King Abdulaziz University
Faculty of Engineering
Department of Electrical Engineering

EE463 | Operating Systems

Project B
OSP2 Memory Module

Date of Submission: 19th of April 2020

Student Name	ID
Rawan Alkhalaf	1605159
Dana Alahdal	1607540

Instructor:
Dr. Abdulghani Al-Qasimi

Table of Contents

Introduction

Memory is an integral part of a computer's hardware whether it's a powerful server, a dedicated system, or a simple microcontroller. In order to process programs on a computer, the said program must first load onto the memory then the CPU fetches it for execution. The concept itself is rather simple, but as computers evolved, the processes allocated into the physical memory are limited to the size of the memory itself. Therefore, the virtual memory is utilized to allow the execution of processes that do not reside nor fit in the physical memory, but rather stores them in a much larger abstract memory.

This project implements the five public classes of the *Memory* module in OSP2 simulator. It utilizes the virtual memory as it employs an enhanced second chance and first in first out page replacement algorithms. In this report, the proposed system's architecture is first discussed. Next, the *Memory* module is implemented in Java Programming Language. The implemented system is then simulated on OSP2 using different parameter files. Finally, the system's performance is studied, and a comparative analysis is conducted between the page replacement algorithms.

System Design

System Architecture

Two different memories are presented in this system: the virtual memory, and the physical memory. The virtual memory is segmented into fixed sized *pages* that fits into the fixed sized *frames* of the physical memory. However, the virtual memory is considerably larger than the size of the physical memory. Hence, the memory management unit (MMU) maps the logical address of a page to the physical address of the frame using a page table. The page table verifies if the page is legal and in memory or not where each page is set to either a valid bit of 1 or an invalid of 0, respectively. Occasionally, when a process tries to allocate a page that is not in the memory, invalid, it initiates a page-fault. Not only that, but sometimes there is no free frame that could accommodate this page due to the full allocation of the physical memory. In order to overcome this issue, this system handles the page-fault by implementing page replacement algorithms, the process of choosing the victim frame to swap out another page and swap in the new page into the frame [1]. This system implements the first in first out (FIFO) and the enhanced second chance algorithms (ESC).

The first implemented page replacement algorithm is FIFO where the oldest residing page in the memory is replaced with the new one. This can be achieved by creating a queue where all the residing pages in the memory are held, so the new page is inserted at the tail while the oldest page is removed from the head. However, the proposed system implements an alternate method which is to timestamp each page, so the oldest page is identified as the one with the longest residing time and is replaced. FIFO is known for its simplicity and ease in programming.

On the other hand, the enhanced second chance (ESC) algorithm is a derivation of FIFO, but it is associated with a reference bit and a dirty bit. The reference bit indicates if the page was recently used or not by setting the bit to 1 and 0, respectively. Whereas the dirty bit indicates if the pages' content was modified. Therefore, this algorithm frees the frame, known as *clean* frame, which has a page that is not dirty, not referenced, not locked and not reserved. To do so, the algorithm should check the frame table a maximum of two times, the reference bit of the frame, if it's set to 1, gives it a second chance of staying in the frame table by clearing it to 0. giving it a second chance to find the desired *clean* frame to replace its page with the newer page. In ESC, the optimum pages to replace are shown in Table 1.

Table 1: ESC reference and dirty bits interpretation

Reference Bit	**Dirty Bit**	**Algorithm Interpretation**
0	0	Best page to replace
0	1	Second best page to replace
1	0	Probably will be used again - keep
1	1	Probably will be used again soon - keep

Implementation

This section provides the detailed implementation of each of the five classes and their methods. A brief introduction is given for the role of each class then the code is presented.

