SYSC4001 Assignment 3

Part I - Concepts

Jason Van Kerkhoven

100974276

Brydon Gibson

100975274

December 9th, 2016

Van Kerkhoven (100974276)
Gibson (100975274)

1a

Memory Slot	Space in Slot (kB)
1	102k
2	205k
3	43k
4	180k
5	70k
6	125k
7	91k
8	150k

Job Number	Memory Slot
1	2
2	4
3	Does not get a spot
4	8

Note that slot 2 is now 83kB.

1b

Job Number	Memory Slot
1	6
2	8
3	2
4	7

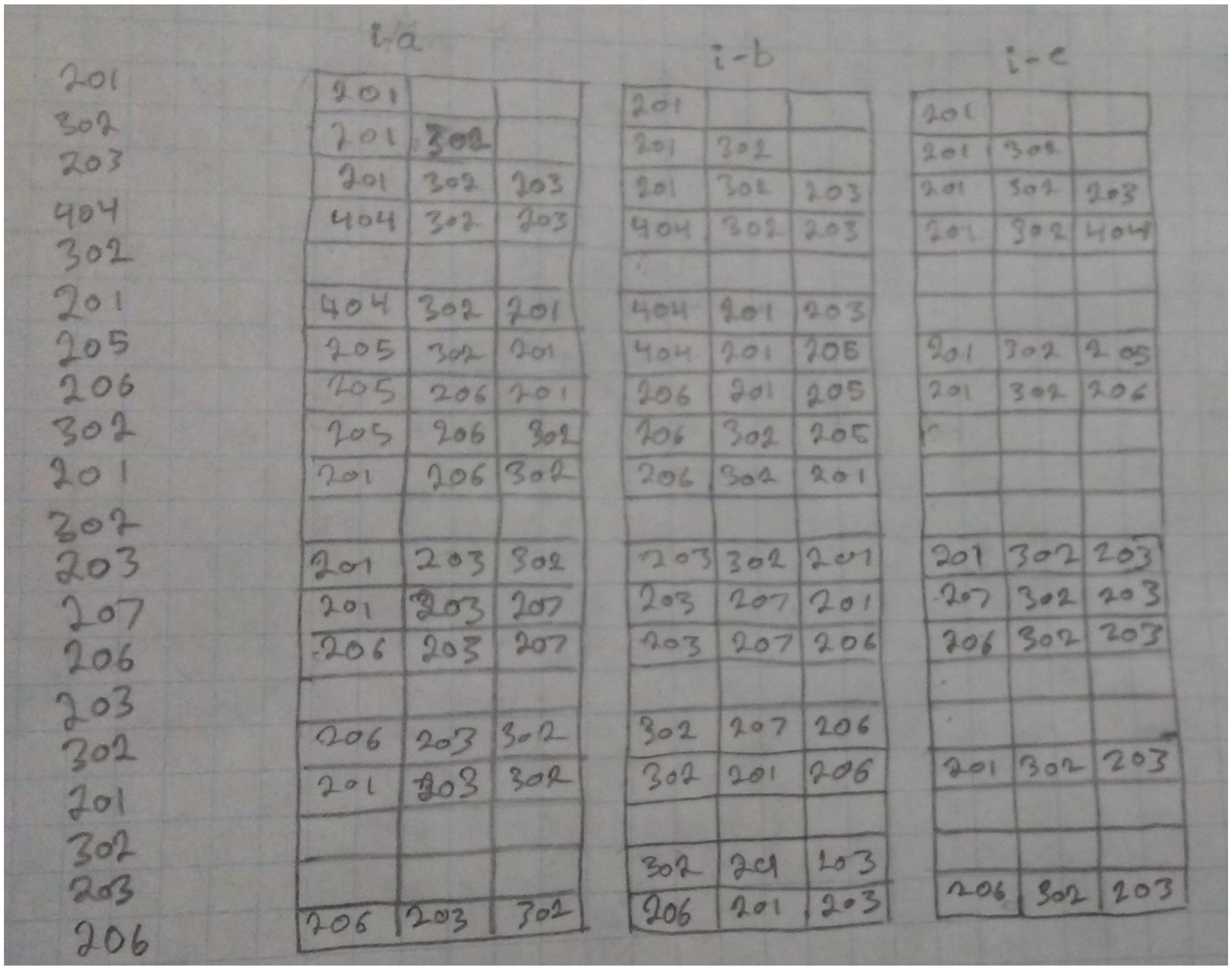
1c

Job Number	Memory Slot
1	2
2	4
3	Does not get a spot
4	8

Note the slot 2 is now 83kB, slot 4 is now 75kB.

