Experimental Analysis of different aspects of Virtual Memory and its Optimizations [*]

Hassan Ali Khan[†]
LUMS - 15030044
SSE CS Department
15030044@lums.edu.pk

ABSTRACT

This paper presents experimental results of different aspects of virtual memory system and its optimization; he effect of TLB hits on page access time of a program (with base page of 4KB and Huge Page of 2MB), demand paging and its effect on overall execution time, cost of large pages allocation with and without page swapping on memory pressure and the time it takes for the allocation of pages in virtual memory(not present on RAM), for both base page and huge page. In following sections, code for the experiments and the graphs are shared but brief results in above mentioned cases are as follows. After rigorous experiments, I calculated the average of assumed possible cases and concludes that TLB if ignored is causing significant performance degradation. Huge pages have their pros and cons as well as they are good when memory is not fragmented but they put a lot of memory pressure. Next I observed the effect of page out on memory allocation and it effects the total physical memory a process can work on as in absence of it program was killed by OS for filling up most of the RAM. Section 1 will introduce the platform used and overviews details of compiler and timer used in above mentioned experiments. Section 2 explains the methodology of experiments and Section 4 will share results.

Keywords

TLB; Huge Page; Base page; Demand Paging; Page Swapping; Zero-out Page

1. INTRODUCTION

In this study I did experiments on the cases for virtual memory as the effect of TLB hits on page access time of a program (with base page of 4KB and Huge Page of 2MB), demand paging and its effect on overall execution time, cost of large pages allocation with and without page swapping on memory pressure and the time it takes for the allocation of pages in virtual memory(not present on RAM), for both base page and huge page.

The results convinced me for the following observations; TLB has a significant effect on program execution as in some tests the performance overhead was exponential when process was not using TLB efficiently. In second case, I have proved demand paging is by default page allocation scheme in Ubuntu 15.10(Linux distribution). Cost is measured by two factors; Memory pressure and Page faults. Huge pages are good with saving page fault penalty but increases memory pressure and vise verse for base pages(with and without demand paging). After all this, I observed effect of demand allocation with multiple page sizes and it turned out to be better than memory allocation at instantiation time in terms of memory pressure.

2. METHODOLOGY

Details for OS and machine used in this study are
-i5 3.3GHz with 4 cores
-8GB of RAM
-L3 cache of 3MB, L2 cache of 25KB and L3 of 32KB
-Ubuntu 15.10(64 bit)
-4KB base page size and 2MB huge page size
-16GB maximum swap file size
-L1 TLB with 64 enteries, L2 TLB with 1024 enteries(for base page)
-l1 TLB with 8 entries, L2 TLB with 1024 entries(for huge page)
- RDTSC timer is used which gives the clock cycle of around nanoseconds. Its own execution time is of few seconds
Following are the details of experiments to validate my results.

2.1 Size of TLB for Base page and Huge page

In this part of study I allocated 30MB of space using mmap() with base page size of 4KB and accessed pages in descending order to keep the starting 64 pages in L1 TLB for a TLB hit in next step, then I accessed those 7680(30*256(pages in one MB)) in ascending order to observe the effect of L1 and L2 TLB and as expected I had spikes in form of CPU cycles consumed for page access at around 65th page and then around 1024th page. I repeated the experiment multiple times to get more accurate results same are the results for huge page of 2MB as showing evident increase in consumed clock cycles after accessing 8 pages.

[*](Produces the permission block, and copyright information). For use with SIG-ALTERNATE.CLS. Supported by ACM.
[†]Dr. Trovato insisted his name be first.

WOODSTOCK '97 El Paso, Texas USA
© 2016 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

Important observations in this part were you have to bind the process to the same CPU to get consistent results and observe effect of TLB and cache correctly as in case when I don't bind the process it might get its next time quantum on other processor and longs delays in page access might result even when pages are present in TLB because of absence of those pages in L1 cache. Spikes in transition from L1 TLB to L2 TLB were relatively low because L2 TLB is present of chip whereas L2 TLB has to fetch from the RAM which causes longer delays.

