Quyen Dang
Hoang Nguyen

# COP4600 Project 2 Report

## I. Introduction

In this project, we are writing a C program to understand how applications reacts to different page replacement algorithms. By implementing First In First Out (FIFO), Least Recently Used (LRU) algorithms, and Segmented First In First Out, we keep track of the pages to be loaded in memory. With different algorithms, we have different result as the number of pages read from disk or the number of pages write to disk. Also, depend on which type of action with the page (Read or Write), the result is affected.

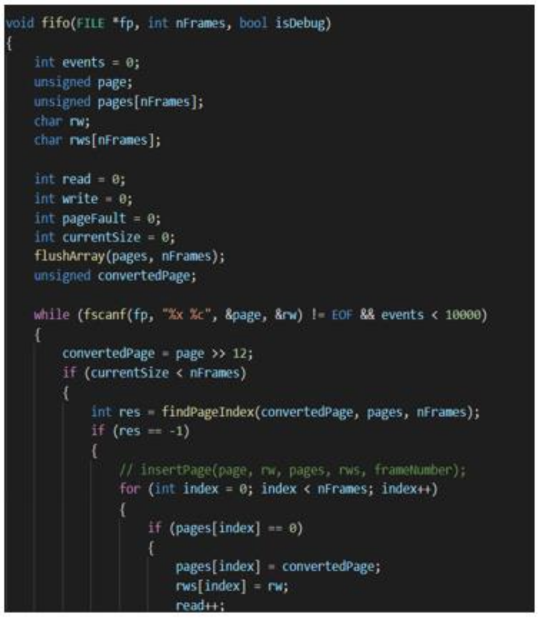
## II. Methods

At program start, we import the trace of the memory into the program. Then depending on the arguments/parameters provided on start of program, we will use either FIFO, LRU, or SFIFO to execute the program.

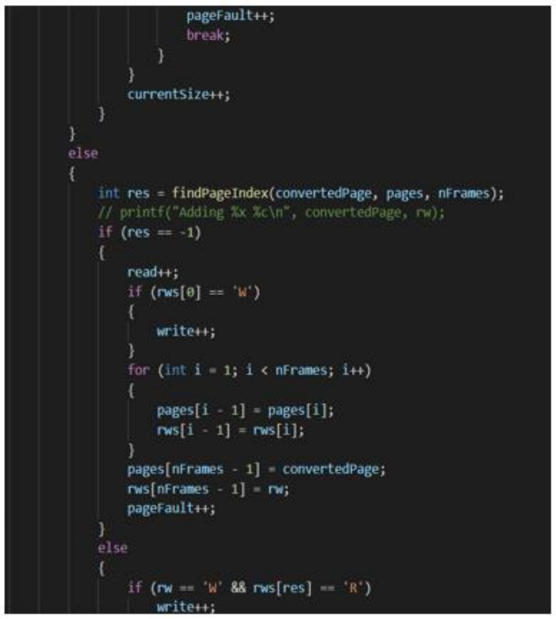
Firstly, we have the page and the frame have same size as 4KB (or $2^{12}$ B), so we have 12 bits of page number (same for virtual and physical address). Therefore, we shift each hexadecimal row in the trace file 12 bits to the right to get the actual page number before loading to the memory. After that, with each policy chosen, we implemented them as following:

### First In First Out (FIFO):

Firstly, while the frame slots are not full, we check whether the page to be loaded into memory (that means the page is already in the memory). If the page was not in the memory, then load it in the first available slot in the queue, and the read count is incensement by one. In case the slots are full, if the page is already loaded, ignore it. If page not loaded, then increment the read count. In addition to that, if the page that is being evicted (or replacement) is write, increment write count. Then load the new page into the top of the queue and repeat the process for every incoming page.



