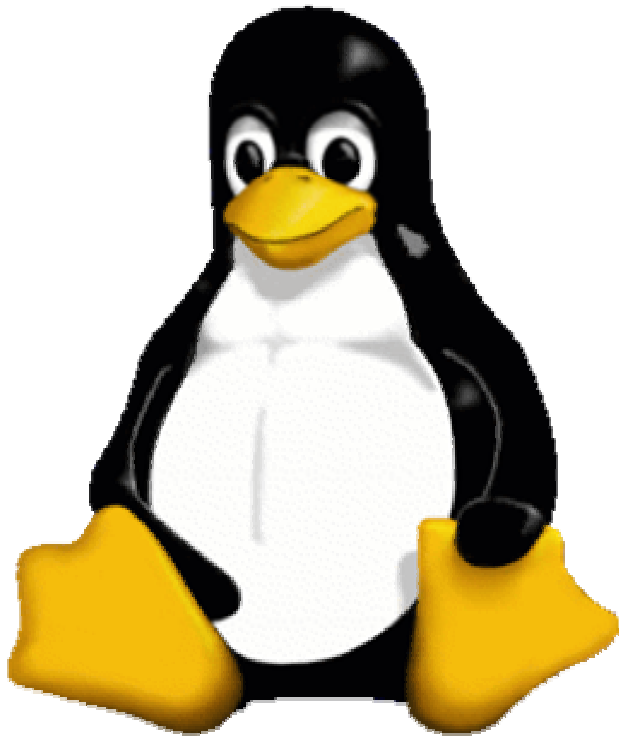
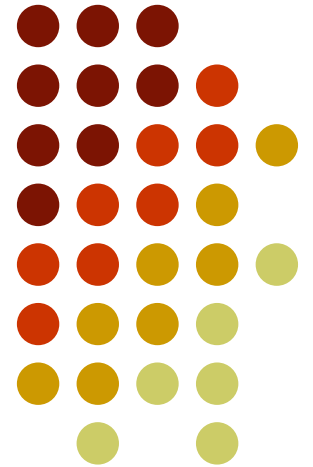


CS 3210

Fall 2005



**Debugging,
GCC Magic**





Kernel Debugging

- Sources:
 - Love 2e: Chapter 18 "Debugging" (2.6)
 - Corbet, Rubini, Kroah-Hartman 3e: Chapter 4 "Debugging Techniques" (2.6)
- Debugging is hard
 - Kernel debugging is harder!
 - Still, many similarities to other large-scale projects
- Need a reproducible bug
 - Intermittent or timing-related bugs very difficult



Types of Kernel Bugs

- Incorrect behaviors
- Corrupt data
- Synchronization errors, races, timing errors
- Performance bugs



Debugging Techniques

- printk()
- Oops
- CONFIG_DEBUG_KERNEL
- SysRq keys
- (Unofficial) kernel debuggers [Not ARM]
 - gdb, kdb, kgdb, nlkd
- /proc
- strace
- User Mode Linux (UML)
- Linux Trace Toolkit (LTT)
- Dynamic Probes (DProbes) [Intel]

printk()



- Very robust! Callable almost anywhere...
- Except very early in boot sequence
- `early_printk()`
- Circular log buffer
 - `klogd` (user space) reads `/proc/kmsg`
 - sends (via `syslogd`) to `/var/log/messages`
 - read with `dmesg`
- Log-levels (message priorities) 0 (high) .. 7 (low)
 - `/proc/sys/kernel/printk` (threshold)
 - `KERN_EMERG`, `_ALERT`, `_CRIT`, `_ERR`, `_WARNING`, `_NOTICE`, `_INFO`, `_DEBUG`

