

MP3 Report

Virtual Memory Page Fault Profiler

Abhinav Sharma(sharma55)
Saikat Roychowdhury(rychwdh2)
Shyam Rajendran(srajend2)

Implementation details

Resources Registration:

In the module init function, we initialize all the resources that will be needed by the module.

1. Proc Entry - Create proc entry mp3/status and registers file operations for read/write. Process register their process ids by writing in the status proc file. Any process can also read the list of registered process by 'cat /proc/mp3/status' command.
2. Samples buffer - We allocate a buffer for storing samples of minor/major page faults information using vmalloc call. The size of the buffer allocated is 128 pages. (128 * PAGE_SIZE). Thus this buffer can store upto 128 * PAGE_SIZE / sizeof(unsigned long) samples.

(For example if page size is 4096 bytes and size of unsigned long is 8 bytes the buffer can hold 65536 values.). Buffer is flushed to 0 initially using memset().

3. Character device - we first allocate a device number dynamically for the character device using alloc_chrdev_region() call. Once the major device number is obtained, we register a character device using cdev_alloc() call. We register file operations callback for open, release and the most important mmap call. The open and release functions are just empty implementations. This character device is used as an interface by userspace application to map the samples buffer into its own address space. When user app calls mmap() with the proper character device pointer, kernel module gets a callback to _cdevice_map() method.

Data Structures:

Augmented PCB - We create an augmented process control block to store process related meta-data information. This structure stores the following information:

- a. pid - process id
- b. cpu_util - cpu utilization time
- c. minor_fault - number of minor faults
- d. major_faults - number of major faults
- e. task_pcb - process control block of the process(pointer to kernel data structure)

A linked list of this struct is maintained and accessed synchronously using mutex lock.

Memory Mapping:

- a. Our kernel module exposes interface to allow userspace apps to map the samples kernel mode buffer in their address space. On module init, we register a character device. The major device number for this device can be accessed by performing cat

operation on the /proc/devices. The character device is exposed with the name MP3_CDEV

- b. For mapping the samples buffer, we use `vmalloc_to_pfn()` and `remap_pfn_range()` function. This mapping is done on a per page basis. Since we allocated 128 pages in the module init, we run a loop for 128 pages and first get the base virtual pointer of each page(`samples_buffer + i * num_elements_per_page`) and get its corresponding physical address using `vmalloc_to_pfn()`. We then use `remap_pfn_range()` to map the physical address base pointer of each page and map it to the struct of type `vm_area_struct*` which is passed as input to the function call.

Work Queue implementation:

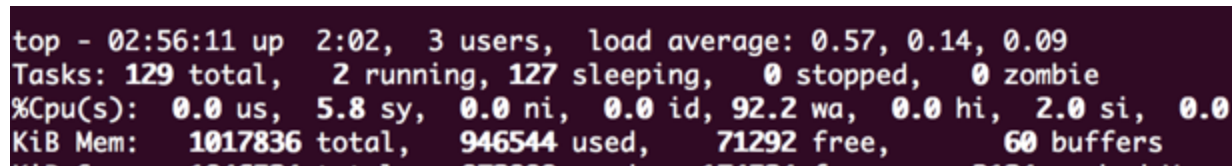
When a process gets registered with our kernel module, we check if this is the first process to be registered in the linked list of registers PIDs(Process ID). If so, then we flush the samples buffer to zero(since it may contain stale information from previous runs) and create a workqueue. A delayed work is then scheduled to run after an interval of 50 milliseconds.(20 times per sec). Once the work is completed another delayed work gets scheduled to run after 50 milliseconds. This goes on until the last process is deregistered and removed the linked list of registered PIDs.

In the delayed work function callback we scan through the list of processes(synchronously using mutex lock) and get stats like major faults, minor faults, utime and stime (using the provided helper function `get_cpu_use()` helper function) for each registered process. These stats for all the process are accumulated and written into sample buffer. We maintain a global write index for the sample buffer, which is incremented every time the samples are written into the buffer. We write four sample values every time, (jiffies, minor fault number, major fault number and the utime+stime) and increment the write index by 4. The write index wraps around if it reaches the end of the buffer.