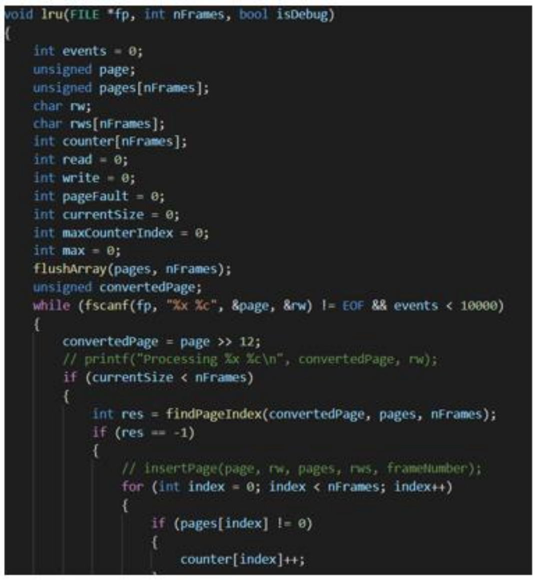
```c
void fifo(FILE *fp, int nFrames, bool isDebug)
{
    int events = 0;
    unsigned page;
    unsigned pages[nFrames];
    char rw;
    char rws[nFrames];

    int read = 0;
    int write = 0;
    int pageFault = 0;
    int currentSize = 0;
    flushArray(pages, nFrames);
    unsigned convertedPage;

    while (fscanf(fp, "%x %c", &page, &rw) != EOF && events < 10000)
    {
        convertedPage = page >> 12;
        if (currentSize < nFrames)
        {
            int res = findPageIndex(convertedPage, pages, nFrames);
            if (res == -1)
            {
                // insertPage(page, rw, pages, rws, frameNumber);
                for (int index = 0; index < nFrames; index++)
                {
                    if (pages[index] == 0)
                    {
                        pages[index] = convertedPage;
                        rws[index] = rw;
                        read++;
```



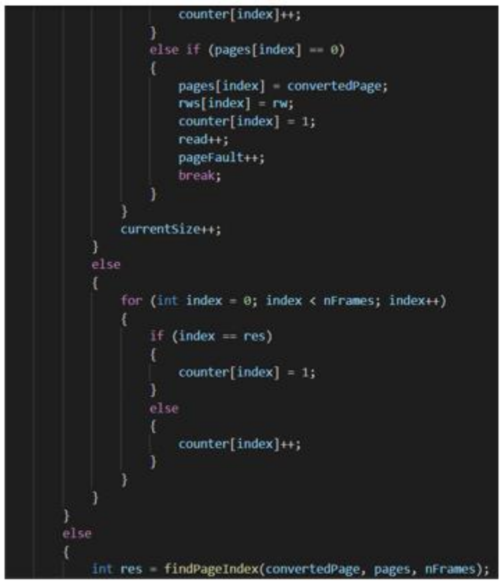
```c
                        pageFault++;
                        break;
                    }
                }
                currentSize++;
            }
        }
        else
        {
            int res = findPageIndex(convertedPage, pages, nFrames);
            // printf("Adding %x %c\n", convertedPage, rw);
            if (res == -1)
            {
                read++;
                if (rws[0] == 'W')
                {
                    write++;
                }
                for (int i = 1; i < nFrames; i++)
                {
                    pages[i - 1] = pages[i];
                    rws[i - 1] = rws[i];
                }
                pages[nFrames - 1] = convertedPage;
                rws[nFrames - 1] = rw;
                pageFault++;
            }
            else
            {
                if (rw == 'W' && rws[res] == 'R')
                    write++;
```

**Least Recently Used (LRU):**

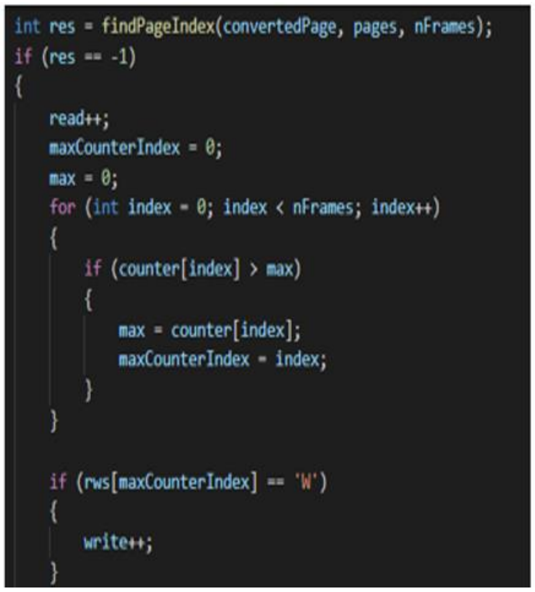
Firstly, we setup a counter array to keep tracking how frequently the page is used in the memory. Then while the frames are not all occupied, we then check if the page is loaded. In case the page is not yet loaded, load the page into the first available slot then set its counter to 1 and increment read by one. Besides, increment counter for all other pages that are already loaded in the frame. In case the page is already loaded, set the counter for that page to 1 and increment the counter for all other pages. When the frames are full, check whether the page is loaded. If not, increment read then find the page that has the highest value in the counter array to remove. Check whether that page is a write operation, increment write count if true. After that, replace that page to the newly loaded page and set the newly loaded page's counter to 1 and increment counter for all other pages. In case the page is already loaded, set its counter to 1 and increment the counter for all other pages. Repeat the process for every event.