Oops



- Kernel exception handler
 - Kills offending process
 - Prints registers, stack trace with symbolic info
- Some exceptions non-recoverable (panic())
 - Prints message on console, halts kernel
 - Oops in interrupt handler, idle (0) or init (1)
- Oops generated by macros:
 - BUG(), BUG_ON(condition)



ksymoops

- Oops must be "decoded"
 - Associate symbolic names with addresses
- Address info lives in System.map
 - Kernel symbol table generated during compile
 - Module symbols included as well
- ksymoops – user-mode program (file access)
- kallsyms
 - 2.6 technique
 - Reads System.map into kernel memory at init

CONFIG_DEBUG_KERNEL



- Many kernel subsystems have extensive debugging that can be compiled in
- Subsystem specific compile-time debugging
 - `_SLAB`, `_PAGEALLOC`, `_SPINLOCK`, `_INIT`,
`_STACKOVERFLOW`, `_ACPI`, `_DRIVER`,
`_PROFILING`
- Info goes to console and `/proc`



Magic SysRq Keys

- Special console key sequences recognized by kernel
 - Useful for "system hangs"
 - Must be compiled in CONFIG_MAGIC_SYSRQ
- Alt-SysRq-<key>
 - h: help, b: reboot, s: sync, u: unmount all, etc.
- Toggle on/off:
 - `/proc/sys/kernel/sysrq`
- Possible to activate remotely by writing char to `/proc`
 - `/proc/sysrq-trigger`

(Unofficial) Kernel Debuggers



- No official kernel debugger!
 - Linus believes developers should deeply understand code and not rely on the "crutch" of an interactive debugger
 - Many still use debuggers from time to time (including Linus)
- gdb (from user-mode)
 - `gdb <kernel image> /proc/kcore`
 - Compile with `CONFIG_DEBUG_INFO` for symbolic info
 - Problem: caches symbol values *sigh*
- kdb (part of kernel) [Intel only]
 - Kernel halted when kdb runs
- kgdb (debug live kernel remotely via serial port)
 - Two separate patches; in flux; originally from SGI
 - Available for many architectures (but not ARM)
- nlkd (new debugger from Novell)

/proc



- Export kernel info via /proc
- Write your own /proc files
 - Can be read-only or read-write
 - Provide a special read_proc() function
 - Register with create_proc_read_entry()
 - Details in CRK
- Cleaner interface (seq_file) in 2.6
 - Better for files with lots of data
 - Iterator style (get_next)



strace

- View entry/exit of user-mode processes via system call interface
- Good for debugging new system calls



User Mode Linux (UML)

- Linux kernel emulation that runs as a user-mode process!
- Implemented as architecture port (arch/um)
- Easy to debug with gdb
- Slow
- Not useful for debugging hardware interactions (emulated)



Linux Trace Toolkit

- Generic event trace framework for kernel
- Includes timing information
- Slows things down but provides relative timing info
- Kernel tracing infrastructure + user-mode applications for viewing (graphs, etc.)
- Many supported architectures including ARM



Dynamic Probes (DProbes)

- Contributed by IBM for IA32
- Allows placement of "probes" anywhere in kernel
- Probes are code written in a special interpreted language
- Executed when flow-of-control reaches probe
- New probes can be inserted without kernel rebuild or reboot



Some Debugging Tricks

- Code conditional on UID
 - `if (current->uid == 7777) { ... }`
- Rate limiting `printk()`
 - Record print time
 - Only print again if interval elapsed

GCC Magic



- Sources
 - Love 2e: Chapter 2 "Getting Started with the Kernel"
 - Griffith: "GCC: The Complete Reference"
 - Hagan, Wall: "The Definitive Guide to GCC"
- Kernel code is written specifically for GCC
- Includes many non-standard GCC extensions



GCC Magic (2)

- `asm` linkage
 - Function attribute that inhibits parameter passing optimization (values in registers) to allow code to be called from assembly code
- named structure initialization
 - ```
static struct foo = {
 member1: value;
 member2: value;
};
```
- `__init`, `__initdata`
  - Attributes marking code and data used only during kernel initialization
  - Marked memory is released late in kernel initialization (`free_initmem()`)
- `unlikely()`, `likely()`
  - Branch annotations allowing `unlikely()` code blocks to be moved out of "normal" code path
  - Enhances pipelining and instruction cache hit ratio