Analysis of log

Case study 1:

Both processes allocate 1GB memory each. The test machine on which the program is run has only 1gb of RAM(as shown in the screenshot (fig 1) of 'top' command while the program was in execution).



```
top - 02:56:11 up 2:02, 3 users, load average: 0.57, 0.14, 0.09
Tasks: 129 total, 2 running, 127 sleeping, 0 stopped, 0 zombie
%Cpu(s): 0.0 us, 5.8 sy, 0.0 ni, 0.0 id, 92.2 wa, 0.0 hi, 2.0 si, 0.0
KiB Mem: 1017836 total, 946544 used, 71292 free, 60 buffers
```

Fig 1

In both work1 and work2, we observe thrashing. Below are the log plots of minor and major page faults.

Profile 1: (nice ./work 1024 R 50000 & nice ./work 1024 R 10000 &)

Profile 2: (nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &)

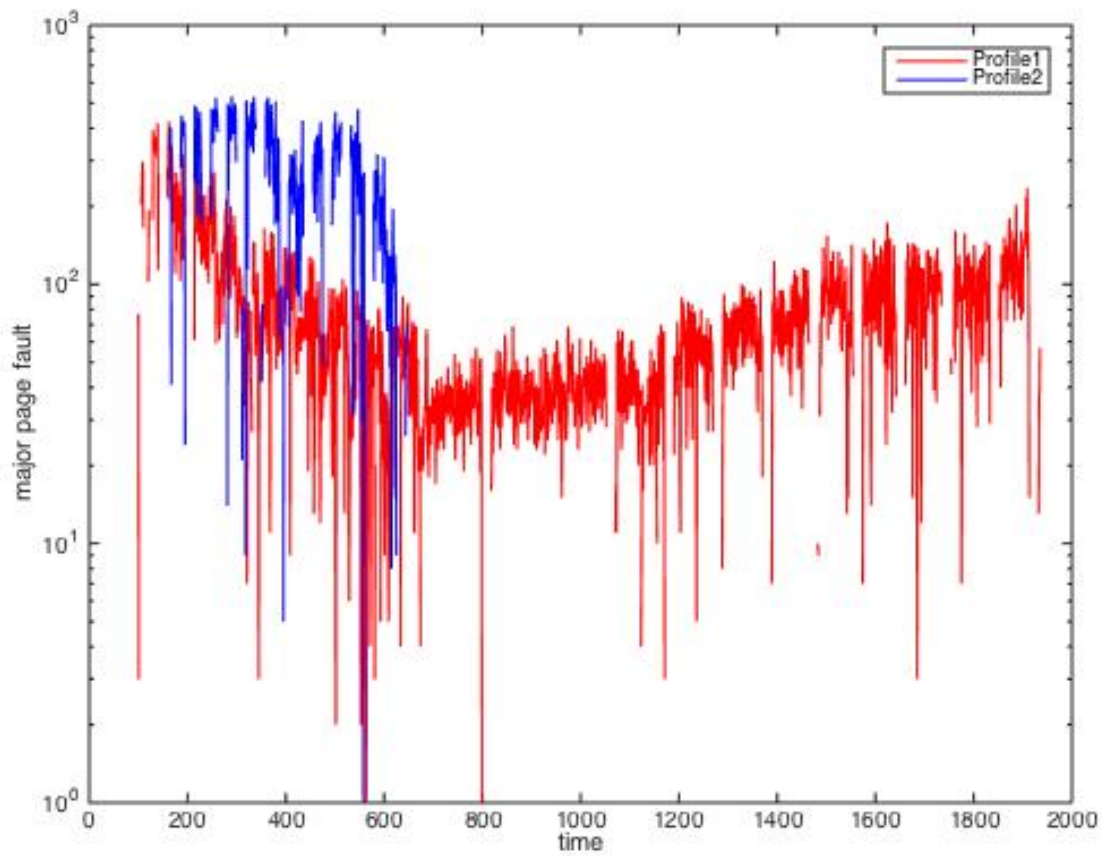