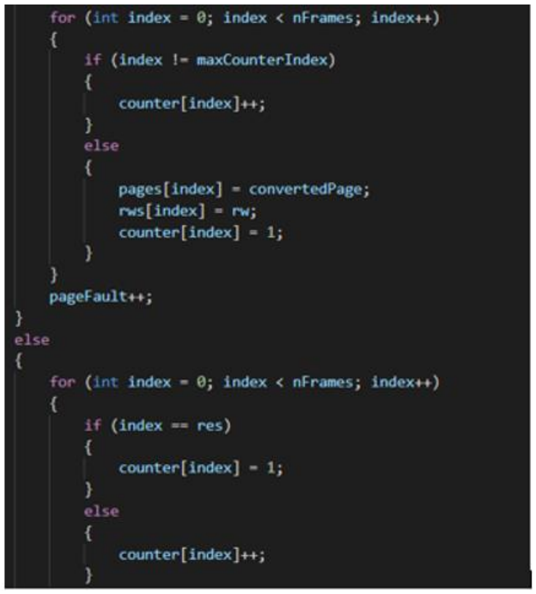
```c
void lru(FILE *fp, int nFrames, bool isDebug)
{
    int events = 0;
    unsigned page;
    unsigned pages[nFrames];
    char rw;
    char rws[nFrames];
    int counter[nFrames];
    int read = 0;
    int write = 0;
    int pageFault = 0;
    int currentSize = 0;
    int maxCounterIndex = 0;
    int max = 0;
    flushArray(pages, nFrames);
    unsigned convertedPage;
    while (fscanf(fp, "%x %c", &page, &rw) != EOF && events < 10000)
    {
        convertedPage = page >> 12;
        // printf("Processing %x %c\n", convertedPage, rw);
        if (currentSize < nFrames)
        {
            int res = findPageIndex(convertedPage, pages, nFrames);
            if (res == -1)
            {
                // insertPage(page, rw, pages, rws, frameNumber);
                for (int index = 0; index < nFrames; index++)
                {
                    if (pages[index] != 0)
                    {
                        counter[index]++;
```



```c
                        counter[index]++;
                    }
                    else if (pages[index] == 0)
                    {
                        pages[index] = convertedPage;
                        rws[index] = rw;
                        counter[index] = 1;
                        read++;
                        pageFault++;
                        break;
                    }
                }
                currentSize++;
            }
            else
            {
                for (int index = 0; index < nFrames; index++)
                {
                    if (index == res)
                    {
                        counter[index] = 1;
                    }
                    else
                    {
                        counter[index]++;
                    }
                }
            }
        }
        else
        {
            int res = findPageIndex(convertedPage, pages, nFrames);
```



