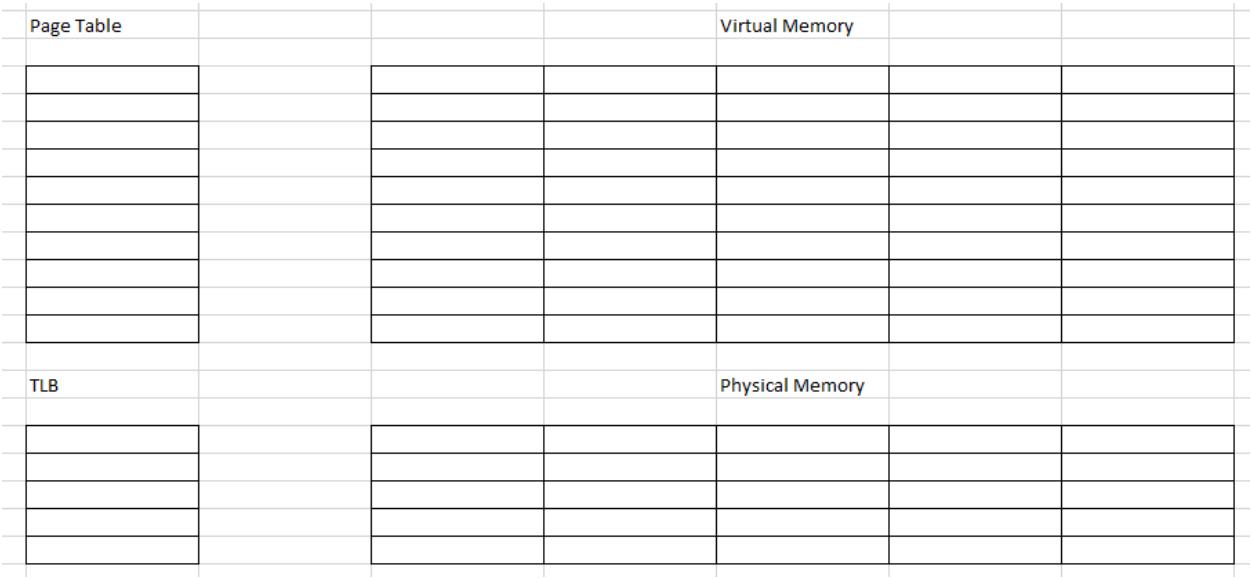
CSE312

FINAL REPORT

Ahmet Ergani

161044011

DESIGN



I created a page table to keep the data about virtual memory entries and a TLB table to keep the data about physical memory entries.

Page Table:

The main purpose of my page table is to directly check whether a certain frame is loaded into memory or not (isValid). And if it is loaded, page table shows the physical memory index of that frame(index).

```
17  class PageEntry
18  {
19  public:
20      bool isValid;
21      int index;
22  };
23
```

TLB:

The main purpose of TLB is to check a physical memory space's emptiness (isOccupied) modification status (isModified) and to store which frame is loaded into this space (frameNo). It also

```
31  class TLBEntry
32  {
33  public:
34      bool isOccupied;
35      bool isModified;
36      bool isReferenced;
37      int frameNo;
38      int usage;
39  };
```

helps us with the page replacement algorithms. For example I store isReferenced status in it's entries for NRU and a usage integer for LRU

Physical Memory:

I created a dynamic 2d array to hold the frames.

set:

First I calculated which frame has the given index (frameNo). Then I calculated which element of that frame stands for the given index (dataIndex). Then I check the pageTable if this frame is loaded into memory

```
216     if(pageTable[frameNo].isValid)
217     {
218         physicalMemory[pageTable[frameNo].index][dataIndex] = value;
219 
220         tlbTable[pageTable[frameNo].index].isModified = true;
221         tlbTable[pageTable[frameNo].index].isReferenced = true;
222         tlbTable[pageTable[frameNo].index].usage = 0;
223     }
```

If it is, I directly set the value of the necessary index of the Physical memory and set the modified bit to true.

