Efe Beydogan

21901548

CS342 HW2

Section 2

## Q1)

Available:

A B C D

6 3 5 4

Max:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 9 | 5 | 5 | 5 |
| P1 | 2 | 2 | 3 | 3 |
| P2 | 7 | 5 | 4 | 4 |
| P3 | 3 | 3 | 3 | 2 |
| P4 | 5 | 2 | 2 | 1 |
| P5 | 4 | 4 | 4 | 4 |

Allocation:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 2 | 0 | 2 | 1 |
| P1 | 0 | 1 | 1 | 1 |
| P2 | 4 | 1 | 0 | 2 |
| P3 | 1 | 0 | 0 | 1 |
| P4 | 1 | 1 | 0 | 0 |
| P5 | 1 | 0 | 1 | 1 |

Need:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 7 | 5 | 3 | 4 |
| P1 | 2 | 1 | 2 | 2 |
| P2 | 3 | 4 | 4 | 2 |
| P3 | 2 | 3 | 3 | 1 |
| P4 | 4 | 1 | 2 | 1 |
| P5 | 3 | 4 | 3 | 3 |

Step 1) Initialize Work = [6, 3, 5, 4], Finish = [False, False, False, False, False, False]

Steps 2 & 3) Find a process i such that Need[i] <= Work && Finish[i] == False

Work = Work + Allocation[i], Finish[i] = True

1) P1: Need[1] = [2, 1, 2, 2] <= Work = [6, 3, 5, 4] && Finish[1] == False
   Work = [6, 4, 6, 5] and Finish = [False, True, False, False, False, False]

2) P2: Need[2] = [3, 4, 4, 2] <= Work = [6, 4, 6, 5] && Finish[2] == False
   Work = [10, 5, 6, 7] and Finish = [False, True, True, False, False, False]

3) P0: Need[0] = [7, 5, 3, 4] <= Work = [10, 5, 6, 7] && Finish[0] == False
   Work = [12, 5, 8, 8] and Finish = [True, True, True, False, False, False]

4) P3: Need[3] = [2, 3, 3, 1] <= Work = [12, 5, 8, 8] && Finish[3] == False
   Work = [13, 5, 8, 9] and Finish = [True, True, True, True, False, False]

5) P4: Need[4] = [4, 1, 2, 1] <= Work = [13, 5, 8, 9] && Finish[4] == False
   Work = [14, 6, 8, 9] and Finish = [True, True, True, True, True, False]

6) P5: Need[5] = [3, 4, 3, 3] <= Work = [14, 6, 8, 9] && Finish[5] == False
   Work = [15, 6, 9, 10] and Finish = [True, True, True, True, True, True]

Step 4) After the last iteration in the previous step, we can't find a process with Finish[i] == False, as all the processes could finish. The safe sequence is: <1, 2, 0, 3, 4, 5>. **Hence, the current state is safe.**

**P5 requests (3, 2, 3, 3):**

Available:

A B C D

6 3 5 4

Max:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 9 | 5 | 5 | 5 |
| P1 | 2 | 2 | 3 | 3 |
| P2 | 7 | 5 | 4 | 4 |
| P3 | 3 | 3 | 3 | 2 |
| P4 | 5 | 2 | 2 | 1 |
| P5 | 4 | 4 | 4 | 4 |

Allocation:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 2 | 0 | 2 | 1 |
| P1 | 0 | 1 | 1 | 1 |
| P2 | 4 | 1 | 0 | 2 |
| P3 | 1 | 0 | 0 | 1 |
| P4 | 1 | 1 | 0 | 0 |
| P5 | 1 | 0 | 1 | 1 |

Need:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 7 | 5 | 3 | 4 |
| P1 | 2 | 1 | 2 | 2 |
| P2 | 3 | 4 | 4 | 2 |
| P3 | 2 | 3 | 3 | 1 |
| P4 | 4 | 1 | 2 | 1 |
| P5 | 3 | 4 | 3 | 3 |

Step 1) [3, 2, 3, 3] <= Need[5] = [3, 4, 3, 3], so it can proceed to the next step.

