OS lab 2

Tobias Mattsson, DVAMI21h

Samuel Nyberg, DVAMI21h

Home assignment 3

In program test1 we allocate a matrix that has 32x1024 rows and 32x1024 columns of integers. That matrix is worth roughly 4.3 Gb of memory.

In program test2 we write a matrix of the size 16x1024 by 16x1024 with integers. That is about one Gb of data each iteration.

Home assignment 4

Vmstat:

How much memory is used by a process: **free**

The number of swapped ins and outs: **si** & **so**

The number of IO-blocks read and written **bi** & **bo**

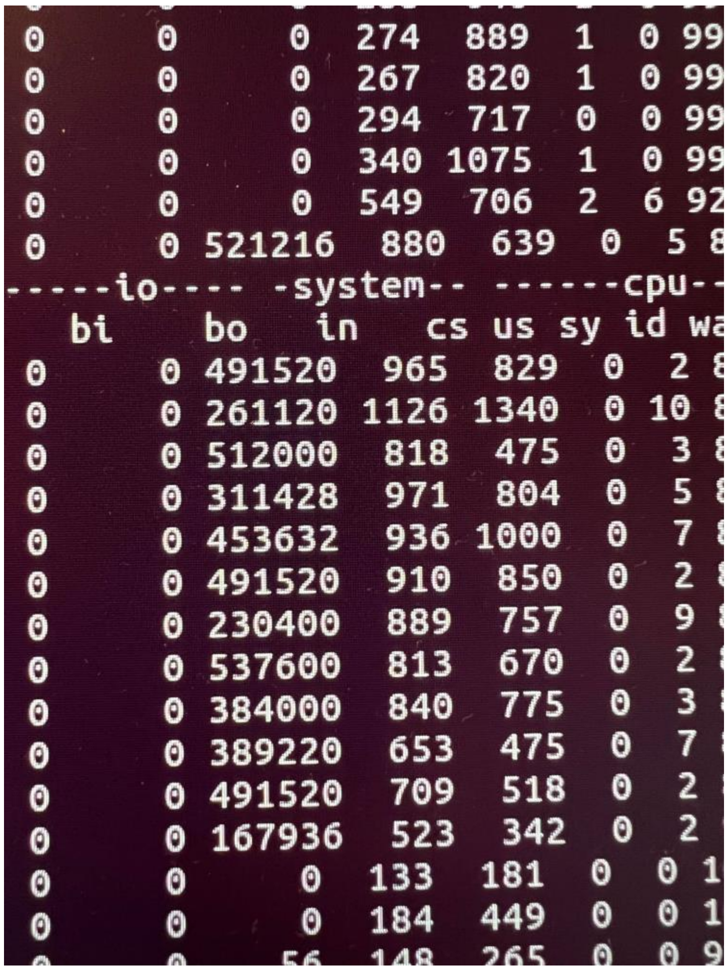
Top:

To see the CPU utilization, we use command **%CPU** and to see CPU task time we use the command **TIME** or **TIME+** if we want time timed to hundredths.

Task 1:

When running test1 we get a CPU utilization of 100% and the free memory seen in vmstat dropped from 12074904 kb down to 7872148 which when calculating the used memory corresponds to 12074904−7872148 = 4202756 = 4.2Gb. This is almost the same as in Home Assignment 3. And since test1 had 100% Utilization on the CPU it’s a CPU-bound program since it’s runtime is completely affected by the speed of the processor.

Test2 had 12073960 kb free memory which dropped to 11020104 during execution resulting in 12073960 − 11020104 = 1053856 = 1Gb of memory was allocated and used during test2. This corresponds to Home Assignment 3. If we see how much that is written to the text file, we can summaries all kb from bo and we get a total of 5.2Gb which corresponds to running the loop five times which is done in the executable. Since Test2 is using only 30% of the CPU we can conclude that task2 is memory bound.



```
 0     0      0  274  889  1  0 99
 0     0      0  267  820  1  0 99
 0     0      0  294  717  0  0 99
 0     0      0  340 1075  1  0 99
 0     0      0  549  706  2  6 92
 0     0 521216  880  639  0  5 8
-----io---- -system-- ------cpu--
   bi    bo   in   cs us sy id wa
 0     0 491520  965  829  0  2 8
 0     0 261120 1126 1340  0 10 8
 0     0 512000  818  475  0  3 8
 0     0 311428  971  804  0  5 8
 0     0 453632  936 1000  0  7 8
 0     0 491520  910  850  0  2 8
 0     0 230400  889  757  0  9
 0     0 537600  813  670  0  2
 0     0 384000  840  775  0  3
 0     0 389220  653  475  0  7
 0     0 491520  709  518  0  2
 0     0 167936  523  342  0  2
 0     0      0  133  181  0  0 1
 0     0      0  184  449  0  0 1
 0     0     56  148  265  0  0 9
```

Task 2:

Run1 is a bash file that executes five programs sequentially. It executes test1 three times in a row and after that test2 two times. In Run2 the bash file executes the series of test1 and the series of test2 simultaneously. It does this by creating its own shell for test2 to run in. Both run1 and run2 were tested on a computer in lab 332.

