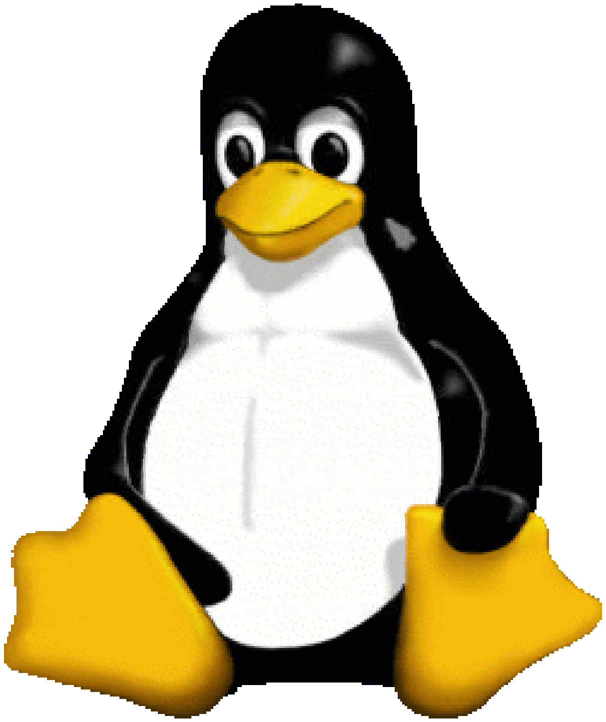
The Linux Kernel: Debugging



Accessing the “Black Box”

- Kernel code:
  - Not always executed in context of a process.
  - Not easily traced or executed under a conventional debugger.
    - Hard to step through (& set breakpoints in) a kernel that must be run to keep the system alive.

- How, then, can we debug kernel code?

Debugging by Printing

- **`printf`**’s are a common way of monitoring values of variables in application programs.
- Cannot use **`printf`** in the kernel as it’s part of the standard C library.
- **`printk`** is the kernel equivalent:
  - Messages can be classified according to their loglevel.
  - e.g. **`printk(KERN_DEBUG “I have an IQ of 6000.\n”);`**
  - Details found in **`kernel/printk.c.`**

Using /proc Filesystem

- See Rubini page 74.
- Can use **`/proc`** virtual filesystem to create file nodes for reading kernel data.
- Entries in **`/proc`** can be configured like any file and can refer to devices too!
- Reading a **`/proc`** entry causes data to be **generated**. This is different than reading a file whose contents existed before the read call.
  - Try doing **`‘ls –l /proc’`** to see the file sizes.

Debugging System Faults

- Oops Messages:
  - Usually generated by kernel when dereferencing invalid address.
  - What about other hardware detected faults?
  - Processor status is dumped to screen, including CPU register values.
    - Generated by **`arch/*/kernel/traps.c`**.
  - Can check **`/var/log/messages`** to see fn before oops message.
  - Can **`‘cat /proc/ksyms’`** to see address of function where PC was (value in **`EIP`** register) at time of fault.

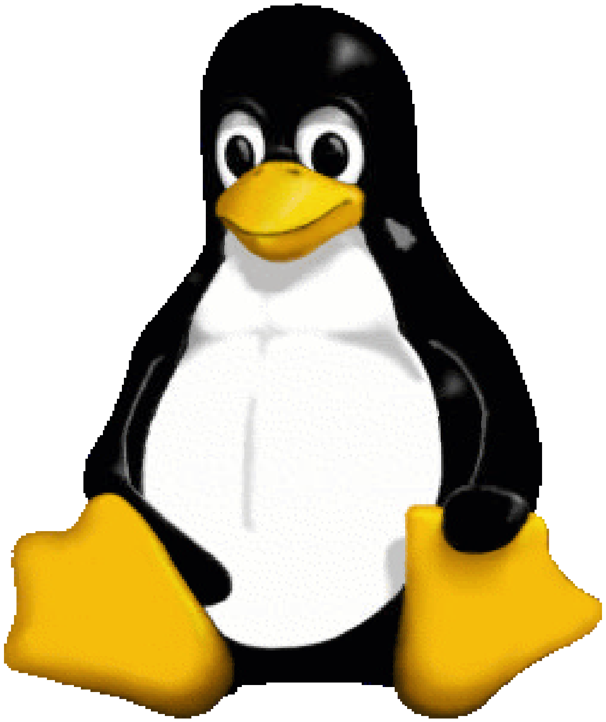
Other Debugging Methods

- Using a debugger:
    - e.g. **`gdb vmlinux /proc/kcore`** enables symbols to be examined in the uncompressed kernel image.
    - Assumes kernel built with symbols not stripped (-g option). Will be huge!
    - **`kcore`** is a core file representing the “executing kernel”. It is as large as all physical memory.
        - You cannot run the kernel image being debugged – it will seg fault! Hence this method is only good for symbol examination.
    - Other methods: **`kgdb`**, remote debugging.

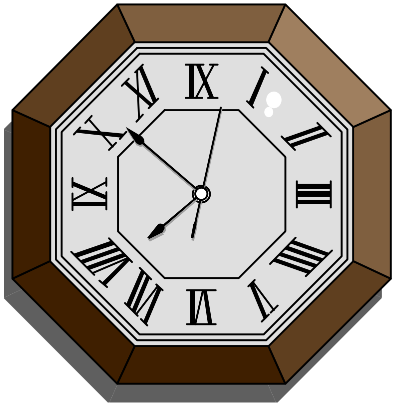
Message Logging

- **`<linux/kernel.h>`** defines the loglevels.
    - 8 loglevels available.
- If priority of message is less than **`console_loglevel`** priority, printk message is displayed.
- If **`klogd`** and **`syslogd`** are running, messages are logged in **`/var/log/messages`**.
- **`/etc/syslog.conf`** tells **`syslogd`** how to handle messages.

