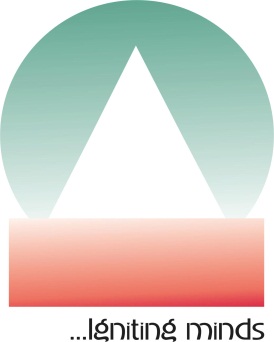


**POSTGRADUATE ENGINEERING AND MANAGEMENT PROGRAMME – (PEMP)**

|  |  |
| --- | --- |
| **ASSIGNMENT** | |
| **Module Code** | ESD2528 |
| **Module Name** | Advanced System Programming |
| **Course** | M.Sc[Engg] in Real-Time Embedded Systems |
| **Department** | Computer Engineering. |

#### 



|  |  |
| --- | --- |
| **Name of the Student** | Kushal J |
| **Reg. No** | CHB0412002 |
| **Batch** | Part Time 2012 |
| **Module Leader** | Jishmi Joc Chondal |

M.S.Ramaiah School of Advanced Studies

Postgraduate Engineering and Management Programmes(PEMP)

**#470-P Peenya Industrial Area, 4th Phase, Peenya, Bengaluru-560 058**

**Tel; 080 4906 5555, website: www.msrsas.org**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Declaration Sheet | | | | | | | | |
| Student Name | Kushal J | | | | | | | |
| Reg. No | CHB0412002 | | | | | | | |
| Course | RTES | | | | | Batch | Part-Time 2012. | |
| Batch | Part-Time 2012 | | | | | | | |
| Module Code | ESD 2528   |  | | --- | |  | | | | | | | | |
| Module Title | Advanced System Programming | | | | | | | |
| Module Date | |  | | --- | | 15-12-2013 | | | to | | 01-02-2014 | | | |
| Module Leader | Jishmi Jos | | | | | | | |
| **Extension requests:**  Extensions can only be granted by the Head of the Department in consultation with the module leader. Extensions granted by any other person will not be accepted and hence the assignment will incur a penalty. Extensions MUST be requested by using the ‘Extension Request Form’, which is available with the ARO.  **A copy of the extension approval must be attached to the assignment submitted**. | | | | | | | | |
| **Penalty for late submission**  Unless you have submitted proof of mitigating circumstances or have been granted an extension, the penalties for a late submission of an assignment shall be as follows:   * Up to one week late: Penalty of 5 marks * One-Two weeks late: Penalty of 10 marks * More than Two weeks late: Fail - 0% recorded (F)   All late assignments: must be submitted to Academic Records Office (ARO). It is your responsibility to ensure that the receipt of a late assignment is recorded in the ARO. If an extension was agreed, the authorization should be submitted to ARO during the submission of assignment.  **To ensure assignment reports are written concisely, the length should be restricted to a limit indicated in the assignment problem statement. Assignment reports greater than this length may incur a penalty of one grade (5 marks). Each delegate is required to retain a copy of the assignment report.** | | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the PEMP Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the student | |  | | | | | Date |  |
| Submission date stamp  (by ARO) | |  | | | | | | |
| Signature of the Module Leader and date | | | | Signature of Head of the Department and date | | | | |
|  | | | |  | | | | |

# Abstract

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Often Linux has been defined as General Purpose Operating System. But in recent years, because of its popularity and ease of use, Linux kernel has been used in many embedded systems for hard-real time and soft real-time applications. Part A of the assignment describes the advantages, ways and disadvantages of creating a patch on Linux kernel so that it can be used for hard real-time embedded applications.

System calls are a way to access and perform a specified operation in the “kernel space”. System calls are also ways to access kernel code, hardware and other devices present in the system. Part B section of the assignment, we design and implement and test a system call which is sued to calculated the resource usage of a given process.

In the Part C of the assignment, an application program is developed to analyze the system performance using the developed system call. The application program performs a lot of file opening/closing operations as well as heap allocations so as to increase the virtual memory allocated for the process. In turn the number of major page faults, minor page faults and number of page frames are calculated at different time intervals. It also analyses the results and describes the performance of the system using many graphical plots.

# Contents

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Contents

[Declaration Sheet ii](#_Toc379011140)

[Abstract iv](#_Toc379011141)

[Contents v](#_Toc379011142)

[List of Tables vi](#_Toc379011143)

[List of Figures vii](#_Toc379011144)

[List of Symbols viii](#_Toc379011145)

**[Chapter 1: Pre-emptible Linux kernel for hard real-time applications](#_Toc379011146)** [9](#_Toc379011146)

[1.1 Introduction- Hard Real-Time Systems and Linux kernel 9](#_Toc379011147)

[1.2 Hard real time requirements and RTOS 11](#_Toc379011148)

[1.3 Pre-emptible kernel approach to RTOS design 12](#_Toc379011149)

[1.4 PREEMPT\_RT: making Linux pre-emptible kernel 12](#_Toc379011150)

[1.5 Conclusion 13](#_Toc379011151)

**[Chapter 2: Design and Implementation of a System Call](#_Toc379011152)** [15](#_Toc379011152)

[2.1 Introduction 15](#_Toc379011153)

[2.2 System Call Design 15](#_Toc379011154)

[2.3 Algorithm Flowchart 16](#_Toc379011155)

[2.4 Algorithm Steps 17](#_Toc379011156)

[2.5 Implementation of sys\_pgfltstat system call 17](#_Toc379011157)

[2.6 Code 19](#_Toc379011158)

[2.7 Test program 21](#_Toc379011159)

[2.8 Results and Analysis 22](#_Toc379011160)

**[Calculating page fault rate using system call](#_Toc379011161)** [24](#_Toc379011161)

[3.1 File operation and Virtual memory Usage Application 24](#_Toc379011162)

[3.2 Calculation of Page frames 30](#_Toc379011163)

[3.3 Graphical representation and analysis 31](#_Toc379011164)

**[References](#_Toc379011165)** [36](#_Toc379011165)

# List of Tables

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# List of Figures

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
| **Figure No.** | **Title of the figure** | **Pg.No.** |
| Figure 1.1 | Title of the figure | 13 |
| Figure 1.2 | Title of the figure | 15 |
| Figure 2.1 | Title of the figure | 19 |
|  |  |  |

< The Figure numbers have to be based on the chapter number>

# List of Symbols

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**PART-A**

# **Chapter 1: Pre-emptible Linux kernel for hard real-time applications**

## Introduction- Hard Real-Time Systems and Linux kernel

A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. Different approaches exist to make linux kernel a real-time kernel. Some of them are:

**Thin-kernel approach -** In this approach, a small real-time kernel runs beneath the linux kernel, meaning that the real-time kernel has higher priority than the linux kernel. Real-time tasks are executed by real-time kernel and normal linux programs are executed when no real-time tasks have to be executed, as real-time kernel has higher priority. Linux can be considered as idle task for the real-time scheduler. When this idle task runs, it executes its own scheduler and schedules normal linux processes[2]. This approach is shown in figure 1.1.

