LAB 2: MEMORY SYSTEMS

Ahmad Tarshehani Samar Albardan
DVAMI19

Home Assignment 3. Test programs

How much dynamic memory is allocated in the test1 program (i.e., how large is struct link)?

The dynamic memory of allocated by calloc is 1 block for 4 gigabytes
4294967296 bytes ≃ 4 GB
Size = 32*1024
Struct link {int x [size][size]}
4 [(32*1024) * (32*1024)] = 4294967296 B

How much data is written to the temporary file /tmp/file1.txt in each iteration of the program test2?

The file have about 1,1 GB (1073741824 KBytes)
SIZE = 16*1024 = 16384
Struct link = 4 * 16384 * 16384 = 1073741824 Kbytes
And in each iteration is 214 748 364 bytes about 20.4 MB because it is five iterations and so 1073741824 / 5 = 214 748 364 bytes per iteration.

Home Assignment 4. vmstat and top

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?

The amount of memory a process uses can be found inside the memory section in the cache/swpd column that gives how many bytes the cache stored so the cpu can handle.
The number of swap can be found inside the swap section in the si and so columns and shows how many swap processes move in and out the memory.
The number of I/O blocks read and written can be found inside the io section in bi and bo columns which represent the data that transfers between virtual memory and block devices in the disk.

Where in the output from the top do you find how much CPU time and CPU utilization a process is using?

We can find the time that the process uses in column TIME+that shows time of the executing program until the program finished. I would say in column number 11.
The percent of CPU that each program use is in column number 9 under name %CPU that shows the utilization.

Task 1. CPU-bound vs. I/O-bound processes

Execute the test program test1. How much memory does the program use? Does it correspond to your answer in Home Assignment 3?

After we run the program three times we got that the program use about 4 GB and we did by checking the free column in the vmstat command and took the memory value in kib before we execute the program and subtract it with the memory value after the execution which in average about 4 204 244 kib. 12 260 504 - 8 056 260 = 4 204 244 kib. By checking the VIRT column in the top command

,which shows the total amount of virtual memory used by the task, we got 4198656 kib which also means it is about 4 GB.
Yes, it correspond our answer

What is the cpu utilization when executing the test1 program? Which type of program is test1?
After we ran the program Test1 a few times we saw in the top command that the CPU utilization was stretching between 99% to 100% to run all 10 iterations. The test 1 is a computationally-intensive program that uses cpu-bound because the time for it to complete a task is determined principally by the speed of the central processor: processor utilization is high, like we got 100%.

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?
Before we run the Test2 program the free memory in vmstat command was around 12568724 KiB and after we run it the free memory reduced to 11494402 KiB which means the memory that the program used is about 1074322 KiB and by checking also the VIRT column in the top command was also 1052928 KiB which match with the file1.text = 1,1 GB. The total block the program write are over 5 million blocks.
Yes it corresponds to our answer.

What is the cpu utilization when executing the test2 program? Which type of program can we consider test2 to be?
When we run the program Test2 the CPU utilization stretches between 30% to 40% and the program type is I/O bound because the time it takes to complete a computation is determined principally by the period spent waiting for input/output operations to be completed.

Task 2. Overlapping CPU and I/O execution

What is the difference between them in terms of how they execute the test programs?
The run1 executes the program in sequence order with just one process at the same time. But when we run2 executed the two programs, test1 & test2 worked at the same time in parallel processes.

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?
As many as we run the program we will got that run1 will take about 47s to execute and as we know the program execute on one process so the we can see from %CPU when run1 start with task1 the cpu utilization start low to 100% and after the tenth iteration the cpu restart from beginning to 100 but this time their low be lower than the first low ,for example first low of test1 was 73% and second one was 43%, and the cpu again restart after the tenth iteration from low to 100% then the test2 begin and the cpu utilization restart between 30% and 37% to max between 40& and 45% and in second time the cpu restart under 30% to max 38% and that means that cpu and memory can do the task faster when it do it many time in the row.

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?
As many as we run the program we will got that run2 will take about 27s to execute and as we know the program execute in parallel processes so the we can see from %CPU when run2 start with task1 and task2 run in same time and same as run1 the cpu utilization for task 1 start low to 100% and after

the tenth iteration the cpu restart from beginning to 100% and same time the task2 cpu do the same as run1.