Class FrameTableEntry

A frame table is an array that holds the physical frames of the main memory. This class only enables the entries in the frame table so that a frame can be declared of as a `FrameTableEntry` object in other classes. Since this class extends `IflFrameTableEntry`, it inherits the methods of its superclass. The attributes of the said frame entry can be obtained and manipulated by the inherited methods. Such methods indicate if a thread reserved this frame (reserved flag), a page occupied the frame, the lock count on the frame, and if the contents of the frame are altered from the previous page that occupied it (dirty flag).

FrameTableEntry.FrameTableEntry(int frameID)

The constructor of this class only calls `super(frameID)` to refer to the superclass. Thus, it enables the inheritance of all the superclass methods mentioned above to the indicated frame.

```
public FrameTableEntry(int frameID)
    {
        super(frameID);
    }
```

Class PageTableEntry

Similar to `FrameTableEntry`, the class `PageTableEntry` implements the page entries, associates the pages with the frames, and detects pagefaults. Also, it inherits methods from its superclass, `IflPageTableEntry`.

PageTableEntry.PageTableEntry(PageTable ownerPageTable, int pageNumber)

This is the constructor method for `PageTableEntry`. It only denotes that this page entry is to `ownerPageTable` page table and holds the integer `pageNumber`. Not only that, but the constructor also creates a time stamp to the page to keep track of its residing time in the frame for when invoking the FIFO page replacement algorithm in `PageFaultHandler` class.

```
private long timeStamp;
```

```
public PageTableEntry(PageTable ownerPageTable, int pageNumber)
{
        super(ownerPageTable, pageNumber);
        this.setTimeStamp(HClock.get());

}
```

PageTableEntry.getTimeStamp()

This is only a simple *get* method to return the timestamp of the frame. This method is used in other classes whenever the timestamp of the page is needed.

```
public long getTimeStamp()
{
       return timeStamp;
}
```

PageTableEntry.setTimeStamp (long time)

A *set* method is also required to associate the time with the page.

```
public void setTimeStamp(long time)
{
       this.timeStamp = time;
}
```

PageTableEntry.do_lock(IORB iorb)

If a page is allocated into a frame, then this page is said to be *locked* in the frame. In simple terms, this can be done by incrementing the lock count of the frame (declared of type `FrameTableEntry` as discussed in the previous section) only if the page is valid (legal and in memory). Once the frame's lock is incremented, the program returns `SUCCESS`.

However, a pagefault is initiated if the page is invalid. One of the inherited methods from the superclass, `getValidatingThread()`, identifies the thread that caused the pagefault as `pagefaultThread` below. If the thread (`pagefaultThread`) is not involved in the pagefault and the page is not valid, then a *`pagefaulthandler`* is initiated to the argument thread that is referencing the page (*`iorbThread`*).

On the other hand, if the thread `pagefaultThread` is involved in the pagefault and is not equal to the page's thread `iorbThread`, then this page's thread will be suspended until the page is valid. In addition, the program must return failure if the created IORB thread gets killed.

```
public int do_lock(IORB iorb)
{
        ThreadCB iorbThread = iorb.getThread();

       if(!this.isValid())
       {
       ThreadCB pagefaultThread = this.getValidatingThread(); //Thread that caused pagefault

             if (pagefaultThread == null)
             {
                  //When page in invalid, initiate pagefault
                  if (PageFaultHandler.handlePageFault(iorbThread,MemoryLock,this) == FAILURE)
                      return FAILURE;
              }

             else
             {
                   if (pagefaultThread != iorbThread)
                  {
                        iorbThread.suspend(this); //wait until the page becomes valid
                        if (iorbThread.getStatus() == ThreadKill)
                            return FAILURE;
                   }
              }
         }
              /*Page is valid*/
              //Increment lock count on the frame associated with the page

              FrameTableEntry frame = this.getFrame();
              frame.incrementLockCount();

              return SUCCESS;
    }
```

PageTableEntry.do_unlock()

Here, the frame is unlocked once the page is detached from the frame. Instead of incrementing the lock, the frame here decrements the lock count. It should be noted that the lock count must always be a positive integer, never negative.

```
public void do_unlock()
    {
        // your code goes here
        if(getFrame().getLockCount()<= 0)
        {
            return;
        }
```

```
        else
        {
             getFrame().decrementLockCount();
        }

    }
```

Class PageTable

Unlike the frame table, the page table is a virtual memory that is not set in the architecture of OSP2. Therefore, the page table needs to be implemented while initializing its pages and freeing the frames with terminated tasks. Just as all the other classes, this class inherits from its corresponding superclass, `IflPageTable`.

