SYSC4001 - Assignment 3 Part III
Zaineb Ben Hmida - 101302936
Sohaila Haroun - 101297624

1. [0.6 marks]
Consider the following page reference string in an Operating System with a Demand Paging memory management strategy:
415, 305, 502, 417, 305, 415, 502, 518, 417, 305, 415, 502, 520, 518, 417, 305, 502, 415, 520, 518

(i) [0.3 marks] Assume we have 3 Frames Allocated. How many page faults will occur with 3 frames allocated to the program using the following page replacement algorithms?

a) FIFO (First-In-First-Out)

~~415~~	417	417	~~417~~	305	305	~~305~~	518	518	~~518~~	502	502	~~502~~	518					
305	~~305~~	415	415	~~415~~	502	502	~~502~~	417	417	~~417~~	415	415	415					
502	502	~~502~~	518	518	~~518~~	520	520	~~520~~	305	305	~~305~~	520	520					

16 page faults
4 page hits Highlighted
20 references in total
4/20 = 0.2 therefore 20% hit ratio

(b) LRU (Least Recently Used) [Student 1: explain the LRU algorithm]

~~415~~	417	~~417~~	502	502	~~502~~	305	305	~~305~~	520	520	~~520~~	305	305	~~305~~	520	520		
305	305	305	~~305~~	518	518	~~518~~	415	415	~~415~~	518	518	~~518~~	502	502	~~502~~	518		
502	~~502~~	415	415	~~415~~	417	417	~~417~~	502	502	~~502~~	417	417	~~417~~	415	415	415		

19 page faults
1 hit
20 references in total
1/20 = 0.05 therefore 5% hit ratio

(c) Optimal

415	~~415~~	~~502~~	518	518	518	518	518	518	518
305	305	305	~~305~~	~~415~~	~~502~~	520	520	520	520
~~502~~	417	417	417	417	417	~~417~~	~~305~~	~~502~~	415

12 page faults
8 hits Highlighted
20 references in total
8/20 = 0.4 therefore 40% hit ratio

(ii) [0.1 marks] Assume that the case above is repeated with 4 Frames Allocated. Repeat all three algorithms from Part (i) with 4 frames allocated to the program. Show your work for each of the algorithms. Calculate the hit ratio for each of the algorithms

a) FIFO (First-In-First-Out)

~~415~~	(Hit 415)	518			518			518	(Hit 518)		~~518~~	502			502				
305	(Hit 305)	~~305~~	(Hit 305)		415			415			415		~~415~~	(Hit 415)		518			
502	(Hit 502)	502			~~502~~	(Hit 502)		520			520		520	(Hit 520)		520			
417		417	(Hit 417)		417			~~417~~	(Hit 417)		305		305			305			

10 page faults
10 hits
20 references in total
10/20 = 0.5 therefore 50% hit ratio

(b) LRU (Least Recently Used) [Student 1: explain the LRU algorithm]

415	(Hit 415)	415	~~415~~	305	305	305	~~305~~	518	518	518	~~518~~	415	415	415					
305	(Hit 305)	~~305~~	417	417	417	~~417~~	520	520	520	~~520~~	502	502	502	502					
502	(Hit 502)	502	502	~~502~~	415	415	415	~~415~~	417	417	417	~~417~~	520	520					
~~417~~		518	518	518	~~518~~	502	502	502	~~502~~	305	305	305	~~305~~	518					

17 page faults
3 hits
20 references in total
3/20 = 0.15 therefore 15% hit ratio

c) Optimal

415	Hit	~~415~~	Hit	~~502~~	520		520	520	Hit										
305	Hit	305	Hit	305	~~305~~	Hit	502	502											
~~502~~	Hit	518		518	518	Hit	518	518	Hit										
417		417	Hit	417	417	Hit	~~417~~	415											

6 page faults
11 hits
20 references in total
11/20 = 0.55 therefore 55% hit ratio

(iii) [0.2 marks] Based on your results, answer the following questions: Which algorithm performs best with 3 frames and why? Which algorithm performs best with 4 frames and why? How do the results change when more frames are allocated? What is the relationship? Why is the Optimal algorithm impractical in real-world operating systems? Compare the performance of FIFO and LRU. When might FIFO be better or worse than LRU?