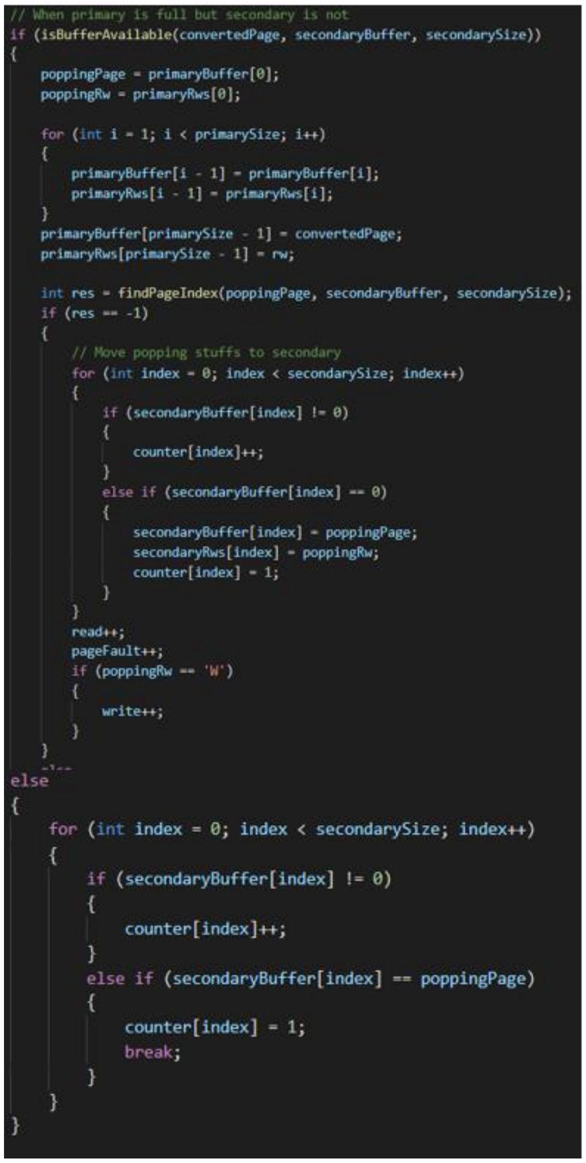
```c
int res = findPageIndex(convertedPage, pages, nFrames);
if (res == -1)
{
    read++;
    maxCounterIndex = 0;
    max = 0;
    for (int index = 0; index < nFrames; index++)
    {
        if (counter[index] > max)
        {
            max = counter[index];
            maxCounterIndex = index;
        }
    }

    if (rws[maxCounterIndex] == 'W')
    {
        write++;
    }
```



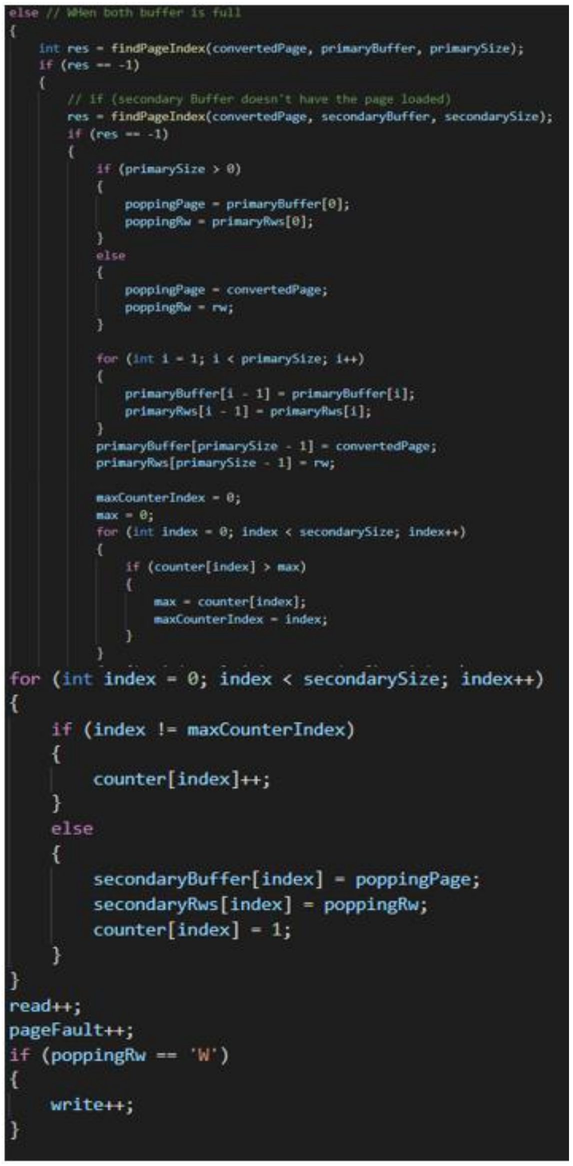
```c
    for (int index = 0; index < nFrames; index++)
    {
        if (index != maxCounterIndex)
        {
            counter[index]++;
        }
        else
        {
            pages[index] = convertedPage;
            rws[index] = rw;
            counter[index] = 1;
        }
    }
    pageFault++;
}
else
{
    for (int index = 0; index < nFrames; index++)
    {
        if (index == res)
        {
            counter[index] = 1;
        }
        else
        {
            counter[index]++;
        }
```