PageTable.PageTable(TaskCB ownerTask)

In this method, the page, a variable inherited from `IflPageTable`, is declared as an array of type `PageTableEntry`. According to the size (no. of address bits) of a single page, the size of the page table which is the maximum number of pages is determined as follows:

$$Page\ Table\ size = 2^{page\ size}$$

Furthermore, the pages are initialized by calling the constructor of the `PageTableEntry` class for each page number. A for loop is used to initialize each page according to the size its array. With this, the page table is constructed with initialized pages within it.

```
public PageTable(TaskCB ownerTask)
{
   super(ownerTask);

   // page table is an array of size equal to the maximum number of pages allowed
   //calculate maximal number of pages allowed
   int MaxnumberOfPages = (int)Math.pow(2, MMU.getPageAddressBits());

   pages = new PageTableEntry[MaxnumberOfPages];

    // initialize each page with page table entry
    for (int i = 0; i < MaxnumberOfPages; i++)
    {
        pages[i] = new PageTableEntry(this, i);
    }
}
```

PageTable.do_deallocateMemory()

When a task is terminated, the frame in the main memory, frame table, that was previously occupied by the said task should be freed. This is done by looping through the entire frame table, retrieving the frame, the frame's associated page, and checking if the frame's occupying task is the same as the terminated one and the page is not null. If so, the frame should set its associated page to null, clean the page, unset the reference bit, and unreserve the frame that was reserved by the said task.

```
public void do_deallocateMemory()
{
        TaskCB pagetableTask = getTask();
        for(int i = 0; i < MMU.getFrameTableSize(); i++)
        {
            FrameTableEntry frame = MMU.getFrame(i);
            PageTableEntry page = frame.getPage();
            if(page != null && page.getTask() == pagetableTask)
            {
                frame.setPage(null);
                frame.setDirty(false);
                frame.setReferenced(false);
                if(frame.getReserved() == pagetableTask)
                    frame.setUnreserved(pagetableTask);
            }
        }
}
```

Class MMU

The memory management unit is a hardware that handles all the memory operations within an operating system. Hence, this class is a representation of the said hardware in the sense that it is able to map the page from the virtual memory to the frame in the main memory. In short, it manages all memory operations associated with the processor.

MMU.init()

This constructor of the MMU initializes the frame table itself. Since the frame table is the main memory that is part of the operating system's architecture, its size is known. Each real frame is then allocated to the entries of type FrameTableEntry. Here, two global variables are initialized, `Cursor`, which points to the frame in the table, and `wantFree,` which indicates the number of free frames needed by the thread causing the page fault.

```
public static int Cursor;
public static int wantFree;
public static void init()
{
	Cursor=0;
	wantFree=1;

	for (int i = 0; i < MMU.getFrameTableSize(); i++)
	{
		setFrame(i, new FrameTableEntry(i));
	}

}
```

MMU.do_refer(int memoryAddress, int referenceType, Thread CB)