Run1 showed a CPU utilization of 100% while running test1 and 28-34% while running test2. The entire run took 1.46.43 minutes. The CPU utilization was tracked with the top command.

Run2 showed a CPU utilization of 100% and 30% at the same time showing the work split over two subshells where test1 runs in one shell and test2 runs in another. Run2 takes 27.64 seconds to complete, this is much faster than Run1 since this type of bash execution splits the processes so that they are run simultaneously giving a better performance when measured in time. This makes it so test2 doesn’t need to wait for all test1 processes to finish before commencing.

Task 4:

mp3d.mem

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

mult.mem

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	58
1024	38245	16855	2890	1124	479	204	57	57

Task 5:

Based on the values in Table 1 and Table 2, answer the following questions.
What is happening when we keep the number of pages constant and increase the page size? Explain why!

The page faults decrease. The reason is that each page covers a higher range of memory addresses resulting in higher chance of corresponding page already being load

What is happening when we keep the page size constant and increase the number of pages? Explain why!

The page faults decrease. The reason is that more pages are loaded at the same time increasing the probability of the corresponding page for the memory reference to already be loaded.

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?

There are a lot of potential reasons for that. One reason could be due to good or bad temporal locality. Good temporal locality means that the same data is accessed repeatedly over a short timeframe, in that case a smaller page size could be effective. On the other hand, if your application has poor temporal locality, a larger page size could help by loading more data into memory at once, reducing the number of page faults caused by frequent switching between data.

Focus now on the results in Table 2 (matmul). 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?

The number of page faults between page size 256 and 512.
With a larger page size, each page contains more data, which reduces the total number of pages that need to be loaded into memory to do the same set of work. This can lead to fewer page faults, especially if the program accesses data in contiguous blocks.

***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?***

Between 64 and 128 pages the faults don't decrease anymore.
One reason could be because when the demand for pages increases it can exceed the available physical memory resulting in increased page faults. This scenario is called thrashing.

Task 7:

mp3d.mem

Page size	1	2	4	8	16	32	64	128
128	55421	16973	11000	6536	1908	996	905	796
256	54337	14947	9218	3811	794	684	577	417
512	52577	11432	6828	1616	603	503	362	206
1024	51804	10448	5605	758	472	351	167	99

Task 8:
Between LRU and FIFO, LRU is the algorithm that gets the lowest number of page faults. This is because LRU changes out the least recently used page to store a new page. This is much more effective because pages that are used more frequently are more likely to already be stored inside of the physical pages thus generating less page faults since we get hits instead of misses.

LRU and FIFO can sometimes generate the same amount of page faults, this happens when we have only one physical page. The logic behind this is that no matter the algorithm, if we only have one physical page to store our pages in, we either get a hit meaning that the page we try to access already is loaded and the algorithm does nothing. Or we get a miss, and the existing page is swapped for the newly loaded one. Since we only have one page, we don't have any specific priority to take into account, thus the number of page-faults stays the same.

The other time the page faults would be the same is if we have more physical pages than we have virtual pages. Then the algorithms priority on how to change the pages wouldn't be used since we will never get a full list of pages, if a page isn't already loaded, we can just add it and next time it appears it will be a hit.

Task 10

mp3d.mem

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	470	340	228	173
1024	51804	10389	3367	496	339	213	107	99

Task 11

Based on the values in Table 1, Table 3, and Table 4, answer the following questions. As expected, the Optimal policy gives the lowest number of page faults. Explain why!

The optimal policy is the fastest since it has the ability to see into the "future" and can decide on which page is the furthest away from being used, thus being able to swap this out for the new page.

Optimal is considered to be impossible to use in practice. Explain why!

Optimal is considered to be impossible to apply in the real world since it requires the algorithm to be able to know what addresses and memories are going to be loaded and used in the future which we can't predict.

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?

FIFO and LRU do have the same number of page faults as Optimal algorithm when there is just one physical page to work with. This is because no matter the algorithm if there is a miss the one and only page will need to be swapped for the new one and if there is a hit the page loaden inside the physical page is kept. It's the same logic over all the three algorithms when there is just one page since we don't have any specific order for multiple pages to keep track of. There is one more time that page faults are the same between algorithms that is explained in task 8.

Implementation:

All algorithms just fill the physical pages in the beginning until it’s full. Then the real algorithm takes place.

Fifo:

We made an array with a static pointer that operated with modulo to keep track of the page that would be changed.

LRU:

We sent the index the hit page is positioned on and moved it forward and moved the rest of the pages backwards one priority spot. When there was a miss, all priorities were moved one step meaning the last one was removed, and the new loaded page was put in latest used.

Optimal:

The optimal algorithm calculates all pages before commencing. The algorithm takes place when a miss happens (page fault). Then it checks for each of the loaded pages when they will appear next time. If a page is never loaded again that will immediately be switched for the new page. Otherwise, it takes the page that is loaded furthest away in time.