Group_09_MP3

Daniel Petrisko (petrisk2)
Xuefeng Zhu (xzhu15)
Yanwen Li (li206)
Abhishek Sharma (sharma51)

Part 1: Implementation & Design Decisions

Given below are the steps we followed in order to implement a kernel module for Virtual Memory Fault Profiler.

<u>Step1:</u> Create a directory entry “/proc/mp3” within the Proc filesystem and created a file entry “/proc/mp3/status/” inside the directory

```
/* Helper function to create the directory entries for /proc */
void create_mp3_proc_files(void);
```

<u>Step 2:</u> Callback functions for write in the entry of the proc filesystem. The write function has a switch to separate each type of message (R: registration, U: unregistration).Detailed will be showed in following steps.

```
/* Occurs when a user runs ./process > /proc/mp3/status
   Input format should either be: "R PID"
                                  "U PID"
*/
ssize_t write_proc(struct file *filp, const char *user, size_t count, loff_t *offset);


  switch(type)
  {
    case 'R':
      register_handler(pid);
      break;
    case 'U':
      unregister_handler(pid);
      break;
  }
```

<u>Step 3:</u> Declare and initialize a linked list “pid_sched_list” that contains our own PCB. We add additional state information of task to Linux PCB including the pid, process utilization, major fault count and minor fault count of the corresponding process.

```
/* The linked list structure */
struct mp3_pcb {
  struct list_head list; /* Kernel's list structure */
  struct task_struct* linux_task;
  unsigned long pid;
  unsigned long utilization;
  unsigned long major;
```

```
   unsigned long minor;
};

/* Initialized the linked list */
INIT_LIST_HEAD(&mp3_pcb.list);
```

Step 4: Implement registration and unregistration functions. The registration function adds the process to the PCB list. If it is the first one in the PCB list, we need to create a workqueue. The unregistration function delete the process from the PCB list. If the PCB list is empty after deletion, we will delete the workqueue as well. Here we use delay_work.

```
/* Helper function to register a task */
void register_handler(unsigned long pid);

/* Helper function to unregister a task */
void unregister_handler(unsigned long pid);
```

Step 5: Use vmalloc() to allocate a memory buffer in kernel memory when kernel module is initialized.

```
prof_buffer = vmalloc(NPAGES * PAGE_SIZE);
```

Step 6: Implement a workqueue that periodically measures the major and minor page fault counts, and CPU utilization of process in the PCB list and save the measured information to the memory buffer.

```
monitor_work = (struct delayed_work*)kmalloc(sizeof(struct delayed_work), GFP_KERNEL);
INIT_DELAYED_WORK(monitor_work, monitor_wq_function);
schedule_delayed_work(monitor_work, msecs_to_jiffies(0));

/* Callback for the work function to monitor jobs */
void monitor_wq_function(struct work_struct *work)
```

The sample rate of profiler must be 20 times per second.

```
schedule_delayed_work(monitor_work, msecs_to_jiffies(1000 / 20));
```

Step 7: Use character device driver to transfer profiled data directly from kernel to user process. Here we use three callback functions: open, close and mmap. The open and close callback handler are defined as empty functions.

```
/* Drive open op */
int open_drive(struct inode *inode, struct file *file){
   return 0;
}

/* Drive close op */
int release_drive(struct inode *inode, struct file *file){
   return 0;
}

/* Drive mmap op */
int mmap_drive (struct file *file, struct vm_area_struct *vma){
   unsigned long pfn;
   int i;

   for (i = 0; i < NPAGES; i++){
       pfn = vmalloc_to_pfn(prof_buffer + i*PAGE_SIZE);
       remap_pfn_range(vma, vma->vm_start + i*PAGE_SIZE, pfn, PAGE_SIZE, PAGE_SHARED);
   }

   return 0;
}
```

Step 8: Memory leak checks. Our kernel module ensures that the resources that were allocated during its execution are freed which includes freeing allocated memory, stopping pending work function, destroying timers, slab allocator, character device driver and proc filesystem entry.