Which algorithm performs best with 3 frames and 4 frames and why?
The optimal algorithm consistently performs the best when allocating 3 or 4 frames. With a 40% hit ratio, optimal performs best with 3 frames. Compared to FIFO with 20% hit ratio and LRU with a 5% hit ratio, the optimal algorithm performs the best by far. Similar results are seen using 4 frames with a 55% hit ratio optimal algorithm again performs best. Compared to FIFO with a 50% hit ratio and LRU with a 15% hit ratio, the optimal algorithm performs the best among the other algorithms. The optimal algorithm performs the best due to its nature where it always replaces the page that will not be used for the longest amount of time, therefore minimizing the most amount of page faults and resulting in the most hits.

How do the results change when more frames are allocated? What is the relationship?
When more frames are allocated, all the algorithms tend to perform better. As can be seen when comparing part i and ii, where FIFO gives a hit ratio of 20% in 3 frames and 50% in 4 frames, LRU gives a hit ratio of 5% in 3 frames and 15% in 4, and finally optimal provides a hit ratio of 40% using 3 frames and 55% using 4. The results show that the hit ratio increases with 4 frames meaning less page faults and more hits and a better overall result.

Why is the Optimal algorithm impractical in real-world operating systems?
The optimal algorithm is impractical in real-world operating systems, as you must now the future for it to work. The optimal algorithm works by replacing the page that will not be used for the longest amount of time, meaning the system must know the future references which is not possible in practice. So while in theory it is the best algorithm in practice it is not feasible.

Compare the performance of FIFO and LRU. When might FIFO be better or worse than LRU?
Comparing FIFO and LRU, using 3 frames FIFO provides a hit ratio of 20% and LRU one of 5%, and using 4 frames FIFO provides a hit ratio of 50% and LRU one of 15%. These results show that FIFO has a far better performance than LRU in both 3 and 4 frames for this reference string. However, generally LRU performs better in scenarios where recently used pages are highly likely to be used again in the near future which is common in real world systems (principle of locality). FIFO may perform better than LRU in predictable patterns like the ones from this question, where older pages are less likely to be used again.

[Student 1: explain the LRU algorithm]

LRU is the least recently used algorithm, where we replace the page that has not been used for the longest period of time. In the LRU algorithm, when a page must be replaced, the page that has not been used for the longest period of time is replaced by the new page. LRU uses the recent past as an approximation of the future, mimicking the optimal page-replacement algorithm, but looking backward in time rather than forward, which is not feasible in real-world applications.

[Student 2: explain the LFU algorithm]

The Least Frequently Used (LFU) algorithm is a *counting* page replacement algorithm. In counting algorithms, each page has 1-2 bits in the page table that is dedicated to a counter that keeps track of the number of references made to that page. The LFU algorithm then replaces the page that has the smallest count.

2. [0.3 marks] Consider a system with memory mapping done on a page basis. Assume that the necessary page table is always in main memory. A single main memory access takes 120 nanoseconds (ns).
(a) [0.1 marks] How long does a paged memory reference take in this system without a TLB? Explain your answer.
Memory access time = page table + data
Memory access time = 120ns + 120ns = = 240ns

In a system with memory mapping done on a page basis every memory access first reads the page table (always in main memory for this example) to find the address in memory and then access the data in that address in memory. Since each memory access takes 120 ns, the total time for a paged memory reference without a TLB is 240ns.

(b) [0.1 marks] If we add a Translation Lookaside Buffer (TLB) that imposes an overhead (t) of 20 ns on a hit or a miss. If we assume a TLB hit ratio of 95%, what is the effective memory access time (EMAT)? Explain your answer.
EMAT = h(t + m) + (1 - h)(t + m + m)
EMAT = 0.95(20 + 120) + (1 - 0.95)(20 + 120 + 120)
EMAT = (0.95×140)+(0.05×260)
EMAT = 146ns

A Translation Lookaside Buffer (TLB) caches recent page table entries. Adding a TLB means that on a hit, we no longer require reading the page table from main memory, but it imposes an overhead of 20ns per lookup. With a 95% hit ratio, most of the time, the page table does not need to be read from main memory, but on a miss, the memory system must read the page

table from main memory and the data, plus the 20ns overhead per lookup. The effective memory access (EMAT) is therefore 146ns.