Step 2) [3, 2, 3, 3] <= Available = [6, 3, 5, 4], so the process doesn't need to wait.

Step 3) Pretend to allocate the resources to the process:

Available = [3, 1, 2, 1]

Need[5] = [0, 2, 0, 0]

Allocation[5] = [4, 2, 4, 4]

New state of the tables:

Available:

A B C D

3 1 2 1

Max:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 9 | 5 | 5 | 5 |
| P1 | 2 | 2 | 3 | 3 |
| P2 | 7 | 5 | 4 | 4 |
| P3 | 3 | 3 | 3 | 2 |
| P4 | 5 | 2 | 2 | 1 |
| P5 | 4 | 4 | 4 | 4 |

Allocation:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 2 | 0 | 2 | 1 |
| P1 | 0 | 1 | 1 | 1 |
| P2 | 4 | 1 | 0 | 2 |
| P3 | 1 | 0 | 0 | 1 |
| P4 | 1 | 1 | 0 | 0 |
| P5 | 4 | 2 | 4 | 4 |

Need:

| | A | B | C | D |
|---|---|---|---|---|
| P0 | 7 | 5 | 3 | 4 |
| P1 | 2 | 1 | 2 | 2 |
| P2 | 3 | 4 | 4 | 2 |
| P3 | 2 | 3 | 3 | 1 |
| P4 | 4 | 1 | 2 | 1 |
| P5 | 0 | 2 | 0 | 0 |

Run the safety check algorithm:

Step 1) Work = [3, 1, 2, 1] and Finish = [False, False, False, False, False, False]

Steps 2 & 3) Find a process i such that Need[i] <= Work && Finish[i] == False

Work = Work + Allocation[i], Finish[i] = True

Unable to find such a request

Step 4) Finish[i] == False for every i, so **there is no safe sequence if we allocate P5's request. Hence, the changes must be reversed and the request shouldn't be granted.**

## Q2)

a) EAT = 2 * 150 + ε – α where ε = access time to TLB and α = hit ratio. There is no TLB in this case, so **EAT = 300 ns**. One access for the page table, another one to retrieve the info from the memory.
b) EAT = 0.85 x (10 + 150) + 0.15 x (10 + 150 + 150) = **182.5 ns**.
c)

   Answer for a) if two level paging used: 150 + 150 + 150 = **450 ns**. Two accesses for two page tables and one access to retrieve the info from memory.

   Answer for b) if two level paging used: 0.85 x (10 + 150) + 0.15 x (10 + 150 + 150 + 150) = **205 ns**.

## Q3)

a) Page offset is 12 bits, and physical addresses are 48 bits, so PFN is 48 – 12 = 36 bits. Hence, **36 bits** are used to store a frame number in a PTE.
b) In a 4th level page table, there are 2^9 entries. Each entry maps a page of size 2^12 bytes. Hence, each 4th level page table maps 2^9 * 2^12 = 2^21 = 2 MB.

   0x000000300000 = 3 x 2^20. 3 * 2^20 / 2^21 = 3 / 2 = 1.5. This means, the mapping for this process starts at the half of the 1st indexed page table in level 4. Also, the process occupies 6 MB virtual memory, so it must span 6 / 2 = 3 page tables in the 4th level. This means, 1.5 + 3 = 4.5, so 1st, 2nd, 3rd and 4th indexed page tables in level 4 are used. To map these, only the 0 indexed page table in level 3, and 0 indexed page table in level 2 are required. Finally, there is one top level page table. In total, 7 page tables are needed. Each of these tables have 2^9 entries and each entry is 8 bytes. Putting this all together, total space required for the page tables is: **(4 + 1 + 1 + 1) * 2^9 * 8 = 7 * 2^12 = 28 KB**.

c)

**4th level:**

Code: 0x000001000000 = 2^24, 2^24 / 2^21 = 8. The code segment starts from the 8th indexed table in level-4, and it is 128 KB. One page table in level 4 maps 2 MB of virtual memory, so for the code segment only the 8th indexed table is enough.

