KOÇ UNIVERSITY

COMP 304: Operating Systems

Project - 3

Spring 2023

Ali Gebeşçe, 64294

Yakup Enes Güven, 64045

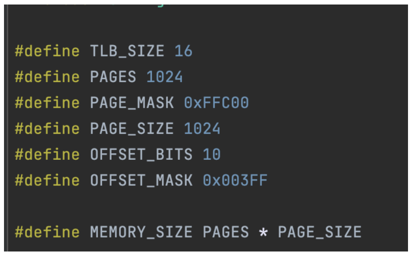
Table of Contents

Important Note

Please note that all project components were completed by both team members during in-person and online meetings. However, only one team member's computer was used to commit the corresponding work after these meetings. As a result, Git commits do not accurately reflect individual contributions. Nevertheless, we made an effort to ensure that both team members' Git accounts contributed equally in terms of the weight of points for the project.

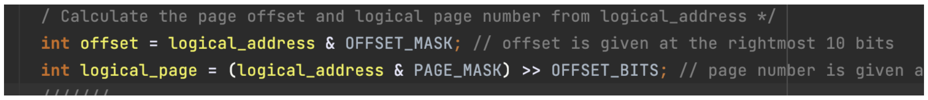
A. Part I [1, 2]

- The masks are comprised of 32-bit binary numbers that are utilized in a bitwise-AND (&) operation with logical addresses to ascertain the offset and logical page. These masks are 32-bit quantities as they correspond to the size of an int, the datatype used for defining logical addresses. The 10th to 19th bits of the logical address retain the logical page value, while the offset value is stored on the bits from the 0th to the 9th. These specific bits are set to 1 in the masks, and the others are filled with zeroes. It's worth noting that the logical_page value needs to be right-shifted by OFFSET_BITS to align the logical page value with the least significant bits.We defined masks as follow:



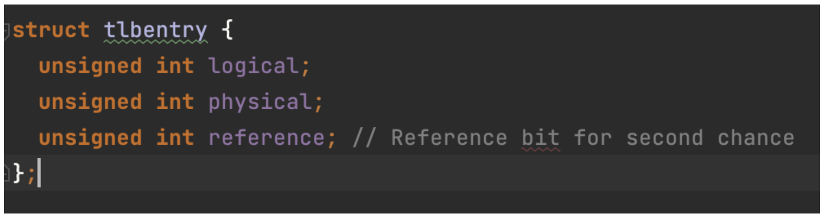
```
#define TLB_SIZE 16
#define PAGES 1024
#define PAGE_MASK 0xFFC00
#define PAGE_SIZE 1024
#define OFFSET_BITS 10
#define OFFSET_MASK 0x003FF

#define MEMORY_SIZE PAGES * PAGE_SIZE
```



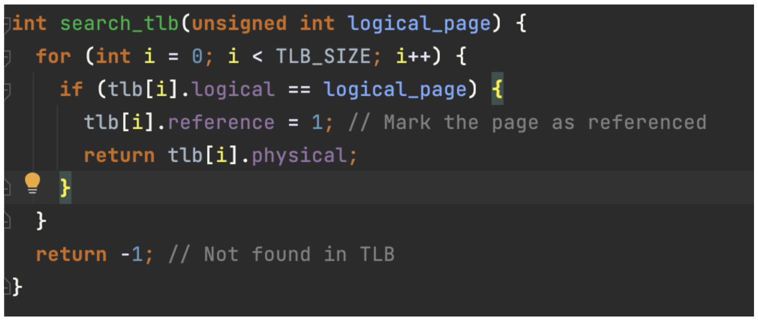
```
/ Calculate the page offset and logical page number from logical_address */
int offset = logical_address & OFFSET_MASK; // offset is given at the rightmost 10 bits
int logical_page = (logical_address & PAGE_MASK) >> OFFSET_BITS; // page number is given a
```

- Then, to implement second chance algorithm we changed the structure of tlb to implement second chance algorithm as you can see below:



