Hao Zheng hzheng19
Zhili Feng zfeng6
Joo Won Lee jlee164

Implementation

First, we got the basic skeleton from MP1 and MP2.

For “mp3.c”, we kept the basic structure almost identical to MP2. We had a struct called ‘task_node_t’. This struct contains a task_struct pointer, a process id, list_head for the linked list structure, and minimum fault and maximum fault variable and lastly, the cpu_utilization, which is the process utilization variable.

Then, we basically declared the same variables as MP2 with some additions: delayed_workqueue pointer, memory buffer pointer, buffer iterator, cdev pointer, and dev_t variable. After that, we created a mutex for the process list, a task_struct for the dispatching thread and initialized our task list.

In the function ‘update_process_info()’, we call the given ‘get_cpu_use()’ on each of our nodes and compute the corresponding min fault, max fault and cpu_utilization variable.

In the function ‘write_process_to_shared_mem_buffer()’, we add each node’s total major fault, minor fault, and cpu_utilization variables to our global variables. To store this, we use a 4-index partition; in the first index, we store the jiffies; in the second, we store the minimum fault, in the third, we store the major fault, and in the fourth and last, we store the cpu_utilization. If we go over the boundary (A.K.A. if the page overflows), we set the index back to 0.

The function ‘_delayed_workqueue_init()’ initializes the ‘measure_info_obj’ variable, which can be done using kmalloc.

The function ‘_measure_info_worker()’ calls these three functions in the order listed above.

The function ‘mp3_read()’ is extremely similar to ‘mp2_read()’ for MP2 in every single way.

‘init_node()’ simply enables us to find a task by their pid and set it to be the new task.

The function ‘add_to_list()’ is similar to the function of the same name for MP2 but for MP3, it only adds the new node to the tail. It does not have any states like MP2 did.

The function ‘destruct_node()’ destroys the node at a given point.

The function ‘find_task_node_by_pid()’ returns the node that corresponds to the given pid. This function traverses through the list and looks for the node with the pid. If there are no nodes with the given pid, we simply return NULL.

The function ‘reg’ adds the new process to the PCB linked list and creates a workqueue job if it is the first one. This is done by calling the function ‘_delayed_workqueue_init()’.

The function ‘delete_workqueue()’ checks if there are any delayed workqueues. And if measure_info_obj is not NULL, we call the ‘cancel_delayed_work()’ function on measure_info_obj then free the memory space for it. After that, we proceed to flush and destroy the workqueue.

The ‘unreg()’ function deletes the requesting process from the PCB list. It does so by finding the node and then calling the ‘destruct_node()’ function on that node.

In ‘mp3_write’, the beginning part is very similar as ‘mp2_write’ except we do not do a ‘admission control’ like we did for MP2. We also removed a conditional check where buf[0] = ‘D’.

The ‘cdev_mmap’ function keeps looping while there is a page to be processed. In the loop, we use ‘vmalloc_to_pfn()’ function to get the physical address of a virtual address of the buffer. Then, the function ‘pfn_to_page’ gives us the struct page pointer for the given page frame number. After that, we set the bit to PG_reserved and proceed to remap kernel memory to userspace. Using this function, we add up all the appropriate values accordingly.

The function ‘init_cdev’ calls ‘calloc_chrdev_region()’ to register a range of char device numbers. Upon this, we initialize my_cdev, which is the cdev we will use.

The function ‘allocate_shared_mem_buffer’ initializes the shared memory buffer, and use SetPageReserve() to set its PG_reserved bit.

‘mp3_init’ is invoked when a new module is loaded. This is also very similar to ‘mp2_init’, but instead of keeping track of the currently running task, we create our workqueue, allocate memory space for our memory buffer and initialize our iterator and cdev.

‘mp3_exit’ also executes similar things as ‘mp2_exit’, clear up all the memory. But the main difference is, instead of calling the ‘kthread_stop()’ function, we use the ‘delete_workqueue()’ function defined above and ‘vfree()’ our memory buffer. In addition, we also call ClearPageReserve() to clear the reserved bit.

To run the program

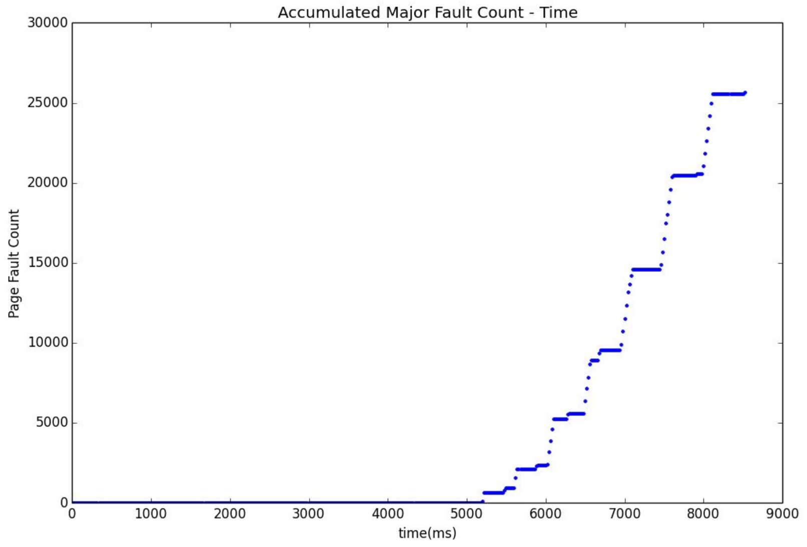
Type ‘make’ in terminal to compile the code.