(c) [0.1 marks] Why does adding an extra layer, the TLB, generally imp rove performance?
Are there situations where the performance may be worse with a TLB than without one? Explain all cases.
Adding an extra layer, the TLB, generally improves performance because it allows some memory access to avoid the main memory page lookup step, which takes time as it is an extra main memory access per main memory access, and this reduces the total amount of memory operations required overall (if the hit ratio is good). This is especially beneficial when the same page is accessed repeatedly. Situations where the performance may be worse with a TLB than without one are when the TLB hit rate is very low and the overhead imposed by the TLB is high. Consider the case where the page's entry is not in the TLB, this memory access will both require 2 main memory accesses and has already been affected by the TLB overhead (t + 2m).This results in slower memory accesses when using both the TLB and the normal memory access, resulting in a larger effective access time than a system without a TLB.

3. [0.3 marks] Consider a system with a paged logical address space composed of 128 pages of 4Kbytes each, mapped into a 512 Kbytes physical memory space. Answer the following questions and justify your answers.
(a) [0.1 marks] What is the format and size (in bits) of the processor's logical address?
Logical address space = 128 pages
Pages = 4 Kbytes
Pages = 4 x 1024 = 4096 bytes

Number of bits per page number = $\log_2$(logical address space)
Number of bits per page number = $\log_2$(128)
Number of bits per page number = 7 bits

Number of bits per offset = $\log_2$(pages)
Number of bits per offset = $\log_2$(4096)
Number of bits per offset = 12 bits

Processor's logical address size = 7 + 12
Processor's logical address size = 19 bits

Processor's logical address format:
[7-bit page number | 12-bit offset]

(b) [0.1 marks] What is the required length (number of entries) and width (size of each entry in bits, disregarding control bits) of the page table?

Logical address space = 128 pages
Pages = 4 Kbytes
Pages = 4 x 1024 = 4096 bytes
Physical memory space = 512 Kbytes
Physical memory space = 512 x 1024 = 524,288 bytes

Number of frames = Physical memory space ÷ Pages
Number of frames = 524,288 ÷ 4096 or (512KB/4KB)
Number of frames = 128 frames

Number of bits for the frame = $\log_2$(Logical address space)
Number of bits for the frame = $\log_2$(128)
Number of bits for the frame = 7 bits

Therefore, the required length (number of entries) of the page table is 128 pages, and the required width (size of each entry in bits, disregarding control bits) of the page table is 7 bits.

(c) [0.1 marks] What is the effect on the page table width if now the physical memory space is reduced by half (from 512 Kbytes to 256 Kbytes)? Assume that the number of page entries and page size remain the same.

If the physical memory is halved, the page table width will be 1 bit smaller. (this is clear just from understanding how bits work but here are some calculations to prove it:)
.
Number of frames = Physical memory space ÷ Pages
Number of frames = 256KB/4KB
Number of frames = 64 frames

Number of bits for the frame = $\log_2$(Logical address space)
Number of bits for the frame = $\log_2$(64)
Number of bits for the frame = 6 bits

[Student 1: explain what the physical memory space is]

Physical memory space is the actual hardware memory, RAM, of the computer. Physical memory space refers to the limited, real memory location that the CPU can directly access. The physical memory is divided by the operating system into frames, or fixed-size memory blocks, used to work with virtual memory. Data in physical memory is volatile and is lost when the CPU powers off.

[Student 2: explain what the logical memory space is]

The logical memory space is the set of addresses that a program *thinks* it has while it is running.

These addresses (logical/virtual) are handled by the CPU when the program executes instructions.

Logical memory is independent of where the program is actually stored in physical RAM.

