Samuel George
Comp 530H
November 16th, 2017

Fault Handler behavior in a Demand Paging System

Introduction:

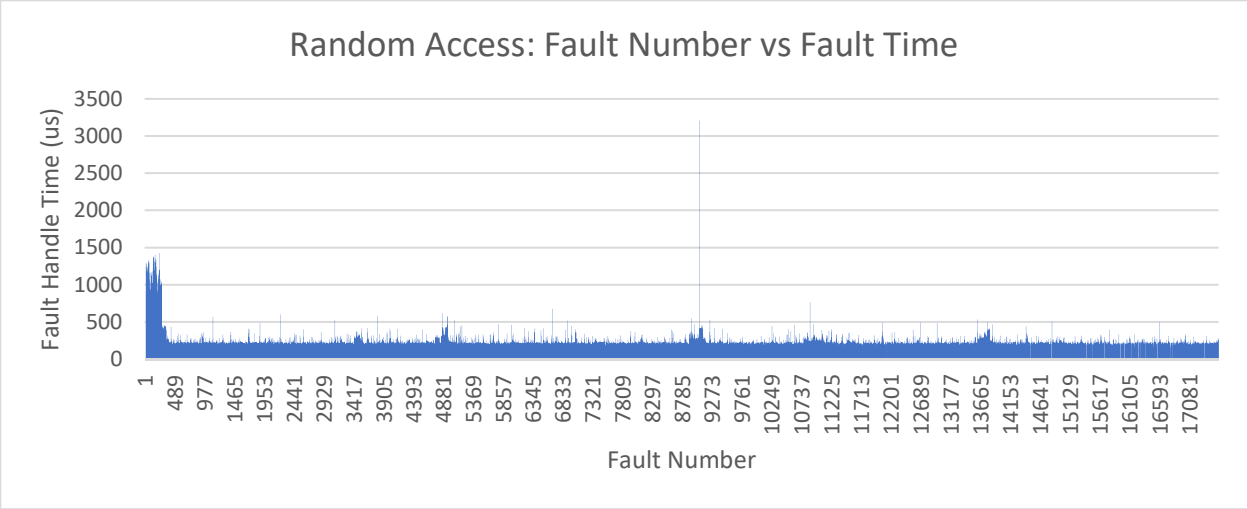
For Homework Six I created a kernel module that tracks the page faults generated by any given process. Page faults are generated both when a file needs to retrieve a mapping for a page in memory as well as when a file is missing from memory and needs to be loaded into memory. The system we tested uses demand paging, which loads pages into memory as they are needed by the program. This is in contrast to anticipatory paging that loads all of the pages a program may use into memory at once. Therefore, my fault tracking program allows you to analyze the page behavior of a process in real-time. Different programs were written to access a memory mapped file randomly, sequentially, and stepwise in order to analyze how demand paging works in the system. Based on my knowledge of demand paging I would expect that the random access of a memory mapped file would spend the most time faulting. I would also expect the sequential program to be the most efficient in terms of time spent faulting. Finally, I would expect the stepwise performance to be somewhere in between sequential and random access.

Methods:

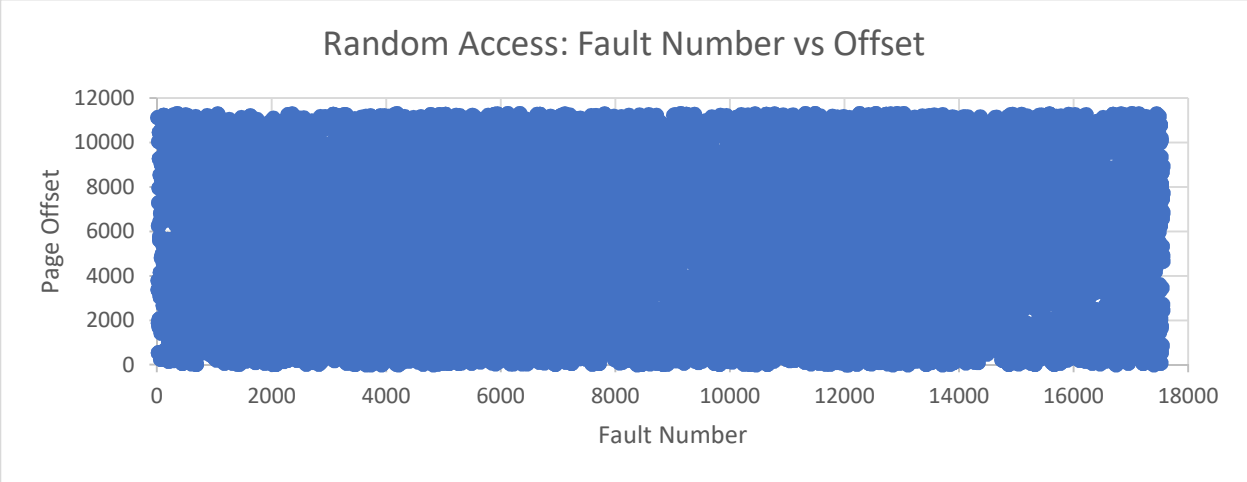
I wrote three programs in order to test my kernel module. All three begin by mapping a file and then calling my kernel module. The first program then accesses the mapped file randomly, requesting random addresses inside the memory mapped file with no pattern. The second program accesses the memory mapped file in sequential order, starting at its lowest index and going all the way through the file sequentially. The final program accesses memory in a stepwise manner, jumping 49152 bytes till the end of the file and then going back to the beginning. Both the sequential and stepwise end up accessing every byte in the file. The random program has no guarantee of accessing all bytes of the mapped file, but does the same number of memory accesses. The kernel log module records overall fault number, VMA fault number, area address, virtual address, page offset, page frame, and time to complete the systems fault call. For my experiments I isolated the VMA responsible for the mapped file. Fault count numbers and times are for that VMA only. Each experiment was run after a full reboot of the machine so that physical memory was clear of all test files. The kernel module prints output using a trace_printk call that records data in the /sys/kernel/debug/tracing/trace file. Performance was evaluated using a number of factors including average fault time, total faults, and total amount of time spent faulting. These same metrics were evaluated for faults that returned no page in vm_fault, meaning they were actual I/O accesses and not mapping. Data was collected for each experiment and analyzed.

Results:

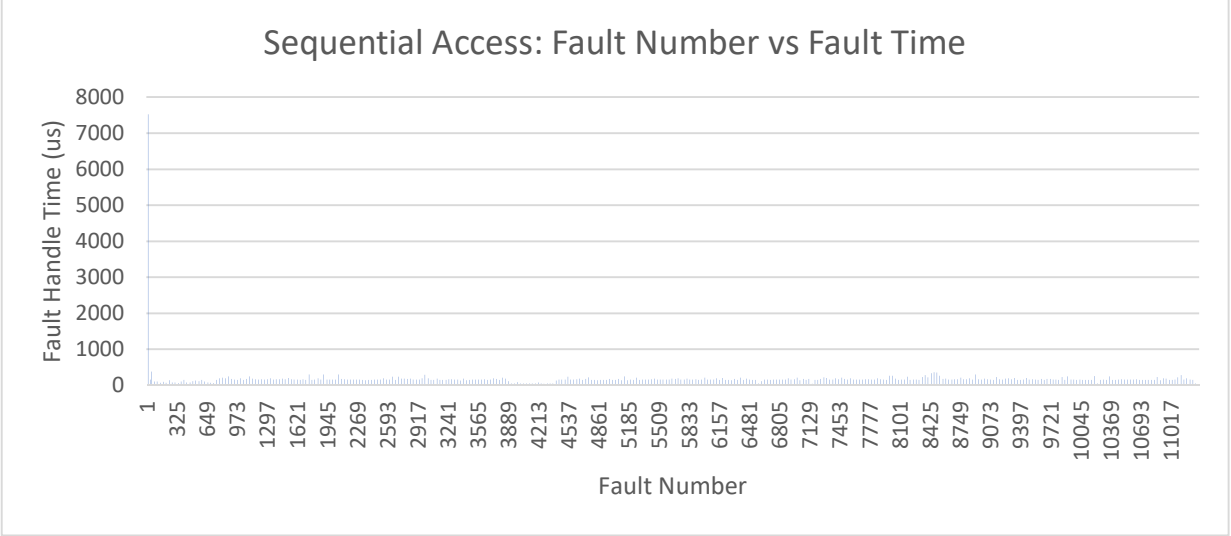
Below are chart showing the fault number and time to handle the fault based on fault number within the VMA containing the mapped file. This chart is for the random-access program. Times are in microseconds.



