CS695 Assignment 4

Mridul Agarwal & Shantanu Welling
210050100 & 210010076

3rd May 2024

Problem

The Working set size gives an estimate of the amount of memory a process requires to keep running without swapping the memory contents to the disk. It is possible that our process has been allocated GBs of memory to it, however it just uses around 10-15Mb actively, and rest of it quite rarely That's the working set size: the "hot" memory that is frequently used. It can be worth knowing if there is a significant difference between a process's RSS and the amount of memory actually in use.
Why do we need to measure WSS?

- You are sizing main memory for an application, with the intent to keep it from paging (swapping). WSS will be measured in bytes over a long interval, say, one minute.
- You are optimizing caches, WSS will be measured in unique pages (or cachelines) accessed over a short interval, say, one second or less.
- Optimizing the performance of programs by identifying memory usage hotspots
- this information can be helpful when partitioning the system between containers or setting control-group limits

Approaches

A general approach to estimate WSS of any process, is to unset some kind of flag, every polling interval, set it whenever a particular portion of memory is accessed, and get an aggregate of the number of memory locations accessed every polling interval. Following are the 3 specific approaches that the linux kernel uses/can use to estimate the Working Set Size.

Linux Reference Page Flag

This approach uses the ability to set and read the referenced page flag from user space. The referenced page flag is really the PTE accessed bit (_PAGE_BIT_ACCESSED in Linux). A page is considered referenced if it has been recently accessed via a process address space, in which case one or more PTEs it is mapped to will have the Accessed bit set, or marked accessed explicitly by the kernel. The tool used to estimate using this approach, takes as input, the pid of the process, whose WSS is to be estimated, and the polling interval

1. Reset referenced page flags for a process
2. Sleep for the duration of polling interval

3. Read referenced page flags

This reset is done by writing a value of 1 to the clear_refs file in the same /proc directory. The problem with this approach is that the linux, uses the reference flag to swap pages to memory. The "referenced" state of each page is used by the memory-management subsystem itself to make decisions on which pages to evict. Resetting every page to the "not referenced" state will thus perturb page reclaim, and probably not for the better. If a page's access bit is unset, the kernel considers this as a potential page to be swapped to the disk in case of memory pressure. So, on explicitly unsetting this bit, the kernel might swap the working pages to memory when other, idle pages our available in the memory. If these measurements are to be made often, it would be good to have a less invasive way to make them.

Linux Idle Page Flag

This approach uses Idle and Young page flags for more reliable working set size analysis, and without drawbacks like changing the referenced flag which could confuse the kernel reclaim logic. These extra idle and young flags are only in the kernel's extended page table entry (page_ext_flags). The idle page tracking API is located at `/sys/kernel/mm/page_idle` . The file implements a bitmap where each bit corresponds to a memory page. The bitmap is represented by an array of 8-byte integers, and the page at PFN #i is mapped to bit #i%64 of array element #i/64, byte order is native. When a bit is set, the corresponding page is idle. In order to estimate the amount of pages that are not used by a workload one should:

1. Mark all the workload's pages as idle by setting corresponding bits in `/sys/kernel/mm/page_idle/bitmap`. The pages can be found by reading `/proc/pid/pagemap`
2. Wait for the polling interval
3. Read `/sys/kernel/mm/page_idle/bitmap` and count the number of bits set.

The kernel internally keeps track of accesses to user memory pages in order to reclaim unreferenced pages first on memory shortage conditions.
When a dirty page is written to swap or disk as a result of memory reclaim or exceeding the dirty memory limit, it is not marked referenced.
When a page is marked idle, the Accessed bit must be cleared in all PTEs it is mapped to, otherwise we will not be able to detect accesses to the page coming from a process address space. To avoid interference with the reclaimer, which, as noted above, uses the Accessed bit to promote actively referenced pages, one more page flag is introduced, the Young flag. When the PTE Accessed bit is cleared as a result of setting or updating a page's Idle flag, the Young flag is set on the page. The reclaimer treats the Young flag as an extra PTE Accessed bit and therefore will consider such a page as referenced.

Invalidating Cache Mappings

This approach was presented in the [memmgmt] paper. Each sampled page is tracked by invalidating any cached mappings associated with its PPN, such as hardware TLB entries, present bit in page tables and virtualized MMU state. The next guest access to a sampled page will be intercepted to re-establish these mappings, at which time a touched page count t is incremented. At the end of the sampling period, a statistical estimate of the fraction of memory actively accessed by the VM is $f = t/n$. This approach requires setting up page fault trap handlers and modifying the kernel.