In a virtual memory system, this exists because of virtual memory, which separates the program's view of memory (logical addresses) from the actual hardware memory (physical addresses). Because of this separation:

- A program does not need to have all its code in RAM at once—only the parts being executed.
- The logical address space can be much larger than the physical RAM, since virtual memory can use disk space to extend it.
- Multiple processes can each have their own logical address space, making it appear as if each process has "its own private memory".

Quick summary (mostly for myself):
The compiler generates relative addresses (like using offsets).
When a program runs, the CPU generates/uses logical addresses (also known as virtual addresses).
The MMU translates logical addresses → physical addresses (for ex: using page tables).

4. [0.1 marks] Explain, in detail, the sequence of operations and file system data structure accesses that occur when a process executes the lseek(fd, offset, SEEK_END) system call.
Consider a system using a hierarchical directory structure and assume the file described by the file descriptor (fd) is not currently open by any other process.

When a process executes the lseek(fd, offset, SEEK_END) system call:

It creates a "software interrupt" which causes a switch to kernel mode and a contact switch. The provided interrupt vector is then used to get the first address of the corresponding ISR from the vector table, which will be the first address of the lseek routine.

The operating system then uses the file descriptor to look up the corresponding entry in the process's file descriptor table, which points to the system-wide open file table entry for that file. From there, the OS accesses the file's inode, which contains the current file size which is needed because SEEK_END means the new position is calculated relative to the end of the file. If the inode is not already cached, it must be read from disk. The kernel then computes the new offset as file size + offset and updates the stored file position in the open file table entry. Importantly, the directory structure is not accessed during this call because the file is already open, and no data blocks are touched until the process actually reads or writes. Finally, the new position is returned to the process.

5. File System Organization
a) [0.1 marks] (from Silberschatz) Consider a file system that uses inodes to represent files. Disk blocks are 8Kb in size, and a pointer to a disk block requires 4 bytes. This file system has 12 direct disk blocks, as well as single, double, and triple indirect disk blocks. What is the maximum size of a file that can be stored in this file system?

Direct blocks:

12 * 8KB = 96KB

Single indirect:
1 block, 2^11 = 2048 pointers -> 2048 * 8 KB
= 16,384 KB = 16 MB

Double indirect:
2048 (first level) * 2048 (second level) * 8 KB
= 4,194,304 blocks * 8 KB
≈ 32 GB

Triple indirect:
2048^3 × 8 KB
= 8,589,934,592 blocks × 8 KB
= ~68,719,476,736 KB ≈ 64 TB (* this doesn't feel right, but I tried following with the textbook and got lost)

Direct: 12 × 8192 = 98,304 bytes (≈ 96 KB)
Single indirect: 2048 × 8192 = 16,777,216 bytes (≈ 16 MB)
Double indirect: $2048^2 \times 8192 = 34,359,738,368$ bytes (≈ 32 GB)

Triple indirect: $2048^3 \times 8192 = 70,368,744,177,664$ bytes (≈ 64 TB)
Total maximum file size = 96 KB+16 MB+ 32 GB+ 64 TB
Total ≈ 64 TB (dominated by triple indirect)

Maximum file size ≈ 64 TB.

Or is it:
direct= 12*8KB = 48 KB = 384000
indirect= 12*8KB*8KB
Double indirect= 12*8KB*8KB *8KB
Triple 12*8KB*8KB *8KB*8KB

b) [0.1 marks] Explain what you can do in case (a) if you need to store a file that is larger than the maximum size computed. Give an example showing how you can define a larger file, and what the size of that file would be.

If you need a larger file you can (A) split the logical file across multiple files and manage them at the application level or with a volume layer, (B) reformat the filesystem with a larger block size (or use a filesystem with wider pointers/offsets), or (C) use a distributed/cluster filesystem or logical volume that presents a larger address space.
If the on-disk inode structure can address more space, the kernel's in-memory file-offset width (e.g., 32-bit signed) may still limit usable file size to $2^{31}-1$ bytes unless the OS/filesystem uses 64-bit offsets. (according to textbook)