This method takes as parameters the physical memory address, the memory reference type, and the thread that made the reference. In order to return the page of the thread's logical memory to which the reference was made. It first computes the page number using $Page\ Number = \frac{Memory\ address}{2^{virtual\ address\ bits - page\ address\ bits}}$. Then, that page is checked for validity, if it is, the referenced and dirty bits of the frames are set and the page returned. Else, if the page has no validating thread, a page fault is initiated using `CPU.interrupt(PageFault)`. But if a page fault is caused by a different thread, the thread that made the reference is suspended. Finally, the referenced and dirty bits of the frames are set anf the page returned.

```
static public PageTableEntry do_refer(int memoryAddress, int referenceType, ThreadCB thread)
{
        // find the page to which the reference was made
        // pageNo = address/2^offset
        int pageNumber = memoryAddress / (int)Math.pow(2.0, MMU.getVirtualAddressBits() -
MMU.getPageAddressBits());

         //get pageTableEntry
          PageTableEntry page = getPTBR().pages[pageNumber];

          if(page.isValid())
          {
                    page.getFrame().setReferenced(true);

                    if (referenceType == GlobalVariables.MemoryWrite)
                    {
                        page.getFrame().setDirty(true);
                    }

                    return page;
           }
           else
           {
                    if(page.getValidatingThread() == null)
                    {
                        //if the pageFault occurs due to the original thread, set pagefault as
true and handle the pagefault

                        InterruptVector.setInterruptType(referenceType);
                        InterruptVector.setPage(page);
                        InterruptVector.setThread(thread);

                        CPU.interrupt(PageFault);

                        if(thread.getStatus() == GlobalVariables.ThreadKill)
                        {
                            return page;
                        }
                    }

                    else
                    {
                        //if pagefault occurs due to some other thread, suspend this thread
and wait

                        thread.suspend(page);
```

```
                    if(thread.getStatus() == GlobalVariables.ThreadKill)
                    {
                        return page;
                    }
                }
        }

        page.getFrame().setReferenced(true);

         if (referenceType == GlobalVariables.MemoryWrite)
         {
                //set the dirty bit as true if reference type was write
                page.getFrame().setDirty(true);
         }

         return page;
}
```

Class PageFaultHandler

This class is initiated in the case of page fault due to three memory reference types, memoryLock, memoryRead, and memoryWrite. If no free frame are available, this class will initiate the page replacement algorithm, ESC or FIFO, to find a victim frame for swapping out its current page and swapping in the new page. If there is a free frame, only the swapping in operation if preformed.

PageFaultHandler.do_handlePageFault(Thread CB thread, int referenceType, PageTableEntry page)

This method take as input the thread that caused the page fault, the memory reference type, and the page that needs to be swapped in. it returns an integer ***FAILURE*** if the page that caused the page fault is not valid or null, and if a thread is killed during swapping procedures. ***SUCCESS*** is returned if a page fault was successfully handled and the page replacement algorithm has worked. It returns ***NotEnoughMemory*** if all the frames in the frame table are locked or reserved.

The first thing this method does is check for the conditions of ***FAILURE*** and ***NotEnoughMemory*** and returns them if the conditions are met. If not, a system event object is initialized for the page fault and the thread that caused it suspended. Next, the validating thread, which is the thread that caused the page fault on the page, is set for that page.

Next, the number of free frames is checked using the created *numFreeFrames*(), if there are no free frames, a page replacement algorithm, fifo or ESC, is called to find the victim frame. If there are, the created method *getFreeFrame*(), is called to assign a free frame.

Once a victim frame is chosen, this method reserves the frame if it's not already reserved. Then checks if the frame contains a dirty page (a page that is not null) to perform the swap out operation by first obtaining the swap out file using `getSwapFile().write()` from the OpenFile class, which is a handle that contains information about the disk blocks used by the file. Then the frame is freed, and the page table is updated. After, swapping out, a swapping in operation is preformed using `getSwapFile().read()`, to insert the new page into the frame. if while preforming any of these two operations the thread is killed, then the method returns failure after notifying the threads. If the frame contains a null page (clean), then only swapping in is performed.

Finally, the page table is updated and the refenced and dirty bits are set then the threads are notified and the thread that caused the page fault is dispatched.