2i

a) 15 page faults
b) 16 page faults
c) 11 page faults



	i-a			i-b			i-c		
201	201			201			201		
302	201	302		201	302		201	302	
203	201	302	203	201	302	203	201	302	203
404	404	302	203	404	302	203	201	302	404
302									
201	404	302	201	404	201	203			
205	205	302	201	404	201	205	201	302	205
206	205	206	201	206	201	205	201	302	206
302	205	206	302	206	302	205			
201	201	206	302	206	302	201			
302									
203	201	203	302	203	302	201	201	302	203
207	201	203	207	203	207	201	207	302	203
206	206	203	207	203	207	206	206	302	203
203									
302	206	203	302	302	207	206			
201	201	203	302	302	201	206	201	302	203
302									
203				302	201	203			
206	206	203	302	206	201	203	206	302	203

2ii

a) 8 page faults.

	FRAME 1	**FRAME 2**	**FRAME 3**	**FRAME 4**	**FRAME 5**
201	201				
302	201	302			
203	201	302	203		
404	201	302	203	404	
302					
201					
205	201	302	203	404	205
206	201	302	206	404	205
302					
201					
302					
203	201	302	206	203	205
207	201	302	206	203	207
206					
203					
302					
201					
302					
203					
206					

b) 10 page faults.

	FRAME 1	FRAME 2	FRAME 3	FRAME 4	FRAME 5
201	201				
302	201	302			
203	201	302	203		
404	201	302	203	404	
302					
201					
205	201	302	203	404	205
206	206	302	203	404	205
302					
201	206	201	203	404	205
302	206	201	302	404	205
203	206	201	302	203	205
207	206	201	302	203	207
206					
203					
302					
201					
302					
203					
206					

c) 7 page faults.

	FRAME 1	**FRAME 2**	**FRAME 3**	**FRAME 4**	**FRAME 5**
201	201				
302	201	302			
203	201	302	203		
404	201	302	203	404	
302					
201					
205	201	302	203	404	205
206	201	302	203	206	205
302					
201					
302					
203					
207	201	302	203	206	207
206					
203					
302					
201					
302					
203					
206					

3a
A paged memory reference would take 500 *ns*. This is because two different memory reads must be done, each taking 250 *ns*. The first read is to get the page table, while the second is to actually read the contents of memory.

3b
The effective memory access time is 330 *ns*. This is because the effective value is the approximate weighted average of purely the overhead time (which occurs 80% of the time), and a full memory access in addition to the overhead time (occurs 20% of the time). The weighted average of this is then added to the memory read time to determine the effective memory read access.

$$t_{eff} = t_{read} + \sum w_i t_i$$
$$t_{eff} = 250ns + (0.2)(250ns + 30ns) + (0.8)(30ns)$$
$$t_{eff} = 250ns + 80ns$$
$$t_{eff} = 330ns$$

3c
Adding a TLB would improve the overall performance, as it can quickly and easily access the page table. However, this is contingent with the TLB hitting the correct page relatively frequently. If the TLB misses frequently, the overhead associated with accessing it causes the average page access time to become slower. Therefore, the average access time after adding a TLB is inversely proportional to the frequency of hits in the TLB.

