# Question 1

## 1a)

i)

Shortest Remaining Time (best for minimal turnaround) (note the answer is not SJF since the question asks for preemptive)

ii)

FCFS

Order: R1, R2, R3

Per-request turnaround: R1: 100, R2: 101, R3: 111

Average turnaround: (100 + 101 + 111) / 3 = 104

Throughput: 3 / 111 ~= 0.027 requests / time

SJF

Order: R2, R3, R1

Per-request turnaround: R2: 1, R2: 11, R3: 111

Average turnaround: (1 + 11 + 111) / 3 = 41

Throughput: 3 / 111 ~= 0.027 requests / time

iii)

FCFS is not predictable – the more jobs there are on the system, the longer the turnaround time for the user’s job.

RR is not predictable – each job gets a fair share of the CPU, but the more jobs there are, the less frequent these shares are.

Assuming the user can create a process with a priority higher than anything else running on the system at the time, simple priority scheduling would ensure their process will be run to completion as soon as possible, i.e. turnaround time = job size. Otherwise there could be other, higher-priority requests running on the system during the execution of the user’s job => not predictable.

iv)

No, it would be extremely difficult to implement (the two processes don’t share CPU nor memory), and VM2 would have nothing to spend its CPU time on (it cannot donate its CPU time to VM1).

>> It is possible if you mess with the hypervisor scheduler .

>> The OS should be responsible for controlling priority of processes, as these VM’s are on two separate machines, the two operating systems would have to be modified to work together, this creates a lot of overhead work, that in practice may not be feasible.

## 1b)

i) Note that our locking is out of order.

P1:

lock(R1)

lock(R2)

...

P2:

lock(R2)

lock(R1)

...

ii) Note this time we do our locking in-order.

P1:

lock(R1)

lock(R2)

...

P2:

lock(R1)

lock(R2)

...

## 1c)

i)

RAID-0, since data can be written in parallel to multiple disks, so given N disks, we can expect a speed-up factor of N.

RAID-1 needs to update mirrors in parallel which makes it slower.

ii)

Under RAID-0: NG gigabytes

Under RAID-1: NG/2 gigabytes

iii)

She just wants to get both

[RAID-0+1](https://en.wikipedia.org/wiki/Nested_RAID_levels" \l "RAID_01_(RAID_0+1)) provides striping as well as mirroring:

|  |  |  |  |
| --- | --- | --- | --- |
| Disk 1 | Disk 2 | Disk 3 | Disk 4 |
| A1 | A2 | A1 | A2 |
| A3 | A4 | A3 | A4 |
| A5 | A6 | A5 | A6 |
| … | | | |

Writing and reading can be done in parallel across two disks, providing ~2x faster speed. The data is also mirrored so data is always recoverable if a single disk is lost (in some cases, e.g. disk 3 and 4 failing, the data is still intact on disks 1 and 2).

**Alternative Hybrid Raid 1-0:**

|  |  |  |  |
| --- | --- | --- | --- |
| Disk 1 | Disk 2 | Disk 3 | Disk 4 |
| 1 | 1 | 2 | 2 |
| 3 | 3 | 4 | 4 |
| 5 | 5 | 6 | 6 |
| ... | | | |

# Question 2

## 2a)

1. Page table entry – which physical frame the page currently resides in
2. Dirty bit – whether or not the page has been modified in memory without being saved to disk
3. Valid bit – whether or not the page is actually present in memory

## 2b)

10 bits used for the byte offset – pages contain 2^10 = 1024 addresses. Assuming byte-addressable memory, one page is 1024 bytes = 1 KiB.

## 2c)

2^16 addresses; assuming byte-addressable memory again, this means the addressable memory is 2^16 bytes = 64 KiB.

64MiB = (2^16 possible frames \* 2^10 size of each frame)

<https://cs.stackexchange.com/questions/56271/finding-maximum-virtual-and-physical-memory> im looking at this, for reference, but not 100% sure if its correct.

## 2d)

|  |  |  |  |
| --- | --- | --- | --- |
|  | VA | Page no. | Physical address |
| 0xF1F | 0000 11 | 11 0001 1111 | 3 | 0x7C00 + 0x031F = 0x7F1F |
| 0xC000 | 1100 00 | 00 0000 0000 | 48 | N/A |
| 0x403 | 0000 01 | 00 0000 0011 | 1 | 0x4C00 + 0x0003 = 0x4C03 |
| 0x0 | 0000 00 | 00 0000 0000 | 0 | 0x1800 + 0x0 = 0x1800 |

## 2e)

Assuming the memory is byte-addressable.

Use the least significant bit to represent valid, and second least significant bit to represent dirty:

0x1803

0x4C03

0x5400

0x7C03

0x1402

**Alternatively:**

Using two (hex) bytes (or one decimal):

* 6 most significant bits are frame number (discard unused 10 bits)
* 2nd least is dirty bit
* least is valid bit

0x1B = 0x18 + 11

0x4F = 0x4C + 11

0x54 = 0x54 + 00

0x7F = 0x7C + 11

0x16 = 0x14 + 10

## 2f)

IDK for sure, but this makes sense to me.

* Use Bank switching <https://en.wikipedia.org/wiki/Bank_switching>
* This basically means that we divide the physical memory into banks of 64KB
* We then have a page table in each bank (this means a process can have multiple page tables)
* Whenever we switch bank we change the base offset of our address relative to the bank index. For example all bank 0 addresses start from 0x0, when we switch to bank 1 the addresses now start from 0x(whatever 64KB is in hex).
* We can identify what bank an address is by doing (address/64KB)
* A process can now use a large amount of memory under the proposed scheme by adding page entries to the current banks page table. If the page table becomes full, then it switches to the next bank, and continues to add the rest of the pages to this table.

Sum1 elses answers

* Use larger PTE + page switching – allows storage of larger physical addresses; however, with this change, there are more frames than there are pages, so the process still only has a 64KBis virtual address space – accessing more would imply switching out pages in the table. Using larger PTE would also mean that larger memory will need to be allocated for the page tables.
* Address larger parts of memory – e.g. word-addressable memory without modifying the PTE size would support physical memory of up to 128 KiB; would be a bit more difficult to port existing code

Alternative -

Use frame swapping so that when the process asks for more memory than we have in free frames we would swap frames into disk memory thus freeing up frames.

I thought this question was talking about using (for example) a 2-level page table, so have bits for an “inner” and “outer” page# indexes in the “virtual address” specified... is this not right? *Inner page tables are good for when the virtual address space is large since we don’t need to have the page mappings for the whole virtual address space in memory at once, but it still only has the same amount of addresses i.e. doesn’t increase the size of the physical memory supported.* Alright, thank you :)