QNode **array;  //indicates the aggregation of the qnodes within the array
} Hash;


QNode* newQNode(unsigned pageNumber)  //used for the creation of a Queue node
{
QNode* temp = new QNode;
temp->pageNumber = pageNumber;  //creation of of a Queue nodes tables
temp->prev = temp->next = NULL;
return temp;
}

Queue* createQueue(int numberOfFrames) //this is for the empty nodes of Queue. It
implements that some frames are either to be dropped, or to be moved higher up. Thus the
reasoning why the queue is left empty. This is for any sort of changes.
{
Queue* queue = new Queue;
queue->count = 0;
queue->front = queue->rear = NULL;
queue->numberOfFrames = numberOfFrames;
return queue;
}


Hash* createHash(int capacity)  // this is exactly like the empty queue, however it is also for the
hash. But unlike the frames, this is for the capacity. The size of the capacity can either increase
or decrease depending on the qnode.
{
Hash* hash = new Hash;
hash->capacity = capacity;
hash->array = new QNode* [hash->capacity];
int i;
for(i = 0; i < hash->capacity; ++i)
hash->array[i] = NULL;
return hash;
}

int AreAllFramesFull(Queue* queue)  //this function is used to check to see if there is any space
available within the memory.
{
```

```
return queue->count == queue->numberOfFrames;
}

int isQueueEmpty( Queue* queue )  // this is used to check if the queue in use is empty
{
return queue->rear == NULL;
}


void deQueue( Queue* queue )  // This command is used in order to delete a frame from within
the queue node. This is a frame that was used the least and has no importance to the algorithm
{
if (isQueueEmpty(queue))
Return;
if (queue->front == queue->rear)
queue->front = NULL;
QNode* temp = queue->rear;
queue->rear = queue->rear->prev;
if (queue->rear)
queue->rear->next = NULL;
free(temp);
queue->count--;
}

void Enqueue(Queue* queue, Hash* hash, unsigned pageNumber) //This is used in order for
the addition of a page. The page number is one that'll be added to both the queue and hash so
that its included within their tables.
{
if (AreAllFramesFull(queue))  // indicates that a page can be dropped
   {
hash->array[queue->rear->pageNumber] = NULL;  //page is dropped if full
deQueue(queue);
   }
QNode* temp = newQNode(pageNumber);
temp->next = queue->front;
if (isQueueEmpty(queue))
queue->rear = queue->front = temp;
Else
{
queue->front->prev = temp;
queue->front = temp;
}
hash->array[pageNumber] = temp;
```

```
queue->count++;
}

void ReferencePage(Queue* queue, Hash* hash, unsigned pageNumber)  //This is a reference
to a page number that is chosen by the user. However this page is coming from the memory
and or cache.
{
QNode* reqPage = hash->array[pageNumber];
if (reqPage == NULL)
Enqueue(queue, hash, pageNumber);
else if (reqPage != queue->front)
{
reqPage->prev->next = reqPage->next;
if (reqPage->next)
reqPage->next->prev = reqPage->prev;
if (reqPage == queue->rear)
{
queue->rear = reqPage->prev;
queue->rear->next = NULL;
}
reqPage->next = queue->front;
reqPage->prev = NULL;
reqPage->next->prev = reqPage;
queue->front = reqPage;
   }
}


int main()  // this is the start of our main algorithm. With all the coding above, it is used to
designated the proper value that we could input below.
{
Queue* q = createQueue(4);  // this creates the queue at a size of 4. Meaning after 4 pages, the
least used one of the four is removed.
Hash* hash = createHash( 10 ); // Since of value is from 0-7 I decided to use 10. This allows the
table to run without a fault or even redundancy.

// the lines below give the readings for both the hash and queue within the reference table. As
we can see from the output, 5 and 2 are dropped because they are the ones used the least.
ReferencePage(q, hash, 5);
ReferencePage(q, hash, 2);
ReferencePage(q, hash, 4);
ReferencePage(q, hash, 6);
ReferencePage(q, hash, 3);
ReferencePage(q, hash, 7);
cout<<q->front->pageNumber<<"  ";
cout<<q->front->next->pageNumber<<"  ";
cout<<q->front->next->next->pageNumber<<"  ";
cout<<q->front->next->next->next->pageNumber<<"  ";
return 0;
}
```

Output



C:\Users\Ryan F\Documents\test 1.exe

```
7  3  6  4
--------------------------------
Process exited after 4.107 seconds with return value 0
Press any key to continue . . .
```

Subproject #2:

Important Notes:

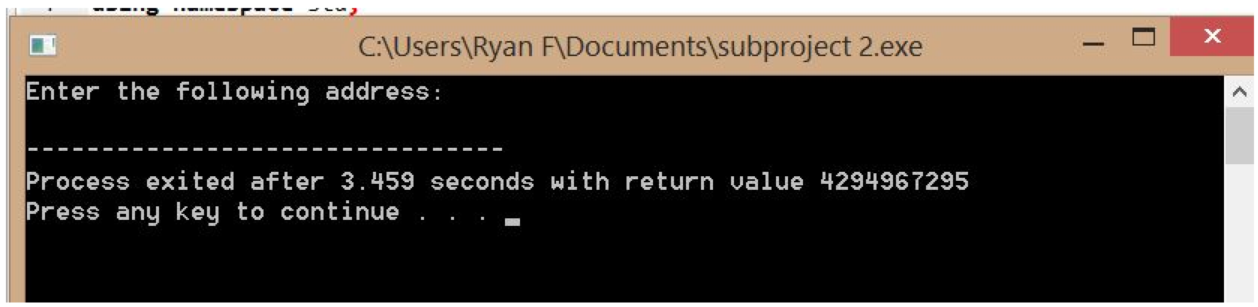
I used a C++ Program

For this project, we had to make the page number to 4. Since that the page number was 4, I converted it from 4 kilobytes to 4000 bytes. In this cause it was 4096 bits. This would end up being my page size. I next set the argument line set to less than 2. This is so that the image is loaded and displayed. This is because the page number is 4, thus finding a number that won't override the size. Next, I used the cout input "Enter the following address:." This portion of the project requires a particular value inputted. That would give us a page number and offset.

```
#include<iostream>
#include<stdlib.h>
#define PAGE_SIZE 4096  //this is used to define the kernel headers
using namespace std;
```

```
int main(int argc,char **argv)
{

if ( argc < 2 )  // My count for the algorithm will be less than 2
{
cout << "Enter the following address:\n"; // the first write out command
return -1;  // this is only used when an error occurs. This can be if a value is higher/lower than
the address  set
}

unsigned int address = atoi(argv[1]); //used for a higher range of values . This means that there
are no sort of constraints.

unsigned int page_number = address / PAGE_SIZE; // No constraints for the page size

unsigned int offset = address % PAGE_SIZE; // also no constraints for the page size percentage

cout << "The following address will be: " << address << " This will provide the page number: "
<< page_number << " And the offset will be equal to: " << offset << "\n";
}
```

Output



```
Enter the following address:

--------------------------------
Process exited after 3.459 seconds with return value 4294967295
Press any key to continue . . .
```