CS 423
Operating System Design: MP3 Walkthrough

Andrew Yoo
(All content taken from a previous year's walkthrough)
• **Understand** the Linux virtual to physical page mapping and page fault rate.

• **Design** a lightweight tool that can profile page fault rate.

• **Implement** the profiler tool as a Linux kernel module.

• **Learn** how to use the kernel-level APIs for character devices, `vmalloc()`, and `mmap()`.
• Performance gap between memory and disk
  – Registers: \(~1\)ns
  – DRAM: \(50-150\)ns
  – Disk: \(~10\)ms, hundreds times slower than memory!

• Performance of the virtual memory system plays a major role in the overall performance of the Operating System

• Inefficient VM replacement of pages
  – Bad performance for user-level programs
  – 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 like accessing null pointer) that needs to be handled, OS takes care of that.
• **Major page fault**
  – Handled by using a disk I/O operation
  – Memory mapped file
  – Page replacement / Cold Pages
  – *Expensive as they add to disk latency*

• **Minor page fault**
  – Handled without using a disk I/O operation
  – `malloc()`, `copy_on_write()`, `fork()`
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
MP3 Overview

- **Work Process 1 (100MB)**
- **Work Process 2 (10MB)**
- **Work Process 3 (1GB)**
- **Monitor Process**

Linux Kernel

MP3 Profiler Kernel Module

CPU Utilization

Degree of Multiprogramming

Post-Mortem Analysis

Disk
• Major page fault

• Minor page fault

• CPU utilization
  – Calculated as a rate
    • For task $T$: $U_T = \frac{\text{cpu time}_T}{\text{wall time}} = \frac{\text{stime}_T + \text{utime}_T}{\text{jiffies}}$
    • stime: Time spent in kernel space
    • utime: Time spent in user space
Thrashing
Measurement

• **Accuracy of Measurement**
  – Many profiling operations are needed in a short time interval.

• **Copy to user space causes a significant performance overhead**

• **Solution: Use Shared Memory**
Memory Map

Virtual Addr.

Physical Addr.

Virtual Addr.

mmap()
• A character device driver is used as a control interface of the shared memory
  – Map Shared Memory (i.e., mmap()): To map the profiler buffer memory allocated in the kernel address space to the virtual address space of a requesting user-level process

• Shared memory
  – Normal memory access: Used to deliver profiled data from the kernel to user processes
• Three types interfaces between the OS kernel module and user processes:
  – a Proc file
  – a character device driver
  – a shared memory area
• Proc filesystem entry (/proc/mp3/status)
  – **Register:** Application to notify its intent to monitor its page fault rate and utilization.
    • ‘R <PID>’
  – **Deregister:** Application to notify that the application has finished using the profiler.
    • ‘U <PID>’
  – **Read Registered Task List:** To query which applications are registered.
    • Return a list with the PID of each application
• **Work program** (given for case studies)
  – A single threaded user-level application with three parameters: 
    *memory size*, *locality pattern*, and *memory access count* per iteration
    • Allocates a request size of virtual memory space (e.g., up to 1GB)
    • Accesses them with a certain locality pattern (i.e., random or temporal locality) for a requested number of times
    • The access step is repeated for 20 times.
  – Multiple instances of this program can be created (i.e., forked) simultaneously.
• **Monitor application** is also given
  - Requests the kernel module to map the kernel-level profiler buffer to its user-level virtual address space (i.e., using `mmap()`).
    • This request is sent by using the character device driver created by the kernel module.
  
    - The application reads profiling values (i.e., major and minor page fault counts and utilization of all registered processes).
  
    - By using a pipe, the profiled data is stored in a regular file.
      • So that these data are plotted and analyzed later.
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
  – Top half
  – Bottom half

• Better performance because :
  – quick response to interrupts
  – by deferring non-time-sensitive part of the work to later
Work Queue

- Bottom-half mechanism used to defer work
- Work queues run in process context.
  - Work queues can sleep, invoke the scheduler, and so on.
  - The kernel schedules bottom halves running in work queues.

- The work queue execute user’s bottom half as a specific function, called a work queue handler or simply a work function.

- Linux provides a common work queue but you can also initialize your own
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 * );`
• 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 );`
• **Flush_work()**: to flush a particular work and block until the work is complete
  
  – \texttt{int flush\_work( struct work\_struct \*work );}

• **Flush_workqueue()**: similar to flush_work() but for the whole work queue
  
  – \texttt{int flush\_workqueue( struct workqueue\_struct \*wq );}
Creating/Destroying a Work Queue

• **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 );`
• Initialize data structure
  – `void cdev_init(struct cdev *cdev, struct file_operations *fops);`

• Add to the kernel
  – `int cdev_add(struct cdev *dev, dev_t num, unsigned int count);`

• Delete from the kernel
  – `void cdev_del(struct cdev *dev);`
static int my_open(struct inode *inode, struct file *filp);

static struct file_operations my_fops = {
    .open = my_open,
    .release = my_release,
    .mmap = my_mmap,
    .owner = THIS_MODULE,
};
• Gets Page Frame Number
  – \texttt{pfn} = \texttt{vmalloc\_to\_pfm(virt\_addr)};

• Maps a virtual page to a physical frame
  – \texttt{remap\_pfm\_range(vma, start, pfn, PAGE\_SIZE, PAGE\_SHARED)};
    (see http://www.makelinux.net/ldd3/chp-15-sect-2)
More Questions?

• Office hours

• Campuswire