Part 2: Details of how to run your program

Step 1:
cd MP3, run “make”
sudo insmod mp3.ko
-install the module

cat /proc/devices
<check the created device’s major number>
sudo mknod node c <major # of the device> 0
sudo chmod a+r+w node

Step 2:
nice ./work 1024 R 50000 & nice ./work 1024 R 10000 &
./monitor > profile1.data
-run two work processes with random access

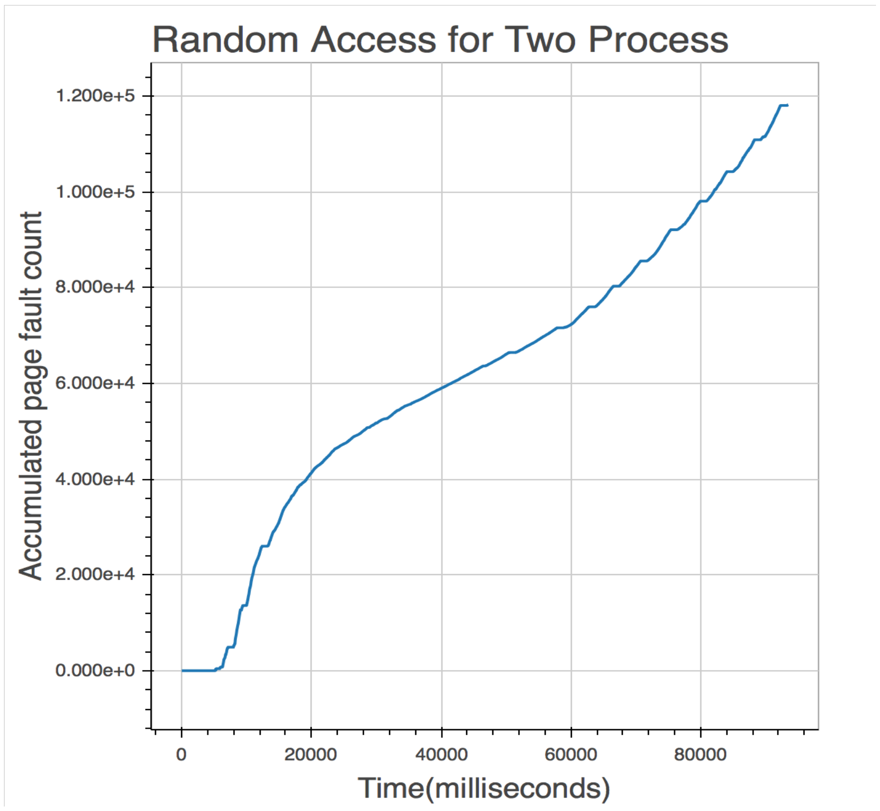
Step 3:
nice ./work 1024 R 50000 & nice ./work 1024 L 10000 &
./monitor > profile2.data
-run two work processes one with random access, one with locality-based access.

Step 4:
sh multiprogramming.sh
-run N copies of work process ./work 200 R 10000, where N=1,5,7,11