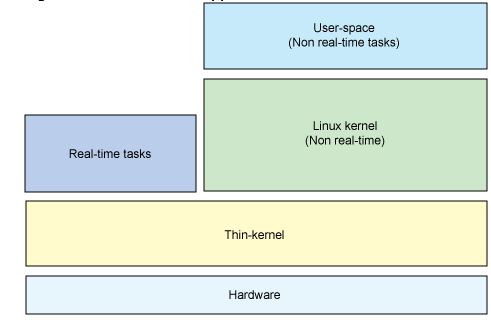


Figure 1.1: Thin kernel approach for real-time linux architecture.

The disadvantage of thin kernel is that it is difficult to debug the programs as the real-time and non-real time tasks are independent. Also, a non-real-time task does not have full linux platform support. Examples: RTLinux, RTAI(real-time application interface) and Xenomai.

**Nano-kernel approach –** In this approach kernel code is minimized to a great extent and becomes only a hardware abstraction layer as shown in figure 1.2. The nano-kernel provides hardware resource sharing for multiple operating systems operating at higher layer. As the nano-kernel abstracts hardware, it can provide higher priority for real-time tasks and thus supports hard-real time systems. When hardware events occur, the kernel queries each operating system in a chain to see which will handle the event.

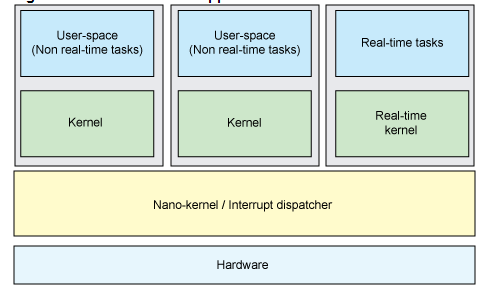


Figure 1.2: Nano kernel, which is an hardware abstraction layer.

Example:  Adaptive Domain Environment for Operating Systems (ADEOS)

* **Resource kernel approach -** In this approach, a module is added to the kernel to provide reservations for various resources like CPU, network and disk bandwidth. The reservation promises time-multiplexing and has several reserved parameters such as period of occurrence and required processing time. The resource kernel provides a set of API’s to allow tasks to request these reservations as shown in figure 5. The scheduler uses earliest deadline first algorithm to handle dynamic scheduling workload. Example: CMU's Linux/RK, where linus kernel is added as loadable module.

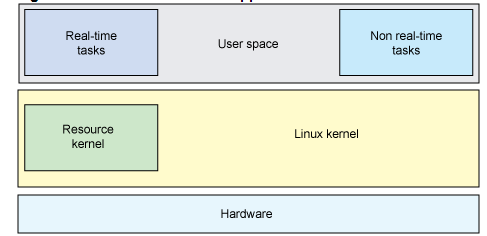


Figure 1.3 : Resource kernel approach to resource reservation.

|  |
| --- |
| Hard real time requirements and RTOS Requirements for a hard real-time systems are minimal latency during task switching, minimal jitter, run-to completion, preemptive multi-tasking, priority inheritance and principally meet strict deadlines.  **Response time and kernel pre-emption** |

Hard real time means strict about adherence to each task deadline. When an event occurs, it should be serviced within the predictable time at all times in a given hard real time system. The time between the arrival of the interrupt and dispatching of the required task (assuming it's the highest-priority task to dispatch) is called the **response time***.* For real time, the response time should be deterministic and operate within a known worst-case time as shown in figure 1.4

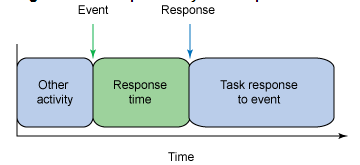


Figure 1.4: Response time should be in terms of a few micro-seconds in a hard real-time system.

For responding to an event/interrupt, the kernel should be pre-empted (from its previous task). The preemption period for the hard real time task in worst case should be less than a few micro-seconds.

Automobile engine control system and anti-lock brake are the examples of hard real time systems.

**Periodic task scheduling**

The kernel of a hard real-time system should schedule tasks at regular periodic intervals of time. The kernel processing should be pre-empted to allow higher priority processes be executed at desired periods. Figure 1.5 shows periodic task scheduling scheme.

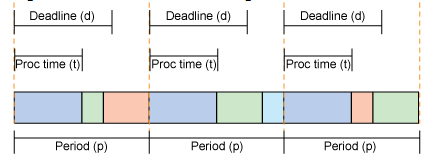


Figure 1.5: Periodic task scheduling of a hard real-time kernel.

In the diagram shown in figure 1.5, for the periodic interval shown in green, the kernel is pre-empted to perform other higher priority tasks. This also invariably means that there should be absolutely **no** deadlines occurring during any process execution.

## Pre-emptible kernel approach to RTOS design

Preemption means that a running task can be suspended while another task takes its place as the currently running process. Once a task has consumed its allotted quantum, or CPU time slice, the scheduler will suspend it and run the next task in the queue. Kernel operations are pre-emptible in a real-time operating system.

Advantages and disadvantages of kernel preemption – A preemptive kernel has the choice to suspend a running task even in kernel mode. This is a requirement for hard-real systems to meet its deadline. This poses a challenge for the user as their execution time is very short but not constant. For instance, under a user preemptive model, the time required to complete a *malloc()* function call depends on heap fragmentation, size of the requested memory block and system load. Thus one call may take 10ms to complete whereas another might take 50ms. A task that must terminate within 40ms might not finish on time if another process calls *malloc()* in between.

Under a kernel pre-emptive model, the scheduler can preempt *malloc()* system call and enable a real-time task to finish. **Kernel preemption improves the responsiveness of some notorious applications that causes system to freeze up for a short while during certain activities**.

On the downside, kernel preemption will require kernel overhaul and complicate kernel’s code.

To support pre-emptibility, kernel data must be explicitly protected using mutexes or spinlocks. The pre-emptible kernel patch (supported by MonoVista) uses spinlocks. Kernel preemption is disabled when spinlocks are held. The OS will impose an upper bound on how long preemption is held off and interrupts disabled; this allows developers to ascertain worst-case latencies.

## PREEMPT\_RT: making Linux pre-emptible kernel

The key point of the PREEMPT\_RT patch is to minimize the amount of kernel code that is non-preemptible, while also minimizing the amount of code that must be changed in order to provide this added preemptibility. In particular, critical sections, interrupt handlers, and interrupt-disable code sequences are normally preemptible. The PREEMPT\_RT patch leverages the SMP capabilities of the Linux kernel to add this extra preemptibility without requiring a complete kernel rewrite. In a sense, one can loosely think of a preemption as the addition of a new CPU to the system, and then use the normal locking primitives to synchronize with any action taken by the preempting task.

