Virtual Memory Page Fault Measurement (MP3 Q&A)

Shiguang Wang

Mar 21th, 2014

Objectives

- Learn the basics of Linux Virtual to Physical Page Mapping
- Learn the concept of Page Fault and Page Fault rate
- Design a lightweight tool that can profile page fault rate
- Implement a profiler tool as Linux Kernel Module
- Learn the Linux kernel-level API for:
  - Work queue
  - Character Device Driver
  - `vmalloc`
  - `mmap`
- Analyze the profiled data for some test scenarios

Introduction

- There is a huge performance gap between the memory and the hard disk
- As a result, the performance of the virtual memory system plays a major role in the overall performance of the Operating System
  - E.g. Inefficient replacement of memory pages, affects user level programs by:
    - Increasing the Response time
    - Lowering the Throughput

Page Fault

- Page Fault is a trap to the software raised by the hardware when:
  - A program accesses a page that is mapped in the Virtual address space but not loaded in the Physical memory
- In general, OS tries to handle the page fault by bringing the required page into physical memory.
- The hardware that detects a Page Fault is the Memory Management Unit of the processor
- However, if there is an exception (e.g. illegal access) that needs to be handled, OS takes care of that

Minor Page Fault (or Soft Page Fault)

- The page is loaded in memory but not marked in the MMU as loaded
  - E.g. Memory is shared by different programs and the page is already in memory for other programs
- The page fault handler of OS needs to fix the MMU entry
- No need to read the page into memory

Major Page Fault (or Hard Page Fault)

- The page is not in physical memory when needed
- OS page fault handler has to:
  - Find a free page in memory
  - Or choose a page memory to be evacuated, write back that page memory to disk
  - Bring in the required page from disk
- Major Faults are expensive as they add disk latency

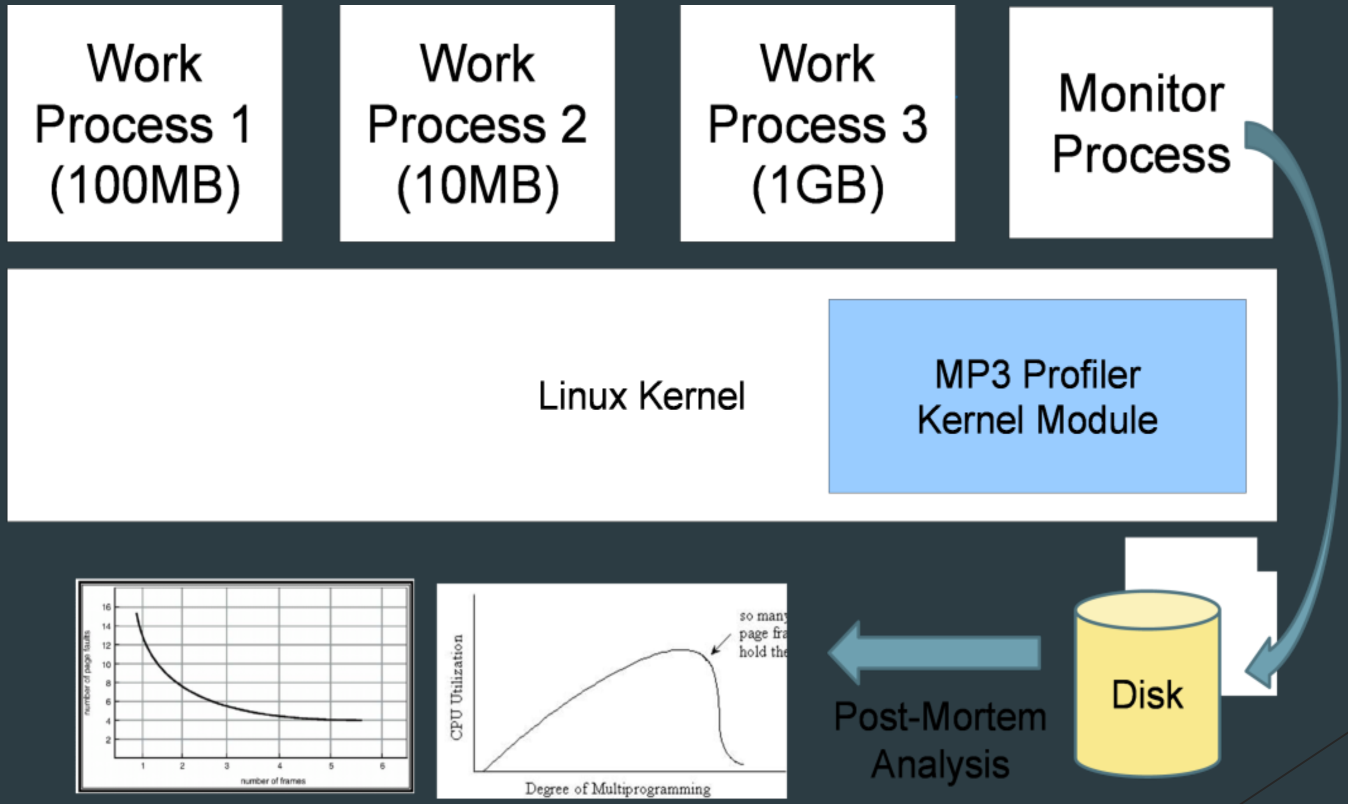
Effect of Page Fault on System Performance

