UM-Dearborn - CIS-450-002 Operating Systems

Student: Demetrius Johnson

Professor: Dr. Jinhua Guo

Homework 2

April 11, 2022

Due Date: April 11, 2022

CIS 450 - WINTER 22 – HW2 – Demetrius Johnson

CIS450/ECE478 Homework #2, Due Monday, Apr. 11, 2022

Question 1:

1. (10 points) Consider a logical-address space of 32 pages of 1,024 bytes each, mapped onto a physical memory of 128 frames
   a. How many bits are in the logical address?
      - Since we have a total of 32 pages = 2^5,
        - we can use **5 bits** for the address of the page number,
      - and since each page contains 1,204 bytes = 2^10,
        - then each byte in a page (the offset address in a given page) can be represented by **10 bits**.
      - Thus,
        - we can use a page# + offset = 5 + 10 = **15-bit logical address**.

   b. How many bits are in the physical address?
      - Since we have a total of 128 frames, 1 frame = 1 page; pages = 128 = 2^7
        - we can use **7 bits** for the address of the page number,
      - and since each page contains 1,204 bytes = 2^10,
        - then each byte in a page (the offset address in a given page) can be represented by **10 bits**.
      - Thus,
        - we can use a page# + offset = 7 + 10 = **17-bit logical address**.

Question 2:

2. (10 points) For each of the following virtual addresses, compute the virtual page number and offset for a 4 KB (4096) page and for an 8KB (8192) page: 20000, 32768, 60000, and 90000.

- Address (in decimal) for **20000**:
  - 4KB page# + offset =
    - Page# → 20000/4096 = 4, remainder 3616/4096 = **on page 4**
    - Offset → remainder = 3616/4096 = **offset value is 3616**
  - 8KB page# + offset =
    - Page# → 20000/8192 = 2, remainder 3616/8192 = **on page 2**
    - Offset → remainder = 3616/8192 = **offset value is 3616**
- Address (in decimal) for **32768**:
  - 4KB page# + offset =
    - Page# → 32768/4096 = 8, remainder 0/4096 = **on page 8**
    - Offset → remainder = 0/4096 = **offset value is 0**
  - 8KB page# + offset =
    - Page# → 32768/8192 = 4, remainder 0/8192 = **on page 4**
    - Offset → remainder = 0/8192 = **offset value is 0**
- Address (in decimal) for **60000**:
  - 4KB page# + offset =
    - Page# → 60000/4096 = 14, remainder 2656/4096 = **on page 14**
    - Offset → remainder = 2656/4096 = **offset value is 2656**

- 8KB page# + offset =
  - Page# → 60000/8192 = 7, remainder 2656/8192 = **on page 7**
  - Offset → remainder = 2656/8192 = **offset value is 2656**

- Address (in decimal) for **90000**:
  - 4KB page# + offset =
    - Page# → 90000/4096 = 21, remainder 3984/4096 = **on page 21**
    - Offset → remainder = 3984/4096 = **offset value is 3984**
  - 8KB page# + offset =
    - Page# → 90000/8192 = 10, remainder 8080/8192 = **on page #**
    - Offset → remainder = 8080/8192 = **offset value is 8080**

Question 3:

3. (10 points) A computer has 32-bit virtual addresses and 4 KB pages. Suppose, a process needs 12 MB, the bottom 4 MB for program text, the next 4 MB for data, and the top 4 MB for the stack. How many page table entries are needed in the page table if traditional (one-level) paging is used? How many page table entries are needed for two-level paging, with 10 bits in each part?

I made this table to help me visualize the question:

MB#	32-BIT Address (4096MB) VIRTUAL MEMORY TABLE for a 12MB process
4095	.STACK Segment (4MB)
4094	
4093	
4092	
… [8 to 4093] …	~Rest of memory (4084MB)~ (empty/unused/invalid)
7	.DATA Segment (4MB)
6	
5	
4	
3	Program Text = .CODE Segment (4MB)
2	
1	
0	

