Runtime and Postmortem Analyze of the Linux Network Stack

*Tracing and Probing Techniques from IRQ Context to User Space*

Hagen Paul Pfeifer

hagen@jauu.net

Protocol Labs

16. April 2011
Agenda

▶ Goals:

1. Provide information how to analyze the network stack
2. What tools are available to spot corner cases

▶ Agenda

• Tool Overview
• Tools in Detail
• Mostly Kernel Space, but some User Space tools are considered
• „War Stories“
Tools Overview

- trace-cmd
- perf
- netstat
- tc
- ethtool
- ss
- ftrace
- kprobe
- perfmon2
- proc/net/snmp

- What tools are available, when to use, how to use
- But: all of these tools require a fairly good understanding of the network stack. This talk is about to bring some light into the interaction and provide starting points.
Tools Overview II

- Starting with basic tools
- Going deeper, get into the details by using more specialized tools
Netstat

netstat -s

IP:
  3229907 total packets received
  
  [...]
  108 dropped because of missing route

Icmp:
  ICMP input histogram:
    destination unreachable: 57

Tcp:
  117181 active connections openings
  105765 passive connection openings
  559 connection resets received
  2775 segments retransmitted
  2133 resets sent

TcpExt:
  70382 delayed acks sent
    Quick ack mode was activated 772 times
  25235253 bytes directly received in process context from prequeue
  1641950 packet headers predicted
  450 packets collapsed in receive queue due to low socket buffer
  277 connections reset due to unexpected data
  
  [...]

Runtime Analyze of Linux Network Stack
Ethtool

▶ ethtool -S eth0

NIC statistics:
  rx_packets: 1172472
  tx_packets: 784183
  rx_bytes: 156112238
  tx_bytes: 72181402
  rx_broadcast: 2826
  rx_errors: 0
  tx_errors: 0
  tx_dropped: 0
  collisions: 0
  rx_crc_errors: 0
  rx_no_buffer_count: 0
  tx_fifo_errors: 0
  tx_tcp_seg_good: 47
  tx_tcp_seg_failed: 0
  alloc_rx_buff_failed: 0
  rx_dma_failed: 0
  tx_dma_failed: 0
[...]
Ethtool

- Ethtool is an excellent tool to discover CPU, BIOS, RAM and Network Card issues!
- `rx_no_buffer_count` → if CPU cannot keep new buffers to the hardware fast enough (e.g. increasing interrupt rate, increase RX descriptor ring, eliminate locking issues, ...)
- `rx_missed_errors` → Not enough bus bandwidth, host is too busy
Ethtool

- `ethtool -g eth5`

  Ring parameters for eth5:
  Pre-set maximums:
  RX: 4096
  TX: 4096
  Current hardware settings:
  RX: 256
  TX: 256

- 256 good compromiss:
  - 256 descriptors: 4096 byte, one page (256 * 16)
  - 512 descriptors: 8192 byte, two pages
  - Allow to buffer more incoming packets, increase system memory usage\(^1\), degrade cache line efficiency – especially with routing, cacheline efficiency is crucial

  ![see drivers/net/e1000/e1000_main.c:e1000_clean_rx_irq()](drivers/net/e1000/e1000_main.c:e1000_clean_rx_irq())

\(^1\)beside 16 byte descriptor, a receive buffer is also allocated 2048, 4096, 8192 or 16384 bytes; depending on the MTU
Strace and Ltrace

\texttt{strace -tt -T -e trace=network -p $(pidof mutt)}

- \texttt{-T} time spent in system call
- \texttt{-tt} time of day, include microseconds
- \texttt{trace=network} trace all network related system calls
- Don’t forget \texttt{read()}/\texttt{write()}/\texttt{splice()} in the case it is used to tx/rx data via socket(s)
AF Packet and PCAP

- Understand the internals requires a understanding of the algorithm - why versus how
- Why is partly described in RFCs, IEEE publications and other standard documents
- Why can also be observed by looking from a higher level: monitoring traffic
  - Raw sockets and PCAP provide a wide set of tools
  - TCP Slow Start, window limited sender, neighbor discovery, . . .