Note that this statement of philosophy should not be taken too literally, for example, the PREEMPT\_RT patch does not actually perform a CPU hot-plug event for each preemption. Instead, the point is that the underlying mechanisms used to tolerate (almost) unlimited preemption are those that must be provided for SMP environments.

The [CONFIG\_PREEMPT\_RT patch set](http://www.kernel.org/pub/linux/kernel/projects/rt/)  allows nearly all of the kernel to be preempted, with the exception of a few very small regions of code ("raw\_spinlock critical regions"). This is done by replacing most kernel spinlocks with mutexes that support [priority inheritance](https://rt.wiki.kernel.org/index.php/Priority_inheritance), as well as moving all interrupt and software interrupts to kernel threads

The CONFIG\_PREEMPT\_RT features are:

* Making in-kernel locking-primitives (using spinlocks) preemptible though reimplementation with rtmutexes.
* Critical sections protected by i.e. spinlock\_t and rwlock\_t are now pre-emptible. The creation of non-preemptible sections (in kernel) is still possible with raw\_spinlock\_t (same APIs like spinlock\_t)
* Implementing priority inheritance for in-kernel spinlocks and semaphores.
* Converting interrupt handlers into pre-emptible kernel threads: The RT-Preempt patch treats soft interrupt handlers in kernel thread context, which is represented by a task\_struct like a common user-space process. However it is also possible to register an IRQ in kernel context.
* Converting the old Linux timer API into separate infrastructures for high resolution kernel timers plus one for timeouts, leading to user-space POSIX timers with high resolution[5].
* **Disadvantage** -  
  The normal Linux kernel allows preemption of a task by a higher priority task only when the user space code is getting executed.
* In order to reduce the latency, the CONFIG\_PREEMPT\_RT patch forces the kernel to non-voluntarily preempt the task at hand, at the arrival of a higher proiority kernel task. This is bound to cause a reduction in the overall throughput of the system since there will be several context switches and also the lower priority tasks won't be getting much a chance to get through.

## Conclusion

The general idea of Real-time (RT) Linux is that a small real-time kernel runs beneath Linux, meaning that the real-time kernel has a higher priority than the Linux kernel. Real-time tasks are executed by the real-time kernel, and normal Linux programs are allowed to run when no real-time tasks have to be executed. Linux can be considered as the idle task of the real-time scheduler. When this idle task runs, it executes its own scheduler and schedules the normal Linux processes. Since the real-time kernel has a higher priority, a normal Linux process is preempted when a real-time task becomes ready to run and the real-time task is executed immediately.

**PART-B**

# **Chapter 2: Design and Implementation of a System Call**

\_\_\_\_­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## Introduction

The *pgfltstat()*  implemented in this assignment provides the number of major page faults, number of minor page faults for a given input process(es). The system call was implemented on linux kernel 3.12.6 and tested on Ubuntu 13.04 LTS machine.

The following sections describe about the details of the system call.

## System Call Design

The system call is prototyped as shown below:



System call name - sys\_pgfltstat

Input arguments:

***int val*** - describes whether the page fault statistics is required for a single process , a group of processes or all the processes running in the system.

***val*** *🡺 -1* means page fault statistics are calculated for the process whose ***pid*** is the second input

***val*** *🡺 group id means* page fault statistics are calculated for all the processes belonging to the group indicated by the group id.

***val*** *🡺 -2 means* page fault statistics are calculated for all the processes running in the system

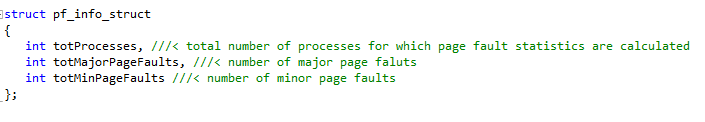
2)

***pid\_t pid*** *–* describes the process id of the process for which page fault statistics have to be calculated.

This field is invalid if val = -2 or if val = group id.

3)

***pf\_info\_struct \*info*** is a user definedinput output variable which gives the page fault statistics. ***pf\_info\_struct*** has the following fields,



## Algorithm Flowchart

Exit

Get the pid value from pid\_t structure

Find the resource usage for the process, update the ***pf\_info\_struct*** array

(groupid of process = val?

Val = -1?

(/proc/dir exists?

Read /proc directory

Start

Val = -2?

Val = group id?

If (/proc/dir exists?

Read /proc directory

Yes

NO

NO

Yes

Yes

Yes

Yes

Yes

NO

NO

NO

Figure 2.1: Flow diagram explaining the logic for calculating the page faults.

## Algorithm Steps

Initially the input ‘val’ is compared with different assigned values.

If val = -2, then all the directories in the /proc folder is read and the pid for each process is stored. For each pid value, the resource usage is calculated using the function getrusage().



The getrusage() returns the a structure of type *rusage*. The fields of *rusage* are shown below:

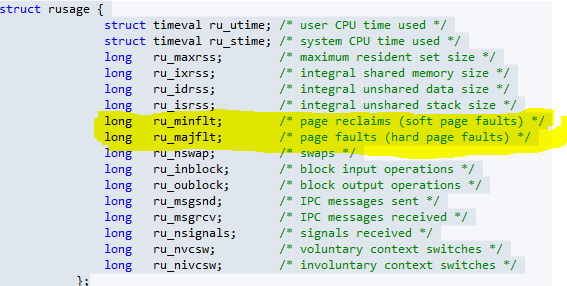


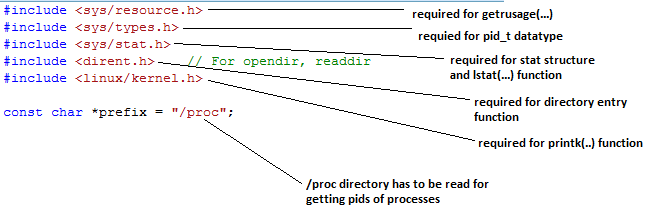
Figure 2.2: rusage structure which has major page faults and minor page faults highlighted in yellow.

* Major page fault and minor page fault are given by the fields ru\_minflt and ru\_majflt.
* If *val = -1*, the pid\_t is a signed integer type which is passed as input to the function ***getrusage()***. The major page faults and minor page faults are extracted using the above procedure.
* Else, the value is compared with each group id working on the system and the page faults are recovered for all the processes of the indicated group. The processes belonging to a particular group is found out by scanning the contents of /proc directory.

## Implementation of sys\_pgfltstat system call

The linux kernel source code is expanded into the folder /usr/src/linux-3.12.6 directory. A directory named /usr/src/linux-3.12.6/pgfltstat is created which contains the file that has implementation of the system call. The C implementation file is named as process.c and the contents are shown below:

Include header files:



Code snippet 1



Code snippet 2

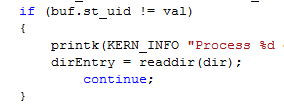
First open and read the ***/proc*** directory to find all the current running processes as shown in code snippet 2.

Each running process will have a directory entry in the /proc directory. The name of the directory will indicate the *pid* of each running process.