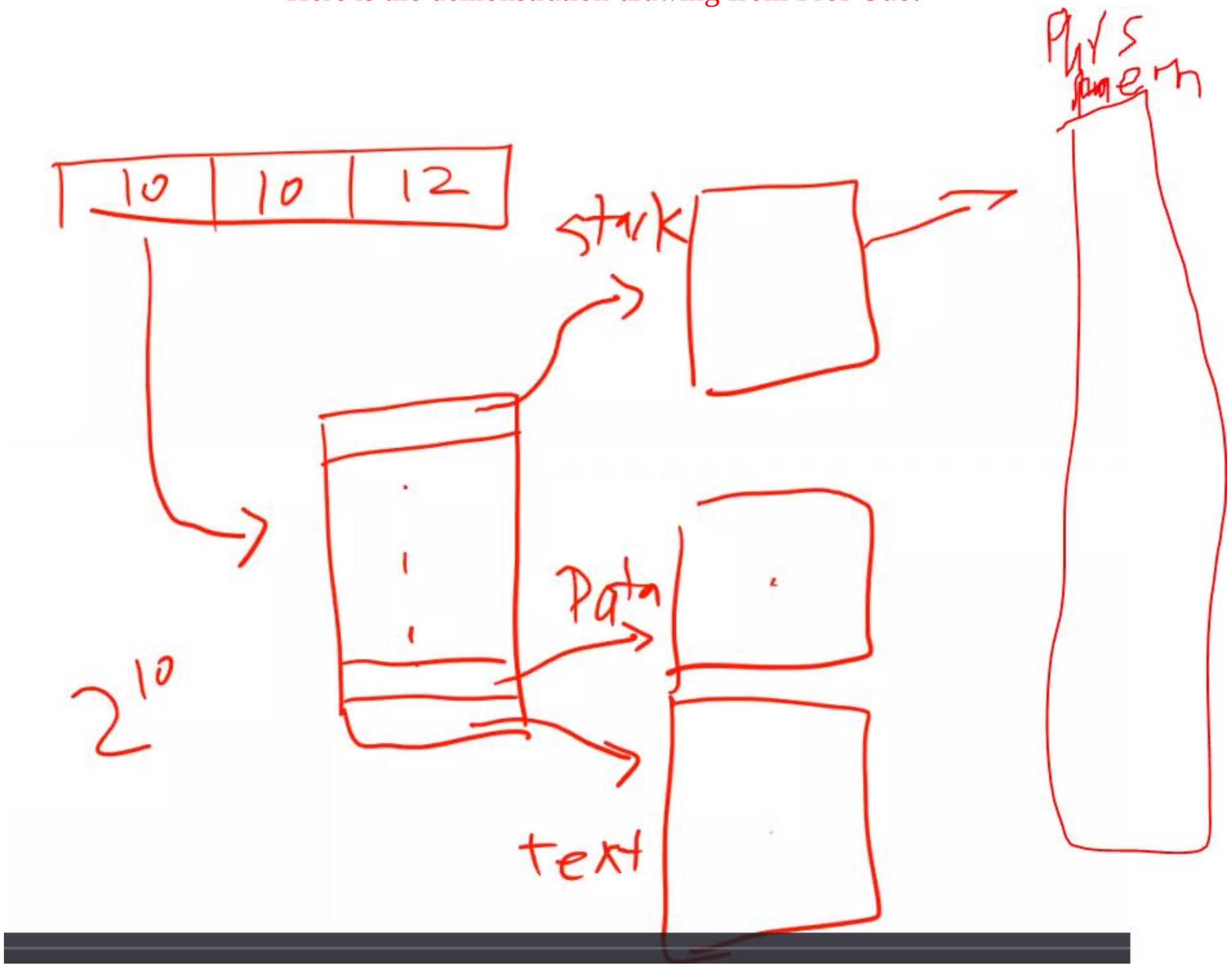
As shown above, majority of the memory is unused, but we still need a page table that can cover all of the addresses of the 32-bit (4096MB = 4GB) memory.

- **Traditional one-level paging:**
  - Each page (virtual) will map to a frame (physical) address location in memory. Since each page = 4KB, thus 4GB memory will be split into:

        - 4GB/4KB = (4*1024*1024*1024)/(4*1024*1024) = **1,048,576 page table entries needed!**
    - To reference each page, we would need:
        - 1,048,576 = 2^20 = **20 bits for the page number address**.
    - And to reference the byte offset in each page, we need:
        - Each page = 4KB = 4096bytes = 2^12 = **12 bits for byte offset address in each page.**
    - I notice that 20bits for page address + 12bits for offset = 32bits → the size of the virtual memory address (and physical memory address size).
    - For a 12MB process, we would only be using:
        - 12MB/4KB = 3,072 pages out of the 1,048,576 page table entries needed to build the page table with one-level paging; the large amount of entries needed is very wasteful (since page table is stored in main memory; ideally, for such a small process it would be better to somehow have a smaller page table that is the exact size needed to do the address translation) because the stack is all the way at the top address spaced as far as possible from the data and code segments.
            - Of course, if one or many processes use all of the memory (or most of it, and often), then one-level paging wouldn't be so wasteful (if at all).
- **Two-level paging (much less wasteful!)**
    - Use multi-paging: split the single page into two pages → thus split 2^20 pages, which uses a 20-bit reference address, into two 10-bit addressable pages:
        - Each of the 2^10 (page level 1) points to → 2^10 (page level 2), where each level uses 10 bits = 20 bits total; thus we still have 2^20 pages, but we only need the first-level page = 2^10 = **1,024 (level 1) page table entries.**
            - this saves a lot more space, even with max separation of the code and data segments from the stack, **we only need a table that contains 1,024 entries** (the first page reference). Each level-one page will reference a level-two page with 2^10 more entries, which then finally we use the final 12 bits to get the offset of the acquired page# (if you combine level-one and level-two page addresses == final page# address == 10+10 = 20 bits)) mapped to the actual physical memory (offset at a given frame).
            - In our case, the bottom 2 pages in the first-level page table will be used for the CODE and DATA segments, which each will map to 2^10 =1024 pages, where **each** page is 4KB; 1024pages*4KB = 4MB → exactly the size needed for those segments. Then, the same situation will occur for the STACK since it is also 4MB, however, it will be the top-most entry in the level-one page table. **Only** the remaining 1,024 - 3 = 1,021 level-one page table entries will be wasted while that space is not used, but all remaining level-two page table entries (1024[level-one]*1021[level-two] = 1,045,504 entries) will not

