

Recitation Week #9

Scheduling Algorithms and Kernel Debugging

Alejandro Cabrera
Operating Systems
COP4610 / CGS5765

Today's Recitation

- Kernel Debugging
- Scheduling Algorithms
- Project 3 Hints
- Synchronization Primitives Brief

Kernel Debugging

- Configuring the Kernel With Debugging in Mind
- Monitoring
- /proc/ Programming for Debugging
- Anatomy of a Kernel OOPS
- Defensive Programming

Configuring the Kernel for Debugging

- Available (possibly relevant to this course) options:
 - Timing info on printk
 - Enable `__deprecated` logic
 - `debugfs`
 - Detect hung tasks
 - SLUB debugging
 - Kernel Memory Leak Detector
 - All mutex/lock debugging
 - `kmemcheck`
 - Check for stack overflow
 - Debug linked list manipulation
 - Kernel Tracing Framework

SLUB Debugging

- Inserts special *poison* bits into each kernel memory allocation.
- If an OOPS occurs- a kernel segfault-, you'll see that special pattern arise if the OOPS was created by your module.

Kernel Memory Leak Detector

- Maintains a queue in debugfs.
- Reports all memory leaks triggered by any kernel operation.
- A great way to discover all your memory leaks, as there are **NO FALSE POSITIVES**.
 - If the kernel memory leak detector signals a memory leak, investigate the cause!
- Requires mounting debugfs.
 - `sudo mount -t debugfs nodev /sys/kernel/debug/`
 - `more /sys/kernel/debug/kmemleak.txt`

Check for Stack Overflow

- Stack overflows are a dangerous and silent error.
 - A stack overflow may overwrite data in an adjacent function call, producing undefined behavior.
- In short: enable all checks for stack overflows.

Debugging by Monitoring

- Monitoring refers to the strategic insertion of calls to `printk()`.
- Allows prefixing of priority to message:
 - _ `KERN_EMERG` /* critically urgent */
 - _ `KERN_ALERT`
 - _ `KERN_CRIT`
 - _ `KERN_ERR`
 - _ `KERN_WARNING`
 - _ `KERN_NOTICE`
 - _ `KERN_INFO`
 - _ `KERN_DEBUG` /* least urgent */
- No priority `printk()` defaults to some priority.
- Read messages by typing `dmesg` or by issuing `sudo more /proc/kmesg`.
 - _ `sudo more /proc/kmesg` – monitor mode

procfs for Debugging

- General process:
 - Identify data to monitor in your module
 - Create a proc entry to monitor this data
 - Run your module
 - Query `/proc/<entry>` for that information at any time

Anatomy of a Kernel OOPS

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU:      0
EIP:      0060:[<d083a064>]      Not tainted
EFLAGS: 00010246      (2.6.6)
EIP is at faulty write+0x4/0x10 [faulty]
eax: 00000000    ebx: 00000000    ecx: 00000000    edx: 00000000
esi: cf8b2460    edi: cf8b2480    ebp: 00000005    esp: c31c5f74
ds: 007b    es: 007b    ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460 cf8b2460
       ffffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005 cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005 00000005
Call Trace:
[<c0150558>] vfs_write+0xb8/0x130
[<c0150682>] sys_write+0x42/0x70
[<c0103f8f>] syscall_call+0x7/0xb
Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

Inspecting EIP

EIP: 0060: [<d083a064>]

Not tainted

EFLAGS: 00010246 (2.6.6)

EIP is at faulty_write+0x4/0x10 [faulty]

- EIP – Instruction Pointer, the last instruction executed prior to the OOPS.
- Best hint the kernel can give you without any debugging effort.

Inspecting Call Trace

Call Trace:

[<c0150558>] vfs_write+0xb8/0x130

[<c0150682>] sys_write+0x42/0x70

[<c0103f8f>] syscall_call+0x7/0xb

- Series of function calls leading to OOPS.
 - Similar to gdb bt/backtrace.

Defensive Programming

- Infinite loops and deadlocks at the kernel level hang your machine
 - Ctrl-Alt-Del has NO effect
 - Ctrl-C does not matter
 - Ctrl-D does not matter
 - You may only **reboot**
- How do you protect yourself?
 - Use `schedule()` strategically

Invoking the Scheduler

```
#include <linux/sched.h>

void schedule(void);
```

- Relinquishes CPU time to other processes.
- Insert in the middle of a loop you suspect may not terminate
- Insert in a region that uses mutexes or semaphores for locking.
 - DO NOT insert in a spin-lock protected region.

Debugging Tools not Covered

- LTT – Linux Tracing Framework
- Dprobes – Dynamic Probes
- gdb – Invoking gbd on the kernel image
- kgdb – A remote debugger for the kernel
- Magic SysRq
- debugfs – In detail!
- procfs – seq_file interface for large proc entries.
- printk – Rate limiting, turning on/off

Scheduling *Heuristics*

- FIFO
- Shortest Seek Time First
- SCAN
- C-SCAN
- LOOK
- C-LOOK
- More?

Scheduling Heuristics vs. Scheduling Algorithms

- A **heuristic** does not make any guarantees on the performance it is expected to provide.
 - No lower bounds, no upper bounds, no worst case.
- A heuristic is implemented because it *appears* to work well in most cases in practice.
- An **algorithm** models the problem and input, and making certain assumptions, guarantees certain bounds on performance.

Recent Result: Precise Disk Scheduling Algorithm is NP-Hard

- Disk Scheduling problem to optimize throughput proven to be NP-Hard.
 - Reduction to Traveling Salesman Problem
- Basically, defining some manner of cost function on picking up a request, say seek time, it is *really* expensive computationally to pick out the best request processing order.
- Details here:
 - M. Andrews, M. A. Bender, L. Zhang. *New Algorithms for Disk Scheduling*. Bell Labs. 2008.

