Sitare Arslantürk, 57677

COMP 304 Operating Systems: Assignment 3

Due: May 13, midnight

Problem 1:

	Code Sharing	**External Fragmentation**
Segmentation	In segmentation, code sharing is easy. It could be managed by assigning each sharable section of code to its own segment. Then, a single identical mapping allows two processes to share the code.	The segmentation suffers from external fragmentation as the memory is filled with variable sized blocks. A segment of a process is laid out contiguously in memory and as segments of old processes are replaced by new processes external fragmentation can occur.
Paging	Paging allows processes to share code. Particular set of pages needs to be mapped to the shared code. Each section of sharable code would need its own set of pages, in which internal fragmentation might be an issue.	Paging does not suffer from external fragmentation. It allows a process to be stored in a non-contiguous method which solves the problem of external fragmentation**.**
Contiguous Allocation	In contiguous allocation code sharing is nearly impossible so it does not allow processes to share code because each process has an independent memory space and a process' virtual memory segment cannot be broken into segments.	Contiguous allocation suffers from external fragmentation because address spaces are all located contiguously and holes develop as old processes die and new processes start. A solution to this is compaction where the used memory allocations are moved to one side in order to use the free memory available.

Problem 2:

$2^{32} \div 2^{13} = 2^{19}\ pages$

For each page, one page table entry is required. So, there will be 2^{19} entries.

Each entry is 4 bytes => $2^{19} * 2^{2} = 2^{21}$ bytes or 2 MB

The process uses 100 pages * 4 bytes = 400 bytes or 0.4 KB

However, its page table will require all pages in order to perform the translation so it will need 2 MB.

Problem 3:

a) 5 Frames

LRU	FIFO	Optimal
8	**10**	**7**

i) **LRU**

	1	2	3	4	2	1	5	6	2	1	2	3	7
Frame1	1					NF				NF			
Frame2		2			NF				NF		NF		
Frame3			3					6					
Frame4				4								3	
Frame5							5						7
Faults = All – Not Faults = 13 – 5 = 8													

ii) **FIFO**

	1	2	3	4	2	1	5	6	2	1	2	3	7
Frame1	1					NF		6					
Frame2		2			NF				NF	1			
Frame3			3								2		
Frame4				4								3	
Frame5							5						7
Faults = All – Not Faults = 13 – 3 = 10													

iii) **Optimal**

	1	2	3	4	2	1	5	6	2	1	2	3	7
Frame1	1					NF				NF			
Frame2		2			NF				NF		NF		
Frame3			3									NF	
Frame4				4									7
Frame5							5	6					
Faults = All – Not Faults = 13 – 6 = 7													

b) 6 Frames

LRU	FIFO	Optimal
7	**7**	**7**

i) **LRU**

	1	2	3	4	2	1	5	6	2	1	2	3	7
Frame1	1					NF				NF			
Frame2		2			NF				NF		NF		
Frame3			3									NF	
Frame4				4									
Frame5							5						7
Frame6								6					

Faults = All – Not Faults = 13 – 6 = 7

ii) **FIFO**

	1	2	3	4	2	1	5	6	2	1	2	3	7
Frame1	1					NF				NF			7
Frame2		2			NF				NF		NF		
Frame3			3									NF	
Frame4				4									
Frame5							5						
Frame6								6					

Faults = All – Not Faults = 13 – 6 = 7

iii) **Optimal**

	1	2	3	4	2	1	5	6	2	1	2	3	7
Frame1	1					NF				NF			
Frame2		2			NF				NF		NF		
Frame3			3									NF	
Frame4				4									
Frame5							5						7
Frame6								6					

Faults = All – Not Faults = 13 – 6 = 7

Problem 4

	1	**2**	**3**	**4**	**5**	**3**	**4**	**1**	**6**	**7**	**8**	**7**	**8**	**9**	**7**	**8**	**9**	**5**	**4**	**5**	**4**	**2**
Frame1	1				5				6	7	8	7			N F							2
Frame2		2						1										5		N F		
Frame3			3			N F							8			N F						
Frame4				4			N F							9			N F		4		N F	
Counter	1	1	1	1	1	2	2	2	1	1	1	2	2	1	3	3	2	2	3	3	4	2

Faults = 22 – 7 = 15

A good page replacement algorithm can reduce the page faults, when the program is executing, and effectively increase the system's efficiency. The performance of the system increases when less page faults is made. Let's compare the algorithm's page fault with that of optimal and LRU. The LRU algorithm produces 13 page faults while the optimal algorithm produces 11 page faults. Although not effective as LRU or Optimal, I would still consider this a good algorithm depending on its usage.

