DV1629 Lab 2

Emil Karlström and Samuel Jonsson

December 3, 2021

Home Assignment 3

Study the small test programs in Listing 1 (`test1.c`) and Listing 2 (`test2.c`), so you understand what they are doing.

1. How much dynamic memory is allocated in the `test1` program (i.e., how large is struct link)?
   Answer: Roughly 4 GB of data.

2. How much data is written to the temporary file `/tmp/file1.txt` in each iteration of the program `test2` ?
   Answer: Roughly 1 GB of data.

Home Assignment 4

Study the man pages of `vmstat` and top so you understand how these two commands work. Specifically, understand the parameters that you can give to the commands and the output format of the commands.

1. In which output columns of `vmstat` can you find the amount of memory a process uses, the number of swap ins and outs, and the number of I/O blocks read and written?
   Answer: The amount of memory a process uses is found in the `buff` column, swap In/Out are found in the `si/so` columns, blocks read/written are found in the `bi/bo` .

2. Where in the output from top do you find how much CPU time and CPU utilization a process is using?
   Answer: `TIME+` for CPU time, `%CPU` for CPU utilization.

Task 1

1. What is the CPU utilization when executing the test1 program? Which type of program is `test1` ?
   Answer: 100%, which means it is CPU bound.

2. Execute the test program `test2` . How much memory does the program use? How many blocks are written out by the program? How does it correspond to your answer in Home Assignment 3?
   Answer: 1 GB of RAM used, 6 blocks outputted. This corresponds well with our calculations in Home Assignment 3.

3. What is the CPU utilization when executing the `test2` program? Which type of program can we consider `test2` to be?
   Answer: Roughly 40%, this means it is I/O-bound.

Task 2

1. What is the difference between them in terms of how they execute the test programs?
   Answer: `run1` runs the programs in sequence, while `run2` runs three instances of `test1.c` in sequence in the background and also runs two instances of `test2.c` at the same time.

2. Execute the script run1 and measure the execution time. Study the CPU utilization using top during the execution. How long time did it take to execute the script and how did the CPU utilization vary?
   Answer: The execution time of `run1` was roughly 47 seconds. The CPU utilization started at 0% and then went up to 100% as `test1.c` started execution. When `test1.c` finished execution the CPU utilization went down to 0%. Whenever `test1.c` started this same pattern happened. After three instances of `test1.c` had been executed, two instances of `test2.c` ran back-to-back and the CPU utilization reached about 40% before going back down to 0%.

3. Execute the script `run2` and measure the execution time. Study the CPU utilization using top during the execution. How long time did it take to execute the script and how did the CPU utilization vary
   Answer: The execution time of `run2` was roughly 27 seconds. In the "top" program we could see two programs called `test1` and `test2`. `Test1` was using 100% CPU and `test2` was using about 40% of CPU power. This indicates that all instances of `test1` ran parallel to all of the instances of `test2`.

4. Which of two cases executed faster?
   Answer: `run2` ran faster.

5. In both cases, the same mount of work was done. I which case was the system best utilized and why?
   Answer: It is correct that the same amount of work was one, however, in `run2` there were more parallel computing being done which means better use of CPU power. In `run2` the I/O-bound `test2` processing was handed off to another process on the computer, which let the CPU bound `test1` tasks use as much CPU as they wanted.

Task 4

Table 1: Number of page faults for `mp3d.mem` when using FIFO as page replacement policy

No. of pages / Page size	1	2	4	8	16	32	64	128
128	55421	22741	13606	6810	3121	1503	1097	877
256	54357	20395	11940	4845	1645	939	669	478
512	52577	16188	9458	2372	999	629	417	239
1024	51804	15393	8362	1330	687	409	193	99

Table 2: Number of page faults for `mult.mem` when using FIFO as page replacement policy

No. of pages / Page size	1	2	4	8	16	32	64	128
128	45790	22303	18034	1603	970	249	67	67
256	45725	22260	18012	1529	900	223	61	61
512	38246	16858	2900	1130	489	210	59	59
1024	38245	16855	2890	1124	479	204	57	57

Task 5

1. What is happening when we keep the number of pages constant and increase the page size? Explain why!
   Answer: The number of page fault becomes fewer, but not with much. This happens because each page can fit more addresses and therefore can "take" more page faults than a page with a smaller size.