be necessary and thus not waste space in main memory storing empty page table entries.

- Here is the demonstration drawing from Prof Guo:



Question 4:

4. (10 points) Consider the following segment table

Segment	Base	Length
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

What are the physical addresses for the following logical addresses?

a. 0,430

Segment 0, offset 430 (within 600 length limit) → base + offset = 219 + 430 = **0,649**

b. 1,010

Segment 1, offset 10 (within 14 length limit) → base + offset = 2300 + 10 = **2,310**

c. 2,500

Segment 2, offset 500 (**NOT** within 100 length limit) → **illegal; ERROR: segmentation fault.**

d. 3,400

Segment 3, offset 400 (within 580 length limit) → base + offset = 1327 + 400 = **1,727**

e. 4,112

Segment 4, offset 112 (**NOT** within 96 length limit) → **illegal; ERROR: segmentation fault.**

Question 5:

5. (15 points) Consider a demand-paging system with the following time-measured utilizations:

CPU utilization	10%
Paging disk	97.7%
Other I/O devices	5%

For each of the following, say whether it will (or is likely to) improve CPU utilization. Explain your answers.

a. Install a faster CPU
No; the CPU is being underutilized; if we increase its speed, it will only be more underutilized, and in fact, it could create even less utilization since it will be able to context switch faster and thus increase demand paging and potentially thrashing.

b. Install a bigger paging disk.
Yes; this will allow for processes to keep running without having to wait for their page tables to be loaded since there is a higher probability it is still loaded upon context switches because there is a larger paging disk.

c. Increase the degree of multiprogramming
No, otherwise there will be more context switching and more processes in the system to keep page tables for, thus there will be more demand paging and more thrashing (pages replaced when still needed).

d. Decrease the degree of multiprogramming
Yes; for this situation, we need to let the CPU spend more time running processes instead of context switching and dealing with so many processes that demand paging is high.

e. Install more main memory
Yes; if there is more memory, then processes will have access to more memory so that more page tables can be stored in main memory and not have to be paged from the slower disk.

f. Install a faster hard disk
Yes; since demand paging involves reading memory off the disk and into faster main memory, if the disk is faster, then time to load it into main memory will be faster, which will allow more processes to run longer since their page tables have a higher chance of being loaded (since they can be loaded faster before a context switch).

g. Add prepaging to the page-fetch algorithms
Depends; if multiple pages are preloaded and correspond to the given running processes, then there is a good chance that their can be higher probabilities that page tables for given processes and during context switches for other processes will be able to run, which means CPU utilization will increase.

h. Increase the page size
Also depends; larger page sizes means fewer pages; so with so many processes, unless they are sequentially loaded (context switched) in such a way so that the few pages that are already loaded in memory correspond to these processes, then it could actually cause less CPU utilization and even more or the same amount of demand paging. This is because with less pages there can be lower probability that for a given sequence of processes ran on the CPU (via context switch/multiprogramming) their respective pages are already loaded in memory so that they can run without waiting for their page table to be loaded from the disk into memory. In contrast, however as mentioned above, if there is a chance the sequence of processes switched between each other all have their larger page tables in memory, which means there is a higher chance for the CPU to run a process longer since it doesn't have to wait on demand paging.

Question 6:

6. (15 points) Consider the following page-reference string:

   1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

   How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, or seven frames? Remember that all frames are initially empty, so your unique pages will all cost one fault each

   - LRU replacement
   - FIFO replacement
   - MIN, Optimal replacement

Question 7:

7. (10 points) Consider a demand-paging system with a paging disk. The average disk access time is 10 milliseconds, and the average memory access time is 100 nanoseconds. Assume 0.1 percent of the access causes page faults.