Problem 5

a) When the new file with the same absolute path name is created, the links of the old file which continues to exist will now point to the new file. The user will be accessing the old file instead of accessing the new file. In addition, the protection mode for the old file might get used instead of the new file's protection mode.

b) One possible way to solve this problem is to keep a list of links to a file with the file data and remove all links of a file when it is deleted. Another could be to keep a reference count as Unix inode does and only delete the file data when the reference count is 0. The Unix inode is the representation of each file on disk. It has a reference count and the file is not deleted until the reference count is 0. Every time a link is created, the reference count is incremented and every time a link is removed, the reference count is decremented.

Problem 6

a)

1. P1

2	**5**	**7**	**4**	**4**	**1**	**2**	**3**	**7**	**3**	**3**
2	2	2	2	2	2	2	3	3	3	3
	5	5	5	5	5	5	5	7	7	7
		7	7	7	7	7	7	7	3	3
			4	4	4	4	4	4	4	3
				4	4	4	4	4	4	4
					1	1	1	1	1	1
						2	2	2	2	2

Working Set at t=9: {1,2,3,4,7}, size = 5

2. P2

1	**1**	**1**	**3**	**3**	**4**	**4**	**3**	**5**	**6**	**7**
1	1	1	1	1	1	1	3	3	3	3
	1	1	1	1	1	1	1	5	5	5
		1	1	1	1	1	1	1	6	6
			3	3	3	3	3	3	3	7
				3	3	3	3	3	3	3
					4	4	4	4	4	4
						4	4	4	4	4

Working Set at t=9: {3,4,5,6}, size = 4

3. P3

4	**9**	**3**	**2**	**9**	**8**	**7**	**7**	**7**	**1**	**2**
4	4	4	4	4	4	4	7	7	7	7
	9	9	9	9	9	9	9	7	7	7
		3	3	3	3	3	3	3	1	1
			2	2	2	2	2	2	2	2
				9	9	9	9	9	9	9
					8	8	8	8	8	8
						7	7	7	7	7

Working Set at t=9: {1,2,7,8,9}, size = 5

b) Trashing occurs when the total demand frames is bigger than total number of available frames. At t=9 the total demand is 5+4+5 = 14 which is bigger than the total number of available frames which is 10. 14 > 10 results in trashing.
c) Process 1 working Set at t=7: {1,2,3,4,5,7}, size = 6
Process 1 working Set at t=7: {1,3,4}, size = 3
Process 1 working Set at t=7: {2,3,7,8,9}, size = 5

Trashing occurs when the total demand frames is bigger than total number of available frames. To prevent trashing at t=7 the total available frames should be more than the total demand frames which is 6 + 3 + 5 = 14. So, there should be at least 14 available frames.

Problem 7

Program 1) For each page iteration of i in A[0..1023], it will incur one page fault for every time j is accessed. As j accesses the first int it will also incur a page fault. Thus, resulting in 1024 * 1024 = 1048576 total page faults.

Program 2) For each access of the first int, it will incur a page fault. However, the next time it is accessed the page is already present so that will not incur a page fault. Thus, resulting in 1024 total page faults.

Problem 8

a) A process cannot access memory that it does not own because the page tables have valid entries only for the memory the process owns. Every address a process uses is translated by its page table. If attempted to access memory that is on a page the process doesn't own, the hardware looks in the page table and finds that the page isn't valid and signals a segmentation fault that will terminate the attempt. This is done for the sake of memory protection which is necessary to protect the operating system from programs hosted on it, or the programs from each other.

b) The operating system could allow access by setting up valid bits for the entries for the physical pages used by other processes in its page table or providing a system call for reading/writing from other processes.

References

1. courses.cs.washington.edu/courses/cse451/.