Network Programming

Lecture 02

Haitham A. El-Ghareeb

Faculty of Computers and Information Sciences
Mansoura University
Egypt
helghareeb@mans.edu.eg

February 18, 2017

- Flow 5
- NAPI
- Flow 6
- Flow 7

How to Process Interrupt and Received Packet

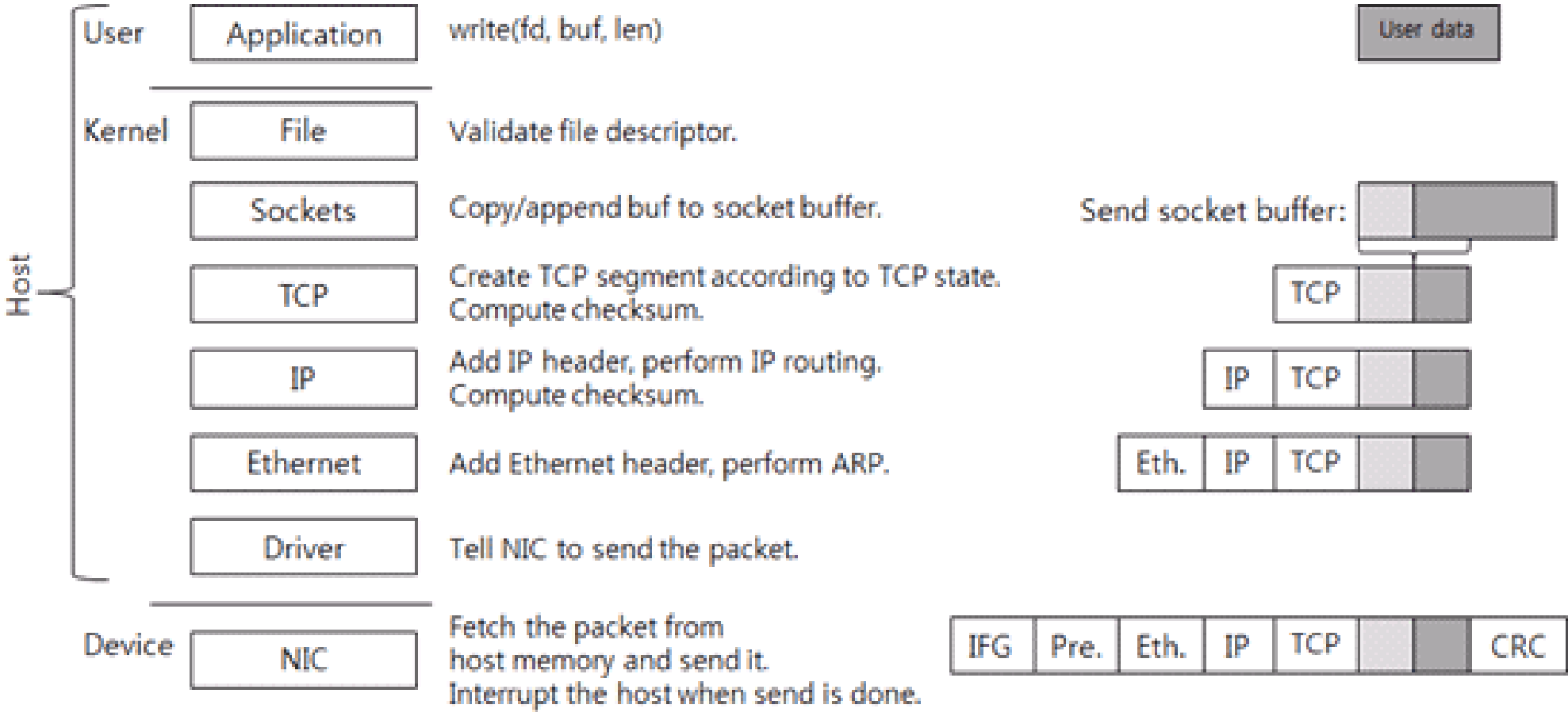
Data Structure

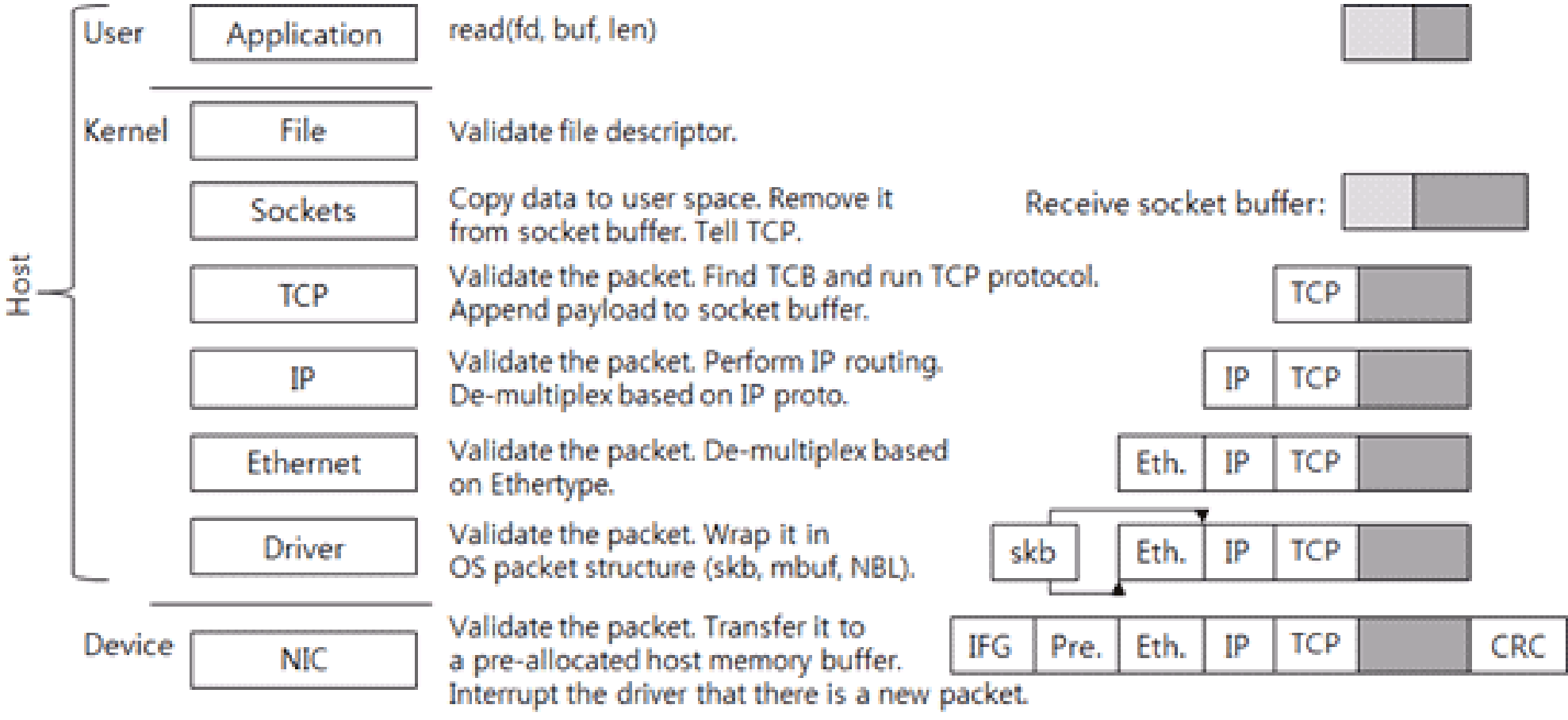
- sk_buff
- TCP Control Block
- TCP Connection Lookup Table

Next Week InchALLAH

Homework

Following content is based on
http://www.cubrid.org/blog/dev-platform/understanding-tcp-ip-network-stack/


Host
User
Application
write(fd, buf, len)
User data
Kernel
File
Validate file descriptor.
Sockets
Copy/append buf to socket buffer.
Send socket buffer:
TCP
Create TCP segment according to TCP state.
Compute checksum.
TCP
IP
Add IP header, perform IP routing.
Compute checksum.
IP
TCP
Ethernet
Add Ethernet header, perform ARP.
Eth.
IP
TCP
Driver
Tell NIC to send the packet.
Device
NIC
Fetch the packet from
host memory and send it.
Interrupt the host when send is done.
IFG
Pre.
Eth.
IP
TCP
CRC


Host
User
Application
read(fd, buf, len)
Kernel
File
Validate file descriptor.
Sockets
Copy data to user space. Remove it from socket buffer. Tell TCP.
Receive socket buffer:
TCP
Validate the packet. Find TCB and run TCP protocol. Append payload to socket buffer.
TCP
IP
Validate the packet. Perform IP routing. De-multiplex based on IP proto.
IP
TCP
Ethernet
Validate the packet. De-multiplex based on Ethertype.
Eth.
IP
TCP
Driver
Validate the packet. Wrap it in OS packet structure (skb, mbuf, NBL).
skb
Eth.
IP
TCP
Device
NIC
Validate the packet. Transfer it to a pre-allocated host memory buffer. Interrupt the driver that there is a new packet.
IFG
Pre.
Eth.
IP
TCP
CRC

NIC

NIC

- NIC writes the packet onto its memory.

NIC

- NIC writes the packet onto its memory.
- NIC checks whether the packet is valid by performing the CRC