Data: 0x000800000000 = 2^35, 2^35 / 2^21 = 2^14. The data segment starts from the 2^14th indexed table in level 4. It spans 2 GB, and (2 * 2^30) / (2 * 2^20) = 2^10, so the data segment spans 1024 tables in level-4.

Stack: 0x0f0000000000 = 2^40 * 15, (2^40 * 15) / 2^21 = 2^19 * 15. The data segment starts from the (2^19 * 15) indexed table in level 4, and it spans 4 MB. 4 / 2 = 2, so 2 level-4 tables are required to map the whole stack segment.

**3rd and 2nd levels:**

Code: The code segment uses the 8th indexed table in level 4, so it must use the 8 / 512 = 0th table in levels 3 and 2.

Data: The data segment starts from the 2^14th indexed table in level 4 and spans 1024 tables. 2^14 / 2^9 = 32. So, in level-3, the mapping for the data segment starts from the 32nd indexed table. Moreover, since the data segment spans 1024 tables in level-4, this means 1024 / 512 = 2 tables are required to map these tables in level-3, since a table in level-3 maps 512 tables in level-4. All in all, tables 32 and 33 are used in level-3 for the data segment. Both of these tables can be mapped by the 0th table in level 2.

Stack: (2^19 * 15) / 2^9 = 2^10 * 15th table in level-3. The stack segment needs 2 level-4 tables, so only one table in level-3 is needed. (2^10 * 15) / 2^9 = 30th table in level-2 must be used to map the level-3 table of the stack segment.

Putting everything together:

Level-4: 1 + 1024 + 2 = 1027 tables

Level-3 = 1 + 2 + 1 = 4 tables

Level-2: 1 + 1 = 2 tables

Level-1: only the top level table, so 1 table

**(1027 + 4 + 2 + 1) * 2^9 * 8 = 1034 * 2^12 = 4136 KB = 4.03 MB**

## Q4)

a) In the below table, the highlighted numbers indicate which page the victim pointer is pointing to at that moment.

| **Page Reference** | **2** | **4** | **1** | **3** | **4 (R bits reset)** | **6** | **3** | **6** | **1** | **2 (R bits reset)** | **1** | **5** | **2** | **5** | **4 (R bits reset)** | **1** | **2** | **4** | **3** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Frame 1** | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 3 |
| Reference | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| **Frame 2** | | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 6 | 6 | 5 | 5 | 5 | 5 | 5 | 2 | 2 | 2 |
| Reference | | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| **Frame 3** | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 4 | 4 | 4 | 4 | 4 |
| Reference | | | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| Page Faults | X | X | X | X | | X | | | | X | X | X | | | X | | X | | X |

b)

| **Page Reference** | **2** | **4** | **1** | **3** | **4** | **6** | **3** | **6** | **1** | **2** | **1** | **5** | **2** | **5** | **4** | **1** | **2** | **4** | **3** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Frame 1** | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 5 | 5 | 5 | 4 | 4 | 4 | 4 | 3 |
| **Frame 2** | | 4 | 4 | 4 | 4 | 6 | 6 | 6 | 6 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| **Frame 3** | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| Page Faults | X | X | X | X | | X | | | | X | | X | | | X | | | | X |

## Q5) 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive is at track 4500.

a) For FCFS, the tracks are visited in the given order.
5200 – 4500 = 700
5200 – 2000 = 3200
9500 – 2000 = 7500
9500 – 4300 = 5200