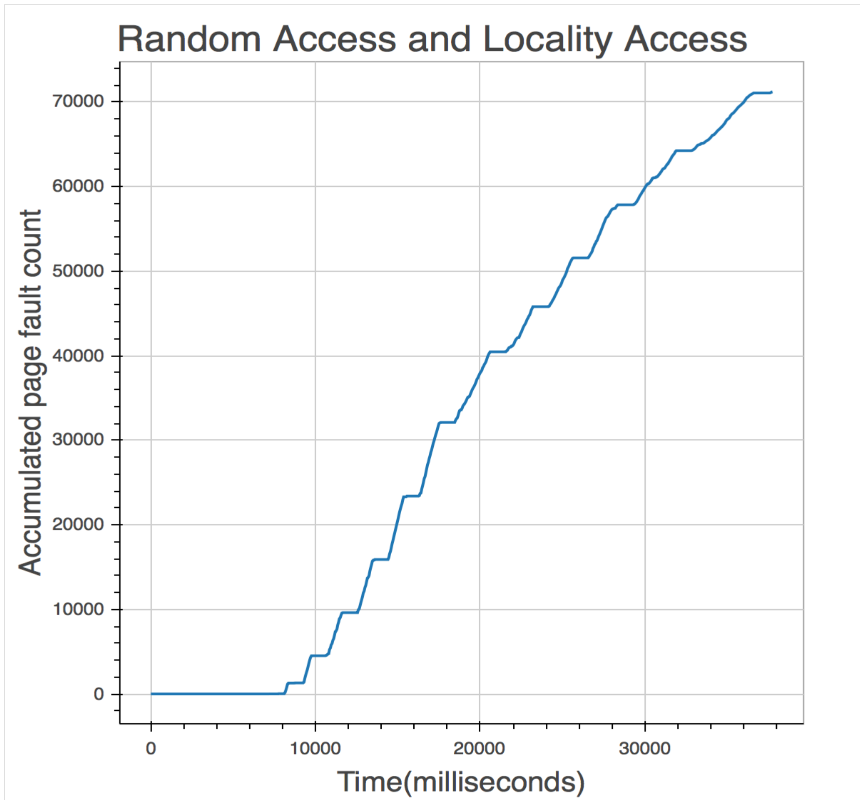
Part 3: Analysis

Case Study 1: Thrashing and locality

Plot:



Plot:



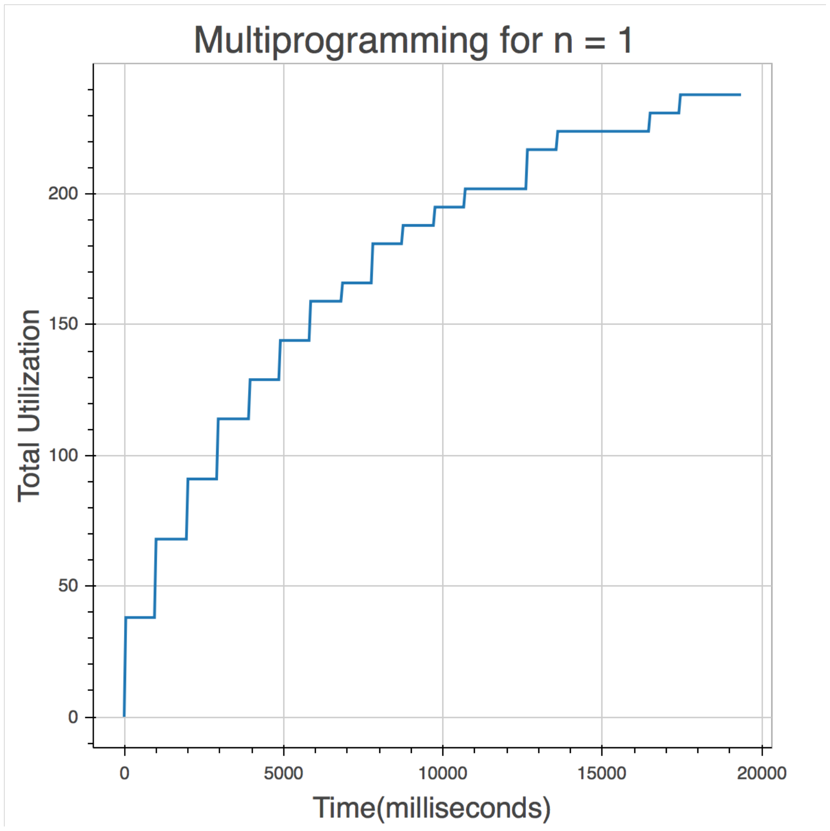
Discussion:

1. It is clear that the page fault with two random processes is much higher compare to one process with random access and other one with localized access.
2. Random access takes a lot more time to complete the work processes than localized access.
3. One possible reason for this behavior is that with localized access, since it keeps one process working with a limited number of pages much fewer than random access one in a period of time, while random access is more likely to swap pages if it is required by another process.