Fig 2

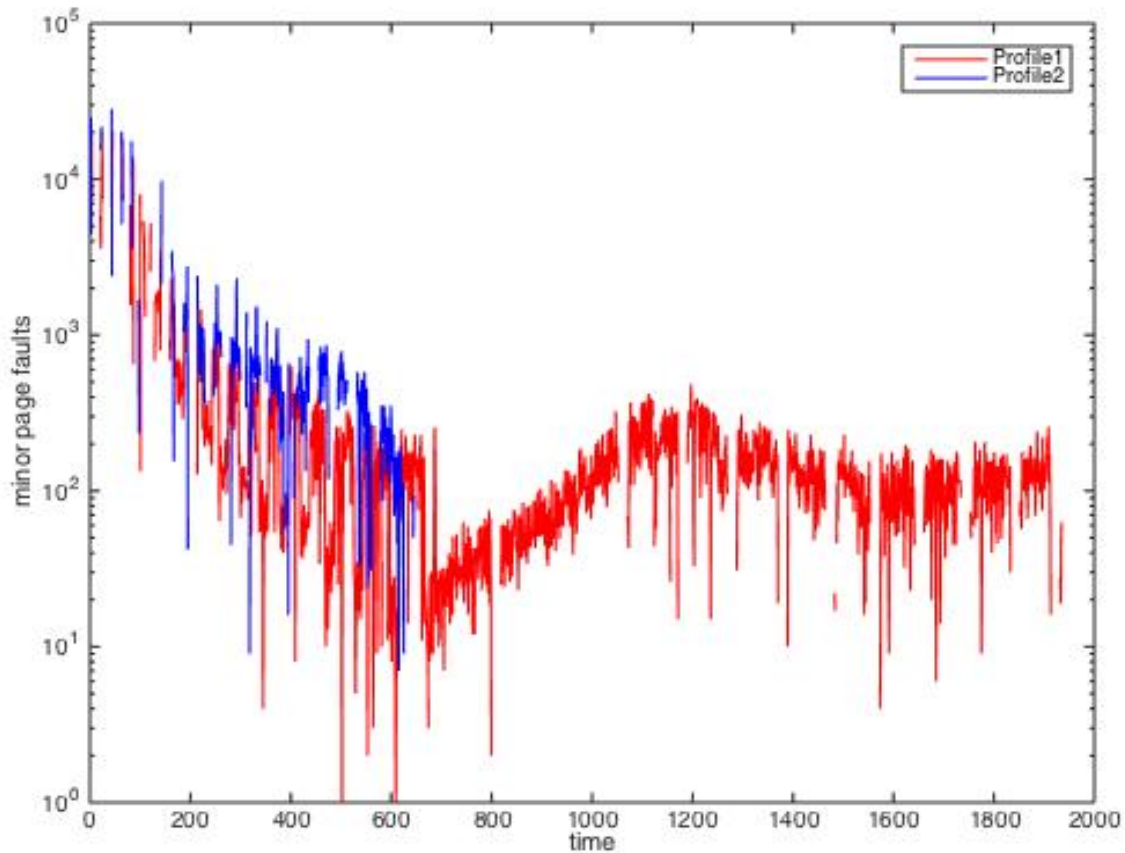


Fig 3

Both the profiles are allocating more than the system memory which is causing page faults. The completion time for **Test case 2**(shown in blue, nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &) is much lesser as compared to **Test case 1**(shown in red, nice ./work 1024 R 50000 & nice ./work 1024 R 10000 &). The completion time for test case 2 may be lesser because the time spent in swapping in pages is lesser for a program which shows locality behaviour in accessing pages.

We however see a discrepancy in the **average number of page faults** (Fig 4) for test case 2, as with better locality behaviour the expected number of page faults should have been lesser than the test case 1.