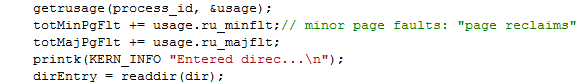
Code snippet 3

The directory name is converted into an integer and checked whether it is a positive number as shown in code snippet 3. If it is a positive number then the ***pid*** is input into the ***getrusage(..)*** to get thepage fault statistics.



Code snippet 4

Snippet 4 shows the segregation of processes according the group ids. The *val* consists of the group id (if *val* is a positive integer).



Code snippet 5

The code snippet 5 shows the implementation of ***getrusage(..)*** function which gives the major page fault and minor page fault.

## Code

The detailed C code implementation of the page fault information syscall is given below:

#include <sys/resource.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h> // For opendir, readdir

#include <linux/kernel.h>

const char \*prefix = "/proc";

struct pf\_info\_struct

{

int numberofProc;

int majPgFault;

int minPgFault;

};

asmlinkage long process(int val, pid\_t pid, pf\_info\_struct\* info)

{

struct rusage usage;

int process\_id=0;

long totMajPgFlt=0;

long totMinPgFlt=0;

struct dirent\* dirEntry = NULL;

struct stat buf;

int target = 0;

int count = 0;

int dummy = 0;

DIR\* dir=NULL;

if (val == -1)

{

printk(KERN\_INFO "Getting resource usage for all the processes\n");

}

else if (val == -2)

{

printk(KERN\_INFO "Getting resource usage for single process\n");

}

else

{

printk(KERN\_INFO "Getting resource usage for group of process\n");

}

if (val != -2)

{

// find the pid's of all the current running processes

dir = opendir(prefix);

dirEntry = readdir(dir);

printk(KERN\_INFO "Entered proc directory\n");

while( dirEntry != NULL)

{

// get info about the node (file or folder)

lstat(dirEntry->d\_name, &buf);

if (val == -1)

{

if (count >= 999)

{

break;

}

process\_id = atoi(dirEntry->d\_name);

printk(KERN\_INFO "Getting resource usage for the process = %d\n", process\_id);

count++;

}

else

{

process\_id = atoi(dirEntry->d\_name);

if (buf.st\_uid != val)

{

printk(KERN\_INFO "Process %d doen not belong to group id %d\n", process\_id, val);

dirEntry = readdir(dir);

continue;

}

if (count >= 499)

{

break;

}

printk(KERN\_INFO "Getting resource usage for the process = %d\n", process\_id);

count++;

}

if (process\_id<0)

{

printk(KERN\_INFO "Not a process 2\n");

dirEntry = readdir(dir);

continue;

}

getrusage(process\_id, &usage);

totMinPgFlt += usage.ru\_minflt;// minor page faults: "page reclaims"

totMajPgFlt += usage.ru\_majflt;

printk(KERN\_INFO "Entered direc...\n");

dirEntry = readdir(dir);

}

if (val == -2)

{

process\_id = (int) pid;

printk(KERN\_INFO "Getting resource usage for the process = %d\n", process\_id);

// find the resource usage for the specified process

getrusage(process\_id, &usage);

totMinPgFlt = usage.ru\_minflt;// minor page faults: "page reclaims"

totMajPgFlt = usage.ru\_majflt;

count = 1;

}

info->majPgFault = totMinPgFlt;

info->minPgFault = totMajPgFlt;

info->numberofProc = count;

return 21;

}

## Test program

The system call was tested on Ubuntu 13.04 LTS system with a linux3.12.6 kernel. The system call was tested using a userspace program shown below.

#include <unistd.h>

#include<linux/kernel.h>

int main(int argc, char\*\* argv)

{

int processNum = 2193;

int inWhat = -2;

int retval = -1;

if (argc == 3)

{

processNum = atoi(argv[1]);

inWhat = atoi(argv[2]);

}

struct pf\_info\_struct info;

pgfltstat (inWhat, processNum, &info); // call to sys\_process(..)

retval=syscall(sys\_process, inWhat, processNum, &info);

if (retval>0)

{

printf(KERN\_INFO "Number of processes = %d\n", info.numberofProc);

printf(KERN\_INFO "Number of major page faults = %d\n",

info.majPgFault);

printf(KERN\_INFO "Number of minor page faults = %d\n",

info.minPgFault);

}

return 0;

}

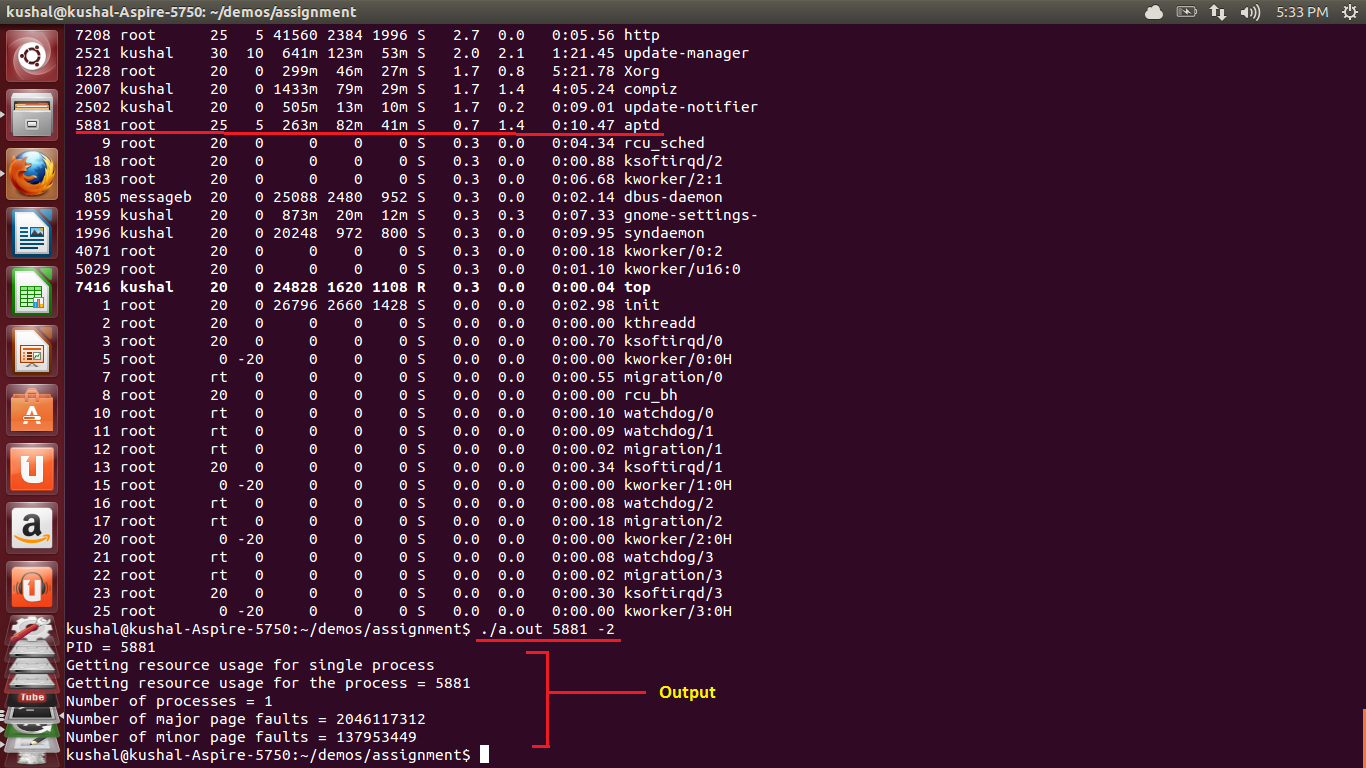
## Results and Analysis

The system call was tested for various processes and the results are shown below:

**Test 1:** Input values are a shown below

Val = -2🡺 means calculate page statistics for a single process

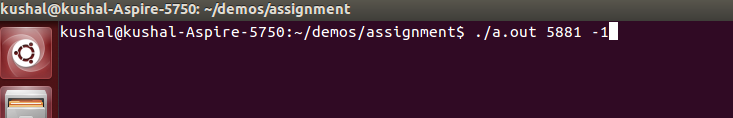
Pid = 5881🡺 processed.

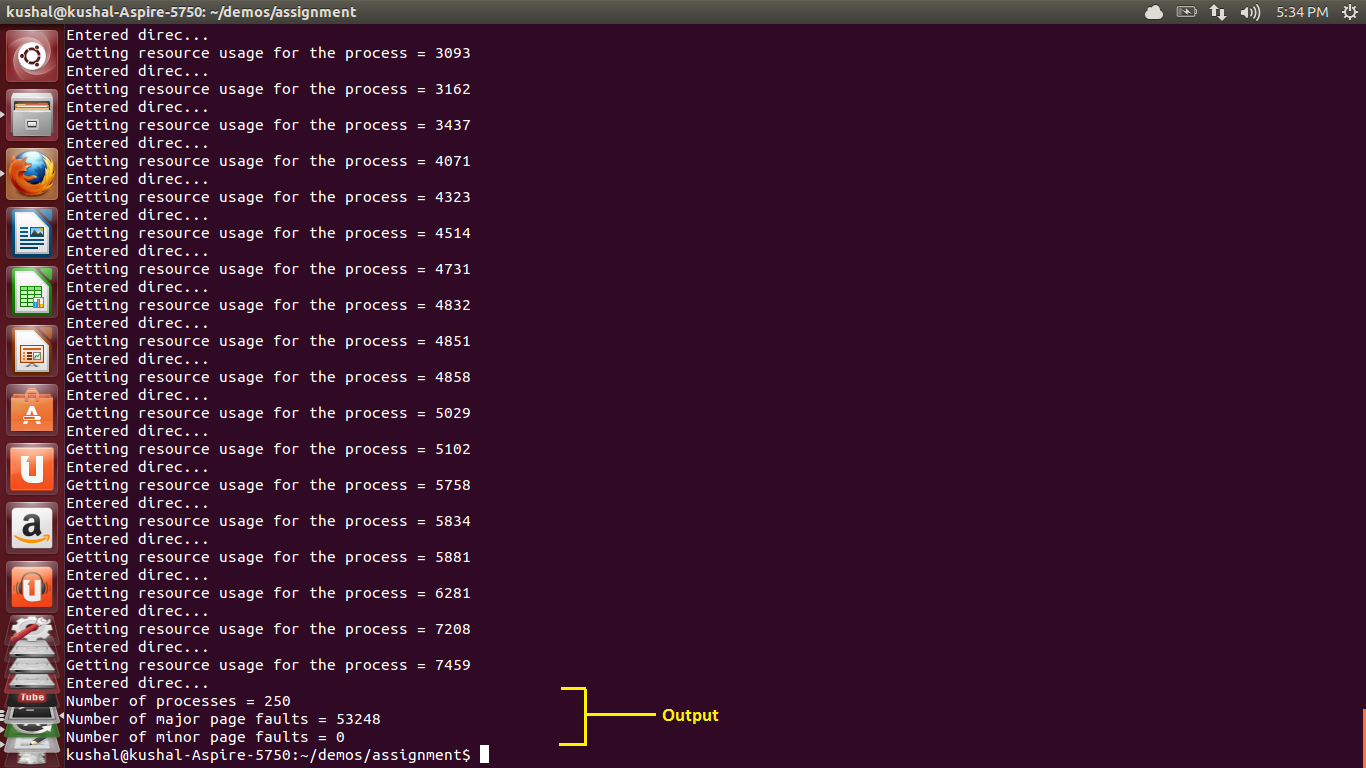


**Test 2:**

Val = -1🡺 means calculate page statistics for all the currently running processes in the system

Pid = 5881🡺 **invalid**.

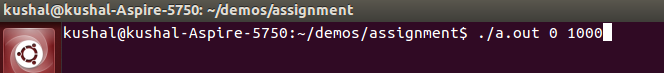


****

**Test 3:**

Val = 1000🡺 means calculate page statistics for all the currently running processes which belong to the group id 1000.

Pid = 5881🡺 **invalid**.



**PART-C**

# **Calculating page fault rate using system call**

\_\_\_\_­­­­­­­­\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

## File operation and Virtual memory Usage Application

The system call developed in Part B is tested using a user-space program which is given below:

(Note the directory scanning part of the code in the implementation of *pgflt\_stat\_info(..)* have been moved out since scanning a directory from inside the kernel space is dangerous and results in crashing of the kernel. This phenomenon was experienced only when the system-call was tested with a malloc’ed and file opening based userspace program. Also when the process gets completed, the directory from the */proc* folder is deleted and again the system-call results in the crash of the kernel).