Else, a page replacement is accomplished. After that process I set the necessary data in both TLB and Page tables and assign the value to the physical memory index returned to me by replacePage().

```
224     else
225     {
226         physIndex = replacePage(frameNo, tName);
227 
228         physicalMemory[physIndex][dataIndex] = value
229 
230         tlbTable[physIndex].isModified = true;
231         tlbTable[physIndex].isOccupied = true;
232         tlbTable[physIndex].frameNo = frameNo;
233         tlbTable[physIndex].usage = 0;
234 
235         pageTable[frameNo].isValid = true;
236         pageTable[frameNo].index = physIndex;
237         tlbTable[physIndex].isReferenced = true;
238 
239     }
```

get:

This process is nearly same as get. There are only 2 differences between them

1) Instead of assigning a value to that index, we just acquire the value of that index and return it
2) We don't set the modified bit to true after the operation

replacePage:

I decide which index will be replaced according to the current page replacement algorithm.

```
329     switch(pageReplacement)
330     {
331         case LRU:
332             physIndex = findLRU();
333             break;
334
335         case NRU:
336             physIndex = findNRU();
337             break;
```

If selected physical memory frame is occupied and has been modified, it's data is saved to disk

```
355     if(tlbTable[physIndex].isOccupied && tlbTable[physIndex].isModified)
356     {
357         lseek(fd, tlbTable[physIndex].frameNo * frameSize * sizeof(int), SEEK_SET);
358
359         for(int i = 0 ; i < frameSize; i++)
360         {
361             val = write(fd, &physicalMemory[physIndex][i], sizeof(int));
362         }
```

Then the necessary frame is read from disk into selected physical memory index

```
419     lseek(fd, frameNo * frameSize * sizeof(int), SEEK_SET);
420     for(int i = 0 ; i < frameSize; i++)
421     {
422         val = read(fd, &data, sizeof(data));
423         physicalMemory[physIndex][i] = data;
424     }
```

REPLACEMENT ALGORITHMS

LRU:

For this algorithm I stored an integer called usage in TLB. Whenever a set or get occurs every TLB entries' usage increments and the subject entry's usage is set to zero. This way I can know that the entry with the biggest usage value is the least recently used at that moment.

```
428  int Memory::findLRU()
429  {
430      int lruIndex = 0;
431      int max = 0;
432      for(int i = 0; i < physicalMemSize; i++)
433      {
434          if (!tlbTable[i].isOccupied)
435          {
436              return i;
437          }
438          else if(tlbTable[i].usage > max)
439          {
440              lruIndex = i;
441              max = tlbTable[i].usage;
442          }
443      }
444      return lruIndex;
445  }
```

FIFO:

For this algorithm I implemented a basic linked-list queue. When a page replacement occurred I dequeued from this queue and enqueued it afterwards.

```
474  int Memory::findFIFO()
475  {
476      for(int i = 0; i < physicalMemSize; i++)
477      {
478          if (!tlbTable[i].isOccupied)
479          {
480              return i;
481          }
482      }
483      return queue.dequeue();
484  }
```

NRU:

```
447  int Memory::findNRU()
448  {
449      for (int i = 0; i < physicalMemSize; i++)
450      {
451          if (!tlbTable[i].isReferenced)
452          {
453              return i;
454          }
455      }
456      return 0;
457  }
```

I created a counter and incremented it in set/get operations. I also marked referenced pages during these operations. When counter reaches a certain point these marks are being reset. In case of page replacement I selected the page without referenced mark.

SORTING ALGORITHMS

Bubble, quick and merge sort methods are implemented simply and traditionally.

```
656      for (int i = 0; i < N; ++i)
657      {
658          indexes[i] = i + N * 3;
659      }
660
661      for (int i = 0; i < N; i++)
662      {
663          for (int j = i + 1; j < N; j++)
664          {
665              if (get(indexes[i], tName) > get(indexes[j], tName))
666              {
667                  int temp = indexes[i];
668                  indexes[i] = indexes[j];
669                  indexes[j] = temp;
670              }
671          }
672      }
673
674      for (int i = 0; i < N; ++i)
675      {
676          values[i] = get(indexes[i], tName);
677      }
678      for (int i = 0; i < N; ++i)
679      {
680          set(i + N * 3, values[i], tName);
681      }
```

Index sort was different. This sorting algorithm is designed to sort collections of large objects and was useless for our case since we were working with integers. So I acted like my objects were large. Created an indexes array and sorted it with a bubble sort-like algorithm. Then I created a values array and filled it using the indexes array. Then I set these values to our memory.

