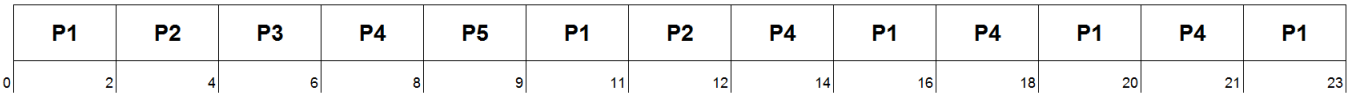
Operating Systems - Exercise 4

A Cache File System: Answers

Itai Tagar - 305392508 - itagar

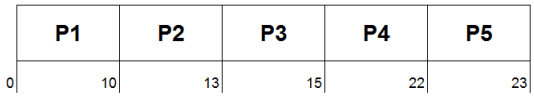
Part I

1. (a) **Round Robin (RR) with quantum = 2:**
   i. Gantt Chart:



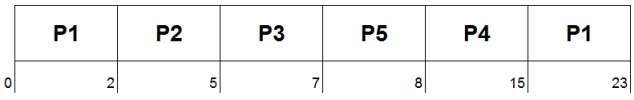
   i. Turnaround Time: $(23 + (12-2) + (6-4) + (21-4) + (9-7))/5 = \mathbf{11.4}$
   ii. Average Wait Time: $(13 + 8 + 0 + 10 + 1) = \mathbf{6.4}$
   - P1 Wait Time: $(9-2) + (14-11) + (18-16) + (21-20) = 13$
   - P2 Wait Time: $(12-4) = 8$
   - P3 Wait Time: 0
   - P4 Wait Time: $(6-4) + (12-8) + (16-14) + (20-18) = 10$
   - P5 Wait Time: $(8-7) = 1$

   (b) **First Come First Serve (FCFS):**
   i. Gantt Chart:



   ii. Turnaround Time: $(10 + (13-2) + (15-4) + (22-4) + (23-7))/5 = \mathbf{13.2}$
   iii. Average Wait Time: $(0 + (10-2) + (13-4) + (15-4) + (22-7)) = \mathbf{8.6}$

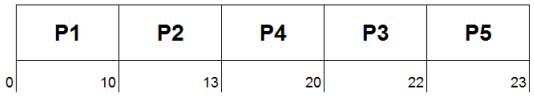
   (c) **Shortest Remaining Time First (SRTF):**
   i. Gantt Chart:



   ii. Turnaround Time: $(23 + (5-2) + (7-4) + (15-4) + (8-7))/5 = \mathbf{8.2}$
   iii. Average Wait Time: $((15-2) + 0 + (5-4) + (8-4) + 0) = \mathbf{3.6}$

(d) **Priority Scheduling:**

i. Gantt Chart:

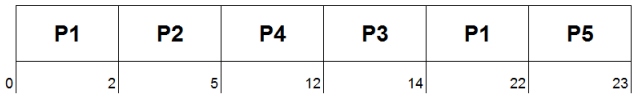


ii. Turnaround Time: $(10 + (13 - 2) + (22 - 4) + (20 - 4) + (23 - 7))/5 = \mathbf{14.2}$

iii. Average Wait Time: $(0 + (10 - 2) + (20 - 4) + (13 - 4) + (22 - 7)) = \mathbf{9.6}$

(e) **Priority Scheduling with preemption:**

i. Gantt Chart:



ii. Turnaround Time: $(22 + (5 - 2) + (14 - 4) + (12 - 4) + (23 - 7))/5 = \mathbf{11.8}$

iii. Average Wait Time: $((14 - 2) + 0 + (12 - 4) + (5 - 4) + (22 - 7)) = \mathbf{7.2}$

2. Consider the case of *Cache Miss*. In such case there will be an access to the main memory, there we will see that the required block is not in our cache and we will have to access the disk to get to our block and then store it in our cache. Note that in this case we will have one access to the main memory and one access to the disk, thus it takes slightly more time then accessing the disk directly.

3. It will be much harder to manage the different algorithms that we used in our *Blocks-Cache* for swapping *pages* because managing the *Blocks-Cache* is done by the operating system and there is plenty of data it can hold and use in order to manage the sophisticated algorithms (such as reference counters and order of use). Swapping pages is done by the hardware itself (*Memory Management Unit*) and the hardware is unable to hold all of the data that the operating system is using in our algorithms. The hardware can use a single bit (The R bit as we saw in class) which indicates a reference and by that it can manage the *Clock-Algorithm* we saw in class.

4.