The user space application to instigate page faults is shown below:

#include <sys/resource.h>

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <dirent.h>

#include <linux/kernel.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

const char \*prefix = "/proc";

struct pf\_info\_struct

{

int numberofProc;

int majPgFault;

int minPgFault;

};

//// function to scan the /proc directory

int dirScan (int val, pid\_t pid, struct pf\_info\_struct \*info)

{

struct rusage usage;

int process\_id=0;

long totMajPgFlt=0;

long totMinPgFlt=0;

struct dirent\* dirEntry = NULL;

struct stat buf;

int target = 0;

int count = 0;

int dummy = 0;

DIR\* dir=NULL;

if (val == -1)

{

printf("Getting resource usage for all the processes\n");

}

else if (val == -2)

{

printf("Getting resource usage for single process\n");

}

else

{

printf("Getting resource usage for group of process\n");

}

if (val != -2)

{

// find the pid's of all the current running processes

dir = opendir(prefix);

dirEntry = readdir(dir);

printf("Entered proc directory\n");

while( dirEntry != NULL)

{

// is a number? (pid)

//if (scanf(dirEntry->d\_name, "%d", &dummy) == 1)

//{

// get info about the node (file or folder)

lstat(dirEntry->d\_name, &buf);

// it must be a folder

//if (buf.st\_mode != S\_IFDIR)

{

//printf("Not a process 1\n");

//continue;

}

//printf("is it a folder");

// check if it's owned by the uid you need

if (val == -1)

{

if (count >= 999)

{

break;

}

process\_id = atoi(dirEntry->d\_name);

printf("1) Getting resource usage for the process = %d\n", process\_id);

count++;

}

else

{

process\_id = atoi(dirEntry->d\_name);

if (buf.st\_uid != val)

{

printf("Process %d doen not belong to group id %d\n", process\_id, val);

dirEntry = readdir(dir);

continue;

}

if (count >= 499)

{

break;

}

printf("Process %d belongs to group id %d\n", process\_id, val);

count++;

}

if (process\_id<0)

{

printf("Not a process 2\n");

dirEntry = readdir(dir);

continue;

}

**returnVal = syscall(315, inWhat, processNum, &info);/// <<<<<<<<<<<<<<<<<<<<<<<<<implemented system call**

totMinPgFlt += usage.ru\_minflt;// minor page faults: "page reclaims"

totMajPgFlt += usage.ru\_majflt;

//}

printf("Entered direc...\n");

dirEntry = readdir(dir);

}

}

if (val == -2)

{

process\_id = (int) pid;

printf("Getting resource usage for the process = %d\n", process\_id);

// find the resource usage for the specified process

**returnVal = syscall(315, inWhat, processNum, &info);/// <<<<<<<<<<<<<<<<<<<<<<<<<implemented system call**

totMinPgFlt = usage.ru\_minflt;// minor page faults: "page reclaims"

totMajPgFlt = usage.ru\_majflt;

count = 1;

}

info->majPgFault = totMinPgFlt;

info->minPgFault = totMajPgFlt;

info->numberofProc = count;

return 21;

}

int main(int argc, char\*\* argv)

{

Getting the process id of the current process(val=-2) **OR** group id(1000=Kushal group id)

int processNum = getpid();//1000;//0

int inWhat = -2;

int returnVal = 1;

struct pf\_info\_struct info;

int loop1 = 0;

Opening an example file( in this case “config.out” is taken as the example file because it has lot of contents. Also allocating1GB of memory

FILE\* fp;

char ch;

int count = 100000000;

char\* str =(char\*) malloc (count\*sizeof(char));

int loop = 0;

FILE\* fpOut = fopen("config.out", "w");

printf("program running.");

FILE\* fpRes = fopen("results.txt", "w");

printf("PID = %d\n", processNum);

long numberOfMajFlts=0;

long numberOfMinFlts=0;

long numberOfProcs=0;

dirScan(inWhat, processNum, &info);

numberOfMajFlts += info.majPgFault;

numberOfMinFlts += info.minPgFault;

numberOfProcs += info.numberofProc;

fprintf(fpRes, "Getting page faults for group process:\n");

fprintf(fpRes, "Number of major page faults = %ld\n", numberOfMajFlts);

fprintf(fpRes, "Number of minor page faults = %ld\n", numberOfMinFlts);

while (1)

{

fp = fopen("config", "r");/// opening a file

if (fp != NULL)

{

printf(".");

do {

if (loop == 100)

{

returnVal = dirScan(inWhat, processNum, &info);

numberOfMajFlts += info.majPgFault;

numberOfMinFlts += info.minPgFault;

//printf("Number of processes = %ld\n", numberOfProcs);

printf("Number of major page faults = %ld\n", numberOfMajFlts);

printf("Number of minor page faults = %ld\n", numberOfMinFlts);

fprintf(fpRes, "Number of major page faults = %ld\n", numberOfMajFlts);

fprintf(fpRes, "Number of minor page faults = %ld\n", numberOfMinFlts);

}

ch = getc(fp);

putc(ch,fpOut);

if (loop<count-1)

{

str[loop++] = ch;

}

Adding 1GB of memory inside the loop

else

{

loop = 0;

loop1++;

printf("Realloc happened\n");

str = (char\*)realloc((void\*)str,count);

}

}while (ch !=EOF);

fflush(stdout);

if ( loop1 >=15)

{

fclose(fp);

break;

}

}

fclose(fp); /// closing a file

fp=NULL;

}

fclose(fpOut); /// closing a file

dirScan(inWhat, processNum, &info);

free(str);

dirScan(inWhat, processNum, &info);

return 0;

}

The results for different values of “*val*” is shown below:

1. val = -2 (page fault statistics for single process)

|  |  |  |
| --- | --- | --- |
| Process Id | Number of major page faults | Number of minor page faults |
| 5881 | 0 | 0 |
| 5881 | 1 | 33204 |
| 5881 | 1 | 33204 |
| 5881 | 1 | 33204 |
| 5881 | 1 |  |
| 5881 | . |  |
| 5881 | . |  |
| 5881 | . |  |
| 5881 | 1 | 33204 |
| 5881 | 0 | 0 |

Current process Id = 5881 (which is running the above the user space application)

Average number of major page faults: 1

Average number of minor page faults: 33204

See Appendix A for detailed results

1. val = 1000( group id- page fault statistics for all the processes belonging to the group)

|  |  |  |
| --- | --- | --- |
| Group Id | Number of major page faults | Number of minor page faults |
| 1000 | 0 | 0 |
| 1000 | 85070 | 0 |
| 1000 | 6055715 | 0 |
| 1000 | 12200211 | 0 |
| 1000 | 18493151 | 0 |
| 1000 | 24934771 | 0 |
| 1000 | 31525071 | 0 |
| 1000 | 38264051 | 0 |
| 1000 | 45151711 | 0 |
| 1000 | 52188287 | 0 |
| 1000 | . | . |
| 1000 | . | . |
| 1000 | 105608303 |  |
| 1000 | 97530731 |  |
| 1000 | 89601839 |  |
| 1000 | 81821627 |  |
| 1000 | 74632543 |  |
| 1000 | . |  |
| 1000 | . |  |
| 1000 | . |  |

Current Group Id: 1000

Average Major Page faults: 53490585.76

Average minor page faults: 0

See Appendix B for detailed results

1. val = -1 and( get the page faults for all the processes running in the system)

|  |  |
| --- | --- |
| Number of major page faults | Number of minor page faults |
| 0 | 0 |
| 85176 | 0 |
| 6030882 | 0 |
| 12123774 | 0 |
| 18363852 | 0 |
| 24751350 | 0 |
| 31286034 | 0 |
| 37968138 | 0 |
| 44797662 | 0 |
| 51774840 | 0 |
| 58899438 | 0 |
| 66171456 | 0 |
| 73590894 | 0 |
| . | . |
| . | . |
| . | . |
| . | . |
| 104743782 | 0 |