A)

a) What is the effective memory access time?
   - 10ms = 1,000,000 * 10 = 10,000,000ns
   - 0.1% = 0.01
   - Effective Access Rate (EAT) = hit time + mis rate * miss penalty =
   - 100ns + 0.01 * 10,000,000ns = **100100ns (ns = nanoseconds)**

B)

b) In order to achieve no more than 5 percent overhead due to demand paging for this system, what should the maximum page fault rate be?

- (100+5)ns = 100ns + miss rate * 10,000,000ns
- **Max miss rate** = ((105ns – 100ns) /10,000,000ns) * 100% = **0.00005%**

Question 8:

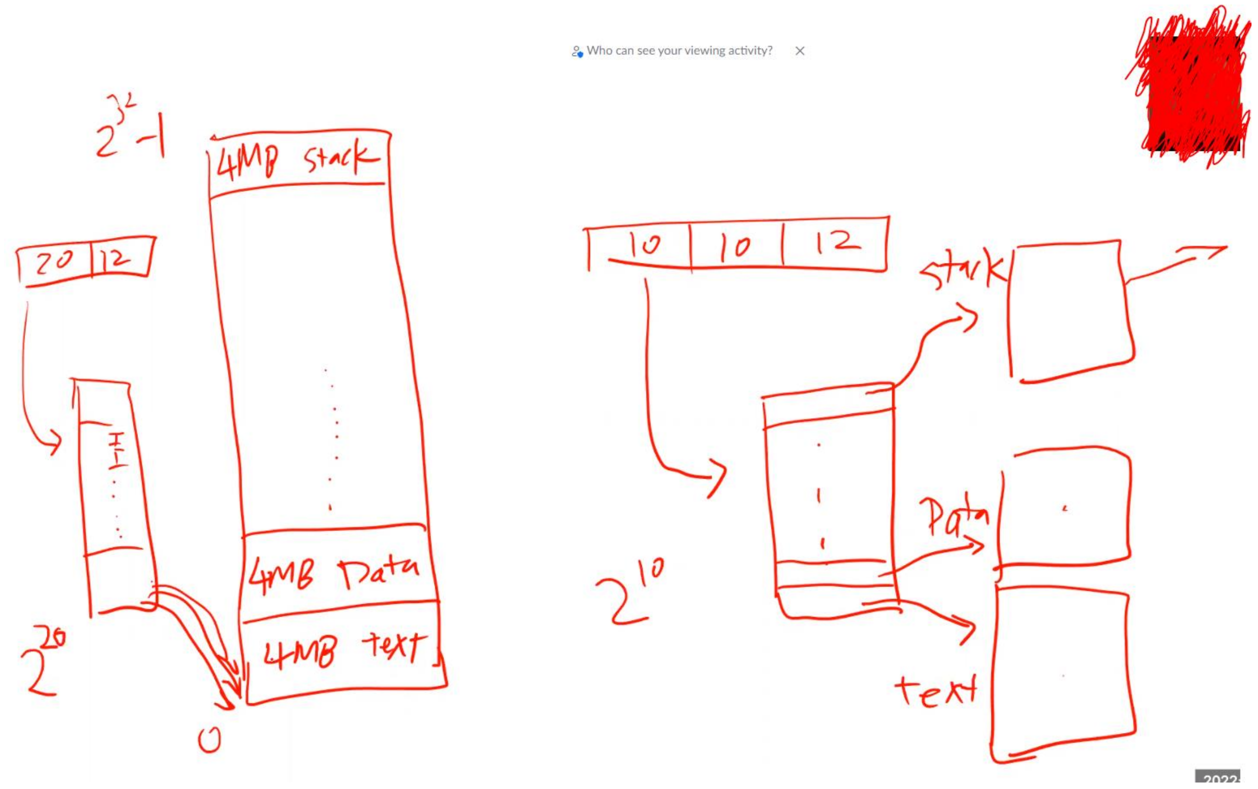
8. (10 points) Contrast the performance of the three techniques for allocating disk blocks (contiguous, linked, and indexed) for both sequential and random file access.
Contiguous is good for both sequential and random access files, since when allocating blocks contiguously, it is easy to access memory sequentially or randomly since you can easily calculate offsets. For Linked, it is only good for sequential since an entire page must be read in before accessing any data in a given block. For indexed, since it only has a table that stores all the disk pointers in one of the allocated blocks, thus, you can easily access data sequentially or randomly with good performance.

Question 9:

9. (10 points) Consider the i-node scheme used by UNIX (12 pointers to blocks, and 1 single, double, and triple indirect block), and suppose that no file cache is in use. How many disk accesses does it take to access the following blocks: 1, 10, 200, 500, 1,000, 2,200, 10,000, 100,000, 1,000,000, 1,400,000, 5,000,000? Blocks are 4K each and the i-node is permanently cached in memory. Each index block can contain up to 256 pointers to data blocks or other index blocks.
   -

Notes/Hints from H2 discussion video

Question 2



[redacted]