CS423: MP3

Brad Misik and Ivan Hu

November 4, 2011

Abstract

Implementation, development, and case studies of a page fault instrumenting Linux kernel module.

1 Introduction

Although we read through and implemented the MP3 spec step-by-step, there were quite a few places where our implementation deviated. Those deviations, as well as a brief overview of the project and the case studies involved, will be outlined in this document.

1.1 Environment Setup

As with MP2, I used my local virtual machine, because the remote machines were painfully slow to work on. I created a git repo of the root folder of MP2 and uploaded it to share with the team.

1.2 Initial Development

The majority of MP3 was just refactoring the solution from MP2. The struct and variable names were modified, some functions were deleted, but the majority of both the structure, variables in use, and functions remained in the code.

2 Implementation Deviations

2.1 Workqueue

For the work queue, we decided to create it when loading our module and destroy it when exiting our module. We had initially followed the spec and had the queue only exist when tasks were in our task list, but the overhead of keeping it in memory is presumed to be negligible, and it makes intuitive sense to have its life span over the life of our module.

2.2 Buffer Allocation and Mapping

In the 2.6 kernel and above, it appears as though the PG_reserved page flags are being phased out, and the more appropriate (and WAY easier way) of allocation a buffer for use in userspace is the vmalloc_user() function. Once using this function for buffer allocation, said buffer may be mmaped into a vma by way of the remap_vmalloc_range() function. That's it - two simple calls and the blunt of this MP was taken care of.

2.3 Work and Monitor

We used a static delayed work structure so we didn't have to go and reinitialize our work every period, or have to worry about timers. This was the most simple solution. Our work handler uses a pointer into our vmalloc'd ring buffer to a sample struct. This sample struct has members for all of the necessary values to be stored each period. We modified the monitor process to use this sample struct to easily iterate through the buffer dumping out its data. The included sample header file also defines the number of (max) samples, to synchronize the buffer information between kernel module and monitor.

3 Case Studies

3.1 Case Study 1: Random vs Local

Graph 1 had a steeper curve than graph 2, though some of that may be distortion because of the scaling of the x axis. The scaling of the x axis is different because graph 1 had a much longer running time (80189 vs 51502 jiffies), which is because graph 1 was made to test random based access, and graph 2 locality based access. Locality based access is faster since it is likely that adjacent memory values are also in already loaded page tables. The random based access needs to swap out more page tables, especially when its buffer size is larger than physical memory. Graph 1 does appear to have more page faults per unit time, which is easy to answer at the surface, for the same reason that random access causes more page faults than locality based access. However, that also means that it takes more time and therefore needs to take longer to recover from those page faults before it has more, probably leading to a cap on the number of page faults possible in one period. The moral of the story is that locality based access is desired if you want to minimize overall page counts and shorten the time taken for a process.

3.2 Case Study 2: Multiprogramming

The utilization increases as N increases. The number of page faults also increase, as well as the completion time. This is a very logical result, considering that more tasks, requiring the same computation, will take a longer time to complete since they are essentially N times the amount of work of one task. The number of page faults also increases because these tasks are competing for pages and

having to swap in and out as the tasks themselves are swapped in and out of using the CPU. The utilization also increases because the scheduler will give less time to all other processes the more processes that are added. Since we're adding more processes under our control, we're seeing more total utilization for them.

3.3 Case Study 3: RT Synchronization

Although the data logging needs to be done in real-time, we have a large 30 second deadline to play with, which, with modern computing machines, allows us to hack together a wide variety of solutions that wouldn't work with smaller deadlines. The problem here is that the monitor process, when reading the buffer, might seek too far ahead of the buffer, or lag too far behind and get passed up by the writer, thus losing samples or recording bad data. The intuitive solution is a shared semaphore (perhaps placed in the shared buffer itself), that will be incremented when the kernel module takes a sample, and decremented when the monitor logs a sample. And in fact, this can also be done so that the monitor only tries to read and decrement the semaphore every 30 seconds when a timer goes off. We would prefer this solution, since it is more elegant, but since we don't know a good way to implement a semaphore in a shared kernel-userspace buffer, we will write pseudocode for something less robust. It would also be trivial to have the monitor process register with the kernel module, and, when the first process is registered, a 30 second timer will be started in the kernel module. When this timer goes off, the kernel module writes the address or index of where it is in the ring buffer and sends a signal to the monitor process. The monitor process then reads up to this address/index and logs the data. It then waits for the next signal. It's a ring buffer, so when the reader or writer reach the end of the buffer, they continue back around.

```
#buffer_position stores the last saved position
#of the kernel module writes to the ring buffer

(module)
register(type, pid, ...):
  if type == MONITOR:
    monitor = pid
  ...

timer_handler:
  *buffer_position = current_sample
  signal(monitor)

(monitor)
signal_handler:
  while current_sample < *buffer_position:
```

```
#assume ++ wraps around the ring buffer if needed
log(current_sample++)
```