Average Major Page faults: 53064469.06

Average minor page faults: 0

See Appendix C for detailed results

## Calculation of Page frames

The virtual memory usage and the ram usage of the current application process are calculated using the following function:

void process\_mem\_usage(double& vm\_usage, double& resident\_set)

{

using std::ios\_base;

using std::ifstream;

using std::string;

vm\_usage = 0.0;

resident\_set = 0.0;

// 'file' stat seems to give the most reliable results

//

ifstream stat\_stream("/proc/self/stat",ios\_base::in);

// dummy vars for leading entries in stat that we don't care about

//

string pid, comm, state, ppid, pgrp, session, tty\_nr;

string tpgid, flags, minflt, cminflt, majflt, cmajflt;

string utime, stime, cutime, cstime, priority, nice;

string O, itrealvalue, starttime;

// the two fields we want

//

unsigned long vsize;

long rss;

stat\_stream >> pid >> comm >> state >> ppid >> pgrp >> session >> tty\_nr

>> tpgid >> flags >> minflt >> cminflt >> majflt >> cmajflt

>> utime >> stime >> cutime >> cstime >> priority >> nice

>> O >> itrealvalue >> starttime >> vsize >> rss; // don't care about the rest

stat\_stream.close();

long page\_size\_kb = sysconf(\_SC\_PAGE\_SIZE) / 1024; // in case x86-64 is configured to use 2MB pages

vm\_usage = vsize / 1024.0;

resident\_set = rss \* page\_size\_kb;

}

## Graphical representation and analysis

* **Page faults vs page frames for single process**

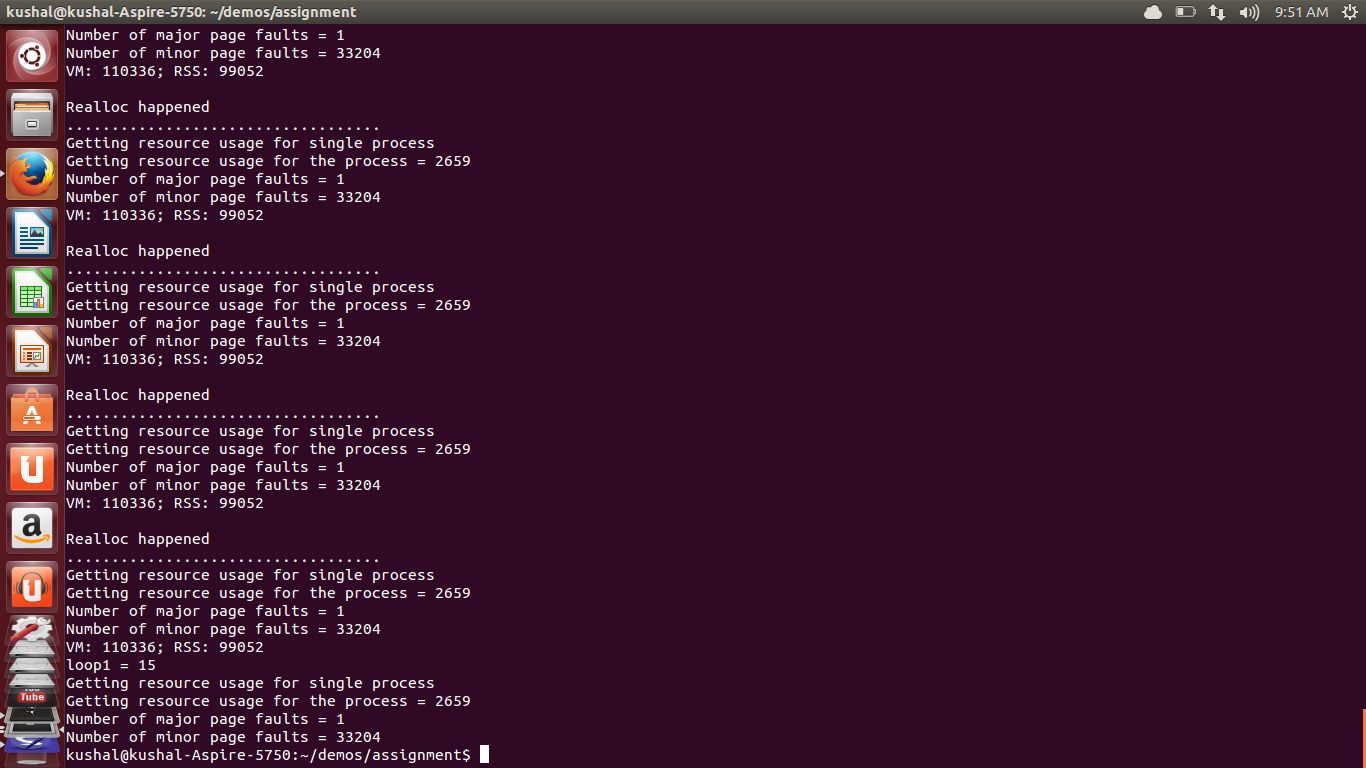
****

Figure 3.1: Result of calculating page faults and page frames for single process

Figure 3.2: Graphical representation of the obtained values. Blue: Major Mage faults, Red: Minor page faults, Maroon: Page frames, Greeen: RAM size

* Analysis - From figure 3.1 and 3.2 we infer that the application program perform a major page fault when the file is opened initially and for subsequent file open’s and closes’ the page fault does not occur as the file is moved to the cache memory and hence the page fault does not occur. Susbsequently the minor page faults staturates at some point because the RAM usage increases. Hence, memory allocation happens inside the RAM.
* **Page faults vs page frames for group of processes**

