Linux Network Layer

Pradipta De
pradipta.de@sunykorea.ac.kr
Today’s Topic

• Network Layer Background
• Linux network data structures
• Network Interrupt Processing
  – Top and Bottom halves (softirq)
• Linux network processing flow
• Advanced Features
Purpose of Network

Send Packet from Host X to Host Y
Network Layers

- A message sent/received from application
- At the transport layer
- At the network layer with network header
- Individual package of transmitted data
Protocol Types

- **Ethernet:**
  - Frame contains header, payload and checksum
  - Header: type, src MAC, dest MAC, length
  - Payload: higher layer headers + data

- **IP**
  - Each device has a IPv4 or IPv6 address, and packets are routed over an Ethernet network
  - Header contains the source and destination IP – at each router the src and dest changes, and a new Ethernet frame is created

- **Transport Layer**
  - end to end protocol – each application assigned a port number
  - TCP: reliable transport
  - UDP: unreliable transport
headers
(cf. Understanding Linux Network Internals)
Network Programming Interface

- **Socket** is the standard interface between user and kernel space
  - Standard socket APIs allow open, close, transmit, receive of messages from/to application

- `int socket (int domain, int type, int protocol)`
  - Domain: `AF_INET`, `AF_INET6`, `AF_LOCAL`, `AF_PACKET`, …
  - Type: `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_SEQPACKET`, …
  - Protocol: TCP, UDP
  - On success, a file descriptor for the new socket is returned

- Lower layers are abstracted from application

- Sockets are the plumbing for client-server apps
  - Server listens on pre-determined port
  - Client connects to server to create a message channel
  - Once server accepts connection, messages are transferred
Message Flow

- Applications connect using socket call
- Socket $\rightarrow$ sys_socketcall
- Message is put in a data structure that holds the data sent by app (plus room for headers)
  - sk_buff: socket buffer data structure
- sk_buff goes through each layer of the network stack, and headers are appended (or taken out)
- Finally, the device sits at the bottom of the network stack
Data Structures

- sk_buff, or Socket Buffer: All network-related queues and buffers in the kernel use this common data structure – contains meta information about socket data
  - Pointers to skb, in/out net_device, …
  - Protocol header pointers
  - Routing related information
  - Various length fields, reference counters
  - Pointers into the data buffer

- net_device: data structure that stores all information specifically regarding a network device (e.g. eth0, lo, …)
  - Global Information (name of the device, Device state, Pointer to next device, Initialization function)
  - Hardware information
  - Interface information
  - Device specific functions
More on net_device

• Netdevice represents a network device
  – Contains hardware and software configs

• Configs:
  – MTU: maximum transfer unit ➔ max frame size
    • e.g. Ethernet MTU is 1500 bytes
  – Dev_addr: mac address of the device
  – int (*hard_start_xmit)(struct sk_buff *skb, struct net_device *dev) ➔
    pointer to device specific transmit function
  – promiscuity: counts how many times device set to promiscuous mode
The structure is used by all the network layers to store their headers, information about the user data (the payload), and other information needed internally for coordinating their work.

**Transport header**: can contain TCP, UDP, ICMP or more

**Network header**: can contain IPv4, IPv6, ARP

**MAC header**: layer 2 header, or the Ethernet header

**stamp**: timestamp of receiving packet

**Net_device**: represents the network card
More on `sk_buff`

- `sk_buff` holds pointer to the data buffer
- Goal is not to reallocate and copy data as packet traverses the network stack
- Data can be added at the front and end of the data buffer

```
struct sk_buff {
    ... 
    unsigned char *head;
    unsigned char *data;
    unsigned char *tail;
    unsigned char *end;
    ... 
};
```
Sk_buff: Utility functions

- `alloc_skb(len, ...)`
- `skb_reserve`
- `skb_put`
- `skb_push`
Transmit and Receive Flow

Reception

Transmission
Deferred Work in Kernel
Recap on Interrupt Processing

- Device raises an interrupt → interrupt controller signals CPU that interrupt occurred
- CPU checks the interrupt request line
  - Uses the interrupt number to trigger interrupt handler
  - Lower priority interrupts may be disabled while servicing interrupt
- Executing program state saved → CPU jumps to interrupt handler → after execution returns to program code
Kernel Contexts

• **Interrupt Context**
  – Uses the context of the process that is interrupted
    • Handler uses kernel stack of interrupted thread
    • Handler cannot block since no independent process structure
  – Example: Interrupt handlers

• **Process Context**
  – Kernel executes on behalf of a user process
    • Example: All syscalls
Interrupt Context

- Perform the most essential task with interrupts disabled, and defer the remaining work for later (with interrupts enabled)
  - Example: Timer interrupt handler
    - Immediate: Increment time of day
    - Defer: recalculate process priority

- Top Half: perform the most time sensitive, hardware specific operation of the interrupt processing
  - Interrupts disabled

- Bottom Half: perform other interrupt related work left unfinished by top half ➔ bottom half context
  - Interrupts enabled
Bottom Half Early Implementation
(2.2 … not present in 2.6)

• Called the BH interface
• Defined an array of 32 statically defined bottom half handlers
  – 18 bottom half handlers were used

• Top half marks a bottom half to be serviced once top half finishes by setting a bit in a 32-bit integer