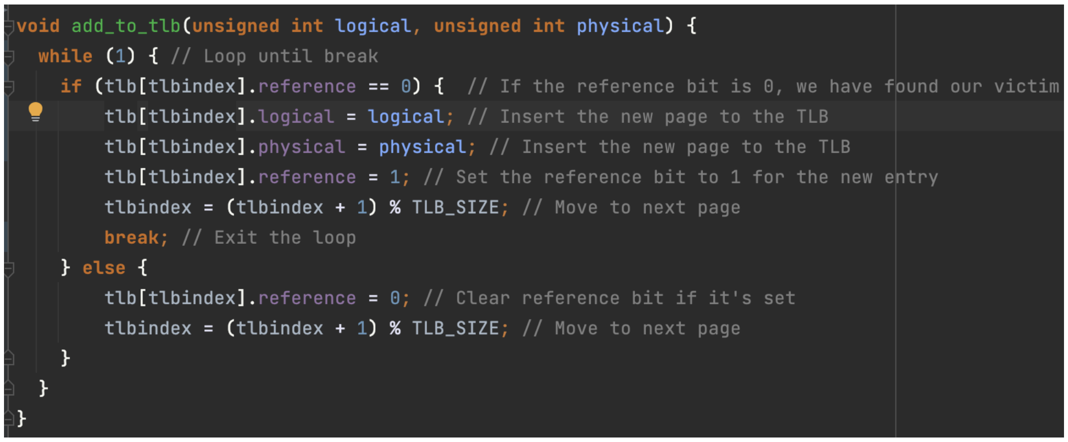
```
struct tlbentry {
  unsigned int logical;
  unsigned int physical;
  unsigned int reference; // Reference bit for second chance
};
```

- Then we implemented the search tlb function. This function searches the TLB for a given logical page. If the page is found in the TLB (a "hit"), it returns the corresponding physical page and marks the page as referenced by setting the reference member to 1. If the page is not found (a "miss"), it returns -1.



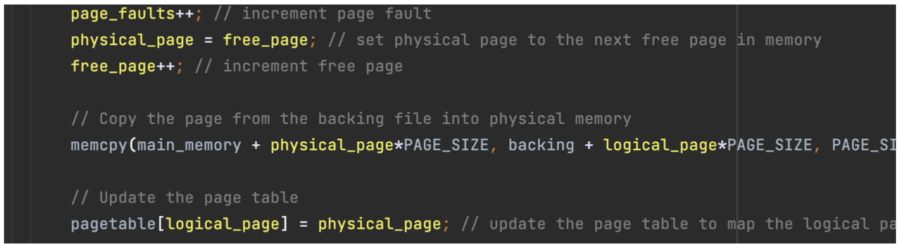
```c
int search_tlb(unsigned int logical_page) {
  for (int i = 0; i < TLB_SIZE; i++) {
    if (tlb[i].logical == logical_page) {
      tlb[i].reference = 1; // Mark the page as referenced
      return tlb[i].physical;
    }
  }
  return -1; // Not found in TLB
}
```

- Then we implemented add_to_tlb function.The function continuously loops through the TLB until it finds an entry whose reference bit is set to 0, which signifies that this entry is available for replacement. Once such an entry is found, the function updates this TLB entry with a new logical address (virtual memory address) and a new physical address. The reference bit for this entry is then set to 1, indicating that it has been recently accessed. The TLB index is then updated to point to the next entry, allowing for a round-robin replacement policy. If a TLB entry with a reference bit set to 1 is encountered, the function simply clears the reference bit and moves on to the next entry, allowing the algorithm to potentially return to this entry in the future if an available slot isn't found.



```c
void add_to_tlb(unsigned int logical, unsigned int physical) {
  while (1) { // Loop until break
    if (tlb[tlbindex].reference == 0) {  // If the reference bit is 0, we have found our victim
        tlb[tlbindex].logical = logical; // Insert the new page to the TLB
        tlb[tlbindex].physical = physical; // Insert the new page to the TLB
        tlb[tlbindex].reference = 1; // Set the reference bit to 1 for the new entry
        tlbindex = (tlbindex + 1) % TLB_SIZE; // Move to next page
        break; // Exit the loop
    } else {
        tlb[tlbindex].reference = 0; // Clear reference bit if it's set
        tlbindex = (tlbindex + 1) % TLB_SIZE; // Move to next page
    }
  }
}
```

- Finally, we handle the page faults.



