/

P4Q7
SMH
P11Q7

Operating System Functions 2000

Time(ms)	1	2	3	4	5	6	7	8	9	10
A	A	A	-	-	|	A	A	-	-	|
B	-	-	B	B	B	-	-	|	B	-
C	-	-	-	-	-	-	-	C	|	-

~~At 9ms, task B is scheduled ahead of task C, and hence task C misses its deadline. EDF would schedule C here ahead of B since although B is more frequent, C is closer to its deadline and B can afford to wait.~~

Paper 4 Question 7 (20 Marks)
Paper 11 Question 7 (20 Marks)

Why is it important for an operating system to schedule disk requests? [2 marks]

Briefly describe each of the SSTF, SCAN and C-SCAN disk scheduling algorithms. Which problem with SSTF does SCAN seek to overcome? Which problem with SCAN does C-SCAN seek to overcome? [5 marks]

Consider a winchester-style hard disk with 100 cylinders, 4 double-sided platters and 25 sectors per track. The following is the (time-ordered) sequence of requests for disk sectors:

{ 3518, 1846, 8924, 6672, 1590, 4126, 107, 9750, 158, 6621, 446, 11 }.

The disk arm is currently at cylinder 10, moving towards 100. For each of SSTF, SCAN and C-SCAN, give the order in which the above requests would be serviced. [3 marks]

Which factors do the above disk arm scheduling algorithms ignore? How could these be taken into account? [2 marks]

Discuss ways in which an operating system can construct logical volumes which are (a) more reliable and (b) higher performance than the underlying hardware. [4 marks]

Should the scheduling of requests to such a logical volume be carried out above or below the abstraction boundary? Justify your answer. [4 marks]

Answers to Paper 4 Question 7/Paper 11 Question 7

Importance of Disk Scheduling

Disks are not random access devices: moving the arm costs a considerable amount of time. There are also costs involved with settling on a given track or switching heads, and [often hidden] issues with caching. For these and other reasons, the OS can considerably reduce the average service time for disk requests by issuing these requests to the hardware in an appropriate way.

Scheduling Algorithms

1 mark per algorithm, 1 mark each for other bits

SSTF is shortest-seek time first; i.e. it always seeks to the closest possible cylinder.

SCAN services requests in order in one direction, and then turns around and services requests in the other direction. I.e. it "scans" from 0 up to 100, and the down to 0, and so on.

C-SCAN is the circular version of scan. As with SCAN is "scans" from 0 up to 100, but unlike SCAN it them simply repeats this action (i.e. it only every scans in one direction).

SSTF is subject to starvation — if new requests keep on arriving for sectors close to the disk head, then outlying cylinders may never be serviced. Even if starvation does not occur, the variance in service time can be high.

In SCAN the worst case service time is two passes of the disk; e.g. consider a request for cylinder 0 just after SCAN has started upward. With C-SCAN the worst case is bounded by a single pass of the disk.

Scheduling Example

1 mark per algorithm.

Since there are 25 sectors per track and 8 tracks per cylinder, we have 100 sectors per cylinder. Hence the sectors { 3518, 1846, 8924, 6672, 1590, 4126, 107, 9750, 158, 6622, 446, 11 }. map to the cylinders { 35, 18, 89, 66, 15, 41, 1, 97, 1, 66, 4, 0 }. Sorting these gives { 0, 1, 1, 4, 15, 18, 35, 41, 66, 66, 89, 97 }.

The disk head is currently at cylinder 10, and so SSTF will service 15, 18, 4, 1, 1, 0, 35, 41, 66, 66, 89, 97. The direction in which the arm is currently traveling does not matter.

For SCAN and C-SCAN the direction does matter. Since the disk head is moving towards 100, SCAN will service 15, 18, 35, 41, 66, 66, 89, 97, 4, 1, 1, 0.

C-SCAN will service 15, 18, 35, 41, 66, 66, 89, 97, 0, 1, 1, 4.

Issues Ignored

The above algorithms use a simplified model of the disk (which is pretty justifiable given underlying complexity / ways in which modern disks hide information from us). Main issues ignored are (a) all based on cylinders and (b) rotational latency. The former means that two 'adjacent' sectors may in fact need a head switch ($\sim$ 2ms) and so a 'more distant' sector might be a better option. For example, the set of requests in the question has two pairs in the same cylinder which are angularly adjacent but on different surfaces. Although all the algorithms schedule these consecutively, we'd be likely to miss them on a real disk and hence take almost a full rotational delay.

Also means that issues of fairness and starvation are not properly addressed — all the above suffer from starvation (or large variance) when an infinite (large) number of requests are queued for the same track. We can solve this with e.g. F-SCAN or N-step SCAN.

Second issue is that a rotation may be much bigger than a cylinder switch (e.g. 8.3ms worst case rotation on a 7200 rpm drive, but moving cylinder might be e.g. 3ms). This also skews the "distance" metric used in the algorithms, but is more difficult to solve in software since the we need very precise information about what's going on. Possibly to solve if we push stuff into the firmware.

Constructing Logical Volumes

Roughly two marks for mirroring, two for striping; single answer with hybrid (e.g. RAID5) also acceptable

The OS can construct a logical volume which is more reliable by *mirroring* data on two or more disks; that is, by writing each piece of data n times, once per disk. This way if $n-1$ disks fail, the data is still recoverable. Mirroring may be done at a byte, block or larger granularity. It is an expensive technique since it requires n times the storage space. [note that it also allows for higher performance reads since any disk may be chosen — see next answer]

The OS can construct higher performance logical volumes by *striping* data across multiple disks; that is, by writing successive portions ("stripes") of data onto disks $1, 2, \ldots n$. This means that reads and writes of $k \times S$ bytes can be done in parallel, where S is the stripe size. In hardware solutions S can be a byte or a block; in software, S will usually by a [physical or logical] block.

Hybrid schemes are also possible which give us less reliability than mirroring (although increased over basic hardware) at a lower [space] cost, along with increase performance for most operations. The most common of these is RAID5 in which an additional parity/ECC stripe is written for each k stripes. For example, with three disks one could write the first block on disk 1, the second on disk 2, and the XOR of these onto disk three. The disk chosen for parity can rotate in a round-robin fashion. Other codes are possible.

Scheduling Logical Volumes

The main answer to this is "it depends". In general it is probably simpler to *not* schedule access to the logical volume but rather simply insert the sub-requests into the scheduling pool for the underlying devices. This is since it is likely that (a) other volumes will be present on the same underlying devices and requests to these should be scheduled fairly/efficiently, (b) there may not be any correlation between the actual sectors required on the various underlying disks.

However for e.g. reading from a mirrored set, the logical volume layer really only needs to read a single copy ⇒ would like to "peek" the data structures and state of the underlying disks. Similarly when multiple disks are being accessed (for read or write), the higher level request is not likely to be satisfied until all (or at least a subset) of these operations have completed; hence it is poor for performance if a request to an individual disk takes considerably longer than those to others.

Probably the best solution is to augment the per-disk scheduling data structures with hints from the logical volume layer such as "perform this read iff a read on another disk has not yet occurred", or "prioritize this write since it is needed for a higher-level write to complete". Alternatively one could construct software stripe/mirror sets so that they used entire identical disks (a la h/w solutions).