Tools
- tcpdump
- wireshark
- tcptrace
- . . .
SS

- To dump socket statistics – ss(8)
- Show all established connection for ssh
  - ss -o state established '( dport = :ssh or sport = :ssh )'
- ss -i -s
- INET_DIAG_INFO
Kprobe

- Debugging mechanism for monitoring events
- Since 2005; developed by IBM
- KProbes as an underlying mechanism, DProbes planned as a Tool (scripting capabilities, et cetera)
- KProbes
  - Handler installed at a particular instruction (pre and post)
- JProbes
  - Get access to a kernel function arguments
- Systemtap uses it
- Documentation: http://lwn.net/Articles/132196/
Kprobe Registration

- implemented by exception handling mechanisms (intr)
- KProbes Interface

```c
struct kprobe {
    struct hlist_node hlist; /* Internal */
    kprobe_opcode_t addr; /* Address of probe */
    kprobe_pre_handler_t pre_handler; /* Address of pre-handler */
    kprobe_post_handler_t post_handler; /* Address of post-handler */
    kprobe_fault_handler_t fault_handler; /* Address of fault handler */
    kprobe_break_handler_t break_handler; /* Internal */
    kprobe_opcode_t opcode; /* Internal */
    kprobe_opcode_t insn[MAX_INSN_SIZE]; /* Internal */
}

int register_kprobe(struct kprobe *p);
```
Kprobe Overhead

- kprobe Overhead (for x86_64)
  - kprobe: 0.5us
  - jprobe: 0.8us
  - return probe: 0.8 us
  - kprobe + return probe: 0.85us
  - kprobe + return probe: 0.85us

- Documentation:
**Systemtap**

- Example: what process allocate pages and report every 5 seconds

```plaintext
global page_allocs

probe kernel.trace("mm_page_alloc") {
    page_allocs[execname()]++
}

function print_count() {
    printf ("%-25s %-s\n", "#Pages Allocated", "Process Name")
    foreach (proc in page_allocs-)
        printf("%-25d %s\n", page_allocs[proc], proc)
    printf ("\n")
    delete page_allocs
}

probe timer.s(5) {
    print_count()
}
```

- Systemtap understands tracepoints too

- Access via kernel.trace() function
Systemtap II

Advantages:

- Scripting support
- Multiple instances
- Type enforcement
Ftrace

- Infrastructure since 2.6.27
- Ftrace can easily trace function calls or use static tracepoints placed in the kernel source
- No dynamic tracepoint on the fly support
- Function tracer implementation via `mcount (-pg, see next foil)`
- Interface:

  ```
  mount -t debugfs nodev /sys/kernel/debug
  sysctl kernel.ftrace_enabled=1
  echo function > current_tracer
  echo 1 > tracing_on
  usleep 1
  echo 0 > tracing_on
  cat trace
  ```

- `Documentation/trace/ftrace.txt`
Function Tracer

- Ingo Molnar and William Lee Irwin created a tracer to find latency problems in the RT branch.
- One feature of the latency tracer was the function tracer.
- Used to show what function where called when interrupts.
- How can it be that if not enabled nearly no overhead can be measured? How is function tracing implemented?
  - If `CONFIG_DYNAMIC_TRACE` is set the gcc compiles with `-pg`?
  - In in turn `mcount()` is placed at every function (this is why inlined function cannot be traced)
  - Gprof is the most prominent user of `mcount()` (`_mcount()` BTW is provided by GLIBC - not GCC)
  - After compilation objedump will search for `mcount()` callees in the `.text` segment (`recordmcount.pl`)
• This script generates a new segment (__mcount_loc) that hold all references to mcount callees

• At boot time the ftrace code iterate over this list and update replace all mcount() calls by nops. Additionally all locations are saved in a list (available_filter_list)

• When traceing is enabled these nops are patched back to call the ftrace infrastructure (mcount() is a stub)
void kfree_skb(struct sk_buff *skb)
{
    if (unlikely(!skb))
        return;
    if (likely(atomic_read(&skb->users) == 1))
        smp_rmb();
    else if (likely(!atomic_dec_and_test(&skb->users)))
        return;
    trace_kfree_skb(skb, __builtin_return_address(0));
    __kfree_skb(skb);
}
Trace Points

▶ cat /sys/kernel/debug/tracing/available_events | wc -l → 778
▶ (or trace-cmd list or perf list | grep Tracepoint)
trace-cmd

Originally a user was constrained to echo and cat to set up a ftrace Tool from Steven Rostedt to hide the detail

Trace points must be a priory available in the kernel

- Scheduling
- Interrupts
- Free skb’s

Set up a tracer:

- `trace-cmd record -e <subsystem:event-name>`
- `trace-cmd record -p function -e sched_switch ls > /dev/null`
- what interrupts have the highest latency
  - `trace-cmd record -p function_graph -e irq_handler_entry -1 do_IRQ sleep 10; trace-cmd report`
- kmalloc calls that were greater than 1000 bytes
• `trace-cmd report -F 'kmalloc: bytes_req > 1000'`

• Function Graph
  
  • `trace-cmd record -p function_graph ls /bin`

• Finding high latency interrupts (one of Stevens examples)
  
  • `trace-cmd record -p function_graph -l do_IRQ -e irq_handler_entry`

  • `trace-cmd report | cut -c32-43 --complement`

`git://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git`
Kernelshark II

Runtime Analyze of Linux Network Stack
Kernelshark II

Runtime Analyze of Linux Network Stack
Perf

- Access to hardware counters (first releases) ²

- First release: Ingo Molnar and Thomas Gleixner; now Arnaldo, Frederic, ...
  - Special hardware registers
  - Count number of certain hardware events like
    - Cache misses
    - TLB missed
    - CPU cycles
    - Branch misprediction
  - Without slowing down the execution (hardware register, say nearly without)

- Interface to kernel tracepoints (examples)
  - Trace points are interesting locations in subsystems, not a record of all executed

²if you build perf by yourself, make sure you build for the actual kernel version: `perf --version` (perf version 2.6.39.rc1.210.g66ee33.dirty) should match `uname -r` (2.6.39-rc1+)
function

- Between these points the must interpolate what happens during that time
  - Collect all kind of performance data
  - Superseeds oprofile, perfmon2, (partly systemtap)
  - Per thread, task, CPU or system wide
## Perf

**Commands:**

<table>
<thead>
<tr>
<th>命令</th>
<th>描述</th>
</tr>
</thead>
<tbody>
<tr>
<td>annotate</td>
<td>Read perf.data (created by perf record) and display annotated code</td>
</tr>
<tr>
<td>bench</td>
<td>General framework for benchmark suites</td>
</tr>
<tr>
<td>list</td>
<td>List all symbolic event types</td>
</tr>
<tr>
<td>probe</td>
<td>Define new dynamic tracepoints</td>
</tr>
<tr>
<td>record</td>
<td>Run a command and record its profile into perf.data</td>
</tr>
<tr>
<td>report</td>
<td>Read perf.data (created by perf record) and display the profile</td>
</tr>
<tr>
<td>sched</td>
<td>Tool to trace/measure scheduler properties (latencies)</td>
</tr>
<tr>
<td>script</td>
<td>Read perf.data (created by perf record) and display trace output</td>
</tr>
<tr>
<td>stat</td>
<td>Run a command and gather performance counter statistics</td>
</tr>
<tr>
<td>timechart</td>
<td>Tool to visualize total system behavior during a workload</td>
</tr>
<tr>
<td>top</td>
<td>System profiling tool.</td>
</tr>
</tbody>
</table>

[...]
Perf stat

`perf stat ls -R > /dev/null`

Performance counter stats for 'ls -R':

```
371,191921 task-clock-msecs # 0,995 CPUs
14 context-switches # 0,000 M/sec
  2 CPU-migrations # 0,000 M/sec
  479 page-faults # 0,001 M/sec
1.208.000.453 cycles # 3254,382 M/sec (scaled from 65,65%)
1.643.737.756 instructions # 1,361 IPC (scaled from 65,67%)
308.823.362 branches # 831,978 M/sec (scaled from 67,44%)
  8.156.463 branch-misses # 2,641 % (scaled from 67,77%)
472.335.871 cache-references # 1272,484 M/sec (scaled from 67,65%)
  4.174.862 cache-misses # 11,247 M/sec (scaled from 66,57%)
```

0,373027705 seconds time elapsed
Perf

Perf list:

cpu-cycles OR cycles [Hardware event]
instructions [Hardware event]
cache-references [Hardware event]
cache-misses [Hardware event]
branch-instructions OR branches [Hardware event]
branch-misses [Hardware event]

[...]

skb:kfree_skb [Tracepoint event]
skb:consume_skb [Tracepoint event]
skb:skb_copy_datagram_iovec [Tracepoint event]
net:net_dev_xmit [Tracepoint event]
net:net_dev_queue [Tracepoint event]
net:netif_receive_skb [Tracepoint event]
net:netif_rx [Tracepoint event]
napi:napi_poll [Tracepoint event]
Perf

