CS307 Project8: Virtual Memory Manager

Junjie Wang 517021910093

May 23, 2019

1 Programming Thoughts

First we have to compute the number of pages, size of each page, the number of frames and the size of each frame and then initialize the page table, memory and TLB table.

1.1 Page Table

Then we implement the logic for page table. Given a page index, if there is a valid frame index in page table, we can simply use the frame index we get and the offset previously computed to get the physical address of the data and then visit the physical memory. If we meet a page fault(no valid frame index), we should first locate to correct position of the BACKING_STORE.bin, and then read in page size bytes of data. Then find a free frame in the physical memory, put the data into that frame and update the page table.

1.2 TLB Table

As for the logic for TLB table, the logic is quite similar to the page table(since TLB is only to make things faster). When a virtual address comes, we first look up its page index in the TLB table. If TLB hits, we can just use the frame index to compute the physical address, visiting the physical memory. If TLB misses, we turn to the page table. Besides, logic for update TLB table when reading data from BACKING_STORE should be added.

1.3 Page Replacement

As for the logic of page replacement algorithm, I implement the LRU algorithm. Since when the number of pages is equal to the number of frames, we can always find a free frame and no page replacement will happen. So the page replacement only works when the size of physical memory is much smaller than the logical memory(e.g. 128 frames and 256 pages). The idea is to initilize the LRU numbers of all frames to **INT_MAX** and each time we visit a frame, we update all the LRU numbers using this strategy: setting the exact frame we are visiting's LRU number to **INT_MAX** and decrease all lother LRU numbers by 1. Each time we choose a victim, we choose the frame with the smallest LRU number.

1.4 TLB LRU

Similar LRU logic applies to the TLB table. Since an entry in TLB table consists of both the page index and frame index, we better define a new data struct to add in the LRU member.

2 Execution Results And Snapshots

```
dreamboy@Elon_Mask:~/OSProjects/project8$ ./manager addresses.txt
Page numbers: 256, Page size: 256
Frame numbers: 256, Frame size: 256
Process finished! Output saved to => ./output.txt
Page fault: 24.376%
TLB hit: 5.594%
```

Figure 1: 256 frames

In 2, the comparsion between the output.txt(the output of the algroithm on addresses.txt) and the correct.txt(the official output for reference) is given.