**Segmented FIFO (SFIFO)**

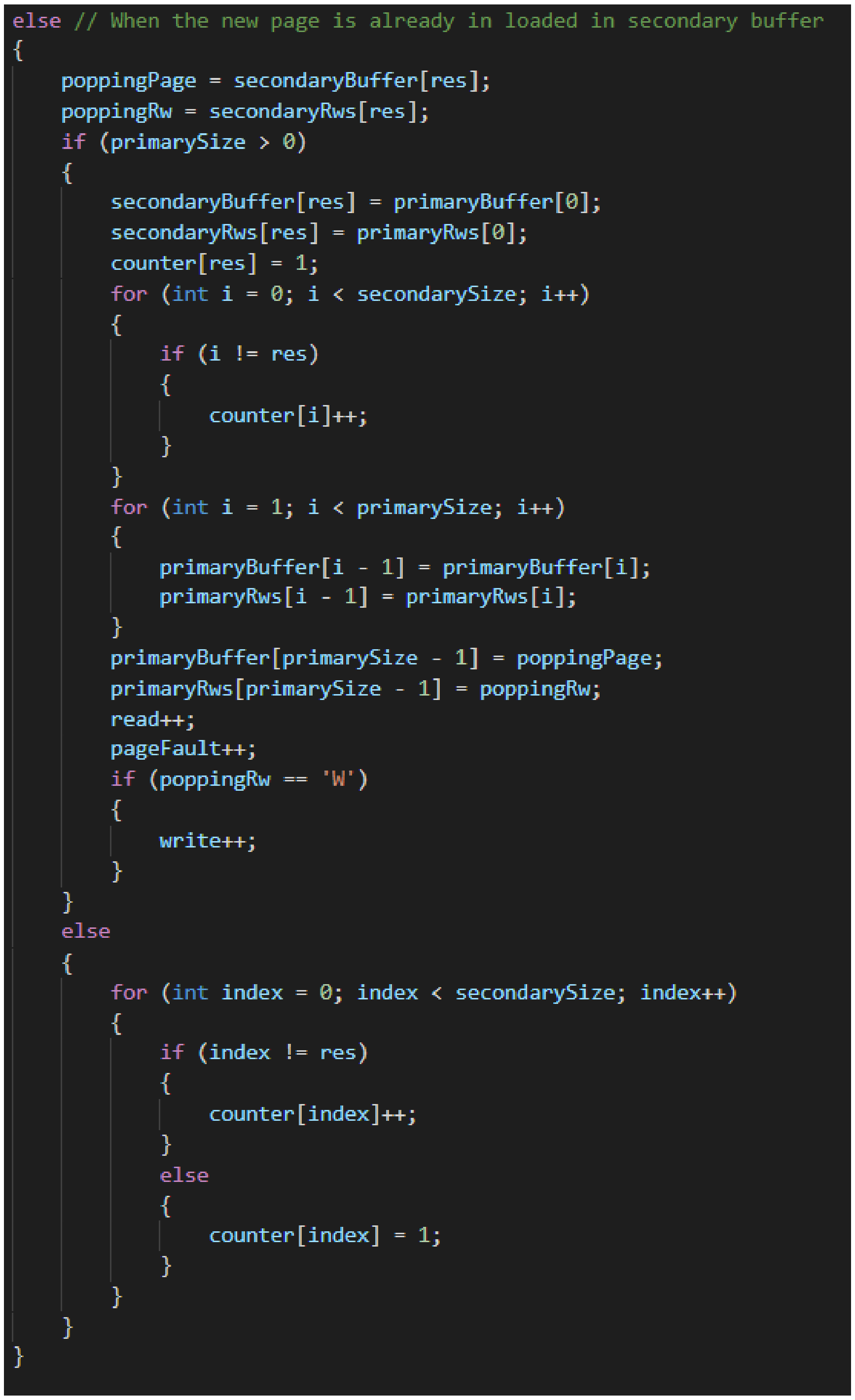
First, we have the “p” is loaded from the argument, which determine the percentage of the total program memory to be used in the secondary buffer (which is using the LRU algorithm). If p = 100, all the pages will use the LRU algorithm to load in memory, and there is no primary buffer, all the frames are secondary buffer. And if p = 0, there is no secondary buffer, all the frames is primary buffer. It is implemented as: while each page is loaded to the memory and the primary buffer are not full, if it is a miss, it would be loaded into the primary buffer (which is using the FIFO algorithm). If a page fault is loaded to the primary buffer when it full, the oldest page in the primary buffer (as FIFO) would move to the top of the secondary buffer. At this time, if the secondary buffer is full, the oldest page there would be remove from memory. However, if the page is load to the secondary buffer, that page would be removed from the secondary buffer, and placed on the top of the primary buffer (as the newest page).



```
// When primary is full but secondary is not
if (isBufferAvailable(convertedPage, secondaryBuffer, secondarySize))
{
    poppingPage = primaryBuffer[0];
    poppingRw = primaryRws[0];

    for (int i = 1; i < primarySize; i++)
    {
        primaryBuffer[i - 1] = primaryBuffer[i];
        primaryRws[i - 1] = primaryRws[i];
    }
    primaryBuffer[primarySize - 1] = convertedPage;
    primaryRws[primarySize - 1] = rw;

    int res = findPageIndex(poppingPage, secondaryBuffer, secondarySize);
    if (res == -1)
    {
        // Move popping stuffs to secondary
        for (int index = 0; index < secondarySize; index++)
        {
            if (secondaryBuffer[index] != 0)
            {
                counter[index]++;
            }
            else if (secondaryBuffer[index] == 0)
            {
                secondaryBuffer[index] = poppingPage;
                secondaryRws[index] = poppingRw;
                counter[index] = 1;
            }
        }
        read++;
        pageFault++;
        if (poppingRw == 'W')
        {
            write++;
        }
    }
else
{
    for (int index = 0; index < secondarySize; index++)
    {
        if (secondaryBuffer[index] != 0)
        {
            counter[index]++;
        }
        else if (secondaryBuffer[index] == poppingPage)
        {
            counter[index] = 1;
            break;
        }
    }
}
```



```
else // WHen both buffer is full
{
    int res = findPageIndex(convertedPage, primaryBuffer, primarySize);
    if (res == -1)
    {
        // if (secondary Buffer doesn't have the page loaded)
        res = findPageIndex(convertedPage, secondaryBuffer, secondarySize);
        if (res == -1)
        {
            if (primarySize > 0)
            {
                poppingPage = primaryBuffer[0];
                poppingRw = primaryRws[0];
            }
            else
            {
                poppingPage = convertedPage;
                poppingRw = rw;
            }

            for (int i = 1; i < primarySize; i++)
            {
                primaryBuffer[i - 1] = primaryBuffer[i];
                primaryRws[i - 1] = primaryRws[i];
            }
            primaryBuffer[primarySize - 1] = convertedPage;
            primaryRws[primarySize - 1] = rw;

            maxCounterIndex = 0;
            max = 0;
            for (int index = 0; index < secondarySize; index++)
            {
                if (counter[index] > max)
                {
                    max = counter[index];
                    maxCounterIndex = index;
                }
            }
for (int index = 0; index < secondarySize; index++)
{
    if (index != maxCounterIndex)
    {
        counter[index]++;
    }
    else
    {
        secondaryBuffer[index] = poppingPage;
        secondaryRws[index] = poppingRw;
        counter[index] = 1;
    }
}
read++;
pageFault++;
if (poppingRw == 'W')
{
    write++;
}
```