- sudo perf record -g ./client -r 0 -s 1073741824 -i 0 -d 0 -n 1 -e localhost
- perf report
Perf Scripts

- Perl or Python
- Generate template, the rest (processing) is up to the user

```bash
perf script record netdev-times
perf script report netdev-times dev=eth5
```

```
21150.714226sec cpu=0
  irq_entry(+0.000msec irq=16:eth5)
    | softirq_entry(+0.004msec)
    |   ---netif_receive_skb(+0.008msec skb=ffff8800cda03400 len=1492)
    |       | skb_copy_datagram_iovec(+0.034msec 8948:wget)
    |       | napi_poll_exit(+0.026msec eth5)
```
Kprobe based Event Tracer

▶ Masami Hiramatsu (26 files changed, 3924 insertions(+), 163 deletions(-))
▶ Based on kprobe (and kretprobe)
▶ Probe various kernel events through ftrace interface
▶ Anywhere where kprobes can probe
▶ Unlike the function tracer, this tracer can probe instructions inside of kernel functions
▶ Allows you to check which instruction has been executed
▶ On the fly
▶ p[:EVENT] SYMBOL[+offset|-offset]|MEMADDR [FETCHARGS]
▶ r[:EVENT] SYMBOL[+0] [FETCHARGS]
▶ LWN Article: http://lwn.net/Articles/343766/
Perf probe

- perf probe ip_rcv
- perf record -e probe:ip_rcv -R -f -g -a
- perf report

100.00%  nyxms2  [kernel.kallsyms]  [k]  ip_rcv
| --- ip_rcv
|     |---100.00%-- netif_receive_skb
|     |     |---100.00%-- napi_gro_complete
|     |     |       | napi_gro_flush
|     |     |       | napi_complete
|     |     |       | e1000_clean
|     |     |       | net_rx_action
|     |     |       |   __do_softirq
|     |     |       |     call_softirq
|     |     |       |       _local_bh_enable_ip
|     |     |       |     local_bh_enable
|     |     |       |     dev_queue_xmit
|     |     |       |     ip_finish_output2
|     |     |       |     ip_finish_output

Runtime Analyze of Linux Network Stack
| ip_output |
| dst_output |
| ip_local_out |
| ip_queue_xmit |
| tcp_transmit_skb |
| tcp_write_xmit |
Trace Packet Flow through Kernel

- Attention: NAPI and non-NAPI flow, take your path)

- What we (probably) want to know:
  - We want to know when the Userspace blocked in recvfrom()
    - $\rightarrow$ recvfrom()
  - We want to know when a NIC interrupt is triggered
    - $\rightarrow$ irq_entry()
      - called by driver when frame received on interface; enqueues frame on current processor’s packet receive queue (non NAPI implementation; NAPI driver call netif_receive_skb)
        - $\rightarrow$ netif_rx
  - We can check when the softirq is raised
    - $\rightarrow$ softirq_raise
  - We can check when the interrupt is exited
• → irq_exit

• We want to know when the softirq is executed
  • → softirq_entry

• We want to know when a packet is get from device queue
  • softirq context
  • → dev.c:__netif_receive_skb()

• We want to know when data is delivery to the user (receive queue)
  • → skb_copy_datagram_iovec()
  • tcp_data_queue(), udp_recvmsg, ...call skb_copy_datagram_iovec()

• We want to know napi loop is leaved
  • → napi_poll_exit()

• We want to know when softirq context is leaved
  • → softirq_exit()

• We want to know when the Userspace Application is scheduled in
• We want to know when the Userspace returned from recvfrom()
  
  • $\rightarrow$ recvfrom()
KVM, Qemu and GDB

- Allow so single step in a real system
- Advantage: analyze the behavior on the fly where you don’t know where to look in advance (e.g. lot of code must be conditionally analysed, without any prior knowledge)
- Use your standard qemu setup with two additional qemu flags: -s and -S
- Both flags instrument qemu to start a qemu gdb server and to break at the beginning
- Setup for the other side:

```
gdb /usr/src/linux/vmlinux
target remote localhost:1234
c
bt
set architecture i386:x86-64:intel
```

- `set architecture i386:x86-64:intel` fix a bug where gdb cannot detect that the target is x86_64
Misc

▶ If all tools fail? `trace_printk()`

▶ Interface:

- `trace_printk()`
  - debug fast path sections that printk is too slow
  - printk may overwhelm the system (livellook)
  - Print to ftrace ringbuffer
  - ms versus us