2. What is happening when we keep the page size constant and increase the number of pages? Explain why!
   Answer: The number of page fault about halves for each increase in the number of pages (except for the last two steps). This happens because we doubles the amount of spaces that exists to save an address in, and therefore the amount of times you have to change it out halves.

3. If we double the page size and halve the number of pages, the number of page faults sometimes decrease and sometimes increase. What can be the reason for that?
   Answer: Since each page is bigger, we have to replace the pages more often, resulting in more page faults sometimes. The reason it only is sometimes is because of how the sequence of addresses behaves. When you have many addresses that pass the "boundary between pages", this

results in more page faults, however when you make the pages bigger, those addresses fall into the same page.

4. Focus now on the results in `Table 2` (`mult.mem`). At some point the number page faults decreases very drastically. What memory size does that correspond to? Why does the number of page faults decrease so drastically at that point?
   Answer: The decrease happens when you increase from 4 pages to 8 pages on page size 128 and 256, as well as between 2 and 4 on page size 512 and 1024.

   These jumps occur are when we reach a hole multiple of a kilobyte. This is important since the memory addresses of `mult.mem` are from the matrix multiplication program from lab 1 where each matrix was 1024×1024 large, which means that most of the addresses in `mult.mem` are close to each other, resulting in entire calculations behind handled within each page, drastically decreasing the number of page faults occurring.

5. At some point the number of page faults does not decrease anymore when we increase the number of pages. When and why do you think that happens?
   Answer: This happens on page count 64 and 128. This likely happens because there are more pages available than is necessary for the task, e.g. in the case of page size 128, we might only need 67 pages, so when e have 128 pages available, we are only using 67 of them.

Task 7

Table 3: Number of page faults for `mp3d.mem` when using LRU as replacement policy.

No. of pages / Page size	1	2	4	8	16	32	64	128
128	55421	16973	11000	6536	1907	995	905	796
256	54357	14947	9218	3811	794	684	577	417
512	52577	11432	6828	1617	603	503	362	206
1024	51804	10448	5605	758	472	351	167	99

Task 8

1. Which of the page replacement policies FIFO and LRU seems to give the lowest number of page faults? Explain why!
   Answer: LRU seems to be better in the middle-case cases, however, only slightly better overall. This is because with FIFO you may encounter situations where you may throw away pages that will be used soon. LRU

fixes this by removing the one that seems to be the one least probable of being used in the future.

2. In some of the cases, the number of page faults are the same for both FIFO and LRU. Which are these cases? Why is the number of page faults equal for FIFO and LRU in those cases? Explain why!
   Answer: The first case is when page count is 1. This is because LRU depends on being able to choose between multiple options which to replace. However, when you only have one page, there are no options. The other is the case where the page size is 1024 and page count is 128. This most likely happens because the pages are so large and the amount of pages are so many that we don't actually use all of the pages.

Task 10

Table 4: Number of page faults for `mp3d.mem` when using Optimal (Bélády's algorithm) as replacement policy

No. of pages / Page size	1	2	4	8	16	32	64	128
128	55421	15856	8417	3656	1092	824	692	558
256	54357	14168	6431	1919	652	517	395	295
512	52577	11322	4191	920	700	340	228	173
1024	51804	10389	3367	496	339	213	107	99

Task 11

1. As expected, the Optimal policy gives the lowest number of page faults. Explain why!
   Answer: This is because when you only replace the page that is the furthest away of being needed, the number of page replacements will decrease drastically.

2. Optimal is considered to be impossible to use in practice. Explain why!
   Answer: Because it is impossible to compute how long it will be before a page is going to be used, unless all software that will run in the system is either known beforehand and you can constantly analyze its memory reference patterns, or its only a class of applications allowing run-time analysis.

3. Does FIFO and/or LRU have the same number of page faults as Optimal for some combination(s) of page size and number of pages? If so, for which combination(s) and why?
   Answer: In the case when we only have one page, the Optimal Algorithm

suffers from the same problem as LRU, that is, there is no possibility to choose which page to replace. You can effectively say that Optimal and LRU are the same as FIFO in this one case as you always replace the first (and only) element.

The second case is where the page size is 1024 and page count is 128. This most likely happens because the pages are so large and the amount of pages are so many that we don't actually use all of the pages, same as with LRU.