Which of the two cases executed fastest?
The second case, run2,
In both cases, the same mount of work was done. I which case was the system best utilized and why?
The second case, run2, because in the second case the system runs in parallel processes which is faster and it utilizes the CPU and the memory in faster and lower percentage than run1.

Task 3. Implementation of FIFO page replacement

FIFO

mp3d.mem

Numbers 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

mult.mem

Numbers 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. Evaluation of the FIFO page replacement policy

What is happening when we keep the number of pages constant and increase the page size? Explain why!
Since the number of frames is equal to the size of the memory divided by the page size, which means an increase in the number of pages will correspondingly decrease the page faults as we can see in the tables. Each process has three sections (text, heap, stack) and if every section has doubled page size that means on average 1,5 pages are unused for each process I would to say 0,5 N blocks lost to fragmentation. The use of larger pages will attract more memory per error, so the number of errors may decrease if there is a limited conflict or a reasonably high spatial area on the page size scale. So first reference page errors will tend to dominate, so larger pages will reduce the number of page errors.

So when we expand the page size we also expand the interval for these memory addresses. So in more explanation if number of pages decrease the CPU will have a very few addresses to refer to and thus increase the number of page faults.now if page size becomes the size of the process then there will almost as many number of pages as the number of processes so CPU will refer to it with no page faults.

What is happening when we keep the page size constant and increase the number of pages? Explain why!
As we can see from the tables above the number of page faults decrease significantly when page size is constant and increase the number of pages. This time when we increase the number of pages that means we have frames to store the reference in it. That causes more intervals and pages which already exist in RAM to increase and page faults decrease significantly. Which means we will have more hits than miss in the table because the reference stored will get longer to be replaced with a new one.

If we double the page size and halve the number of pages, the number of page faults sometimes decreases and sometimes increases. What can be the reason for that?
The reason for that may be Belady's Anomaly because FIFO is not a stack based algorithm which it can be shown that the set of pages in memory for *N* frames is always a subset of the set of pages that would be in memory with *N + 1* frames while in FIFO algorithm , if a page named *b* came into physical memory before a page – *a* then priority of replacement of *b* is greater than that of *a*, but this is not independent of the number of page frames and hence which leads to Belady's Anomaly and sometimes the page faults decreases or increases.

Focus now on the results in Table 2 (matmul). At some point decreases the number of page faults very drastically. What memory size does that correspond to? Why does the number of page faults decrease so drastically at that point?
When the page size is 512 and 1024 for example when the number of pages are 4 the page faults decreases powerfully from 18012 faults to 2900 faults and happen because using larger pages will draw in more memory per fault, so the number of faults may decrease if there is limited contention and/or reasonably high spatial locality at the scale of page size. In more explanation if you have a large page size as 512 or 1024 you will have more hits in the memory frames than miss in the table so that what may lead to the page faults very drastically and in other example when the number of pages is 16.

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?
It is in file mult.mem and when number of pages are 64 and 128 the quantity of the miss and hit in table are the same when the number of pages is 64 and when it is 128 and we think this happen because the frames store the same reference in both 64 and 128 which leads that FIFO algorithm do not replace many pages with new one.

Task 7
Table 3: Number of page faults for mp3d.mem when using LRU as a replacement policy.

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. Comparison of the FIFO and LRU page replacement policies
Which of the page replacement policies FIFO and LRU seem to give the lowest number of page faults? Explain why!
The Least Recently Used algorithm (LRU) gives the lowest number of page faults because the LRU policy is to replace the pages some has not been used in a long time with new pages which means the active pages or may be called in the future will be stored in the frames which decrease the page fault.
The FIFO policy is that the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced, the page in the front of the queue is selected for removal.
So LRU keeps the things that were most recently used in memory which mean that. LRU is, in general, more efficient, because there are generally memory items that are added once and never used again, and there are items that are added and used frequently. LRU is much more likely to keep the frequently-used items in memory that leads to less page faults.