- `trace_clock_global()` may be your friend
Background – GCC Instruments

-`-finstrument-functions`?
  - Insert hooks for function entry and exit

```
0000000000400514 <func>:
400514: 53 push %rbx
400515: 31 f6 xor %esi,%esi
400517: 89 fb mov %edi,%ebx
400519: bf 14 05 40 00 mov $0x400514,%edi
40051e: e8 ed fe ff ff callq 400410 <__cyg_profile_func_enter@plt>
400523: 31 f6 xor %esi,%esi
400525: bf 14 05 40 00 mov $0x400514,%edi
40052a: e8 f1 fe ff ff callq 400420 <__cyg_profile_func_exit@plt>
40052f: 69 c3 ad de 00 00 imul $0xdead,%ebx,%eax
400535: 5b pop %rbx
400536: c3 retq
```

- Support **inlined** function as well
- GCC version 2.95 or later required
- See [?] for an user of `-finstrument-functions`
Background – GCC Instruments

```c
void __attribute__((__no_instrument_function__)) __cyg_profile_func_enter(
    void *this_fn, void *call_site)
{
    int n;

    call_depth++;
    for (n = 0; n < call_depth; n++)
        printf(" ");
    printf("\n", this_fn);
}
```

▶ First argument is the address of the start of traced function

▶ Address to symbol translation can be done via `nm`

[...,]
000000000004004d0 t __do_global_dtors_aux
00000000000400540 t frame_dummy
00000000000400564 T __cyg_profile_func_enter
0000000000040059a T __cyg_profile_func_exit
000000000004005d2 T func
000000000004005f5 T main
[...]
Background – GCC Instruments

▶ -ftest-coverage
  - Instrument the generated code to count the execution frequencies of each basic block

▶ fprofile-arcs
  - count the execution frequencies of each branch

▶ gcov can be used to annotate the source code with this information
Background – GCC Instruments

- **-pg**

  - Adding a call to a function named mcount (or \_mcount\(\), depends on compiler)
  
  - mcount() function of GLIBC
  
  - Lightweight - „only“ function call overhead

```
[...]
4005e4: 55 push %rbp
4005e5: 48 89 e5 mov %rsp,%rbp
4005e8: e8 ab fe ff ff callq 400498 <mcount@plt>
4005ed: 5d pop %rbp
4005ee: 69 c7 ad de 00 00 imul $0xdead,%edi,%eax
4005f4: c3 retq
[...]```
Kernel Oops

BUG: unable to handle kernel NULL pointer dereference at 00000050
IP: [<c12280c0>] tc_fill_qdisc+0x68/0x1e5
*pde = 00000000
Oops: 0000 [#1] SMP
last sysfs file:
Modules linked in:

Pid: 600, comm: qdisc Not tainted 2.6.34 #16 /
EIP: 0060:[<c12280c0>] EFLAGS: 00010282 CPU: 0
EIP is at tc_fill_qdisc+0x68/0x1e5
EAX: 00000000 EBX: ffffffff ECX: 00000000 EDX: c7222070
ESI: c14576e0 EDI: c7115200 EBP: c7239ca0 ESP: c7239c3c
DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
Process qdisc (pid: 600, ti=c7239000 task=c720b700 task.ti=c7239000)
Stack:
00000024 00000014 00000000 c14323a0 c7222060 c7222060 c10a7abd 00001030
<0> 000000d0 c7222060 000000d0 c1228329 000000d0 00000fc4 000000d0 c7115200
<0> 000000d0 00000ec0 c7239cac c12104b1 00000ec0 c1457a98 c7115200 00000258
Call Trace:
[<c10a7abd>] ? __kmalloc_track_caller+0x122/0x131
[<c1228329>] ? qdisc_notify+0x2a/0xc8
[<c12104b1>] ? __alloc_skb+0x4e/0x115
[<c122838a>] ? qdisc_notify+0x8b/0xc8
 [...] 

▶ gdb vmlinuz 

▶ i line *tc_fill_qdisc+0x68 

▶ l sch_api.c:1191 

▶ x/8i *tc_fill_qdisc+0x68
End

- Contact:
  - Hagen Paul Pfeifer <hagen@jauu.net>
  - Key Id: 0x98350C22
  - Key Fingerprint: 490F 557B 6C48 6D7E 5706 2EA2 4A22 8D45 9835 0C22
Perf Internals

- via `perf report -D` recorded raw samples can be displayed

- To disable some cores:
  - add `maxcpus=1` to kernel command line
Trace Points I

CONFIG_FUNCTION_TRACER
CONFIG_FUNCTION_GRAPH_TRACER
CONFIG_STACK_TRACER
CONFIG_DYNAMIC_FTRACE
#include <linux/ftrace.h>
define CREATE_TRACEPOINTS
#include <trace/events/sched.h>

[...]

trace_sched_migrate_task(p, new_cpu);

if (task_cpu(p) != new_cpu) {
    p->se.nr_migrations++;
    perf_sw_event(PERF_COUNT_SW_CPU_MIGRATIONS, 1, 1, NULL, 0);
}

__set_task_cpu(p, new_cpu);
[...]

Runtime Analyze of Linux Network Stack
TRACE Points III

TRACE_EVENT(sched_migrate_task,

    TP_PROTO(struct task_struct *p, int dest_cpu),

    TP_ARGS(p, dest_cpu),

    TP_STRUCT__entry(
        __array( char, comm, TASK_COMM_LEN )
        __field( pid_t, pid )
        __field( int, prio )
        __field( int, orig_cpu )
        __field( int, dest_cpu )
    ),

    TP_fast_assign(
        memcpy(__entry->comm, p->comm, TASK_COMM_LEN);
        __entry->pid = p->pid;
        __entry->prio = p->prio;
        __entry->orig_cpu = task_cpu(p);
        __entry->dest_cpu = dest_cpu;
    ),

    TP_printk("comm=%s pid=%d prio=%d orig_cpu=%d dest_cpu=%d",

Runtime Analyze of Linux Network Stack
__entry->comm, __entry->pid, __entry->prio,
__entry->orig_cpu, __entry->dest_cpu)
);
Gigabit and PCI Bus

- PCI-Express just great for 1Gbps
- `lspci -t -v` and `lspci -vvv`