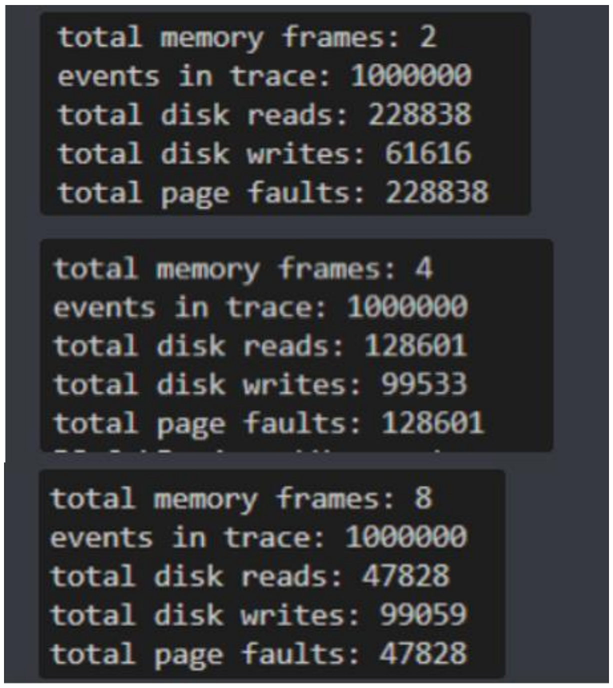
```
else // When the new page is already in loaded in secondary buffer
{
    poppingPage = secondaryBuffer[res];
    poppingRw = secondaryRws[res];
    if (primarySize > 0)
    {
        secondaryBuffer[res] = primaryBuffer[0];
        secondaryRws[res] = primaryRws[0];
        counter[res] = 1;
        for (int i = 0; i < secondarySize; i++)
        {
            if (i != res)
            {
                counter[i]++;
            }
        }
        for (int i = 1; i < primarySize; i++)
        {
            primaryBuffer[i - 1] = primaryBuffer[i];
            primaryRws[i - 1] = primaryRws[i];
        }
        primaryBuffer[primarySize - 1] = poppingPage;
        primaryRws[primarySize - 1] = poppingRw;
        read++;
        pageFault++;
        if (poppingRw == 'W')
        {
            write++;
        }
    }
    else
    {
        for (int index = 0; index < secondarySize; index++)
        {
            if (index != res)
            {
                counter[index]++;
            }
            else
            {
                counter[index] = 1;
            }
        }
    }
}
```