Note: the user option is coded but commented for the sake of running this program on Eclipce.

```
static int pageFault=0; //to count the number of page faults


public static int do_handlePageFault(ThreadCB thread, int referenceType, PageTableEntry page)


{

        if (page.isValid() || page==null)
                return FAILURE;

        int frameCount=0;
        for(int i=0; i<MMU.getFrameTableSize(); i++){
       if(MMU.getFrame(i).getLockCount() != 0 || MMU.getFrame(i).isReserved() == true)
           frameCount++;
    }

    //No frame is available
if(MMU.getFrameTableSize() == frameCount)
    return NotEnoughMemory;

pageFault++;

FrameTableEntry frame=null;

SystemEvent systemEvent = new SystemEvent("PageFault");
thread.suspend(systemEvent);

page.setValidatingThread(thread);

if (numFreeFrames() == 0)
        //if(userOption.equals("Fifo"))
                //frame = Fifo();
                //else if(userOption.equals("SecondChance"))
                //frame=SecondChance();
        //else
                frame=SecondChance();
else
        frame = getFreeFrame();

if(!frame.isReserved())
        frame.setReserved(thread.getTask());

        // if the frame contains a dirty page.

PageTableEntry prevPage = frame.getPage();

if(prevPage != null) {

        if (frame.isDirty()) {

                // swap out
        TaskCB swapTask = frame.getPage().getTask();
        swapTask.getSwapFile().write(prevPage.getID(), prevPage, thread);

        if (thread.getStatus() == ThreadKill) {


                systemEvent.notifyThreads();
                page.notifyThreads();
                ThreadCB.dispatch();
                return FAILURE;
        }

        frame.setDirty(false);

        frame.setPage(null);
        frame.setReferenced(false);
        prevPage.setValid(false);
        prevPage.setFrame(null);
```

```
        } else {
        prevPage.getFrame().setReferenced(false);
        prevPage.setValid(false);

        if(prevPage.getFrame().getLockCount() == 0)
                prevPage.setFrame(null);
        }

page.setFrame(frame);

// Swapping in the pages.

TaskCB task = page.getTask();
task.getSwapFile().read(page.getID(), page, thread);

        if (thread.getStatus() == ThreadKill) {
                if (frame.getPage() != null)
                        if (frame.getPage().getTask() == thread.getTask())
                                frame.setPage(null);

                page.notifyThreads();
                page.setValidatingThread(null);
                page.setFrame(null);

                systemEvent.notifyThreads();
                ThreadCB.dispatch();
                return FAILURE;
        }
} else { //if frame contains clean page

page.setFrame(frame);

 // Swapping in the pages.

TaskCB task = page.getTask();
task.getSwapFile().read(page.getID(), page, thread);

if (thread.getStatus() == ThreadKill) {
        if (frame.getPage() != null)
                if (frame.getPage().getTask() == thread.getTask())
                        frame.setPage(null);

                        page.notifyThreads();
                        page.setValidatingThread(null);
                        page.setFrame(null);

                        systemEvent.notifyThreads();
                        ThreadCB.dispatch();
                        return FAILURE;
                }

}

        frame.setPage(page);
        page.setValid(true);

        frame.setReferenced(true);

        if(referenceType==MemoryWrite)
                frame.setDirty(true);

        if (frame.getReserved() == thread.getTask())
                frame.setUnreserved(thread.getTask());

        page.setValidatingThread(null);
        page.notifyThreads();

        systemEvent.notifyThreads();
        ThreadCB.dispatch();

        System.out.println(pageFault);

        return SUCCESS;
}
```

PageFaultHandler.numFreeFrames()

To find the number of the free frames in the frame table, this method is called, it returns an integer of the number of the free frames. This method check the three conditions that constitute a free frame which are:

1. No page is associated to frame, meaning the frame holds a null page
2. The frame is not reserved by thread
3. The frame is not locked for an I/O operation

```
public static int numFreeFrames()

{
        int freeFrames=0;

        for(int i=0;i<MMU.getFrameTableSize();i++) {
                if(MMU.getFrame(i).getPage()==null &&
                   !MMU.getFrame(i).isReserved() &&
                   MMU.getFrame(i).getLockCount()==0)
                        freeFrames++;
        }

        return freeFrames;

}
```