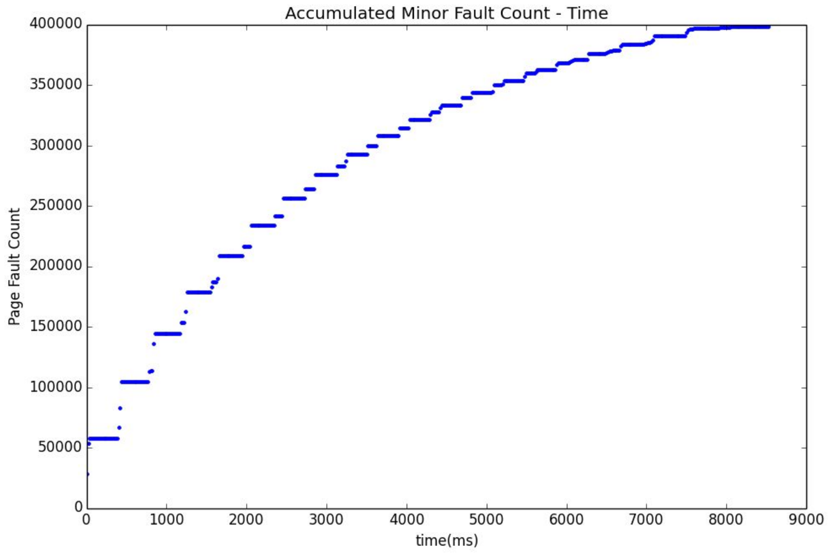
Next, type ‘sudo insmod mp3.ko’.
If ‘node’ exists, then delete the ‘node’ file using ‘rm’ command.
Then, type ‘cat /proc/devices’, and check out the major number of the device.
Type ‘sudo mknod node c <major # of the device> 0’.
To run the case study 1, type ‘nice ./work 1024 R 50000 & nice ./work 1024 R 10000 &’, followed by ‘sudo ./monitor > profile1.data’.
To view the output, type ‘cat profile1.data’.
To run the case study 1, type ‘sudo nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &’, followed by ‘sudo ./monitor > profile2.data’.
To view the output, type ‘cat profile2.data’.
Also, type ‘cat /proc/mp3/status’ to see if all tasks successfully unregistered themselves
In order to uninstall module, type ‘sudo rmmod mp3’.

Analysis

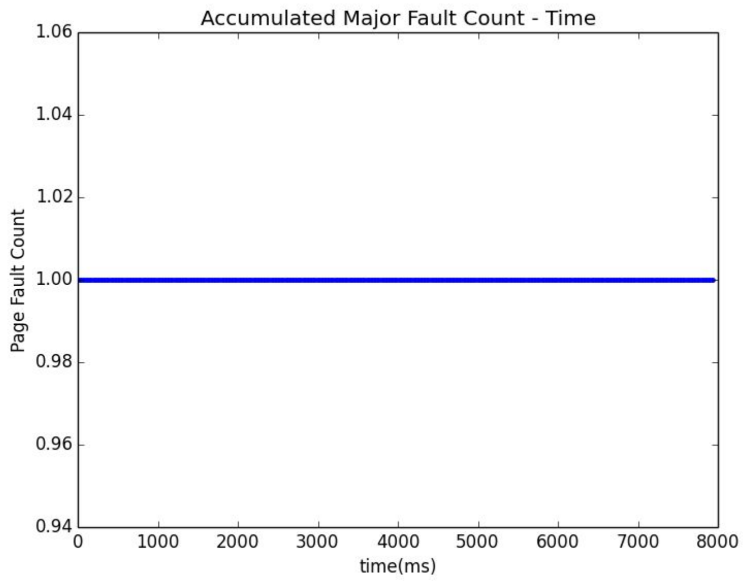
Case Study 1

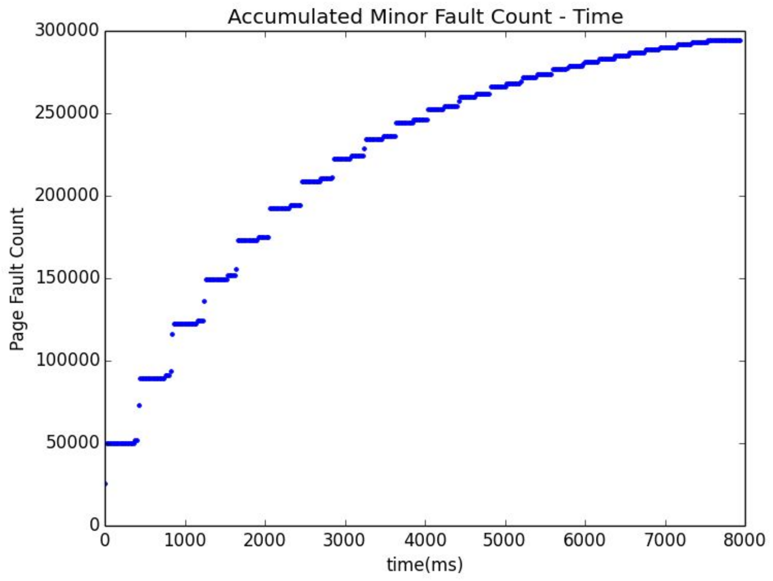
Both two processes are using random access





One process uses random access, while another uses locality access.