After the executions, we output the results to terminal with the format in section III.

## III. Results

Testing the output for first 1,000,000 events from bzip.trace, and with 2, 4, and 8 frames with different algorithm FIFO and LRU, we have the result:

FIFO:

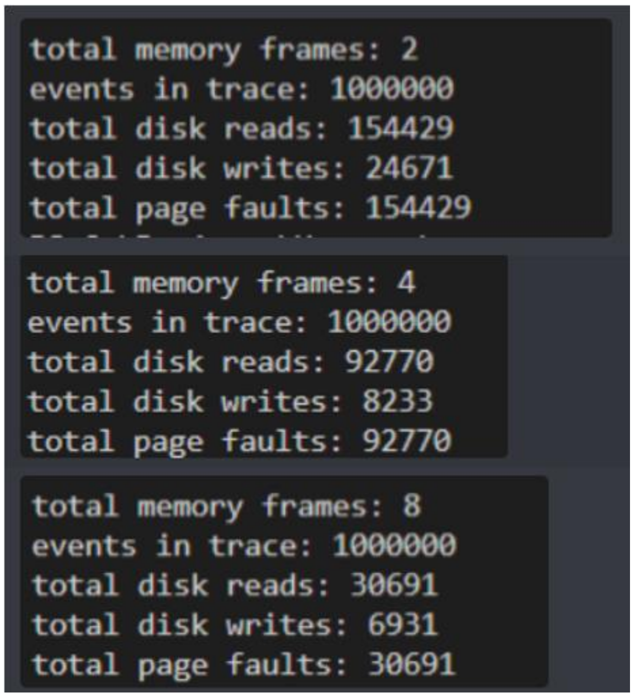


```
total memory frames: 2
events in trace: 1000000
total disk reads: 228838
total disk writes: 61616
total page faults: 228838
```

```
total memory frames: 4
events in trace: 1000000
total disk reads: 128601
total disk writes: 99533
total page faults: 128601
```

```
total memory frames: 8
events in trace: 1000000
total disk reads: 47828
total disk writes: 99059
total page faults: 47828
```

LRU:

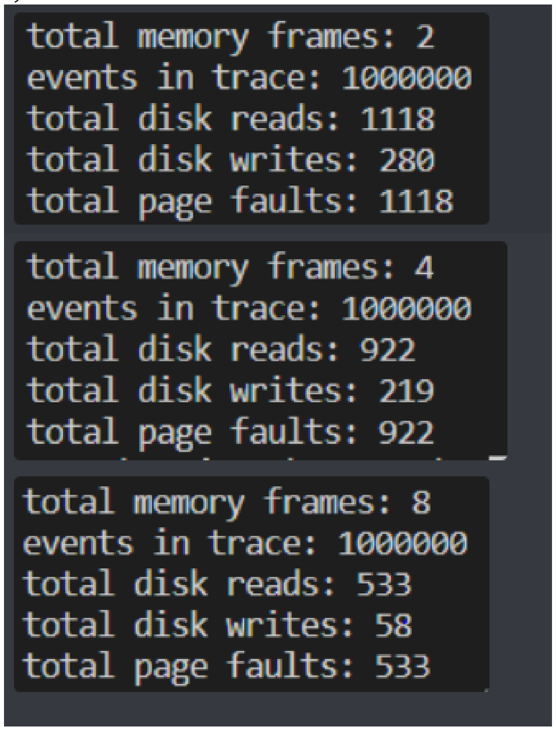


```
total memory frames: 2
events in trace: 1000000
total disk reads: 154429
total disk writes: 24671
total page faults: 154429
```

```
total memory frames: 4
events in trace: 1000000
total disk reads: 92770
total disk writes: 8233
total page faults: 92770
```

```
total memory frames: 8
events in trace: 1000000
total disk reads: 30691
total disk writes: 6931
total page faults: 30691
```