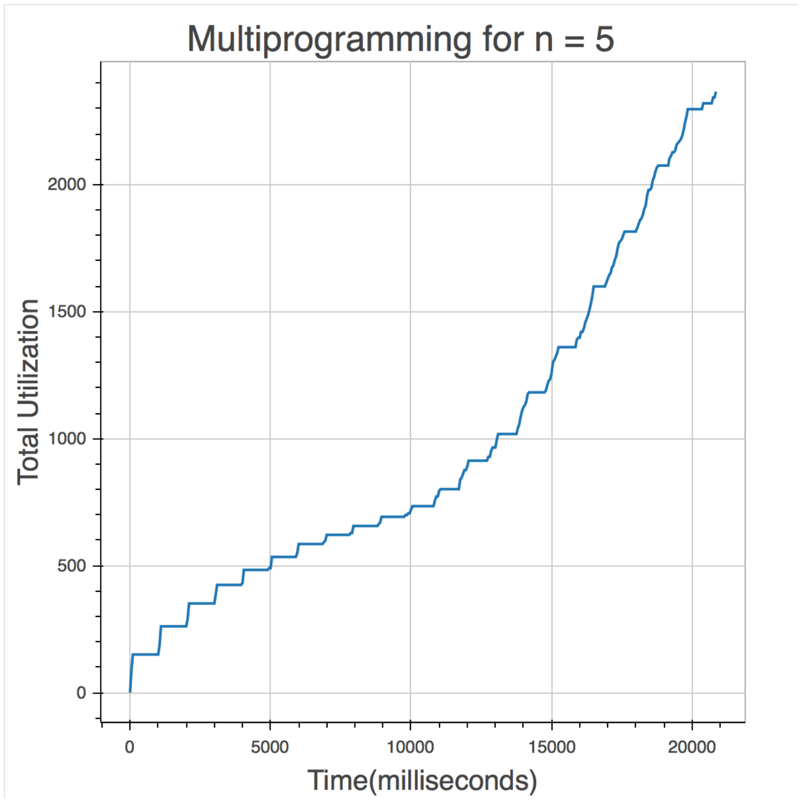
<u>Case Study 2:</u> Multiprogramming

Plot:

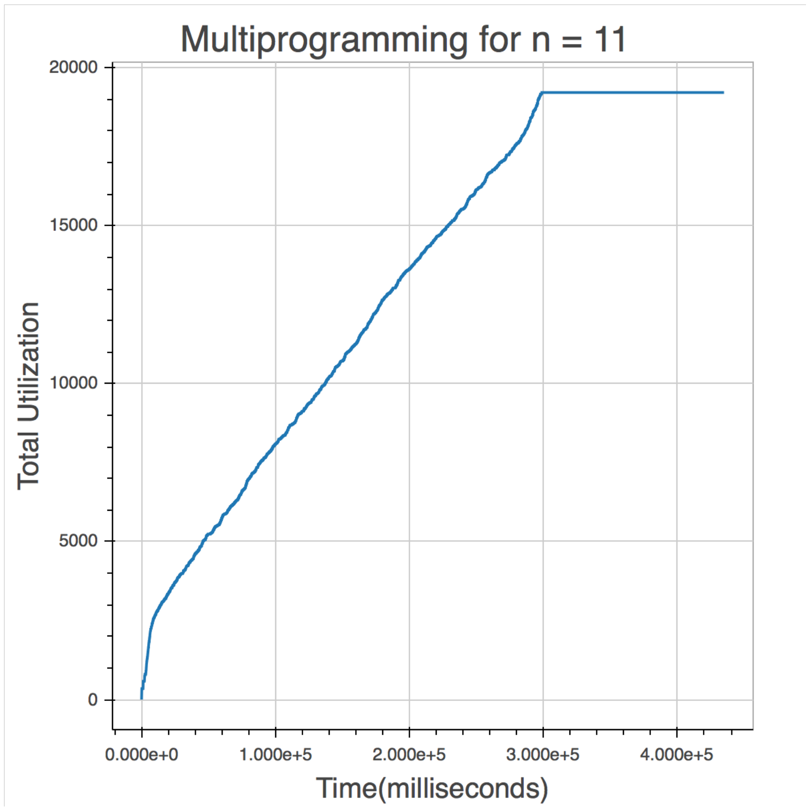
N=1


Multiprogramming for n = 1
Total Utilization
Time(milliseconds)
0
50
100
150
200
0
5000
10000
15000
20000

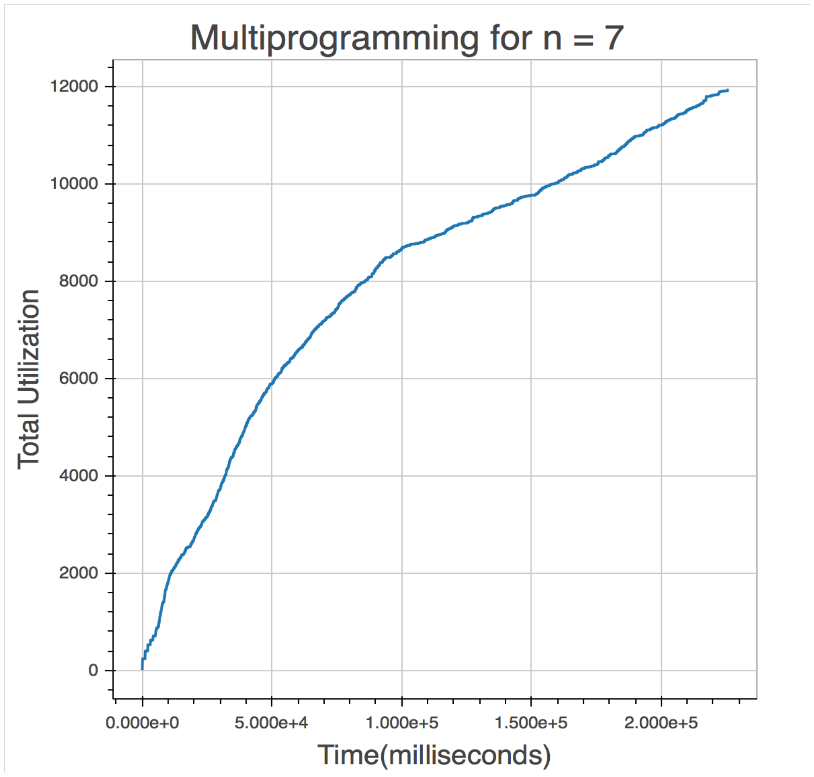
N=5


Multiprogramming for n = 5
Total Utilization
Time(milliseconds)
0
500
1000
1500
2000
0
5000
10000
15000
20000

N=11



Discussion:
1. The time taken to complete for 5 processes is slightly more than the time taken for 1 process. However, for 11 processes, the time to complete increases significantly.
2. One possible reason of this behaviour is thrashing. The efficiency of multiprogramming can only increases till a point, and after exceeding this limit, most of time is used to swap pages.
3. It shows that CPU utilization for 5 processes is much larger than for 1 process, and the CPU utilization for 11 processes increases sharply, but then declines and stay at a constant rate.
4. Therefore as the number of multiprogramming increases, the CPU utilization increases sharply until reaching its maximum.
5. We noticed that when we run processes at n=11, some processes will be closed. We guessed it was due to the fact that system ran out of resource. It lead us to do experiment to find out what will be maximum number of processes. It turned out when n= 7, all processes can be finished. The cpu utilization graph is listed below.



It shows when N=11 some processes gets killed, and N=7 is the maximum number of processes when no process gets killed.