MP3 Case Study And Read Me:

How to run code:

```
make
sudo insmod lall3_mp3.ko
sudo mknod node c 200 0
nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &
sudo ./monitor
```

Design:

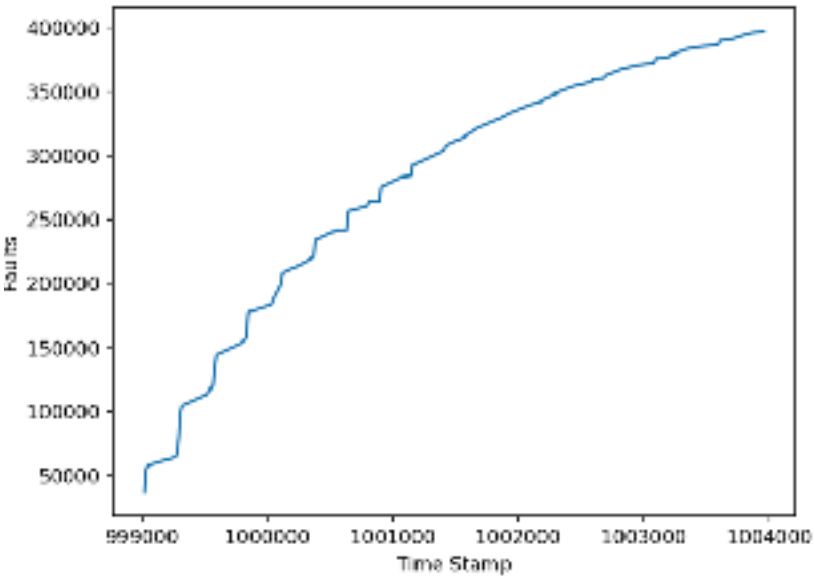
I used the skeleton design from MP2 as a base. I added a few variables to the mp3 struct as shown below

```
//MP3 struct
typedef struct mp3_struct
{
  struct list_head list_node;
  struct task_struct * current_task;
  unsigned long time_used;
  unsigned long major_faults;
  unsigned long minor_faults;


  pid_t pid;

}mp3_t;
```

As for the proc read and write call back functions , I used something similar to that in MP2. The major difference being my register and unregister helper functions.

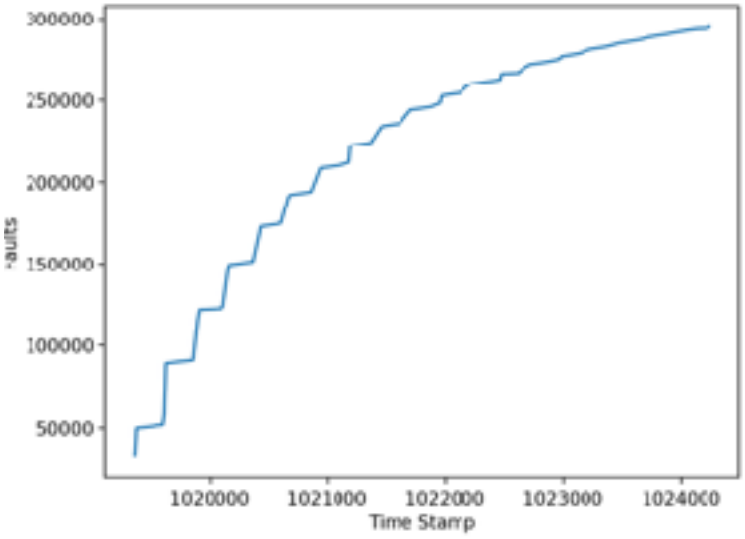
I also implemented new top and bottom half functions to deal with the delayed work.
My delayed work function called delayed_func deals with updating the faults and cpu usage for each program.

Case Study 1:



The graph above is the graph produced using the results generated by running the following code from shell:

$ nice ./work 1024 R 50000 & nice ./work 1024 R 10000 &
$ ./monitor > profile2.data

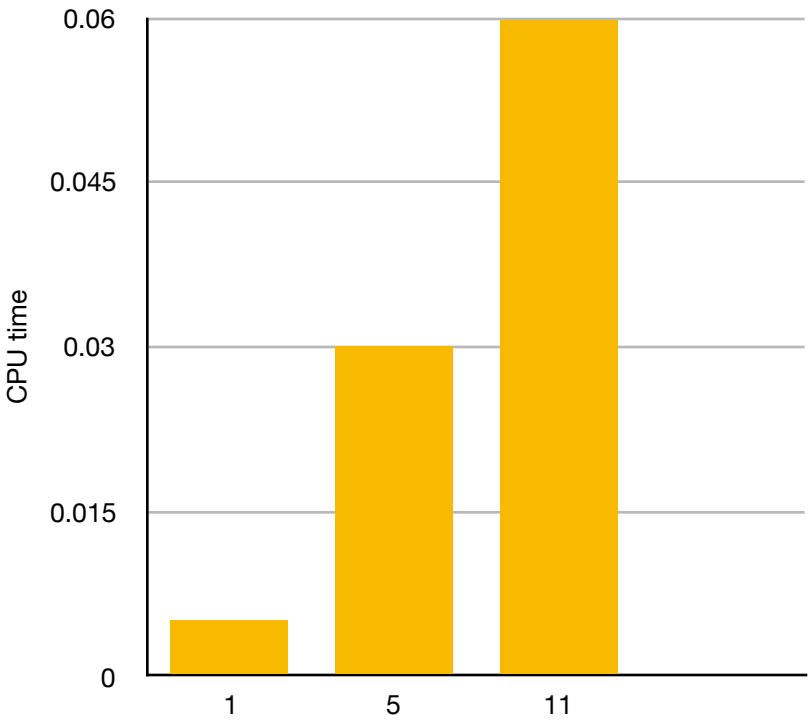


The graph above is the graph produced using the results generated by running the following code from shell:

$ nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &
$ ./monitor > profile2.data

As we can observe from the graph the faults in the second case are about 1000 less then the faults measured on average in the first case. This is because random access, unlike local access, is not able to take advantage of the caching mechanism provided by the kernel.

Case2:



The graph above shows the average cpu usage time measured while running 1, 5 and 11 iterations of the work function simultaneously.

The average cpu times were measured to be 0.005, 0.03 and 0.06 respectively. The increased running time is due to over head and page faults that occur while changing between processes.