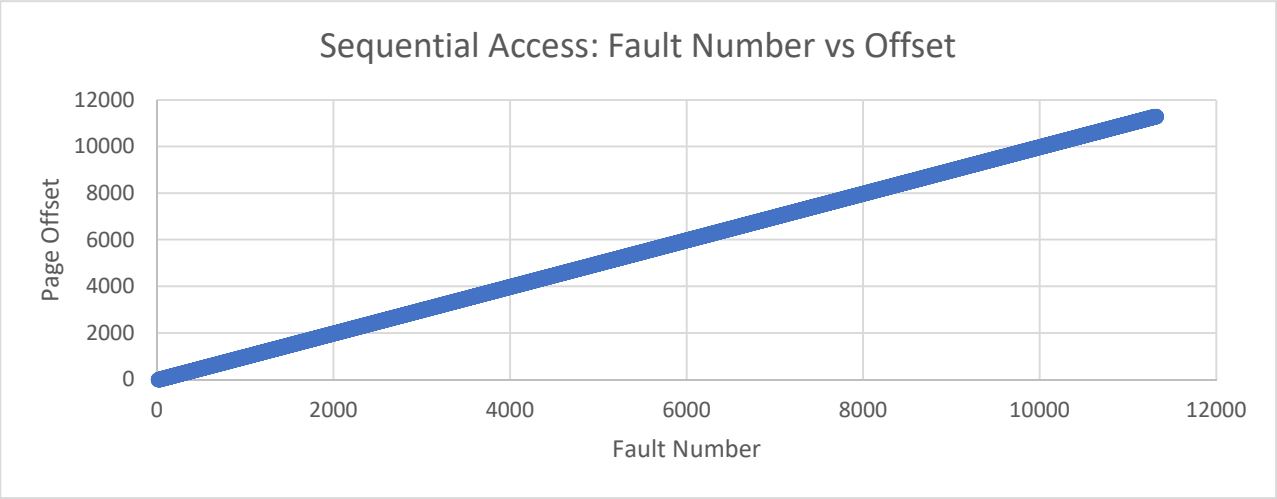
For that same experiment the Page Offset accessed vs fault number is given below.



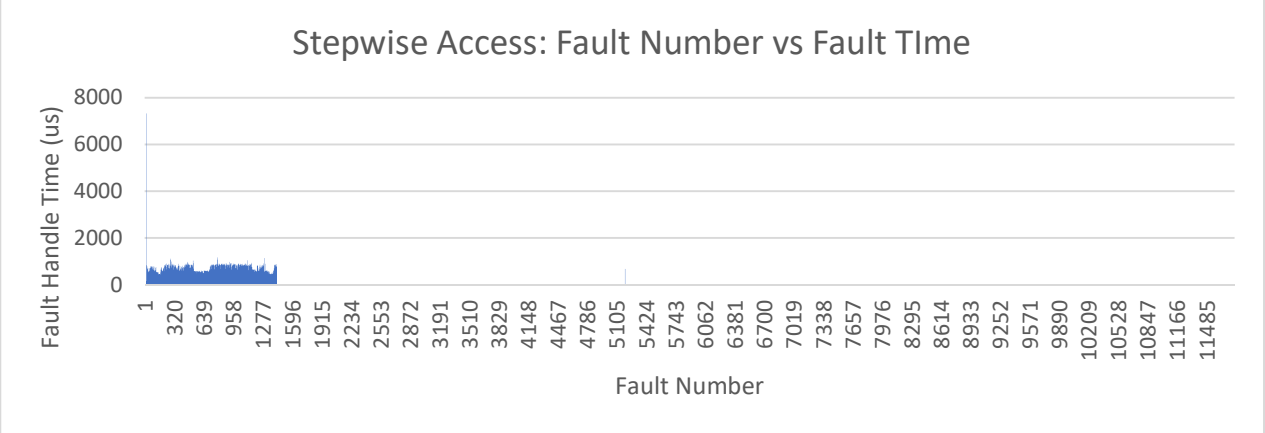
Below are the results of fault number vs time to handle the fault for the sequential access program.