2.2 Memory Allocation for Virtual Memory

I have experimented demand paging by allocating 10MB of memory using mmap() and then used mincore() system call by passing the starting address of allocated memory and vector of size equivalent to number of pages in size of newly allocated memory. The mincore() edits the vector array passed to it by changing the bit of respective page number to 1 in case if the page is available on RAM otherwise make it 0. I later used the same vector array to find if page was present on RAM or not and according to description available in man pages of ubuntu the bits corresponding to the number pages which were not accessed after mmap and before mincore were 0. Which proved that the default page allocation policy is deferred allocation. I repeated the experiment with different test cases such as many accesses before mincore to no access before mincore. I was unable to produce an observable result for second part of this question i-e; when does OS zero out the pages which is initializing the allocated memory with zeros for security reasons; approach I third for this part is, I pointed at random location of allocated memory and after munmap() I tried to deference the pointer to see if it zero out the freed memory or not but it caused a segment fault as I was trying to access an illegal memory in perspective of that running process. C API for allocating and deallocating memory were not right option as even when we free a malloc() array the compiler might not return the memory to OS and keep it in heap with the expectation that it can be used in future. On possible approach could be convert the virtual address of specific page to physical address and look up that address on terminal with kernel commands. I was not able to implement that approach.

2.3 Cost of Using Huge Pages

I know the benefits of huge pages with respect to less page faults already but in this experiment I tried to find the cost of allocating a huge page against base page and the maximum amount of memory that a huge page and base page mmap() can allocate for a process. As expected the cost of allocating a huge page was relatively less than of base page; I allocated the 50MB memory space in first part of this question to find the cost of allocation in both page sizes. In case of huge page there were only 25 pages information we had to insert in Page table where as in 4KB case we had to update meta-data with 12500 new entries if new pages in page table which in turn proves that huge page allocation is faster and cost effective in allocating a memory. The other part of this activity proves some drawbacks of huge pages where we were trying to allocate infinite amount of memory with both page sizes and at the same time accessing those pages with assigning random data at random locations in all the pages to make sure all pages are assigned with frames on RAM, as most of the times RAM is fragmented so allocating 2 MB page is not possible in all cases which in turn proved that process with page size of 4KB was able to get more memory before being killed by the processor. In second run of this experiment I used mlock() to avoid swapping out of pages of processes in result of that experiment processes allocated relatively less memory than the last experiment but the relation between processes with 4KB page size and 2MB page size was same as 2MB process was not able to allocate enough space as of 4KB process .

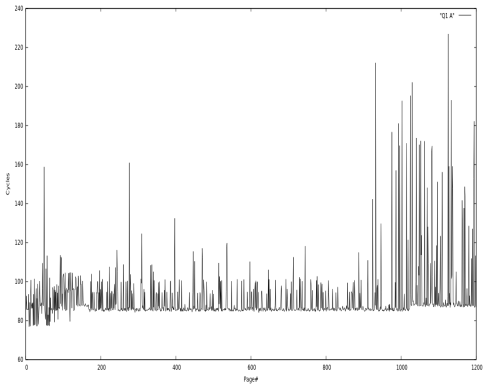


Figure 1: TLB effect in case of 4KB page size

2.4 Importance of Demand Paging(an Optimization in Virtual Memory)

In this experiment of the report I tried reserving and accessing pages in process;for this particular activity I allocated 100MB to find the trade-off between virtual memory with and without demand paging. In separate programs I reserved memory using mmap() with and without MAP POPULATE flag which reserves memory for pages in RAM even before I access all of them. There are two fold to the output of this experiment in case of demand paging there were more page fault which resulted in more consumption of CPU cycles than the process in which we turned demand paging off. But the process with demand paging turned off caused too much pressure on RAM so this scheme should not be the first preference when we have to program data dependant applications. Following are the results and explanation of all experiments.

3. RESULTS

3.1 Size of TLB for Base page and Huge page

In Figure 1 effect of TLB for page size of 4KB is evident such that it is taking a jump on access of 64 pages and a second visible spike near access of 1024 pages. Which in turn proves the TLB L1 size to be 64 enteries and TLB L2 size to be 1024 entries using code of Fig 3. Same way I found the output for huge page TLB too which is 8 entries.

3.2 Memory Allocation for Virtual Memory

Next is the validation of demand paging as explained earlier I allocated memory and use mincore to decide if memory for that page is allocated on ram too or not Figure 4 is the main code for this experiment.