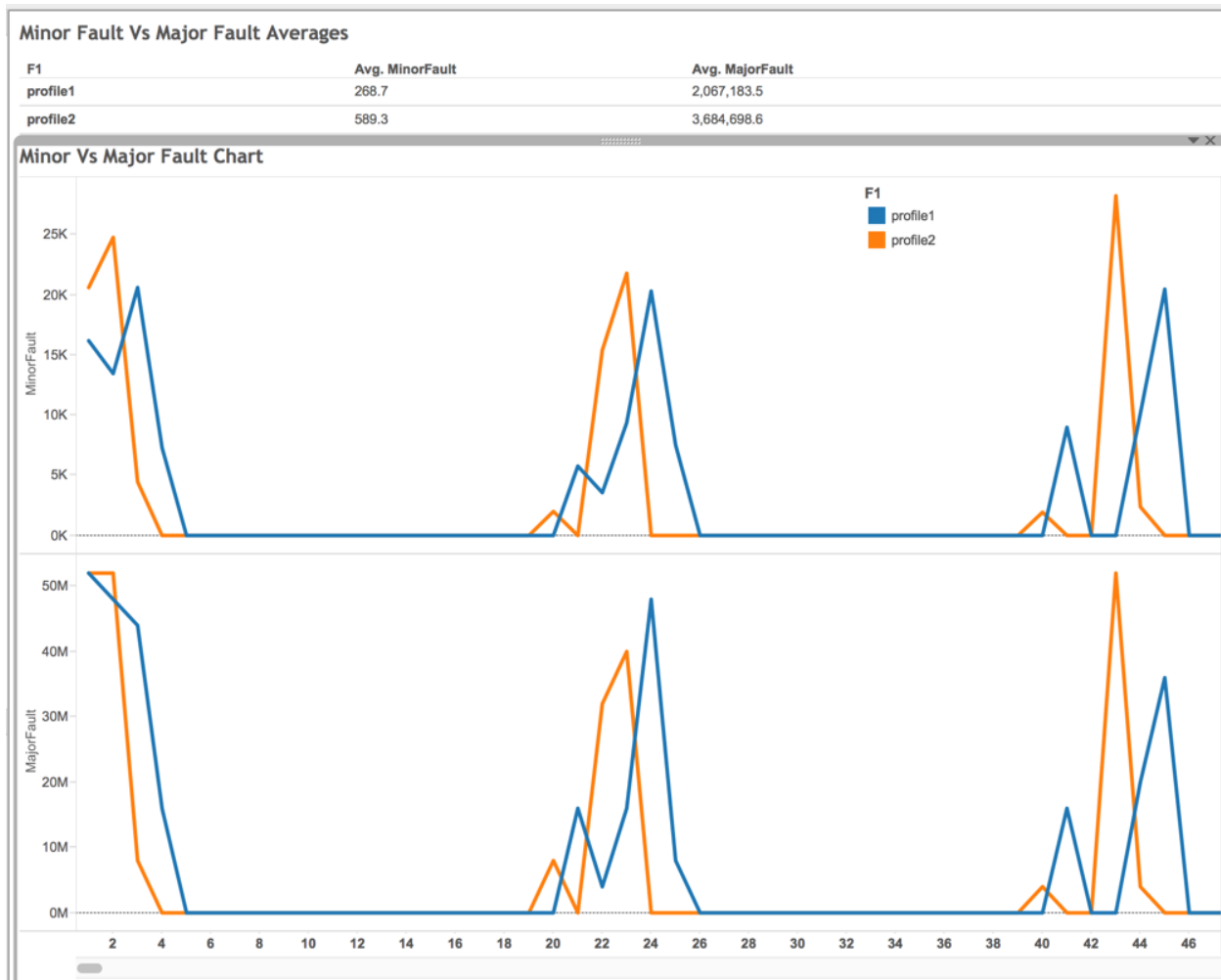


Fig 4

online chart available @

https://public.tableau.com/profile/publish/Plot1VsPlot2/Profile1_Profile2Chart#!/publish-confirm

Case Study 2:

From the above plots we can see that the degree of multiprogramming goes down as thrashing occurs. Starting new threads reduces the page frames available for each thread and hence increases the page fault requests. System throughput plunges as more pages are swapped to and fro and this affects the degree of multiprogramming.