Testing the output for first 1,000,000 events from bzip.trace, and with 2, 4, and 8 frames with SFIFO algorithm P = 50, we have the result:

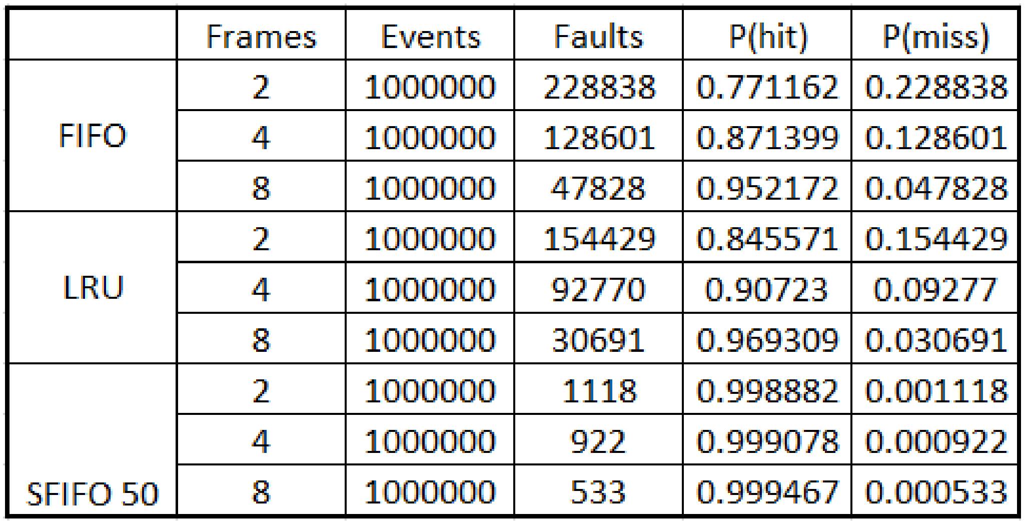


```
total memory frames: 2
events in trace: 1000000
total disk reads: 1118
total disk writes: 280
total page faults: 1118
```

```
total memory frames: 4
events in trace: 1000000
total disk reads: 922
total disk writes: 219
total page faults: 922
```

```
total memory frames: 8
events in trace: 1000000
total disk reads: 533
total disk writes: 58
total page faults: 533
```

Hit and miss rate table:



| | Frames | Events | Faults | P(hit) | P(miss) |
|---|---|---|---|---|---|
| | 2 | 1000000 | 228838 | 0.771162 | 0.228838 |
| FIFO | 4 | 1000000 | 128601 | 0.871399 | 0.128601 |
| | 8 | 1000000 | 47828 | 0.952172 | 0.047828 |
| | 2 | 1000000 | 154429 | 0.845571 | 0.154429 |
| LRU | 4 | 1000000 | 92770 | 0.90723 | 0.09277 |
| | 8 | 1000000 | 30691 | 0.969309 | 0.030691 |
| | 2 | 1000000 | 1118 | 0.998882 | 0.001118 |
| | 4 | 1000000 | 922 | 0.999078 | 0.000922 |
| SFIFO 50 | 8 | 1000000 | 533 | 0.999467 | 0.000533 |

## IV. Conclusions

For this project, we designed and implemented a C program to simulate the behaviors of real-life application when it is executed using three different page replacement algorithms including First In First Out, Least Recently Used, and Segmented First In First Out. After spending 8 days analyzing, understanding, and brainstorming the project description, we spent the rest of the time doing our best to achieve the correct output from the sample given. With the current results, it can be said that with the same events and the same number of frames provided, LRU policy can proceed less page faults than FIFO. However, the LRU algorithm need extra memory to store and count the frequency of the page that accessed in memory, while FIFO does not need that counter. Therefore, to combine those advantage of both algorithms FIFO and LRU, we have the better algorithm is SFIFO, which reduce amount of the fault pages.