PageFaultHandler.getFreeFrame()

This method returns the first free frame of type `FrameTableEntry` by checking the aforementioned conditions.

```
public static FrameTableEntry getFreeFrame()

{
        FrameTableEntry frame =null;

        if(numFreeFrames()!= 0) {

        for(int i=0;i<MMU.getFrameTableSize();i++) {
                if(MMU.getFrame(i).getPage()==null &&
                   !MMU.getFrame(i).isReserved() &&
                   MMU.getFrame(i).getLockCount()==0) {
                        frame=MMU.getFrame(i);
                        break;
                }
        }
        return frame;

        }

        else
                return null;

}
```

PageFaultHandler.secondChance()

This method implements the enhanced second chance page replacement algorithm to return the victim frame for swapping out. It is divided into two for loops. During the first loop, the reference bit of referenced frames is set to false, as to give it a second chance to stay in the frame table, and

the cursor updated using modulus arithmetic by `(MMU.Cursor+1) % MMU.getFrameTableSize()`. Next, the frame is checked if its clean, it's clean if it holds a page, not referenced, not locked, not reserved, and not dirty. If such frame is found, this frame is freed, and the page table is updated by removing the frame's previous page. Still inside the if statement, the method check if the number of the free frames is equal to `wantFree`, if it is, `getFreeFrame()` is returned.

In the second loop, if no frame is returned from the first cycle, the method rechecks the clean frame conditions after the referenced bits are cleared from the first loop and returns the victim frame. Finally, if not free frame is found in the first and second loops, the method return the first dirty frame it found since it is the next best thing.

```
public static FrameTableEntry SecondChance()

{
        FrameTableEntry frame;

        int dirtyFrameID = 0;
        int numCleanframes=0;
        boolean foundDrityFrame=false;

        //phase 1

for(int i=0; i<MMU.getFrameTableSize();i++) {

        frame=MMU.getFrame(MMU.Cursor);

        if(foundDrityFrame==false && frame.getLockCount()<=0 && !frame.isReserved() && frame.isDirty()) {
                dirtyFrameID=frame.getID();
                foundDrityFrame=true;
        }

        if(frame.isReferenced()) {
                frame.setReferenced(false);
                MMU.Cursor  = (MMU.Cursor+1) % MMU.getFrameTableSize();
}


frame=MMU.getFrame(MMU.Cursor);

//find clean frame
if(frame.getPage()!=null && !frame.isReferenced() &&
   frame.getLockCount()==0 && !frame.isReserved() && !frame.isDirty()) {

PageTableEntry prevPage = frame.getPage();

//free the frame
frame.setPage(null);
frame.setReferenced(false);
frame.setDirty(false);

//update page table
prevPage.setValid(false);
prevPage.setFrame(null);

//phase 3
        numCleanframes = numFreeFrames();
        if(numCleanframes==MMU.wantFree)
         return getFreeFrame();
}


}


//cycle again
```

```
if(numCleanframes!=MMU.wantFree) {

        for(int i=0; i<MMU.getFrameTableSize();i++) {

                frame=MMU.getFrame(MMU.Cursor);

                if(foundDrityFrame==false && frame.getLockCount()<=0 && !frame.isReserved() &&
frame.isDirty()) {
                        dirtyFrameID=frame.getID();
                        foundDrityFrame=true;
                }

                if(frame.isReferenced()) {
                        frame.setReferenced(false);
                }



                //find clean frame
if(frame.getPage()!=null && !frame.isReferenced())
        if(frame.getLockCount()==0 && !frame.isReserved() && !frame.isDirty()) {

                PageTableEntry prevPage = frame.getPage();

                //free the frame
        frame.setPage(null);
    frame.setReferenced(false);
    frame.setDirty(false);


    //update page table for previous page
    prevPage.setValid(false);
    prevPage.setFrame(null);


                numCleanframes = numFreeFrames();

        if(numCleanframes==MMU.wantFree)
         return getFreeFrame();

}


MMU.Cursor  = (MMU.Cursor+1) % MMU.getFrameTableSize(); //update cursor using modulus arithmetic

        }
}

//phase 2

        if(numCleanframes<MMU.wantFree && !foundDrityFrame)
                return getFreeFrame();
        else
                return MMU.getFrame(dirtyFrameID);


}
```

