University of Newcastle

School of Electrical Engineering and Computing

**COMP2240 - Operating Systems**

**Workshop 8 Solution**

**Topics: Memory Management & Disk and I/O Scheduling**

|  |  |
| --- | --- |
| **1.** | Consider the following page-reference string: *a,b,d,c,b,e,d,b,d,b,a,c,b,c,a,c,f,a,f,d*. Assume that there are 3 frames available and that they are all initially empty. Complete a figure showing the frame allocation for each of the following page replacement policies:   1. First-in-first-out 2. Optimal 3. Least recently used   Then, find the relative performance of each policy with respect to page faults.  **Answer**:   1. FIFO page-replacement algorithm     Number of Faults = 13 No. of Hits = 7   1. Optimal page-replacement algorithm     Number of Faults = 9 No. of Hits = 11   1. LRU page-replacement algorithm     Number of Faults = 10 No. of Hits = 10  The above data shows that FIFO has highest number of page faults and Optimal has the lowest. It has generally been found that this is the trend for most reference strings. Page faults in FIFO occur more frequently since the pages that are in the memory for the longest time are replaced without regarding the fact that they may have been used quite frequently. Since this problem is overridden in LRU, the latter shows a better performance. |
| **2.** | A process contains eight virtual pages on disk and is assigned a fixed allocation of four page frames in main memory. The following page trace occurs:  1, 0, 2, 2, 1, 7, 6, 7, 0, 1, 2, 0, 3, 0, 4, 5, 1, 5, 2, 4, 5, 6, 7, 6, 7, 2, 4, 2, 7, 3, 3, 2, 3   1. Show the successive pages residing in the four frames using the LRU replacement policy. Compute the hit ratio in main memory. Assume that the frames are initially empty. 2. Repeat part (a) for the FIFO replacement policy. 3. Compare the two hit ratios and comment on the effectiveness of using FIFO to approximate LRU with respect to this particular trace.   **Answer:**   1. LRU:     Hit ratio = 16/33   1. FIFO:     Hit ratio = 16/33   1. These two policies are equally effective for this particular page trace. |
| **3.** | Consider a disk drive with 4,000 cylinders, numbered from 0 to 1,999. The request queue has the following composition:  1045 750 932 878 1365 1787 1245 664 1678 1897  If the current position is 1167 and the previous request was served at 1250, compute the total distance (in cylinders) that the disk arm would move for each of the following algorithms: FIFO, SSTF, SCAN, C-SCAN, LOOK and C-LOOK scheduling.  **Answer:**  **FIFO**  Current head position = 1167  Cylinders served consecutively are:  1045, 750, 932, 878, 1365, 1787, 1245, 664, 1678, 1897  Total distance = 122 + 295 + 182 + 54 + 487 + 422 + 542 + 581 + 1014 + 219  = 3918 cylinders  **SSTF**  Current head position = 1167  Cylinders served consecutively are:  1245, 1365, 1678, 1787, 1897, 1045, 932, 878, 750, 664  Total distance = 78 + 120+313+ 109+ 110 + 852 + 113+ 54+ 128+ 86  = 1963 cylinders  **SCAN**  Current head position = 1167  Previous head position = 1250  So, movement is from right to left.  