In some of the cases, the number of page faults is 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!
It happens when the page number is 1 and page size 128, …, 1024. We can notice on the page faults numbers were the same in both cases FIFO & LRU. The reason for this is just one page fills the frame each time when the system starts working and both algorithms will replace the page with other pages. The same result will be generated in both algorithms when we just fill in 1-page numbers case.
Another case that we can also notice from the tables is when the page number is maximum 128 and page sizes also max 1024. In this case, the page fault is also the same, because the interval of memory space is max page (frames) and page fault fits in it.

Task 10

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. Comparison of the FIFO, LRU, and Optimal page replacement policies

As expected, the Optimal policy gives the lowest number of page faults. Explain why!
The optimal algorithm focuses and looks always to the next instruction and checks if it will be used in the next process and prioritizes them and saves them in memory to use them again if needed instead of replacing them with new pages. The optimal algorithm can predict the most used process and save it until used again, making the optimal algorithm much more affected than the other algorithms.

Optimal is considered to be impossible to use in practice. Explain why!
In this example we know all pages that will work in the system and how long they will stay to finish which make it easy to implement this type of algorithm. The operating system in practice can realize how long it takes past pages to complete their job, but on the contrary it is difficult to predict how long the future pages will stay in memory to finish their work. That makes the optimal algorithm impossible to use in practice.

Does FIFO and/or LRU have the same number of page faults as Optimal for some combination(s) of page size and a number of pages? If so, for which combination(s) and why?
Yes, they have. When the page number is 1 and page size 128, 256, 512, 1024 all types of algorithm we get the same result of page faults .It is because we have just one page to fill the frame when we replace the pages. And when page number 128 and the page size is 1024 we get 99 page faults because the size of page and the number of pages especially have the same page faults.

Code Implementation
FIFO: We implemented the FIFO page replacement algorithm by having one main function that is creating the file that we will work with and calling the FIFO function. FIFO function is a void function, our FIFO structure works as a circular array. So the oldest page will be overwritten by the new page and so on. First we inilitize all the variables that we are working with then we have a while loop that keeps going until there is nothing to read from the file. In our while loop we have a for loop that checks if the memory address already exists in the page table. After that we have an if statement that says if the memory address does not exist in the page table then we will increase the page_fault variable by one, create the first memory in the page and add all the memory references in the page to the memory. At the end of the FIFO function we have a prinf that prints which memory reference we worked with and the page faults in that reference.

LRU: We implemented the LRU page replacement algorithm by also having the main function and the void LRU function. The main function is exactly the same main function we used in the FIFO algorithm. In our LRU function we first initialize all the variables that we are working with, then we have a while loop that keeps going until there is nothing to read from the file. The least recently used page is in the first position of the page table. In the while loop we have a for loop that checks if the page exists in the table and then adds the recently used page to the end of the array. If the page does not exist in the table, then we will replace the LRU page and move the recently used to the end of the array. Lastly we have a printf that prints which memory reference we worked with and the page faults in that reference.

OPTIMAL: Our implementation of the Optimal page replacement algorithm was complex to implement, we have a struct called Array and few help functions that are called in our OPT void function (Optimal function). The Array structure contains three variables, integer pointer array, integer size and integer used_page. These variables will be used throughout the help functions and the OPT function. In the OPT function we begin to declare variable a to of type Array and then call init_array to initialize the array. After that we initialize all the variables that we need to complete the algorithm. Then we have a while loop that keeps going until there is nothing to read from the file and in that while loop we are inserting all the memory references in the order they occur into the array. A for loop comes after the while loop and there starts the optimal page replacement policy, first we check if the page exists in the page table and if the page doesn't exist in the page table it generates a page fault. Thereafter we have another if statement that checks if no_pages (number of pages) is greater than the file_reference (references from file), if that is true then fill the page table with references, increase file_referemce and continue to the next for loop. This for loop will loop through the page table and get the page that is furthest away by initializing the result variable and this variable is calling get_next_pos to get the next position when the page is needed and using the if statement to see if the result is greater than fartherst_pos. After the last for loop we replace the page that is farthest away, increasing file_reference and lastly we have a printf that prints which memory reference we worked with and the page faults in that reference. The main function in this implementation is exactly the same main function in the FIFO and LRU implementation.