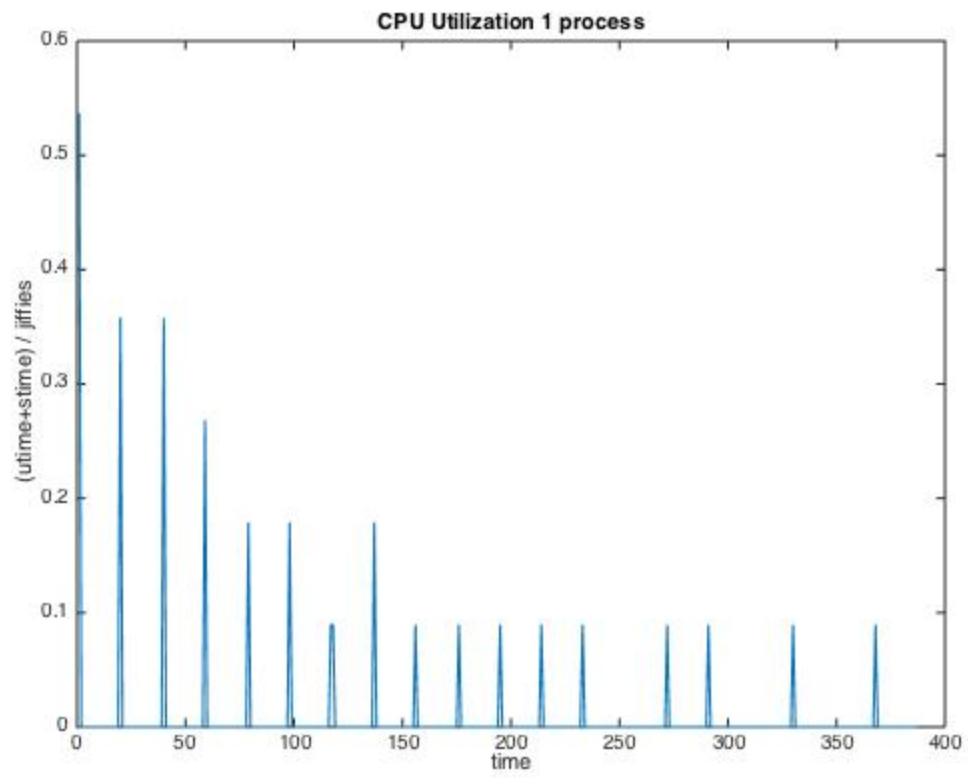


Fig 5

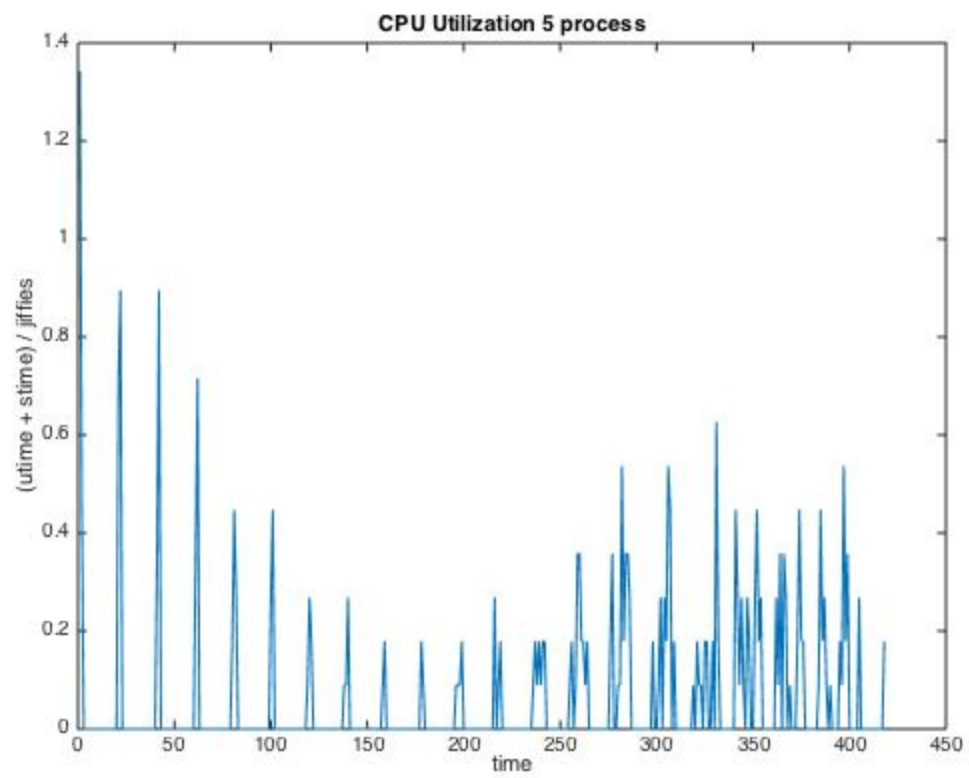