- Major Page Fault are much more expensive. *How much?*
  - HDD average rotational latency : 3ms
  - HDD average seek time: 5ms
  - Transfer time from HDD: 0.05ms/page
    - Total time for bringing in a page = 8ms= 8,000,000ns
  - Memory access time: 200ns
  - Thus, Major Page Fault is 40,000 times slower

Profiling Page Fault Rate

- Page Fault info is only available in kernel address space
- If we want to profile this data in a user space process:
  - Have to switch context between user and kernel space
  - Copy data between these two spaces
  - This has significant performance overhead
- Instead, we use a Linux Kernel Module to extract the page fault rate and CPU utilization

Overview of the MP3



MP3 Introduction

- The emulating user-level test program and the monitor programs are provided ☺
- The major focus is:
  - To build a kernel-module that extracts (major and minor) page fault and utilization information of registered tasks
  - Expose the extracted info by using a memory buffer that is directly mapped into the virtual address space of the monitor process

Proc Filesystem

- We use a single Proc Filesystem entry for registering and unregistering the user-level processes => `/proc/mp3/status`
  - This is similar to MP2
  - It allows 3 operations:
    - Register a user process: `R <PID>`
    - Deregister a user process: `U <PID>`
    - Read registered task list:
      - A user-level application should be able to read all registered user-processes by reading `/proc/mp3/status`

Character Device Driver

- Character Device is used to map:
  - the kernel buffer memory of the profiler Kernel Module to
  - the virtual address space of a requesting user-level process (i.e. the monitor process in MP3)