Cylinders served consecutively are:  1045, 932, 878, 750, 664, <0>, 1245, 1365, 1678, 1787, 1897  Total distance = 122 + 113 + 54 + 128 + 86 + 664 + 1245 + 120 + 313 + 109 + 110  = 3064 cylinders  **C-SCAN**  Current head position = 1167  Previous head position = 1250  So, movement is from right to left.  Cylinders served consecutively are:  1045, 932, 878, 750, 664, <0>, <1999>, 1897, 1787, 1678, 1365, 1245  Total distance = 122 + 113 + 54 + 128 + 86 + 664 + 1999+ 102+110+ 109+ 313+120  = 3920 cylinders  **LOOK**  Current head position = 1167  Previous head position = 1250  So, movement is from right to left.  Cylinders served consecutively are:  1045, 932, 878, 750, 664, 1245, 1365, 1678, 1787, 1897  Total distance = 122 + 113 + 54 + 128 + 86 + 581 + 120 + 313 + 109 + 110  = 1763 cylinders  **C-LOOK**  Current head position = 1167  Previous head position = 1250  So, movement is from right to left.  Cylinders served consecutively are:  1045, 932, 878, 750, 664, 1897, 1787, 1678, 1365, 1245  Total distance = 122 + 113 + 54 + 128 + 86 + 1233 + 110 + 109 + 313 + 120  = 2388 cylinders |
| **4.** | Calculate how much disk space (in sectors, tracks, and surfaces) will be required to store 300,000 120-byte logical records if the disk is fixed sector with 512 bytes/sector, with 96 sectors/track, 110 tracks per surface, and 8 usable surfaces. Ignore any file header record(s) and track indexes, and assume that records cannot span two sectors.  **Answer:**  Each sector can hold 4 logical records.  The required number of sectors is 300,000/4 = 75,000 sectors.  This requires 75,000/96 = 782 tracks, which in turn requires 782/110 = 8 surfaces. |
| **5.** | A disk pack has the following specifications: it comprises 25 double sided disks; each surface of a disk has 480 tracks and a track has 20 blocks in it. Each block is of 2048 bytes, with an inter-block gap of 64 bytes.  Compute the total capacity of a track, the useful capacity of a track (excluding inter-block gap), the total capacity and useful capacity of a cylinder, the total capacity and useful capacity of the disk page, and the percentage of space wasted.  **Answer:**  Given:  Block size B = 2048 bytes  Inter-block gap size G = 64 bytes  Number of blocks per track n = 20  Number of tracks per surface t = 480  Total number of double-sided disks = 25  Total capacity of a track = n × (B + G)  = 20 × (2048 + 64)  = 42240 bytes  = 41.25 KB  Useful capacity of a track = n × B  = 20 × 2048  = 40 KB  Total capacity of a cylinder = no. of sides in a cylinder × total capacity of a track  = 2 × 25 × 41.25  = 2062.5KB  Useful capacity of a cylinder = no. of sides in a cylinder × useful capacity of a track  = 2 × 25 × 40 = 2000KB  Total capacity of the disk pack = no. of cylinders × total capacity of a cylinder  = 480 × 2062.5  = 966.8 MB  Useful capacity of the disk pack = no. of cylinders × useful capacity of a cylinder  = 480 × 2000  = 937.5MB  Percentage of space wasted = wasted space/total space × 100  = (Total capacity – Useful capacity)/Total capacity × 100  = (966.8 – 937.50)/966.8 × 100  = 3.03% |