PageFaultHandler.Fifo()

This method implements the first in first out page replacement algorithm to return the victim frame for swapping out. In the page table entry class, each page is time stamped with its entry to the physical memory time in order to be used in this method. Here, each frame is checked for its time elapsed in the frame table, by its time stamp from the current time which is obtained using `HClock.get()`, and the page with the maximum time is returned to be swapped out.

```
   public static FrameTableEntry Fifo()

    {
        long maxTimeElapsed = 0;
```

```
        long timeElapsed=0;

        FrameTableEntry replaceFrame = null;

        FrameTableEntry frame;

        for(int i=0; i<MMU.getFrameTableSize(); i++){

        frame=MMU.getFrame(i);

        if (frame.getPage()!=null)
                 timeElapsed = Math.abs(HClock.get()-frame.getPage().getTimeStamp());


        if(timeElapsed > maxTimeElapsed && !frame.isReserved() && frame.getLockCount() == 0){

                replaceFrame = frame;

                maxTimeElapsed = timeElapsed;
                System.out.print(" "+replaceFrame+":");
                System.out.print(maxTimeElapsed);
    }

}

    return replaceFrame;
}
```

Simulation Results

With a successful simulation with no errors, the implemented memory Java package was tested with three different parameter files as they provide input data for the simulation. In order to evaluate the system's performance, the only input variable that was manipulated is the number of frames. As shown in Table 2, the results are tabulated according to the implemented Enhanced Second Chance and FIFO algorithms for all three parameters.

Table 2: Simulation Results

	OSP1		**OSP2**		**OSP3**	
No. of Frames in Physical Memory	**15**		**30**		**64**	
Page Replacement	**FIFO**	**ESC**	**FIFO**	**ESC**	**FIFO**	**ESC**
CPU Utilization	95.0796%	97.0192%	89.8492%	88.276794%	98.5772%	94.2352%
Page Faults	1057	1012	1127	1042	921	886
Memory Reads	544	537	686	615	706	672
Memory Writes	546	519	635	627	705	686
Threads Throughput	168/250000 thread/tick	150/250000 thread/tick	144/250000 thread/tick	135/250000 thread/tick	172/250000 thread/tick	142/250000 thread/tick
Page Fault Rate	0.96972	0.95833	0.85314	0.83897	0.65273	0.65243

As shown in Table 2 above, the results are obtained for each simulation of both the page replacement algorithms (FIFO) and (ESC) with increasing the number of frames input. The page replacement algorithms are compared with respect to the following performance measures:

1. ***CPU utilization*:** We want to keep the CPU as busy as possible. CPU utilization should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system) [2]. In all three params file, both algorithms exhibit high CPU Utilization ranging from about 88% to 98%. In addition, both algorithms demonstrate similar behavior since their respective CPU utilization is similar for a given number of frames. However, the second params simulation where the number of frames was 30, both FIFO and ESC show

the least CPU utilization with 89.85% and 88.277%. indicating that CPU utilization may be affected by the size of the physical memory frames.