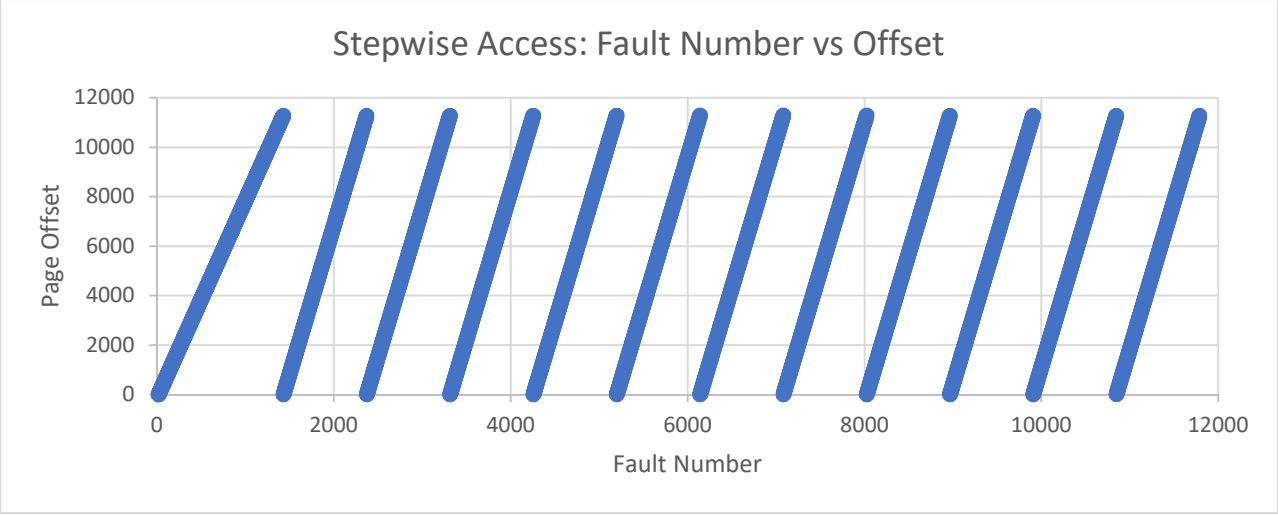
The corresponding offset accessed by the faults is given below:



The results for the fault number versus time to handle the fault for the step-wise program are given below:



The corresponding graph of fault vs offset location is given below:



Discussion/Conclusion:

The offset access patterns showed that the algorithm for accesses were indeed correct, with the random program hitting random offsets, the sequential program accessing offsets in a linear fashion, and the stepwise program jumping offsets and returning back to the beginning in a repeated pattern. With these data confirming the correct types of memory access we can further evaluate the fault times. Below is a table summarizing the fault behavior for each program type:

	Random Access	Sequential Access	Stepwise Access
Average Fault Time (us)	88.48	6.05	30.35
Average No_Page Fault Time (us)	242.76	3951.5	747.12
Total Faults	17542	11302	11771
Total No_Page Faults	6242	2	471
Total Time Faulting (us)	1552140	68396	357288
Total No_Page Time Faulting (us)	1515339	7903	351894
% of Fault Time waiting on I/O	97.62901542	11.55476928	98.49029354