**Supplementary problems:**

|  |  |
| --- | --- |
| **S1.** | Disk-scheduling policies SSTF, SCAN, C-SCAN, LOOK and C-LOOK are not truly fair, that is, starvation can occur. Only FIFO/FCFS is fair.   * 1. Explain why this unfairness assertion is true.   2. Describe a scheme to ensure fairness (other than FIFO/FCFS).   3. Explain why fairness is an important goal in a time-sharing system.   **Answer:**   * 1. New requests for the track over which the head currently resides can theoretically arrive as quickly as these requests are being serviced. Therefore, earlier requests may never be satisfied, as no movement toward them occurs.   2. All requests older than some predetermined age could be ‘forced’ to the top of the queue and an associated bit for each could be set to indicate that no new request could be moved ahead of these requests. For SSTF, the rest of the queue would have to be reorganised with respect to the last of these ‘old’ requests.   3. To prevent unusually long response times. |
| **S2.** | The UNIX kernel will dynamically grow a process’s stack in virtual memory as needed, but it will never try to shrink it. Consider the case in which a program calls a C subroutine that allocates a local array on the stack that consumes 10 K. The kernel will expand the stack segment to accommodate it. When the subroutine returns, the stack pointer is adjusted and this space could be released by the kernel, but it is not released. Explain why it would be possible to shrink the stack at this point and why the UNIX kernel does not shrink it.  **Answer:**  It is possible to shrink a process's stack by deallocating the unused pages. By convention, the contents of memory beyond the current top of the stack are undefined. On almost all architectures, the current top of stack pointer is kept in a well-defined register. Therefore, the kernel can read its contents and deallocate any unused pages as needed. The reason that this is not done is that little is gained by the effort. If the user program will repeatedly call subroutines that need additional space for local variables (a very likely case), then much time will be wasted deallocating stack space in between calls and then reallocating it later on. If the subroutine called is only used once during the life of the program and no other subroutine will ever be called that needs the stack space, then eventually the kernel will page out the unused portion of the space if it needs the memory for other purposes. In either case, the extra logic needed to recognize the case where a stack could be shrunk is unwarranted. Source: [SCHI94]. |
| **S3.** | Windows uses 2-MB large pages because it improves the effectiveness of the TLB, which can have a profound impact on performance. Why is this? Why are 2-MB large pages not used all the time?  **Answer:**  The hit rate on the page table entries cached in the TLB has a big impact on system performance. The operation to walk the page tables and find a missing entry is very expensive. Since the TLB has only a limited number of entries, using 2-MB large pages greatly increases how many virtual addresses can be mapped by the TLB at a time. Large pages will waste large amounts of memory because of the unused space at the end of the final page within a region of the file. So they are effective only when used with very large regions. But even so, they increase memory pressure on memory because a large page is likely to have large amounts of data that is not currently being accessed, and would have been paged out to disk if using 4-KB pages. |
| **S4.** | A certain editor has 100 KB of program text, 30 KB of initialized data, and 50 KB of BSS. The initial stack is 10 KB. Suppose that three copies of this editor are started simultaneously. How much physical memory is needed (a) if shared text is used, and (b) if it is not?  **Answer:**  With shared text, 100 KB is needed for the text. Each of the three processes needs 80 KB for its data segment and 10 KB for its stack, so the total memory needed is 370 KB. Without shared text, each program needs 190 KB, so three of them need a total of 570 KB. |
| **S5.** | Consider a disk drive that has *N* tracks numbered from 0 to (*N* - 1). Assume that requested sectors are distributed randomly and evenly across the tracks of the disk. We want to determine the average number of tracks traversed by a Seek operation. The problem can be partitioned as follows:   1. First, calculate the probability of a Seek operation of length *j* when the head is currently positioned over track *t*. *Hint: You need to determine the total number of combinations, recognising that all track positions for the destination of the Seek are equally likely, as are all starting positions of the disk head prior to the Seek.* 2. Next, calculate the probability of a Seek of length *k*. *Hint: This involves the summing over all possible combinations of movements of k tracks.* 3. Now, calculate the average number of tracks traversed by a seek, using the formula for expected value:   E[x] = x Pr[x = i]   1. Finally, show that for large values of *N* the average number of tracks traversed by a Seek approaches . *(At least you know what answer you are trying to work toward!)*   **Answer:**  It will be useful to keep the following representation of the N tracks of a disk in mind:     1. Let’s use the notation Ps[j/t] = Pr[seek of length j where head is currently positioned over track t]. It is important to recognise that each of the N tracks is equally likely to be requested. Therefore the unconditional probability of selecting any particular track is . We can then state that:   Ps[j/t] = if t ≤ j-1 OR t ≥ N-j  Ps[j/t] = if j-1 < t < N-j  In the first case, the current track is so close to one end of the disk (track 0 or track N-1) that only one track is exactly j tracks away. In the second case, there are two tracks that are exactly j tracks away from track t, and therefore the probability of a seek of length j is the probability that either of these two tracks is selected, which is just .   1. Let Ps[K] = Pr[seek of length K, independent of current track position]. Then:   Ps[K] = x Pr[current track is track t]  =  from part (a) we know that Ps[K/t] takes on the value of for 2K of the tracks, and the value for (N-2K) of the tracks. So:  Ps[K] = x (2K x ) + x ([N-2K] x )  =   1. E[x] = x Pr[x = i]   E[K] = =  = -  = (N +1) -  =   1. It follows directly from (c) above. Given that 1 is insignificant against N2 for large values of N, the above formula approximates |