2. ***Pagefault rate**:* The relationship between the number of frames and the number of pagefaults is not direct. As can be deduced from Table 1, the number of pagefaults alone cannot determine the effect of the alternating number of frames. Hence, the number of pagefaults depends on how many times the memory was referenced whether it is read or written; thus, the pagefault rate is determined from these two variables. The pagefault rate is a measure of how many pagefaults the system exhibits in all the memory references including the memory reads and writes. Typically, it ranges between 0 and 1 where it either indicates no pagefaults or a pagefault has occurred on every reference, respectively. The pagefault rate was calculated using the following formula:

$$Pagefault\ Rate = \frac{No.of\ pagefaults}{memory\ reads + memory\ writes}$$

In the first params file where the number of frames in the physical memory is 15, high pagefault rates of 0.96972 and 0.95833 were achieved for both the FIFO and ESC algorithms, respectively. This means that a pagefault has occurred on most memory references. When increasing the number of frames in the physical memory to 30, the algorithms exhibit almost identical and decreasing pagefault rate; 0.85314 for FIFO and 0.83897 for ESC. The pattern in repeated when further increasing the number of frames from 30 to 64, the pagefault rate decreased to identical values of 0.6527 and 0.65243 for FIFO and ESC. Therefore, these findings indicate that as the number of frames in the physical memory increases, the pagefault rate decreases with nearly identical behaviors of both algorithms.

From both these performance measures, it can be concluded that FIFO and ESC exhibit very similar behaviors. Yet, both extensively depend on the number of frames in the physical memory.

3. ***Throughput***: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit. For long processes, this rate may be one process per hour; for short transactions, it may be ten

processes per second [2]. The throughput for threads was calculated via the following formula:

$$Throughput = \frac{total\ number\ of\ threads}{total\ running\ time}$$

In our simulation, the total running time is fixed at 250,000-time units or ticks. Comparing the three params files simulation, it can be seen that FIFO always has a slightly larger throughput than ESC in all params file.

Conclusion

The memory module of OSP2 simulator was successfully implemented including the `PageTable`, `PageTableEntry`, `FreameTableEntry`, `MMU`, and `PageFaultHandler` classes. In addition, different page replacement algorithms are implemented in the `PageFaultHandler` class, FIFO and ESC.

When evaluating the system's results, it is clear that the number of frames affects the system's pagefault rate regardless of the algorithm. There is an obvious inverse relationship between them; as the number of frames increase, the pagefault rate decrease in both the FIFO and ESC. Therefore, the performance of the system depends on the size of the physical memory.

However, based on the simulation results obtained, there is no difference when it comes to each algorithm's individual behavior. Both show similar pagefault rates and CPU utilization for a given number of frames. This may be because ESC is derived from FIFO, and both algorithms show that a wrong decision of choosing the best page to replace is often taken. So, based on the results of this implementation, the theory that the past is a good indicator of the future does not necessarily stand always.

Summary

Compared to the previous project, this one was much harder. We learned in class the concepts of memory, paging, pagefaults, page replacement. However, implementing these concepts proved more difficult than expected. Reading from the OSP2 book and trying to code the system accordingly is not easy. It took some time to read and understand the new concepts.

The biggest problem we faced was with the Pagefault handler method. In addition, the major issue faced was in the errors obtained when freeing the frame and updating the page table as the simulator kept issuing different errors with different arrangement of the statements. This has led to numerous trials in finding the correct order and location of when to free the frame and update the page table.

From working on the previous work, we gained more expertise as to how to deal with the OSP2 environment. Seeing how the memory module works enabled us to truly understand the working of virtual memory in operating systems and its substantial importance. Seeing how virtual and physical memories interact in operating system enabled us to truly understand the concepts learned in lectures.

All in all, this project was a true challenge. Implementing what we studied in class is an exhilarating experience that enabled us to truly understand the small cogs that drives the whole system. As the last project of our undergraduate journey, this truly pushed us to our limit in a positive way.

References

[1] M. Kifer and S. A. Smolka, Introduction to Operating System Design and Implementation, London: Springer, 2007.

[2] A. Silberschatz, P. B. Galvin and G. Gagne, Operating System Concepts 9th edition, Wiley Publishing, 2012.