Operating Systems - Monsoon 2024

September 24, 2024

Assignment 2 (Total points: 100)
Due: Oct 2, 2024; 11:59 pm

Instructions.
1. Follow all the instructions in the questions closely.
2. Code should be written in C language.
3. Files should be named q[question number].c unless stated otherwise. For example: q1.c
4. Submit a .zip named [RollNo].zip file containing source code files.
5. During the Assignment demos, questions about the code and how it works will be asked. Ensure that you know the code you are submitting.
6. This assignment can be worked on in a group of at most two students.
7. After the deadline, for every 6 hrs, 5% of points will be cut.

Q1. Perform parallel merge sort on a 16-element array. The 16-element array would be divided into two 8-element arrays, and each one would be passed to a child process. Similarly, the children would pass 4-element arrays to their children and so on. Each process will wait for its children and then merge the resulting arrays using a merge operation. Use pipes to pass data among processes. **[10 pts]**
Input: Define a 16-element array in the code. The array must be unsorted.
Output: Print the initial unsorted array and the final sorted array.

Q2. Write a program that simulates an operating system's memory management using segmentation. Your task is to simulate the process of translating logical addresses to physical addresses using the concept of segmentation. Additionally, you will simulate conditions where a segmentation fault occurs. Add your assumptions, if any, to a ReadMe file. **[20 pts]**
Design:

- The system has three segments: Code, Heap, and Stack.
- Each segment has a base and a limit (bounds), which define where the segment starts in memory and its length.
- Assume the system has a **64KB** physical memory, and memory is reserved as follows:
    - Code Segment: Starts at 32KB, size 2KB.
    - Heap Segment: Starts at 34KB, size 3KB.
    - Stack Segment: Starts at 28KB and grows downwards, size 2KB.

- A segmentation fault should be raised if the offset exceeds the segment's limit.
- Use the pseudocode from Chapter 16 for the address translation.

Input: The user can enter a 16-bit logical address in hex format. For example: `a56f`
Output: 16-bit physical address in hex format or segmentation fault.

Q3. Write a program that simulates paging-based memory management and address translation using a Page Table and Translation Lookaside Buffer (TLB). The program should demonstrate how logical addresses are translated into physical addresses using a page table and how the TLB serves as a cache to speed up this process. Add your assumptions, if any, to a ReadMe file. **[20 pts]**

Design:

- Implement a page table that maps virtual page numbers (VPNs) to physical frame numbers (PFNs).
- Assume a fixed page size of 4KB and 64KB of total memory.
- Simulate a TLB that caches recent translations. The TLB size should be small enough that it cannot hold all the pages in the system.
- The TLB should be consulted first to check if the translation exists (TLB hit or miss).
- On a TLB miss, simulate accessing the page table to fetch the PFN and update the TLB.
- Use the pseudocode from Chapter 19 for the address translation.

Input: The user can enter a 16-bit logical address in hex format. For example: `a56f`
Output: The physical address in hex format and a string telling if this was a `TLB hit` or `TLB miss`.

Q4. Write a program to simulate the process of address translation using a two level page table with a page directory for virtual memory management. **[20 pts]**

Design :

- Implement a two-level page table structure. The first level is a Page Directory (**PD**) which points to the Page Tables.
- Assume that a virtual address is divided into three parts
    - Page Directory Index (10 bits): Top level index that maps to an entry in the Page Directory.
    - Page Table Index (10 bits): Second level index that maps to an entry in the page table.
    - Offset (12 bits): Offset within the actual page

- Use the following configuration
    - Virtual Address Length: 32bit (Eg : 0xCCC0FFEE)
    - Page Size: 4KB
    - Page Table Entries: Each page table has 1024 entries (each entry is of 4B).
- Address Translation:
    - Implement an API consisting of two functions, **load** and **store** with the following signature as shown below.

```
uint8_t load(uint32_t va) {
      assert(0 && "TODO : SIMULATE READING FROM va");
}

void store(uint32_t va) {
      assert(0 && "TODO : SIMULATE WRITING INTO va");
}
```

    - Implement these functions to simulate memory access. If the address can be successfully translated to a physical address, then load/store the value into the physical address. If the address cannot be translated i.e., PD or Page Table Entry is invalid (null), simulate a page fault and update the PD and Page Tables accordingly.

Deliverables:

- Hook into your load and store API's to track and display the total number of **page hits** and **page misses**.
- The page table and the page directory size.
- For page misses, show the updates made to the **PD** and **Page Table**.
- Provide a summary at the end showing the **hit/miss** ratio.

Q5. Write a C program to simulate a memory management system that demonstrates page faults and handles them using the clock algorithm for page replacement. Along with the use/reference bit, the dirty bit associated with each page should also be considered. **[30 pts]**

Design

- Simulate a memory of N pages using a fixed-size page table.

- **Input :**
  - **num_frames:** Number of available page slots (frames).
  - **page_requests:** A string representing page requests (Eg. “0,1,1,2,4,2”)
- **Output:**
  - Total page faults
  - Total page hits
  - Hit rate
- Each page can be represented using a simple structure called **struct page** defined as follows

```
struct page {
    uint8_t page_number;
    bool reference_bit;
    bool dirty_bit;
};
```

- Memory frames can be represented using a simple array that holds the page number and any necessary bits.

Algorithm :

- Initialize all memory frames to be empty (-1 or some other value).
- Service each of the page requests from the list passed as input to the program. Eg : [0 1 1 2 4 2]
  - For each page request, check if the page is in memory (check in the frames array).
  - If yes, it's a hit.
  - If no then use the clock algorithm to replace a page and update the reference bit of the corresponding page (struct page instance) if necessary.
  - After servicing all the requests, print out the total faults, total page hits and hit rate.