• Each BH is globally synchronized
  – Two BH cannot run simultaneously, even on different processors ➔ single threaded operation ➔ Low performance and scalability

enum {
    TIMER_BH = 0,
    CONSOLE_BH,
    TQUEUE_BH,
    DIGI_BH,
    SERIAL_BH,
    RISCOM8_BH,
    SPECIALIX_BH,
    AURORA_BH,
    ESP_BH,
    NET_BH,
    SCSI_BH,
    IMMEDIATE_BH,
    KEYBOARD_BH,
    CYCLADES_BH,
    CM206_BH,
    JS_BH,
    MACSERIAL_BH,
    ISICOM_BH
};
Softirq

- Software interrupts or softirq is multithreaded equivalent of BH
- 6 statically defined softirqs

Softirqs key feature difference
- Multiple softirqs can run concurrently
- Same softirq can run on different processors
- Maintain per-CPU list of queues for each softirq type

When is softirq triggered?
- At return from hardware interrupt code or end of top half
- Can be explicitly invoked (e.g. by networking subsystem)
Tasklet

• Tasklet is a special case of softirq, built on top of softirq
  – Uses the TASKLET_SOFTIRQ or HI_SOFTIRQ

• Simpler to use, can be dynamically created (unlike softirq)
  – Two different tasklets can run on different processors, but same type cannot run concurrently (even on different processors)
Ksoftirqd are a set of kernel threads (one on each processor) that helps in executing softirqs.

Why do we need ksoftirqd?

- Problem: “critical tradeoff”
  - Consider high network load \(\rightarrow\) Top half raises softirq at a high rate \(\rightarrow\) softirq processing starves user processes
  - Service softirqs in a bunch after each interrupt \(\rightarrow\) under low load the processing may wait long

- Solution: Do not process reactivated softirq immediately. If number of softirqs grow large, then trigger kernel threads (ksoftirqd) to process them
  - Nice value of ksoftirqd is +19 \(\rightarrow\) will not starve user processes
Other ways to defer work

- **Work Queue**
  - Defer work in process context ➔ bottom half can be scheduled out (can sleep)
  - Workqueue subsystem creates kernel threads (keventd/n) per CPU to handle deferred work

- **Timer driven**
  - Work can deferred by setting a timer
  - e.g. some network card provides a way to trigger interrupts only after fixed intervals
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Softirqs</th>
<th>Tasklets</th>
<th>Work Queues</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execution context</strong></td>
<td>Deferred work runs in interrupt context.</td>
<td>Deferred work runs in interrupt context.</td>
<td>Deferred work runs in process context.</td>
</tr>
<tr>
<td><strong>Reentrancy</strong></td>
<td>Can run simultaneously on different CPUs.</td>
<td>Cannot run simultaneously on different CPUs. Different CPUs can run different tasklets, however.</td>
<td>Can run simultaneously on different CPUs.</td>
</tr>
<tr>
<td><strong>Sleep semantics</strong></td>
<td>Cannot go to sleep.</td>
<td>Cannot go to sleep.</td>
<td>May go to sleep.</td>
</tr>
<tr>
<td><strong>Preemption</strong></td>
<td>Cannot be preempted/scheduled.</td>
<td>Cannot be preempted/scheduled.</td>
<td>May be preempted/scheduled.</td>
</tr>
<tr>
<td><strong>Ease of use</strong></td>
<td>Not easy to use.</td>
<td>Easy to use.</td>
<td>Easy to use.</td>
</tr>
<tr>
<td><strong>When to use</strong></td>
<td>If deferred work will not go to sleep and if you have crucial scalability or speed requirements.</td>
<td>If deferred work will not go to sleep.</td>
<td>If deferred work may go to sleep.</td>
</tr>
</tbody>
</table>

Source: Essential Linux Device Drivers – Chapter 15
Back to Linux Network Subsystem
Frame Reception

- Polling: constantly check the device for incoming frame
  - Can polling be useful?
- Interrupts: device generates an interrupt for each incoming frame

- Receive-livelock: interrupts has highest priority of execution. Under heavy load, interrupts will overrun the input buffer, but bottom half cannot process the packets since top half is continuously scheduled
Frame Reception

• Processing Multiple Frames during an Interrupt
  – On interrupt, handler downloads all frames to kernel input queue
    • Stops at fixed time window or fixed number of frames, otherwise may starve other processes

• New API: combine interrupt with polling
  – Disable interrupt for a device which has frames in its input queue (will be processed with interrupt disabled)
  – Once processing complete, poll the device to check for new frames
Transmission Path

http://www.cs.unh.edu/cnrg/people/gherrin/linux-net.html
Zero Copy Techniques

- Number of data copies is 4
- Number of context switches is 2
Zero Copy Techniques

Dynamic Remapping
- uses mmap system call instead of read
- Data copied from kernel buffer to socket buffer in kernel address space
Zero Copy Techniques

Shared Buffer in Kernel Address Space
• Memory area shared in kernel address space ➞ no data copy required
Zero Copy Techniques

Sendfile system call
- transfer data between file descriptors
- copying done within the kernel, hence more efficient than the combination of read and write which require data transfer to and from user space.

Sendfile with DMA Scatter/Gather Copy
- Eliminates CPU copy between kernel buffer and socket buffer
- Instead of copy, buffer descriptors are passed to the socket
  - The descriptor contains information about the length of the data and where it is.
- Needs Hardware support.
Putting It Together

• Part 1
  – Basic understanding of protocol headers
  – Critical data structures

• Part 2
  – Interrupt Processing - Deferred Functions
  – Network device driver – working principles
  – Zero copy techniques