THREADS

Each thread sets it's statistics data to zero and calls it's sort method (or directly sorts if the algorithm is not complex).

```
639  void * Memory::mergeSortWorker(void *)
640  {
641      readCount[3] = 0;
642      writeCount[3] = 0;
643      pageReplacementCount[3] = 0;
644      pageMissCount[3] = 0;
645      diskReadCount[3] = 0;
646      diskWriteCount[3] = 0;
647      char tName[6] = "merge";
648      int N = (virtualMemSize * frameSize) / 4;
649      mergeSort(N * 2 , N * 3 - 1, tName);
650
651      return NULL;
652  }
```

EXAMPLE RUNS

```
cse312@ubuntu:~/Desktop/final$ ./sortArrays 4 3 5 NRU local 10000 diskFileName.txt
Virtual Memory Size: 32
Physical Memory Size: 8
Frame Size: 16
Replacement Algorithm: NRU
Interval: 10000
File: diskFileName.txt

Array quarters are sorted

FILL : read: 0          write: 512      page miss: 32   page replacement : 24   disk read : 32  disk write : 28
BUBBLE : read: 24126    write: 7870     page miss: 356  page replacement : 348  disk read : 356 disk write : 35
QUICK : read: 1834      write: 918      page miss: 23   page replacement : 15   disk read : 23  disk write : 22
MERGE : read: 896       write: 896      page miss: 27   page replacement : 19   disk read : 27  disk write : 26
INDEX : read: 16384     write: 128      page miss: 104  page replacement : 96   disk read : 104 disk write : 97
CHECK : read: 1016      write: 0        page miss: 24   page replacement : 16   disk read : 24  disk write : 24
```

```
cse312@ubuntu:~/Desktop/final$ ./sortArrays 5 4 6 NRU local 10000 diskFileName.txt
Virtual Memory Size: 64
Physical Memory Size: 16
Frame Size: 32
Replacement Algorithm: NRU
Interval: 10000
File: diskFileName.txt

Array quarters are sorted

FILL : read: 0          write: 2048     page miss: 64   page replacement : 48   disk read : 64  disk write : 56
BUBBLE : read: 395608   write: 133976   page miss: 2734 page replacement : 2718 disk read : 2734        disk write : 2730
QUICK : read: 10829     write: 4984     page miss: 65   page replacement : 49   disk read : 65  disk write : 64
MERGE : read: 4608      write: 4608     page miss: 62   page replacement : 46   disk read : 62  disk write : 61
INDEX : read: 262144    write: 512      page miss: 745  page replacement : 729  disk read : 745 disk write : 740
CHECK : read: 4088      write: 0        page miss: 48   page replacement : 32   disk read : 48  disk write : 48
```

```
cse312@ubuntu:~/Desktop/final$ ./sortArrays 6 5 7 NRU local 10000 diskFileName.txt
Virtual Memory Size: 128
Physical Memory Size: 32
Frame Size: 64
Replacement Algorithm: NRU
Interval: 10000
File: diskFileName.txt

Array quarters are sorted

FILL : read: 0          write: 8192     page miss: 128  page replacement : 96   disk read : 128 disk write : 112
BUBBLE : read: 6222296  write: 2030040  page miss: 27010        page replacement : 26978        disk read : 27010       disk write : 27002
QUICK : read: 59994     write: 17544    page miss: 202  page replacement : 170  disk read : 202 disk write : 201
MERGE : read: 22528     write: 22528    page miss: 198  page replacement : 166  disk read : 198 disk write : 198
INDEX : read: 4194304   write: 2048     page miss: 5023 page replacement : 4991 disk read : 5023        disk write : 4951
CHECK : read: 16376     write: 0        page miss: 96   page replacement : 64   disk read : 96  disk write : 96
```