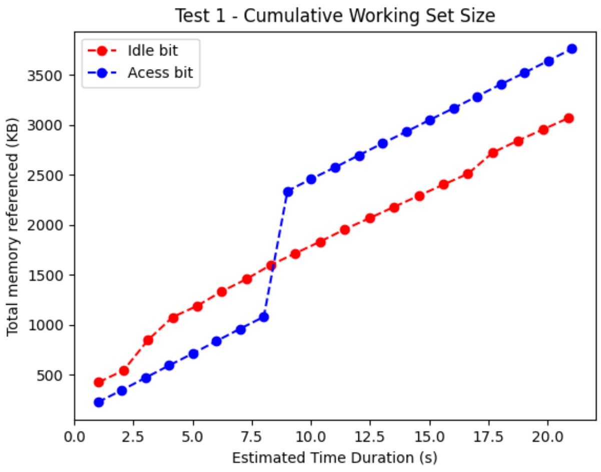
Experiments

We experimented majorly on two type of workloads, and study the WSS pattern over different polling times, and a moving average of the WSS pattern for predictive analysis

Workload 1

This is a constantly increasing trace of memory. We keep allocating and accessing new integer pointers.

```
int a[10000];
int j = 0, n = 1000;
for(int i=0; i<=INT_MAX; i++){
    int *new_int_ptr1 = new int(i);
    int *new_int_ptr2 = new int(i);
    int *new_int_ptr3 = new int(i);
    int *new_int_ptr4 = new int(i);
    a[(i+1)%10000]=*new_int_ptr1;
    a[(i+2)%10000]=*new_int_ptr2;
    a[(i+3)%10000]=*new_int_ptr3;
    a[(i+4)%10000]=*new_int_ptr4;
    usleep(1000);
    if(i==INT_MAX && j!=n) {i=0; j++;}
    else if(i==INT_MAX && j==n) break;
}
```



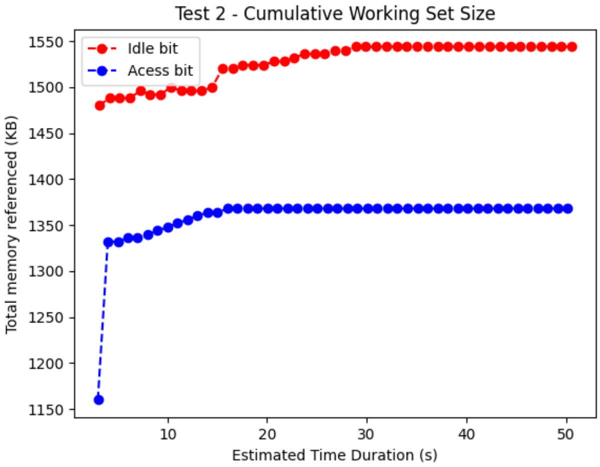
Cumulative WSS for Workload 1 for a polling interval fo 20s

Workload 2

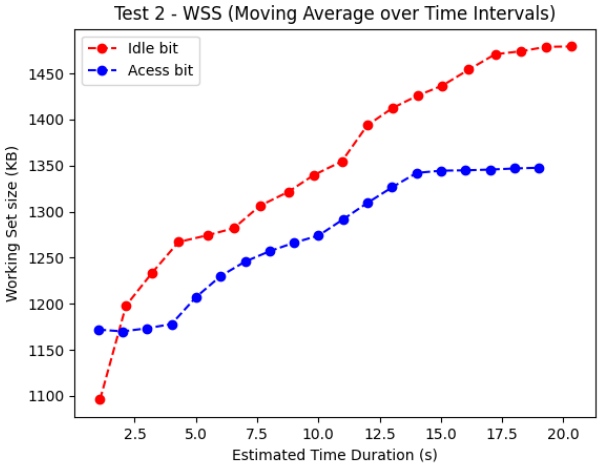
This workload, initializes a big array of multiple pages. This then loops over them and accesses the pages. We however ensure everytime we access the page from memory, by flushing the cache, before every access. The WSS of this workload saturates, when a large polling period is used.

```
int arrsize=30;
int a[1024*arrsize];
```

```
    sleep(10);
    for(int i=0; i<arrsize; i++){
        system("sudo sync > /dev/null 2>&1");
        system("sudo sysctl -w vm.drop_caches=3 > /dev/null 2>&1");
        a[i*1024]=i;
        usleep(50000);
        if(i==arrsize-1) i=0;
    }
```



Cumulative WSS for workload 2 for a polling interval of 50s



Moving average for WSS with a polling interval of 1s

Interesting Question to Explore

Does the bitmap/PTE entries also get modified, when the cache optimizer accesses nearby physical pages?

References

- https://www.brendangregg.com/wss.htmlWSSProfileCharts

- https://www.kernel.org/doc/Documentation/vm/idle_page_tracking.txt
- https://www.cse.iitb.ac.in/ puru/courses/spring2023-24/