The first observation is that sequential access is far more efficient than either stepwise or random access. The total time stepwise spent faulting was 5.2 times greater than sequential and random access was 22.7 times greater than sequential access spent faulting. Looking at the distribution of fault times we see it has one very long fault at the beginning and then almost never faults for a significant amount of time. This indicates that the fault handler has detected that the access is going to be sequential and therefore choose to load all the rest of the pages in the map in a single request. This is supported by the number of No_Page faults (actual calls to disk) being only 2. We also see that the demand paging system and fault handler are specifically tuned to recognize sequential access and the handler will read the entire file at once to save the high cost of seeking out the file repeatedly. The efficiency of the fault handling was also indicated by the percent time waiting on I/O as indicated by No_page time over all faulting time. Performing this analysis shows only 11.55% of the total time faulting was actually time waiting for disk I/O. The rest of the fault time is just the time required to receive the proper pages. In summary, sequential access is able to efficiently Pre-Page and save the huge costs of multiple seeks in the event the whole file is going to be read.

Stepwise shows a similar pattern, with large load times for early faults, but then almost no cost once the file is totally loaded into memory. On the first traversal, all of the pages (even the ones not directly accessed) in the file are loaded into memory and accesses after that are short. However, the fault handler does not seem able to identify even a simple step algorithm and pre-pages only a few pages ahead in the event the accesses are anything but sequential. This is indicated by the number of No_Page faults being 471. The program only pre-pages about 20 or so pages ahead of the requested page on the first traversal. This pre-paging does save efficiency however, since the time spent faulting for random access was 4.34 times higher than the time spent faulting in the stepwise case. Interestingly, the percent time doing I/O was 98.49% of the total time faulting. This indicates that the time required to retrieve a page is a small cost that is quickly overshadowed by the time to do actual I/O in the case of anything except sequential access.

Random-access was by far the most inefficient for accessing all the fields in the file. It spent far more time faulting than any of the other algorithms. It also had a total number of faults about 55% more than either stepwise or sequential. The total number of No_Page faults of 6242 shows that for the file that is about 11000 pages it only pre-paging about 2 pages or less on average for any given access. Looking at the distribution of the fault handling times however, we see that the first several hundred faults pre-paged a larger number of pages, as indicated by the high fault times early. The fault handler then didn't pre-page for most of the duration of the random accesses. Like stepwise, 97.62% of its time faulting was spent doing I/O, so retrieving the proper page once in memory is a small cost. Considering all three tests we can come to some general conclusions.

For all three experiments we see that the fault handler is tuned to balance memory usage with performance. The very fast performance of the sequential case shows that the fault handler is tuned to recognize the common case of sequential access and will pre-page a large file in order to save time faulting. The stepwise and random-access distribution of fault times also show that on repeated early accesses (first 1000 or so faults) the system pre-pages a larger number of pages ahead of the pages accessed. This is likely an attempt to deal with patterned forms of access (such as stepwise). This also implements the principle of locality, which says your access are most likely going to occur within the area you just accessed. After thousands of random accesses however, the fault handler gives up on locality and chooses to stop pre-paging entirely and simply load what is accessed directly in an attempt to save memory.

In conclusion the experiments confirm my hypothesis that sequential access would be the most efficient in terms of time spent faulting followed by stepwise. Random was by far the least efficient. The experiments also show that demand paging is specifically tuned to balance memory usage and performance. The random experiments illustrated this perfectly. If the goal was only to optimize the performance of any one process, then on a random access you would simply load the whole file into memory at that time. Instead the system pre-pages more on early accesses to account for possible locality and then chooses to eliminate pre-paging on later accesses to save memory. The experiments also give some interesting insights into how a programmer should deal with mapped files. In general, if you are going to access a file randomly, it may be more efficient to traverse the whole file sequentially once before doing random accesses. This could potentially save you large amounts of fault overhead in the future when you do your random accesses. This would increase your programs memory usage drastically however, if you are only going to access a small portion of the file. Overall, the experiments showed great insight into the fault handler behavior in a demand paging operating system.

Honor Pledge:

I have neither given nor received unauthorized aid on this assignment.

Samuel George