Report Lab 2 by Vincent Honar & Jonatan Frykmer

Homework Questions

How much dynamic memory is allocated in the test1 program

4*(32*1024) ² bytes

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

A: 4*(16*1024) ² bytes

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?

- memory usage: The difference in the free column from one entry to another
- active swaps: si so
- blocks: bi bo

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

- cputime: TIME+
- CPU utilisation: %CPU

Task Questions

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

The program uses roughly 4244732 bytes and does not correspond to the answer in homework 3.

What is the CPU utilization when executing the test1 program? Which type of program is test1?

The program is CPU bound and utilizes 100% of the CPU

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?

No change in memory usage was detected, 1048792 blocks were written out in each iteration which is 536981504 bytes. The difference is roughly by a factor of 2 (the answer is twice as large as the measured value)

What is the CPU utilization when executing the test2 program? Which type of program can we consider test2 to be?

56% CPU utilisation and the program can be considered to be i/o bound.

Study the two scripts run1 and run2 . What is the difference between them in terms of how they execute the test programs?

Run1 executes sequentially meanwhile run2 executes test1s and test2s parallel

Execute the script run1 and measure the execution time. How long time did it take to execute the script and how did the CPU utilization vary?

2m 35s was the execution time, the program maintained 100 CPU utilisation whilst executing test 1 and dropped to 50% when executing test 2. During the execution of test 2 CPU utilisation varied between 56-42%.

Execute the script run2 and measure the execution time. How long time did it take to execute the script and how did the CPU utilization vary?

1 m 47 s was the execution time. Test1 utilised 100% of the CPU and test 2 varied between 30-45%

In both cases, the same amount of work was done. In which case was the system best utilized?

Run2 utilised the system resources better since one task is CPU bound whilst the other is I/O bound. The processes can work simultaneously since their resource requirements are not completely overlapping.

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

The number of page faults decrease in a varying scale depending on the shape of the references (if the memory references have a great spread in terms of values, the effect is minimised). A bigger page size entails storing more memory in each page which increases the chance of a reference to be contained within one of the existing pages whereas said reference might have no been in those pages before.

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

The number of pagefaults decrease significantly initially and slows decreases marginally as the number pages reaches high values. More pages allow more blocks of memory to be stored which increases the chance of a reference being loaded in already. Although more pages do not necessarily decrease the number of page faults by the same amount during each increase, some references adhere to a block of memory which is only referenced once and does not stand to gain from remaining loaded beyond that point.

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?

Increasing page size only has the potential to decrease page faults if the several references are within the same block of memory. If the references are spread out in a sporadic manner, increasing page size does not load in any relevant references.

At some point in matmul decreases in the number of page faults occurs very drastically. What memory size does that correspond to? Why does the number of page faults decrease so drastically at that point?

When memory size reaches 2048 the number of page faults decreases significantly. This occurs as either most entries in matmul can be contained within a few pages (a considerable number of values are between 22000-23000) or enough pages exist to contain the entire range which reduces the likelihood of a page fault occurring.

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?

This occurs when the page size reaches 64 and is most likely because all references have been loaded in some page

Which of the page replacement policies FIFO and LRU seems to give the lowest number of page faults?

LRU gives the lowest number of page faults which can be explained by the fact that the most used pages in the past have a higher likelihood to be used in the future in comparison to throw out highly used but old memory references.

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?

When the number of pages is 1 or when page size and number of pages are 1024 and 128 the algorithms have the same amount page faults. If only one page exists, neither algorithm has any choice in which page to remove as they have to remove from the same slot each time. In the specific instance where there are 128 pages of size 1024 all references are contained within 99 pages which means that neither algorithm for page replacement is used.

As expected, the Optimal policy gives the lowest number of page faults, why?

Knowing whether a page is going to be used in the near future or at all allows the algorithm to more efficiently select pages to replace. The other algorithms make assumptions on how future accesses will occur whilst Belady's algorithm has enough information to not base itself on guesswork.

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

The optimal algorithm requires knowledge about the future memory references, in practice the pages needed for a program is almost always unknown to the program itself. Therefore, without this information the optimal policy can't be implemented in practise.

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?

The optimal algorithm shares the same number page faults in exactly the same places that LRU and FIFO share with each other. This is solely because at these values, none of the algorithms are ever used.

Implementations

FIFO

The two first inputs are converted from ascii to int through "atoi()", the third input is stored in a char array. A file is opened through the third input with read privileges. An int is created for the purpose of counting the number of page faults, whenever one of the page fault conditions are fulfilled, this is incremented. These steps were done in each implementation.

The program starts by creating a page frame with the size of the desired number of pages, with all values set to –1 indicating that all spots are empty. It then loops through each row in the .mem file, dividing the memory reference by the page size to find which page it belongs to. The program then checks the frame to see if a spot is empty, if it is the page is inserted to that spot and a page fault is added to the counter. If the requested page already exists in the page frame, then the program continues to the next reference. If it doesn't exist a page fault is added to the counter and the new page replaces oldest loaded page with the current reference's page. The oldest will page slot is always the number of previous page faults (not including page faults which stem from empty slots existing) % page size.

LRU

An array is used to create the LRU policy. The array stores the number of rows preceding when the page slot was last accessed. When a page slot is accessed, the array corresponding array slot is incremented. When a reference is not found among the current pages, a loop is started which looks through the "LRU" array for the lowest number. The index of said slot is stored and used to replace the page with the new page.

Optimal

All references are stored in a list (this list will be called ref from now on), the program iterates through the said list, loading the reference's pages in the same manner as previous algorithms. Another array of the same size as the page table is created, this array is tasked with storing the number of references between the next time a page will be used. Once no page for the reference in question can be found a new for-loop, starting at whichever point the outer loop is in begins. This inner loop looks through ref for when and if the currently stored pages will be used. If a page is never used again (indicated by the inner for-loop not finding any reference in ref and thus the other array remaining the same as it did when it initialised), the new reference's page will occupy that slot. If all currently stored pages will be used in the future, the one which is furthest away from where the outer loop's value will be dropped. The other array is reset for future use.

Tables can be found below

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

Page size	Number of pages							
	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 matmul.mem when using FIFO as page replacement policy.

Page size	Number of pages							
	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

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

Page size	Number of pages							
	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

Table 4: Number of page faults for mp3d.mem when using Belady’s algorithm as page replacement policy.

Page size	Number of pages							
	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