4300 – 1500 = 2800
5100 – 1500 = 3600
9600 – 5100 = 4500
9600 – 4000 = 5600
4600 – 4000 = 600
**Total = 33700**

b) SCAN:
The track numbers in sorted order:
1500 2000 4000 4300 [4500>>>] 4600 5100 5200 9500 9600
**Total: (9999 - 4500) + (9999 - 1500) = 13998**

c) SSTF: 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive is at track 4500
4600 – 4500 = 100, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 4600
4600 – 4300 = 300, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 4300
4300 – 4000 = 300, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 4000
5100 – 4000 = 1100, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 5100
5200 – 5100 = 100, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 5200
5200 – 2000 = 3200, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 2000
2000 – 1500 = 500, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 1500
9500 – 1500 = 8000, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 9500
9600 – 9500 = 100, 5200 2000 9500 4300 1500 5100 9600 4000 4600, drive at 9600
**Total = 13700**

## Q6)

a) Transfer time: 4 KB / 30 MB = 1 / 7680 s = 0.13 ms
Time it takes for one rotation: 60 / 3600 = 1 / 60 s = 16.66 ms
Average rotational latency = 16.66 / 2 = 8.33 ms
**I/O time to read one random block from this disk: T = 4 + 8.33 + 0.13 = 12.46 ms**
**Number of transfers per second: 1000 / 12.46 = 80**
**Throughput: (1000 * 4) / 12.46 = 321.03 KB/s = 0.314 MB/s**

b) Transfer time: (4 KB * 512) / 30 MB = 1 / 15 s = 66.6 ms
I/O time to read contiguous blocks from this disk: T = 4 + 8.33 + 66.6 = 78.93 ms
**Number of transfers per second: 1000 / 78.93 = 12**
**Throughput: (1000 * 4 * 512) / 78.93 = 25947.04 KB/s = 25.34 MB/s**

## Q7)

a) 4 KB / 8 B = 2^9, so we can store 2^9 pointers in one block.
There are 10 + 512 + 512^2 + 512^3 ~= 2^27 blocks. Each block is 4 KB, therefore there are 2^27 * 2^12 = 2^39 bytes in total, so **the max file size is approximately 512 GB**.

b) A leaf index block can map 2^9 * 2^12 = 2^21 = 2MB of file data. For 8GB, we need (8 * 2^30) / (2 * 2^20) = 4096 leaf index blocks. For this, we need to use the single level index structure (one leaf index block), the two-level index structure and the three level index structures. We can subtract one table because of the single level index, so we need to have 4095 leaf index blocks from two-level and three-level index structures. The two-level index structure has 512 leaf index blocks, so from the three-level index structure, we need 4095 – 512 = 3583 leaf index blocks. To map 3583 3rd level index blocks, we need 7 second-level index blocks in the three-level index structure (3583 / 512 = 6.99). In total, we need 512 + 7 = 519 second-level index blocks. There are 10 direct pointers, but they only account for 40KB, so they are not enough to decrease the leaf index block size. **The final answer is 519.**

c) i) 2^15 / 2^12 = 8, so the byte at offset 2^15 is in block 8. There are 10 direct pointers, so this block is directly pointed by the inode. We need only **1 disk access** to retrieve the data block.

ii) 2^23 / 2^12 = 2^11 = 2048. So, the byte at this offset is in the 2048th block. The direct pointers point at the first 10 blocks, the single level index structure points at the next 512 blocks, and the two-level index structure points at the next 512^2 blocks, therefore the 2048th block falls within the range of the two-level index structure. Hence, we need to access the top-level table of this structure, then the second-level index table and finally we can retrieve the disk block with another disk access. **We need 3 disk accesses in total.**

iii) 2^32 = 4GB. 4GB / 2MB = 2^11. We need to find where the 2^11th leaf index block is. The single-level and two-level index structures amount to 513 leaf index blocks combined, so we need to look at the three-level index structure. For this, we have to access the top-level table, then a second-level table, a third-level table and finally the block itself. **4 disk accesses are needed.**