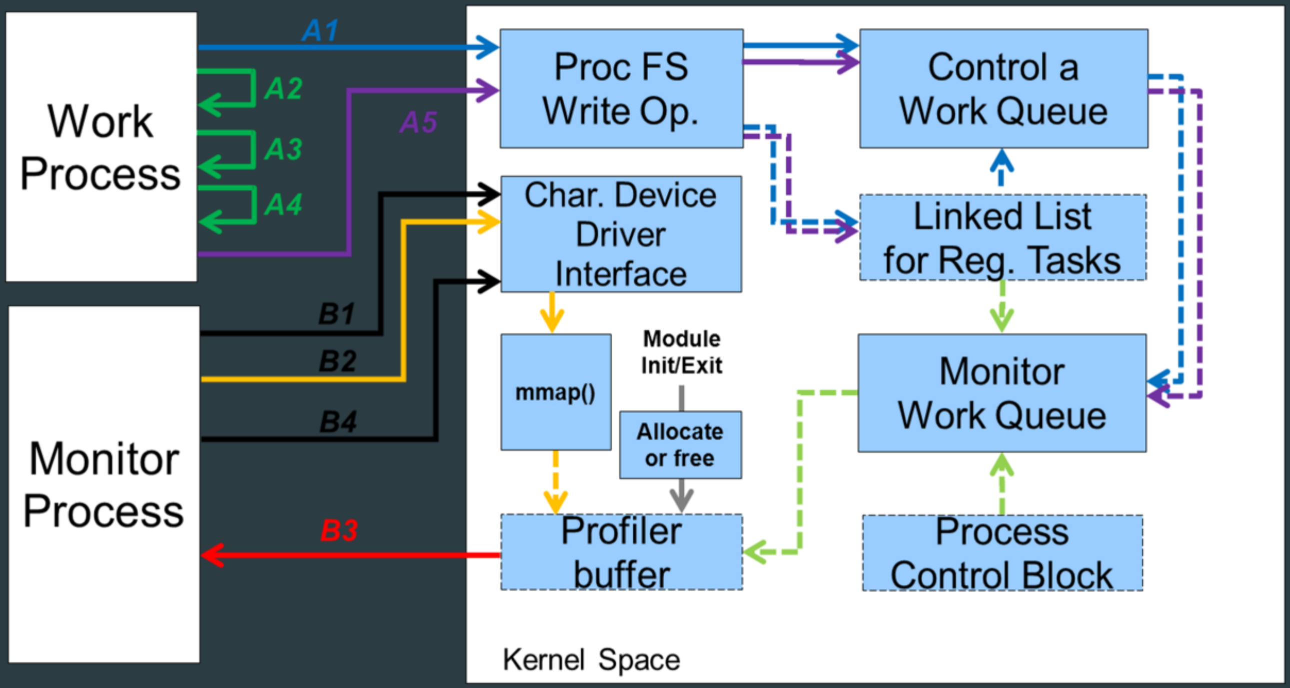
Test Application

- A program that can run as a work process for case studies is given ☺
  - This is a single-threaded user-level application
  - It allocates a requested size of virtual memory space (e.g. up to 1GB)
  - It accesses these allocated memory in a certain locality pattern (i.e. random or temporal locality)
  - Iterates for a requested number of times

Monitor Application

- Monitor applications requests the kernel module to map the kernel-level profiler buffer to its user-level address space, also given ☺
- This request is sent by using the character device driver created by the kernel module
- This application reads the profiling values (major and minor page faults and the utilization of all registered processes) and print these values to a standard output.
- By piping its output to a file, we can store the profiled data into a regular file in order to access them later.

MP3 Details



A1. Register *A2. Allocate Memory Block* *A3. Memory Accesses* *A4. Free Memory Blocks*
A5. Unregister **B1. Open** **B2. mmap()** **B3. Read Profiled Data** ***B4. Close***

Linux Kernel Contexts

- Linux offer 3 contexts for the kernel:
  - Process:
    - executes directly on behalf of a user process
    - All Syscalls run in process context
  - Bottom-Half:
    - Traditionally, lengthy part of code runs here
  - Interrupt
    - Interrupt handlers are run here
- As we move from Interrupt context to process context, higher degree of processor sharing becomes available

Deferring Work

- It is common in kernel code to defer part of the work
- E.g. Interrupt handler code
  - Some or all interrupts are disabled when handling it
  - While handling one, we might lose new interrupts
  - So, make the handling as fast as possible
    - Split interrupt code into:
      - Top Part: that runs within the interrupt context
        - So, interrupts are disabled
      - Bottom Part: The lengthy part of it that runs in process context
        - Now, interrupts are enabled
- The result is better performance because of:
  - quick response to interrupts
  - by deferring non-time-sensitive part of the work to later

Linux Solutions

- Linux provides several options for deferring part of the work:
  - Timers:
    - allow work to be deferred for a certain length of time
  - Bottom Halves:
    - Traditional solution of Linux for deferring work
    - This is replaced by SoftIRQ since Kernel 2.3
  - SoftIRQ:
    - 32 of them are statically defined in Linux kernel
    - Had to be defined at compile time
    - Performed the bottom half of the code within a kernel thread context
    - Source code can be found in ./kernel/softirq.c