It gave the output from 1 to 10 as these are the only

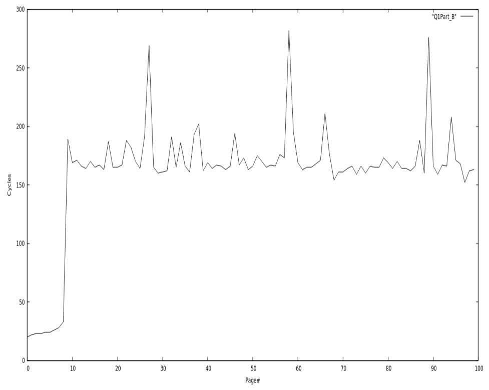
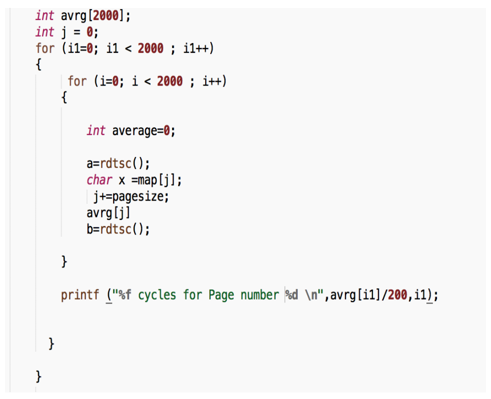
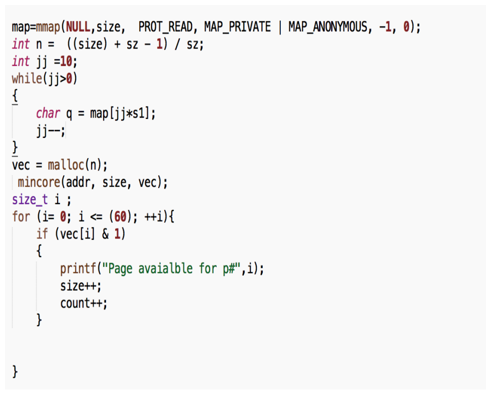


Figure 2: TLB effect in case of 2MB page size



```
int avrg[2000];
int j = 0;
for (i1=0; i1 < 2000 ; i1++)
{
    for (i=0; i < 2000 ; i++)
    {

        int average=0;

        a=rdtsc();
        char x =map[j];
         j+=pagesize;
        avrg[j]
        b=rdtsc();

    }

    printf ("%f cycles for Page number %d \n",avrg[i1]/200,i1);


   }

}
```

Figure 3: Code to find average access time of page



```
map=mmap(NULL,size,  PROT_READ, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
int n =  ((size) + sz - 1) / sz;
int jj =10;
while(jj>0)
{
    char q = map[jj*s1];
    jj--;
}
vec = malloc(n);
 mincore(addr, size, vec);
size_t i ;
for (i= 0; i <= (60); ++i){
    if (vec[i] & 1)
    {
        printf("Page avaialble for p#",i);
        size++;
        count++;
    }


}
```

Figure 4: Code for Mincore use

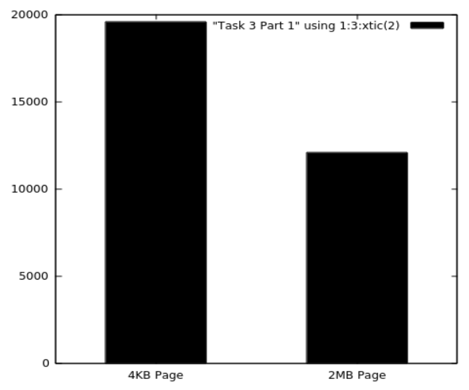


Figure 5: Time to allocate memory with 4KB page size and 2MB page size

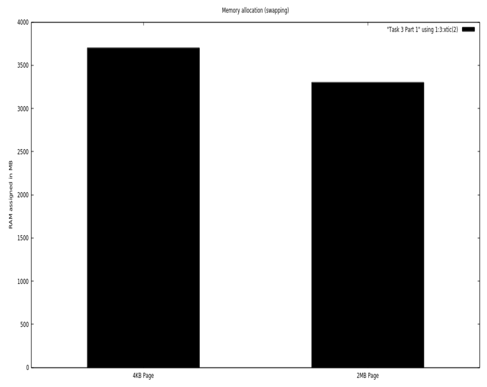


Figure 6: Max allocation with demand paging

pages touched in code. and I was not able to find the time when the memory was zeroed out as physical address against virtual address I was getting in my code was correct and I was unable to monitor the memory location.