Fig 6

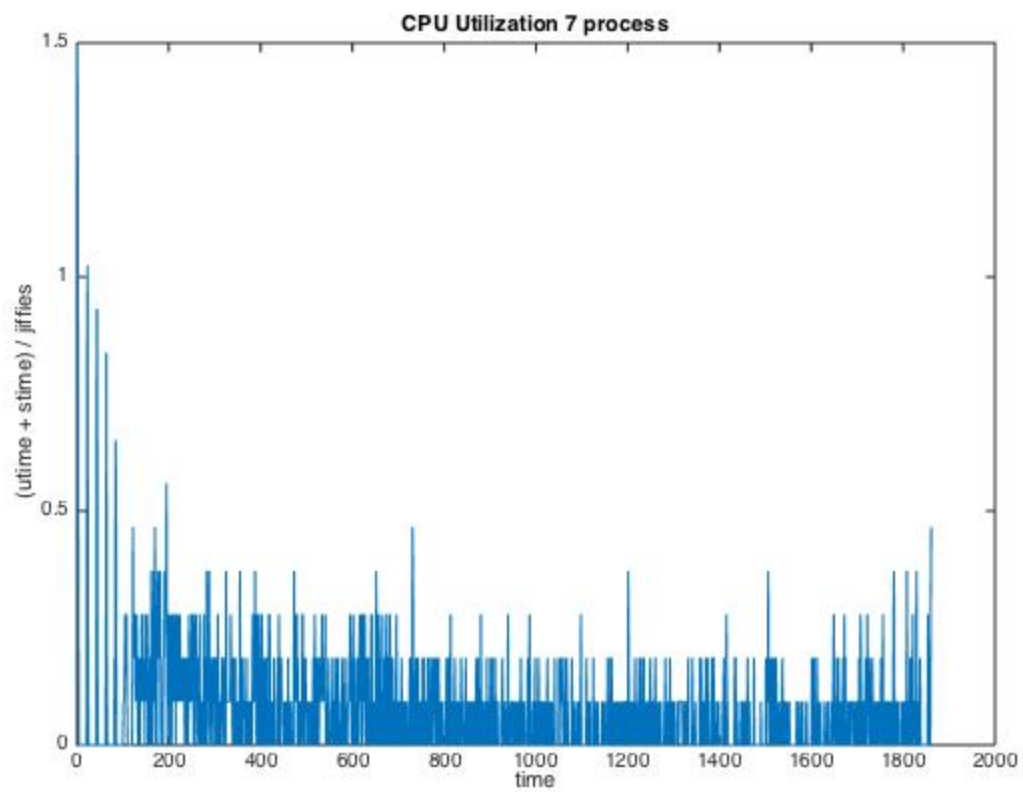


Fig 7