Linux Solutions

- Other Linux solutions for deferring work:
  - Tasklets
    - Allow dynamic creation of deferrable functions
    - Are defined in ./include/linux/interrupt.h
  - Work Queues
    - Starting from Kernel 2.5, Work Queues are defined
    - Allows deferring of the code to outside of the interrupt context and into the kernel process context
    - Are defined in ./linux/workqueue.h

Linux Solutions

- How different entities within Linux kernel be interrupted by others:

	HW-IRQ	Soft-IRQ	Tasklet
Hardware IRQ	+/-	-	-
Software IRQ	+	-	-
Tasklet	+	-	-
System call	+	+	+
Process	+	+	+

Creating/Destroying a Work Queue

- In order to create a work queue, you need to:
  - Call the create_workqueue() function
  - Which returns a workqueue_struct reference
    *struct workqueue_struct *create_workqueue(name);*

- It can later be destroyed by calling the destroy_workqueue() function
  - *void destroy_workqueue(struct workqueue_struct *);*

Creating/Destroying a Work Queue

- The work to be added to the queue is
  - Defined by *struct work_Struct*
  - Initialized by calling the INIT_WORK() function
    - *INIT_WORK(struct work_struct *work, func);*
- *Now that the work is initialized, it can be added to the work queue by calling one of the following:*
  - *int queue_work(struct workqueue_struct *wq, struct work_struct *work);*
  - *int queue_work_on(int cpu, struct workqueue_struct *wq, struct work_struct *work);*

Creating/Destroying a Work Queue

- There are a few helper functions to make the work easier:
  - Flush_work(): to flush a particular work and block until the work is complete
    - *int flush_work(struct work_struct *work);*
  - Flush_workqueue(): similar to flush_work() but for the whole work queue
    - *int flush_workqueue(struct workqueue_struct *wq);*

Creating/Destroying a Work Queue

- There are a few helper functions to make the work easier:
  - Cancel_work(): to cancel a work that is not already executing in a handler
    - The function will terminate the work in the queue
    - Or block until the callback is finished (if the work is already in progress in the handler)
    - *int cancel_work_sync(struct work_struct *work);*
  - Work_Pending(): to find out whether a work item is pending or not
    - *work_pending(work);*

Work Queue Example

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/workqueue.h>
MODULE_LICENSE("GPL");
static struct workqueue_struct *my_wq;
typedef struct {
    struct work_struct my_work;
    int x;
} my_work_t;
my_work_t *work, *work2;
static void my_wq_function( struct work_struct *work)
{
    my_work_t *my_work = (my_work_t *)work;
    printk( "my_work.x %d\n", my_work->x );
    kfree( (void *)work );
}
```

```
int init_module( void ) {
    int ret;
    my_wq = create_workqueue("my_queue");
    if (my_wq) {
        /* Queue some work (item 1) */
        work = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);
        if (work) {
            INIT_WORK( (struct work_struct *)work, my_wq_function );
            work->x = 1;
            ret = queue_work( my_wq, (struct work_struct *)work );
        }
        /* Queue some additional work (item 2) */
        work2 = (my_work_t *)kmalloc(sizeof(my_work_t), GFP_KERNEL);
        if (work2) {
            INIT_WORK( (struct work_struct *)work2, my_wq_function );
            work2->x = 2;
            ret = queue_work( my_wq, (struct work_struct *)work2 );
        }
    }
    return 0;
}
```

Other Possibly Useful API's

- Proc directory and file
  - proc_mkdir_mode(…)
  - create_proc_entry(…)
  - remove_proc_entry(…)
- Profiler buffer
  - vmalloc(…)
  - SetPageReserved(…) // avoid page being swapped out
  - ClearPageReserved(…)
  - vfree(…)
- Device driver
  - alloc_chrdev_region(…)
  - cdev_init(…)
  - cdev_add(…)
  - cdev_del(…)
  - unregister_chrdev_region(…)