# GCC Magic (3)

- `asm()` (`__asm__`)
  - Includes assembly code in C functions
  - Complex syntax including:
    - "local" numeric labels
    - jumps with forward/backward hints
    - ability to load registers with values from variables
- `volatile`, `__volatile__`
  - Variable declaration attribute indicating value may be changed asynchronously by another thread, forcing the compiler to re-load on each use
- `static inline`
  - Inline function expansion
  - Reduces call time by increasing code size
  - Often appear in `.h` files

# GCC Magic (4)

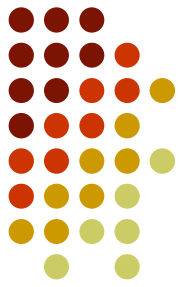


- gotos!
  - Kernel uses a \*lot\* of gotos for efficiency (maybe too many)
  - Common "unrolling" idiom for error exit
- no libc!
  - libc not generally available inside the kernel
  - Possible to call `sys_foo` directly
- no floating point in the kernel!
  - Saving/restoring floating-point registers expensive
  - Floating-point not really necessary in kernel
  - (Not available on some early architectures)
- small kernel stack
  - 8K (2 page) kernel stack per process
  - No large data structures declared on stack
  - Kernel always does dynamic allocation instead



# GCC Magic (5)

- do { s1; s2; } while (0)
  - Strange idiom in macros
  - Allows multiple statement macros in IF of IF-ELSE
  - if (cond) FOO(); else BAR();
- bitops
  - Extensive use of bit operators
  - and (&), or (|), xor (^), not (~)
- function attributes
  - `__attribute__ ((foo))`
- compound statements returning a value
  - `foo = ({ s1; s2; s3; ... });`
  - value returned is value of last statement executed



# GCC Magic (6)

- $x = y ? : z$ 
  - Replaces  $x = y ? y : z$ ;
  - Avoids re-evaluation of  $y$
- function nesting
  - local functions declared inside other functions
  - visible only in declaring function
- label addresses
  - possible to take addresses of labels ( $\&\text{label}$ )

# GCC Magic (7)



- lvalue expressions
  - left-hand side of an assignment (lvalue) can be an expression
  - `((a>5) ? b : c) = 100;`
- case ranges
  - `switch (foo) { case 1..10: bar(); break; ... }`
- function pointers
  - some functions are actually function pointers
  - functionality parameterized at start up
  - confusing when reading code
- `__variable`, `__variable`
  - C runtime reserves names that start with `_`
  - C+ runtime reserves names that start with `__`
  - kernel convention: names that start with `_` shouldn't be used or called directly unless you know what you are doing!



# Kernel Linked Lists

- Source:
  - Love 2e: Appendix A "Linked Lists"
- Standard but unusual linked-list implementation used throughout the kernel
- Circular, doubly linked list
- `<linux/list.h>`
- ```
struct list_head {  
    struct list_head *next;  
    struct list_head *prev;  
};
```
- Key idea: no distinguished head element!
 - List can be traversed starting at any element
 - All elements are conceptually "heads"



Kernel Linked Lists (2)

- list_head element usually embedded in another data structure
 - ```
struct my_struct {
 struct list_head list;
 unsigned long dog;
 void *cat;
};
```
- list\_head initialized at runtime or compile time
  - `INIT_LIST_HEAD(&p->list);`
  - `static LIST_HEAD(fox);`



# Kernel Linked Lists (3)

- Traversing linked lists
  - `struct list_head *p;`
  - `struct my_struct *my;`
  - `list_for_each(p, &mine->list) {`
    - `/* p points to list_head in my_struct */`
    - `my = list_entry(p, struct my_struct, list);`
    - `/* my points to struct containing list_head */`
  - `}`
- `list_entry( pointer to entry, struct type, entry name)`
  - subtracts entry offset from entry address

# Kernel Linked Lists (4)