- **LRU is better than LFU:**
  When each sequence of blocks is read for some interval thus the most recently used block will probably be requested again.
  Consider a cache of size 2 and consider reading a single file of size of 3 blocks. Reading the file's blocks is in the following order:
  $$1, 1, 1, 2, 3, 2, 3, 2, 3, 2, 3$$
  Using LRU we will read the first block from the file into the cache (in the first time there will be a *miss* and in the other two there will be a *hit*). Then we will read the second block to the cache. Now the cache is full and we are trying to read the third block, according to LRU we will remove block number 1 from the cache and replace it with block number 3. From now on all of our reads will be *hits*. So overall we will have 3 *miss* and 8 *hits*.
  Using LFU will do the same until the cache is full. When the cache is full LFU will see that block number 1 has more references than block number 2 thus it will remove it and replace it with block number 3. The same thing will happen to block number 3 and so on. So overall we will have 7 *miss* and 2 *hits*.
- **LFU is better than LRU:**
  When we have some block that is used interchangebly between other reads but it is often requested all along the enture read process.
  Consider a cache of size 2 and consider reading a single file of size of 7 blocks. Reading the file's blocks is in the following order:
  $$1, 1, 1, 2, 3, 1, 4, 5, 1, 6, 7, 1, 1, 1$$
  Using LRU we will read the first block from the file into the cache (in the first time there will be a *miss* and in the other two there will be a *hit*). Then we will read the second block to the cache. Now the cache is full and we are trying to read the third block, according to LRU we will remove block number 1 from the cache and replace it with block number 3. Next we are back to block number 1 but we will have a *miss* and we will replace it with block number 2. Same for reading blocks 4 and 5, we will end up with a full cache that contains $4, 5$ and we will return to blcok 1 and again we will have a *miss*. So overall we will have 10 *miss* and 4 *hits*.
  Using LFU will do the same until the cache is full. When the cache is full LFU will see that blocks 1 has more references than block number 2 thus it will remove it and replace it with block number 3. When block 1 request will come up again we will have a *hit*. The same will be for all of our next reads and we will end up that block number 1 will stay in the cache all the time. So overall we will have 6 *miss* and 7 *hits*.
- **LFU and LRU are both poor:**
  When we read a file sequentially for several times.
  Consider a cache of size 4 and consider reading a single file of size of 5 blocks. Reading the file's blocks is in the following order:
  $$1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5$$
  Using LRU we will read blocks $1 - 4$ and then the cache will be full. Now when we try to read block number 5 according to LRU we will remove block number 1 from the cache and replace it with block number 5. For block number 1 again we will have a *miss* we will have to remove block number 2 from the cache. This cyclic replacement is going on for all the rest of the read requests and **all** of the requests will be *miss*. So overall we will have 15 *miss* and 0 *hits*.
  Using LFU is considered to do the same as LRU because all the references are equal (equals 1) before we are trying to remove a block (at any time) and so in the worst case we will choose to remove the least recently used block. So we are having the same performance as LRU.
  A better algorithm for this scenario will be MRU *for example.*

5. The *new section* controls the incrementing reference counts to try and solve the following problem: In LFU there is an increment of the reference counter each time a block is referenced, so there can be a situation where several blocks are relatively infrequently referenced overall, and yet when they are referenced, due to locality there are short intervals of repeated re-references, thus building up high reference counts. After such an interval is over, the high reference count is misleading because it is due to locality, and cannot be used to estimate the probability that such a block will be re-referenced following the end of this interval.

Part II

1. First note that accessing byte 40,000 requires using *single indirect* because using *direct pointers* can hold only for $(10 \times 2,048) = 20,480$ bytes. The access stages are described as follows:
   (a) Access the *root* inode.
   (b) Access the *root* directory guide block.
   (c) Access the *os* inode.
   (d) Access the *os* directory guide block.
   (e) Access the *readme.txt* inode.
   (f) Access the block pointed by the *single indirect*.
   (g) Access to **read** from the *readme.txt* block that contains byte 40,000.
   (h) Access to **write** to the *readme.txt* block that contains byte 40,000.
   (i) Access to **update** the *readme.txt* inode.

   Overall: **9 Disk Access**.

2. (a) `seek` will cause an interrupt to the operating system, i.e. *trap* because it is a *system call*.
   (b) `read` will cause an interrupt to the operating system, i.e. *trap* because it is a *system call*.
   (c) The actual reading will cause an *hardware interrupt* because to operating system needs to access to the disk in order to read the single byte from the file. (Recall that the operating system can communicate directly with the main memory but not with the disk).

3. (a) We will describe a *Multi-Level Feedback Queue* for this problem as follows:
      i. Number of queues is 2 and denote the queues by Q_1, Q_2.
      ii. The scheduling algorithms are *Round-Robin* for Q_1 and *SRTF* for Q_2. The quantum of Q_1 will be 1.
      iii. Every new task will enter Q_1.
      iv. All the tasks in Q_1 that did not finished their run will be transfered to Q_2.
      v. If we run a task in Q_2 and a new task arrived (according to our definition it will arrive Q_1) we will switch and run the tasks in Q_1.

   In this implementation all the tasks of type a will not wait at all. The tasks of type b is handled by *SRTF* which aims to minimize the waiting time.

   (b) In our algorithm we use preemptions from Q_1 to Q_2 after a single quantum, also we preempt b tasks from Q_2 if a new task has arrived. So our algorithm performs multiple context-switchs which causes more overhead. We could use *FCFS* for example to minimize the overhead.