The Linux Kernel: The Flow of Time



“What time is it?”



- Need timing measurements to:
  - Keep track of current time and date for use by e.g. **`gettimeofday()`.**
  - Maintain timers that notify the kernel or a user program that an interval of time has elapsed.
- Timing measurements are performed by several hardware circuits, based on fixed frequency oscillators and counters.

Hardware Clocks

- Real-Time Clock (RTC):
  - Often integrated with CMOS RAM on separate chip from CPU: e.g., Motorola 146818.
  - Issues periodic interrupts on IRQ line (IRQ 8) at programmed frequency (e.g., 2-8192 Hz).
  - In Linux, used to derive time and date.
  - Kernel accesses RTC through **`0x70`** and **`0x71`** I/O ports.

Timestamp Counter (TSC)

- Intel Pentium (and up), AMD K6 etc incorporate a TSC.
- Processor's CLK pin receives a signal from an external oscillator e.g., 400 MHz crystal.
- TSC register is incremented at each clock signal.
- Using **`rdtsc`** assembly instruction can obtain 64-bit timing value.
- Most accurate timing method on above platforms.

The “PIT”s

- Programmable Interrupt Timers (PITs):
  - e.g., 8254 chip.
- PIT issues *timer interrupts* at programmed frequency.
- In Linux, PC-based 8254 is programmed to interrupt `Hz` (=100) times per second on IRQ 0.
  - `Hz` defined in `<linux/param.h>`
  - PIT is accessed on ports `0x40-0x43`.
- Provides the system “heartbeat” or “clock tick”.

“This’ll only take a jiffy”

- `jiffies` is incremented every timer interrupt.
  - Number of clock ticks since OS was booted.
- Scheduling and preemption done at granularities of time-slices calculated in units of jiffies.

Timer Interrupt Handler

- Every timer interrupt:
  - Update jiffies.
  - Update time and date (in secs & μsecs since 1970).
  - Determine how long a process has been executing and preempt it, if it finishes its allocated timeslice.
  - Update resource usage statistics.
  - Invoke functions for elapsed interval timers.

PIT Interrupt Service Routine

- Signal on IRQ 0 is generated:
- **`timer_interrupt()`** is invoked w/ interrupts disabled (**`SA_INTERRUPT`** flag is set to denote this).
- **`do_timer()`** is ultimately executed:
  - Simply increments **`jiffies`** & allocates other tasks to “bottom half handlers”.
  - Bottom half (bh) handlers update time and date, statistics, execute fns after specific elapsed intervals and invoke **`schedule()`** if necessary, for rescheduling processes.

Updating Time and Date

- **`lost_ticks`** (**`lost_ticks_system`**) store total (system) “ticks” since update to **`xtime`**, which stores *approximate* current time. This is needed since bh handlers run at convenient time and we need to keep track of when exactly they run to accurately update date & time.
- **`TIMER_BH`** refers to the queue of bottom halves invoked as a consequence of **`do_timer()`**.

Task Queues

- Often necessary to schedule kernel tasks at a later time without using interrupts.
  - Solution: Task Queues and kernel timers.
- A task queue is a list of *bottom half handlers*, each represented by a function pointer and argument.
- From **`<linux/tqueue.h>`**:

```
struct tq_struct {
    struct tq_struct *next;
    int sync; /* always 0 initially. */
    void (*routine)(void *);
    void *data;
}
```

Predefined Task Queues

- **`tq_scheduler`**: bottom half tasks in this queue are executed *whenever the scheduler runs*.
  - Both scheduler and bottom halves run in context of process being scheduled out.
- **`tq_timer`**: executed every timer tick at “interrupt time”.
- **`tq_immediate`**: executed either on return from syscall or when scheduler is run.

Useful Task Queue Functions

- **`void queue_task (struct tq_struct *task, task_queue *list);`**
  - Each queued task is removed from its queue after it is executed.
  - A task must be re-queued if needed repeatedly.
- **`void run_task_queue (task_queue *list);`**
  - Not needed unless custom task queues are implemented.
  - Fn is called by **`do_bottom_half()`** for predefined task queues.

Task Queue Example

```
struct wait_queue *waitq=null;

void wakeup_function(void *data) {
       wakeup_interruptible(&waitq);
}

void foo() {
       struct tq_struct bh;
       bh.next=null;
       bh.sync=0;
       bh.routine=wakeup_function;
       bh.data=(void *)some_data;
       queue_task(&bh,&tq_scheduler);
       interruptible_sleep_on(&waitq);
}
```

Kernel Timers

- Like task queues but timer bottom halves execute at predefined times.
- From **`<linux/timer.h>`**:

```
struct timer_list {
    struct timer_list *next;
    struct timer_list *prev;
    unsigned long expires; /* timeout in jiffies. */
    unsigned long data;
    void (*function)(unsigned long);
}
```

Useful Kernel Timer Functions

- **`void init_timer(struct timer_list *timer);`**
  - Zeroes **`prev`** & **`next`** pointers in doubly-linked timer queue.
- **`void add_timer(struct timer_list *timer);`**
  - Adds timer bottom half to kernel timer queue.
- **`int del_timer(struct timer_list *timer);`**
  - Removes timer before it expires.

Kernel Timer Example

```
struct wait_queue *waitq=null;

void wakeup_function(unsigned long data) {
        wakeup_interruptible(&waitq);
}

void foo() {
        struct timer_list bh;
        init_timer(&bh);
        bh.function=wakeup_function;
        bh.data=(unsigned long)some_data;
        bh.expires=jiffies+10*HZ; /* in 10 seconds. */
        add_timer(&bh);
        interruptible_sleep_on(&waitq);
}
```