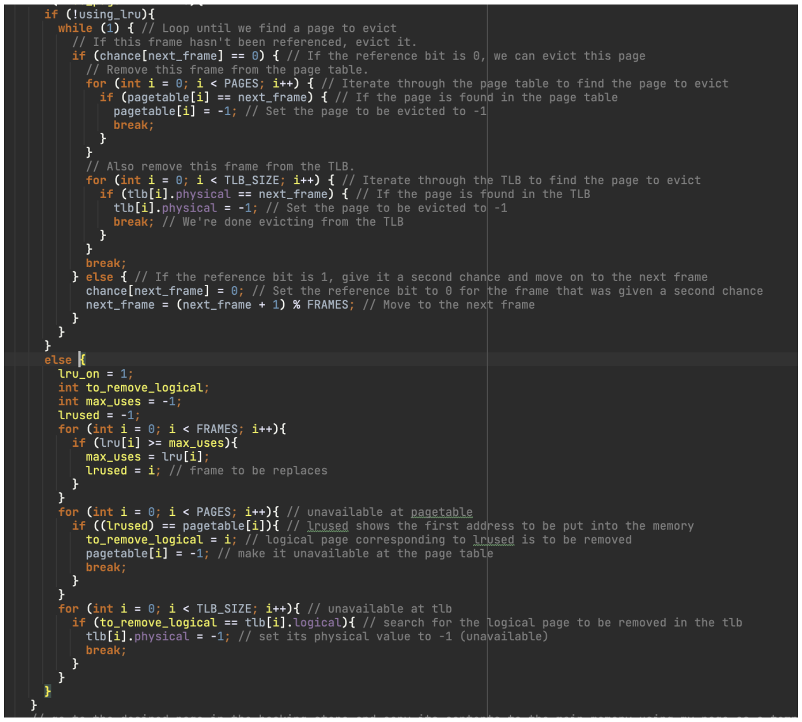
```c
    page_faults++; // increment page fault
    physical_page = free_page; // set physical page to the next free page in memory
    free_page++; // increment free page

    // Copy the page from the backing file into physical memory
    memcpy(main_memory + physical_page*PAGE_SIZE, backing + logical_page*PAGE_SIZE, PAGE_SI

    // Update the page table
    pagetable[logical_page] = physical_page; // update the page table to map the logical pa
```

B. Part II [2, 3, 4]

We have introduced a new constant called FRAMES, which has been set to a value of 256. This value corresponds to the updated main memory size of 256 frames. As a result, the memory size has been adjusted and a new constant called BACKING_SIZE has been established. The mmap function argument has also been updated to align with the new backing size. In addition, an integer array called 'lru' has been created and initialized with zeros. Whenever a memory address is accessed, the corresponding LRU (Least Recently Used) value of that frame is set to 0, while the LRU values for all other frames in the memory are incremented. Once the number of free pages reaches or exceeds the value of FRAMES, we initiate the page replacement process if a page fault occurs.During the page replacement process, the identified page to be replaced, determined by either the Second Chance or LRU algorithm, is removed from the page table and the TLB. Subsequently, the required page is loaded into memory, following the same procedure as described in Part 1.n the LRU substitution algorithm, we cycle through all the frames and select the one that has the highest value in the LRU array. This indicates that the frame has been accessed least recently since the LRU array frames are reset to zero upon access. Once the least recently accessed frame is identified, we load the value into the address of that frame.



```
    if (!using_lru){
      while (1) { // Loop until we find a page to evict
        // If this frame hasn't been referenced, evict it.
        if (chance[next_frame] == 0) { // If the reference bit is 0, we can evict this page
          // Remove this frame from the page table.
          for (int i = 0; i < PAGES; i++) { // Iterate through the page table to find the page to evict
            if (pagetable[i] == next_frame) { // If the page is found in the page table
              pagetable[i] = -1; // Set the page to be evicted to -1
              break;
            }
          }
          // Also remove this frame from the TLB.
          for (int i = 0; i < TLB_SIZE; i++) { // Iterate through the TLB to find the page to evict
            if (tlb[i].physical == next_frame) { // If the page is found in the TLB
              tlb[i].physical = -1; // Set the page to be evicted to -1
              break; // We're done evicting from the TLB
            }
          }
          break;
        } else { // If the reference bit is 1, give it a second chance and move on to the next frame
          chance[next_frame] = 0; // Set the reference bit to 0 for the frame that was given a second chance
          next_frame = (next_frame + 1) % FRAMES; // Move to the next frame
        }
      }
    }
    else {
      lru_on = 1;
      int to_remove_logical;
      int max_uses = -1;
      lrused = -1;
      for (int i = 0; i < FRAMES; i++){
        if (lru[i] >= max_uses){
          max_uses = lru[i];
          lrused = i; // frame to be replaces
        }
      }
      for (int i = 0; i < PAGES; i++){ // unavailable at pagetable
        if ((lrused) == pagetable[i]){ // lrused shows the first address to be put into the memory
          to_remove_logical = i; // logical page corresponding to lrused is to be removed
          pagetable[i] = -1; // make it unavailable at the page table
          break;
        }
      }
      for (int i = 0; i < TLB_SIZE; i++){ // unavailable at tlb
        if (to_remove_logical == tlb[i].logical){ // search for the logical page to be removed in the tlb
          tlb[i].physical = -1; // set its physical value to -1 (unavailable)
          break;
        }
      }
    }
  }
```

C. RUN

We created a new makefile. To test:

- PART1:
  - make p1
- PART2 (Second Chance, -p 0)

    - make p2sc
- PART3 (LRU, -p 1)
    - make p2lru

D. References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts, 10e*
2. *Lecture Notes*
3. https://www.tutorialspoint.com/operating_system/os_virtual_memory.htm
4. https://www.geeksforgeeks.org/operating-systems/