```plaintext
02:00.0 Ethernet controller: Intel Corporation 82574L Gigabit Network Connection
  Subsystem: Intel Corporation Gigabit CT Desktop Adapter
  Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR+ FastB2B- DisINTx-
  Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- latency- 0, Cache Line Size: 64 bytes
  Interrupt: pin A routed to IRQ 16
  Region 0: Memory at fe6e0000 (32-bit, non-prefetchable) [size=128K]
  Region 1: Memory at fe600000 (32-bit, non-prefetchable) [size=512K]
  Region 2: I/O ports at bc00 [size=32]
  Region 3: Memory at fe6dc000 (32-bit, non-prefetchable) [size=16K]
  Expansion ROM at fe680000 [disabled] [size=256K]
  Capabilities: <access denied>
  Kernel driver in use: e1000e
```

- Default IRQ assignments are read from the Differentiated System Description Table (DSDT) table in the BIOS
Modifying the DSDT

- AML table in BIOS

- Build a custom DSDT:

  - aptitude install acpidump iasl
  - acpidump > acpidump.data
  - acpixtract acpidump.data (will generate DSDT.dat and SSDT.dat)
  - iasl -d DSDT.dat
  - vim DSDT.dsl
  - iasl -tc DSDT.dsl
  - cp DSDT.hex /usr/src/linux/include/

  - CONFIG_STANDALONE=n, CONFIG_ACPI_CUSTOM_DSDT=y, CONFIG_ACPI_CUSTOM_DSDT_FILE="DSDT.hex"
Packet Drop

➤ since the kernel has a limit of how many fragments it can buffer before it starts throwing away packet

➤ `/proc/sys/net/ipv4/ipfrag_high_thresh`
`/proc/sys/net/ipv4/ipfrag_low_thresh`

➤ Once the number of unprocessed, fragmented packets reaches the number specified by `ipfrag_high_thresh` (in bytes), the kernel will simply start throwing away fragmented packets until the number of incomplete packets reaches the number specified by `ipfrag_low_thresh`.

➤ Another counter to monitor is IP: ReasmFails in the file `/proc/net/snmp`; this is the number of fragment reassembly failures. If it goes up too quickly during heavy file activity, you may have a problem.
## Softnet

- `/proc/net/softnet_stat`

<table>
<thead>
<tr>
<th>total</th>
<th>dropped</th>
<th>squeezed</th>
<th>collisio</th>
</tr>
</thead>
<tbody>
<tr>
<td>00101ba7</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>0005c045</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>0003a7cf</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00286ae</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>00126a7</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
<tr>
<td>0010b04</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>