3.3 Cost of Using Huge Pages

In first part of this experiment(Figure 5) I came up with the number to allocate memory for both base page and huge page. The cycles taken by allocation with base page were more than cycles consumed by allocation with huge page.
For the second part I did the experiment with and without swapping using mlock() which make the page to stay in memory till the end of program and not allowing to swap it with memory. With swap I was able to assign 3.6GB for the process with page size as 4KB and 3.3GB for the program with page size of 2MB for allocation. I have explained the reason for the difference in the previous section(Fig 6). In case of allocation with demand allocation turned off the max memory allocated in both cases is reduced but relation remains same as 4KB page size program can allocate more space(Fig 7).

3.4 Importance of Demand Paging(an Optimization in Virtual Memory)

In this experiment(Figure 8) I compared the total cycles consumed in access of all the pages of 50MB for both the page sizes, I did bind the process to the same CPU for all

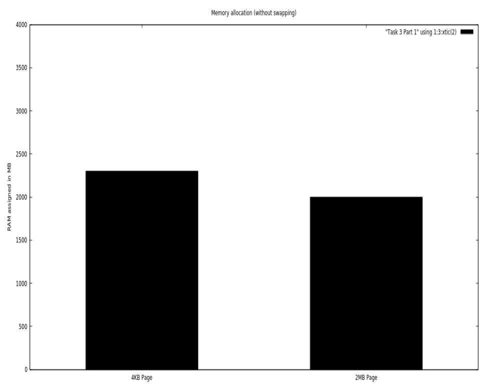


Figure 7: Max allocation without demand paging

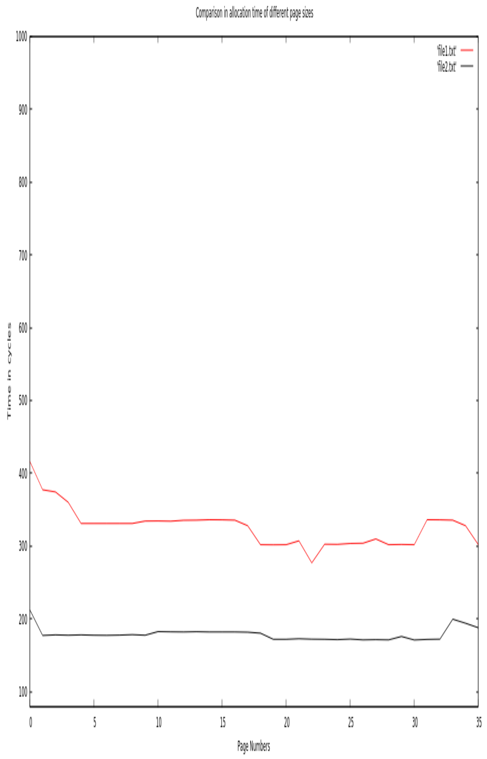


Figure 8: Time to allocate memory with 4KB page size and 2MB page size

the iterations and cleared cache before each iteration for consistent results. Because of resident pages in RAM the page faults number is minimum in the case where I have switched off demand paging by MAP POPULATE flag in mmap. While in the other case the access speed is relatively slow.

4. CONCLUSION

To sum up whole study I would like to state the figures in results of above experiments; I found the size of L1 and L2 TLB for 4KB as 64 and 1024 entries, for 2MB page size L1 and L2 TLB was 8 and 1024 entries I confirmed the test results with cpuid command on Linux. Next I proved that until a page is accessed it is not RAM resident. Second last experiment was for checking the cost of allocating and maximum possible allocation with huge page size in my system which gave result which imply due to fragmentation maximum memory a process with huge page size can allocate is smaller than than of with base page size and in case of no swapping the maximum number came down further. Lastly I experimented on efficiency of allocation scheme without demand paging and discussed that in case of memory extensive processes touching rarely many pages and mostly works on very few pages. Overall the the benefits of virtual memory are much more than trade offs of using it further these optimization like pre-fetching, demand allocation and code sharing make the process take full advantage RAM and even in case where RAM is less than the application memory virtual memory with paging helps app to think they have TBs(in case of 64bit OS) of RAM available to them. Problem of hot pages and cold pages is catered by TLB very efficiently that is after some time few hot pages become available in RAM and TLB as well while cold pages are eliminated and swapped out of TLB and RAM respectively.With the help of different page sizes I can minimize the external and internal fragmentation and in most of the cases avoid page faults too according to the requirements of process.