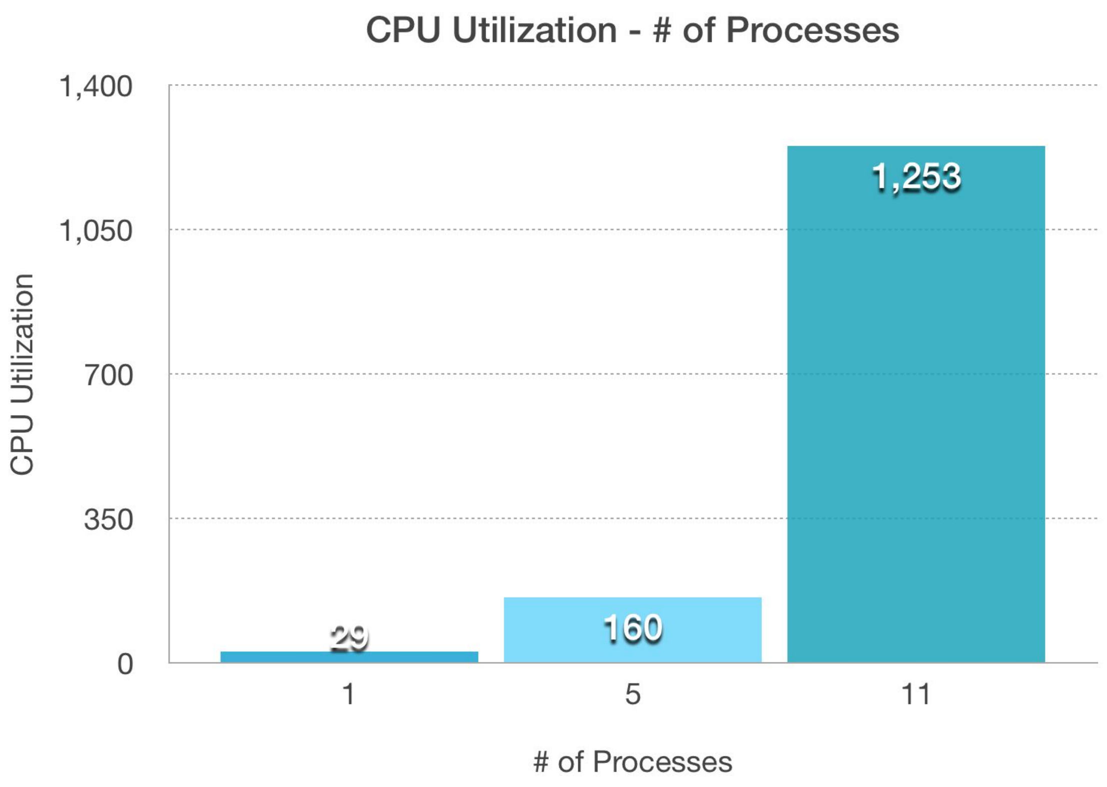
Summary:

From the 4 plots shown above, we can conclude that the one with two random processes has much larger page fault count, both the major fault count and the minor fault count. And it takes more time to run the one with two random processes.

Analysis:

1. Two random access process:
    a. Minor page fault
        i. The reason why random access process has more page fault is that it does not take advantage of spatial locality. Note that the graph is also concave down, this is because as time goes on, we brought more and more memory, therefore the probability a minor page fault happen decreases, the accumulated number of minor page fault does not increase as fast as before.
    b. Major page fault
        i. Since we are accessing memory randomly, then we constantly brought pages into memory. Therefore, after a period of time, we fulfill the page tables. Then if we want to read a new physical memory, a major page fault then happens.
2. One random access process and a locality-based access process
    a. Minor page fault
        i. Decrease significantly compared to two random access processes, because one of the process takes advantages of spatial locality.
    b. Major page fault
        i. Again, one of the process takes advantages of spatial locality, therefore the number decreases significantly.

Case Study 2



Summary:

We ran N (N = 1, 5, 11) processes with same memory size and accesses per iteration to obtain the CPU utilization for each of the cases as shown above. The plot shows that with the increase of the degree of multiprogramming (number of processes running at the same time), the CPU utilization also increases exponentially.

Analysis:

It is obvious that more CPU using is required for more processes running. As we stated above, we can conclude from the plot that the increasing is not linear. This might comes from the increasing number of page fault caused by more processes. The CPU needs to do more I/O in that case.