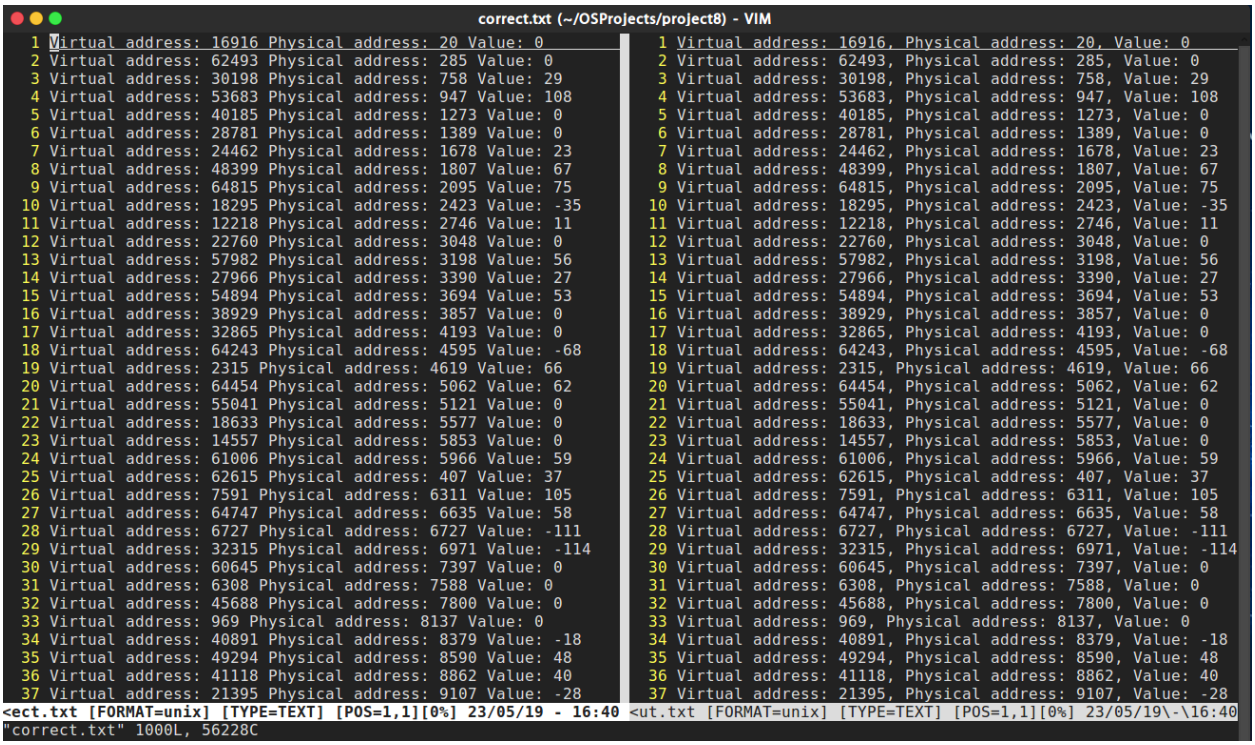

Figure 2: output.txt vs correct.txt

In 3, the execution results of 128 frames are given. Since the only difference only lies in choosing a victim frame, the output should be the same as in 1.

```
dreamboy@Elon_Mask:~/OSProjects/project8$ ./manager addresses.txt
Page numbers: 256, Page size: 256
Frame numbers: 128, Frame size: 256
Process finished! Output saved to => ./output.txt
Page fault: 24.376%
TLB hit: 5.594%
dreamboy@Elon_Mask:~/OSProjects/project8$
```

Figure 3: 128 frames

3 Code Explanation

The data struct for each TLB entry(we add a member to be the lru number) is as follows:

```
/* TLB entry struct */
typedef struct _entry{
   int page_idx;
   int frame_idx;
   int lru;
} entry;
```

The logic of updating LRU numbers for the memory and TLB table

```
/* update the lru for the tlb */
void updateLRUTLB(int page_idx){
   for(int i=0;i<TLB_ENTRIES;i++){
   if(tlb_table[i].page_idx == page_idx) tlb_table[i].lru = MAX_LRU;
   else    tlb_table[i].lru--;

   }
}

/* update the LRU for the memory */
void updateLRUMemory(int frame_idx){
   for(int i=0;i<num_frames;i++){
   if(i == frame_idx) memory_lru[i] = MAX_LRU;
   else    memory_lru[i]--;
   }
}
```

Core logic in the while loop, handling TLB loop up and page table query.

```
if((frame_idx = lookUpTLB(page_idx)) != -1){
   data = memory[frame_idx][offset];
   updateLRUTLB(page_idx);
   tlb_hits++;
}else if(page_table[page_idx] == -1){
```

```
   /* loop up the page table */

   /* -1 means page fault(we don't have a frame number
   * at page_idx in page table */

   /* 1. read data from BACKING_STORE
   * find the position */
   buffer = malloc(sizeof(char) * page_size);
   fseek(fp_back, (page_idx) * page_size, SEEK_SET);
   fread(buffer, sizeof(char), page_size, fp_back);
   /* 2. find a free frame and store the data */
   frame_idx = findFreeFrame();
   if(frame_idx == -1){
      /* can not find a free frame, we must use page replacement */
      /* choose a victim and replace the frames */
      min_lru = MAX_LRU;
      min_lru_idx = 0;
      for(i=0;i<num_frames;i++){
      if(memory_lru[i] < min_lru){
      min_lru = memory_lru[i];
      min_lru_idx = i;
      }
   }
      frame_idx = min_lru_idx;
      updateLRUMemory(min_lru_idx);
   }
   memcpy(memory[frame_idx], buffer, page_size);
   free_frames[frame_idx] = 0;
   /* 3. update the page table and TLB */
   page_table[page_idx] = frame_idx;
   /* choose a victim and update TLB */
   min_lru = MAX_LRU;
   min_lru_idx = 0;
   for(i=0;i<TLB_ENTRIES;i++){
   if(tlb_table[i].lru < min_lru){
   min_lru = tlb_table[i].lru;
   min_lru_idx = i;
   }
   }
   tlb_table[min_lru_idx].page_idx = page_idx;
   tlb_table[min_lru_idx].frame_idx = frame_idx;
   updateLRUTLB(page_idx);

   num_faults++;
}else{
   /* hit in the page table */
   frame_idx = page_table[page_idx];
```

```
    data = memory[frame_idx][offset];
  }
    num_addresses++;
  }
```