Review of Scheduling Heuristics: FIFO

- Method:
 - Service requests in order they arrive
- Pros:
 - + No computational time expended to choose requests
 - + Implementation: As easy as it gets
 - + Fair: Every requests gets a turn
- Cons:
 - - Terrible for random requests!
 - Elevator may backtrack back and forth several times in processing requests
 - - Low throughput potential

Review of Scheduling Heuristics: Shortest Seek Time First

- Method:
 - Service requests near elevator first
- Pros:
 - + Very low computational time expended
 - + Implementation: Fairly easy
 - + Throughput: Fairly high – may minimize seek time
- Cons:
 - - Unfair: May starve distant requests
 - - May ignore distant clusters of requests

Review of Scheduling Heuristics: SCAN

- Method:
 - Service requests in one direction, then go backwards
- Pros:
 - + Very low computational time expended
 - + Implementation: Fairly easy (go up, go down, go up)
 - + Throughput: somewhat high
 - + Fair: No starvation
- Cons:
 - - May take up to two elevator trips to collect a set of requests (up, down, up)

Review of Scheduling Heuristics: Circular-SCAN

- Method:
 - Service requests in one direction
- Pros:
 - + Very low computational time expended
 - + Implementation: Fairly easy (go up, back to start, go up)
 - + Throughput: somewhat high
 - + Fair: No starvation
- Cons:
 - - May go all the way back to start unnecessarily

Review of Scheduling Heuristics: LOOK

- Method:
 - Improvement on SCAN – same method (almost)
 - Uses information of request locations to determine where to stop going in a certain direction.
- Pros:
 - + Low computational time expended
 - + Implementation: Moderate (set upper bound, go up, set lower bound, go down)
 - + Throughput: somewhat high
 - + Fair: No starvation
- Cons:
 - - May miss a new request at outer boundary just as direction change occurs

Review of Scheduling Heuristics: Circular-LOOK

- Method:
 - Improvement on C-SCAN – same method (almost)
 - Uses information of request locations to determine where to stop scan.
- Pros:
 - + Low computational time expended
 - + Implementation: Moderate (set upper bound, go up, set lower bound, go to lower bound, repeat)
 - + Throughput: somewhat high
 - + Fair: No starvation
- Cons:
 - - May miss a new request at outer boundary just as direction change occurs

Review of Scheduling Heuristics: Delayed xxxx

- Method:
 - Potential improvement on xxxx
 - Process requests one cycle – wait next cycle - repeat.
- Pros:
 - ? Variable computational time expended (depends on xxxx)
 - ? Implementation: Variable
 - ? Throughput: potentially highest
 - ? Fair: No starvation?
- Cons:
 - - Wait period length needs to be chosen VERY carefully, else throughput < Standard-xxxx
 - Benchmarking, profiling

A Scheduling Algorithm: CHAIN

- Method:
 - Sort requests and position of elevator by some cost function, such that position of elevator is minimal element.
 - Find longest chain of requests with minimal cost.
 - Repeat once longest current chain is processed
- Pros:
 - + Throughput: Guaranteed high
 - + Fair: No starvation
 - + Proven qualities
- Cons:
 - - High computational time expended ($O(n^3)$)
 - But does it matter? Disk is **SUPER SLOW** anyway.
 - - Implementation: Very difficult
 - - Implementation: **Very difficult**

Project 3 Extra Credit

- Top 5 submissions achieving highest number of passengers transported will receive +10 to their project 3 grade.
- consumer.c and producer.c will now be provided to you.
 - They must NOT be modified to qualify for extra credit consideration.
- Choose and implement your algorithm carefully.
 - Completing project 3 comes first. (choose simple)
 - Getting extra credit comes last. (makes it easy to change later)

Project 3 Hints

- Do It in User Space
- Benchmarking
- Suggested Implementation Order

Project 3 Hints

- Do It in User Space
- Benchmarking
- Suggested Time Line

Do It In User Space

- Particularly for your elevator module, write your representations and scheduling algorithms in user space first.
- Test, test, test, test: make sure you NEVER crash
- Have an external application call your user-space “module” to mimic kernel behavior.
- Port to kernel space once you're certain your project works.
 - At least, once you're sure your *design* works.

Benchmarking

- Once your implementation works flawlessly (to the best of your knowledge), its time to benchmark.
- How do you benchmark your project?
 - Run producer.c
 - Run your elevators (consumer –start [0-2])
 - Make passengers for 10 minutes, then stop
 - Write down number of passengers transported as listed in /proc/elevator
 - Can you improve on this?
- Only your algorithm can raise your score.

Suggested Time Line

- Install custom kernel (2.6.31.1) – Last week
- /proc/currenttime – Last week
- Elevator Representation – By tonight
- Passenger/Request Representation – By tonight
- Elevator Module Design – By Thursday night
 - Elevator Algorithm Chosen – By Friday night
- User space version – By 10/28/09
- /proc/elevator (running) – By 11/06/09
- System Calls – By 11/06/09
- Benchmarking – All last weekend
- Testing – Continuous, of course

Synchronization Primitives

- Semaphores
 - User space
 - Kernel space
- Mutexes
 - User space
 - Kernel space
- Spin locks
 - Kernel space
- Atomic Functions

Next Time:

- Kernel Synchronization Primitives in Detail
- Concurrency vs. Parallelism
- Concurrent Aspects of Project 3
- Global Data vs. Local Data
 - What Should You Synchronize?
- More Project 3 Hints

Any Questions?