4a

Page (9 bits long)	**Offset (11 bits wide)**
0 0000 0000	000 0000 0000

$$page\ bits = log_2\left(\frac{2\ MB}{2\ kB}\right) = log_2\left(\frac{1048576\ B}{2 \cdot 1024\ B}\right) = log_2(512) = 9\ b$$
$$offset\ bits = log_2(2\ kB) = log_2(2048\ B) = 11\ b$$

4b
The page tables needs 9 bits to represent page number, and 9 bits to represent the contents at that entry.
$page\ bits = log_2(Amount\ of\ Pages) = log_2(512) = 9\ b$

4c
If half of the physical memory is removed, but the size of the page table remains the same, the page table is cut in half (256 entries, 8 bits). If there are any frames below the 255th frame, defragmentation will need to occur.

5a

A race condition is when two or more processes/threads running concurrently will make use of shared data in an asynchronous manner, that is, there is no guarantee which process/thread will use the data at any given time. The output of the processes/threads differs depending on which process/thread accessed the data first, as there is no guarantee the data was not modified by said process/thread. An example of this problem can be seen where there are two threads running, sharing an integer *x*, initialized at 0. Thread [*A*] will add 2 to the value and print, whereas thread [*B*] will square to the value and print. If thread [*A*] runs before [*B*], the prints will be "2, 4". However, if [*B*] runs before [*A*], then the prints will be "0, 2".

5b

By blocking interrupts in order to prevent race conditions, there is no way to handle time-critical events if they occur during a critical section of code. If, for instance, you are reading a very large shared array from memory and block interrupts, you are effectively locking the CPU for possible a large duration of time. If the user types a key or the screen needs to be updated, there is no way for the respective response to run, as there is no way to interrupt the process currently reading the very large array. This can, obviously, cause large latencies, and unwanted delay.

6

A semaphore is a hardware supported solution to protecting shared data used in critical sections of code. A semaphore has two basic operations, both supported by the processor architecture, *wait()* and *signal()*. Semaphore operations *wait()* and *signal()* are both designed to be atomic, that is, they cannot be interrupted half-way through completion, and must be run fully before interrupts are allowed. This way, it is guaranteed that only one process/thread interacts with the semaphore at a time. Processes/threads can then use this semaphore to reliably *wait()* for the semaphore to be unlocked (binary/mutex semaphore) or set to the correct value (non-binary/counting semaphore). The semaphore then becomes locked, preventing any other process from accessing it until the program which obtained the semaphore lock releases it using the *signal()* function. Processes/threads waiting for a semaphore lock are placed in a waiting block, and will stay there indefinitely until they obtain the lock (or on occasion, timeout and give up on critical section access).

7

i. *open(mode)* system call is used, given the mode as append-only
ii. The system directory is searched through in order to find the file that needs to be opened.
iii. The files permissions are checked against the user which called the *open()* system call (ie to check if they have permission to access the file in an append-only mode)
iv. The file is added to the open-file table in the system (to avoid having to search for it later)
v. The *open()* system call returns a pointer to the files location in the open-file table that the process can then use for all later interactions with that file.
vi. Once done with the file, the process should use the *close()* system call to remove the file from the open-file table.

8a

$$max = 12direct + 1singleIndirect + 1doubleIndirect + 1tripleIndirect$$
$$max = (12)(8kB) + \left(\frac{8kB}{4B}\right)^1 (8kB) + \left(\frac{8kB}{4B}\right)^2 (8kB) + \left(\frac{8kB}{4B}\right)^3 (8kB)$$
$$max = 98304B + 16777216B + 34359738368B + 70368744177664B$$
$$max = 70403120791552\ B$$
$$max = 65568.02\ GB$$
$$max = 64.03\ TB$$

8b

The only way to increase your maximum file size (without altering the file systems block quantities) is to increase the size of a block. For instance, if you increase the block size from 8 *kB* to 16 *kB*, you increase your maximum file size by approximately 176.5%.

$$max = 12direct + 1singleIndirect + 1doubleIndirect + 1tripleIndirect$$
$$max = (12)(16kB) + \left(\frac{16kB}{4B}\right)^1 (16kB) + \left(\frac{16kB}{4B}\right)^2 (16kB) + \left(\frac{16kB}{4B}\right)^3 (16kB)$$
$$max = 1126174852055040\ B$$
$$max = 1024.25\ TB$$