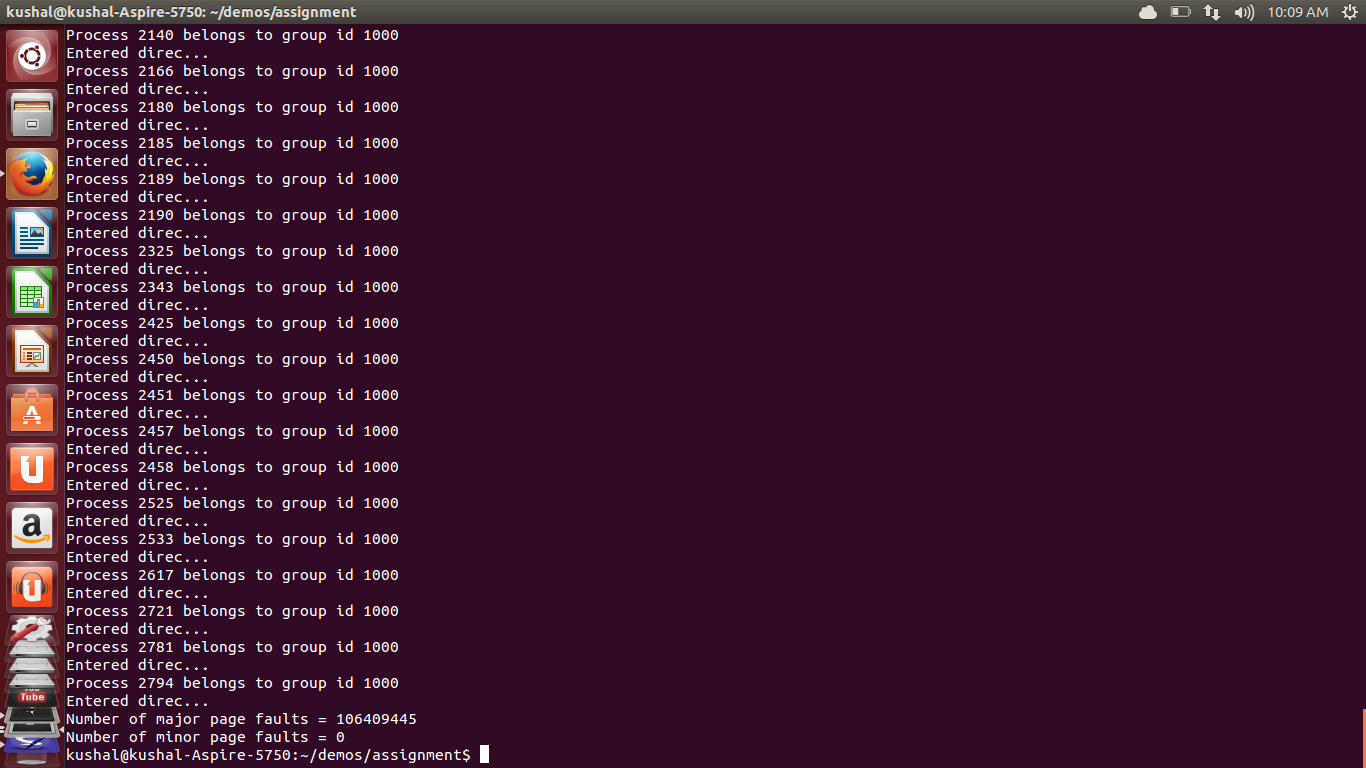
****

Figure 3.3: Results of page faults and page frame calculation for a group of processes

Analysis: From figure 3.3 and 3.4 we see that the magnitude of minor page faults constant while the magnitude of major page faults are increasing exponentially. The above tabulations are recorded when major page faults are calculated for a group of processes with group id = 1000.

Figure 3.4: Graphs showing the rate of change of major faults wrt to the rate of change of page frames.

* **Page faults vs page frames for all processes**

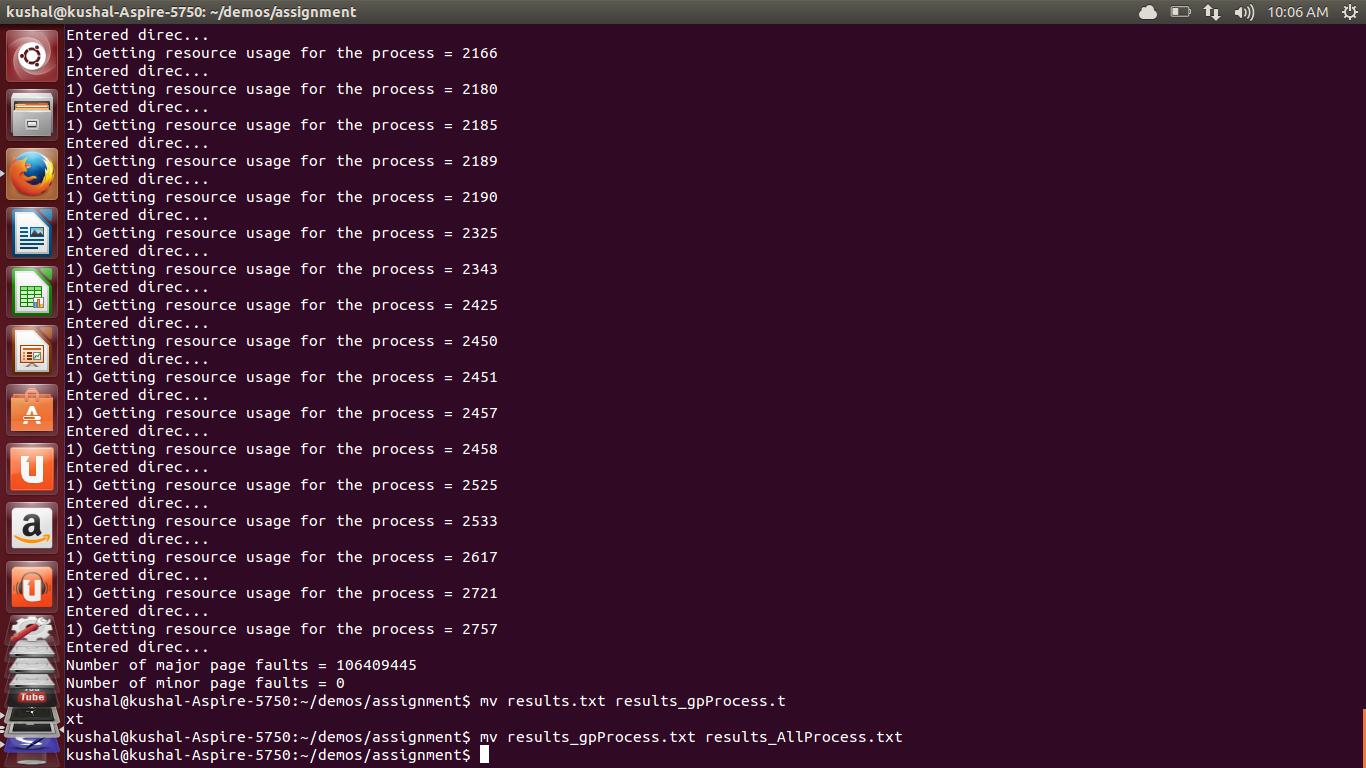


Figure3.6: Page frames rate vs page fault rate for all processes.

The above results are recorded for all the current running processes in the system.

# **References**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. Bernard Blackham, Vernon Tang and Gernot Heiser, Winbond Electronics, *To Preempt or Not To Preempt, That Is the Question*
2. Amit Choudhary, *Implementing a System Call on Linux 2.6 for i386*, 2006−10−27, 1999
3. Freescale Semiconductor, David Beal, *Linux® As a Real-Time Operating System*
4. Amarpreet Singh Ugal , Intel CorporationHard Real Time Linux\* using Xenomai\* on Intel® Multi-Core Processors
5. Leonid Ryzhyk, Yanjin Zhu, Gernot Heiser, The Case for Active Device Drivers
6. *Paul N. Leroux,* RTOS versus GPOS
7. David Beal, Real-time Linux Basics