- NIC send the packet to the memory buffer of the host

NIC

- NIC writes the packet onto its memory.
- NIC checks whether the packet is valid by performing the CRC
- NIC send the packet to the memory buffer of the host
- This buffer is a memory that has already been requested by the driver to the kernel and allocated for receiving packets

NIC

- NIC writes the packet onto its memory.
- NIC checks whether the packet is valid by performing the CRC
- NIC send the packet to the memory buffer of the host
- This buffer is a memory that has already been requested by the driver to the kernel and allocated for receiving packets
- After the buffer has been allocated, the driver tells the memory address and size to the NIC.
- When there is no host memory buffer allocated by the driver even though NIC receives a packet, NIC may drop the packet

Device Driver

- After sending the packet to the host memory buffer, the NIC sends an interrupt to the host OS.

Device Driver

- After sending the packet to the host memory buffer, the NIC sends an interrupt to the host OS.
- Driver checks whether it can handle the new packet or not. So far, the driver-NIC communication protocol defined by the manufacturer is used.

Device Driver

- After sending the packet to the host memory buffer, the NIC sends an interrupt to the host OS.
- Driver checks whether it can handle the new packet or not. So far, the driver-NIC communication protocol defined by the manufacturer is used.
- When the driver should send a packet to the upper layer, the packet must be wrapped in a packet structure that the OS uses to understand the packet.

Device Driver

- After sending the packet to the host memory buffer, the NIC sends an interrupt to the host OS.
- Driver checks whether it can handle the new packet or not. So far, the driver-NIC communication protocol defined by the manufacturer is used.
- When the driver should send a packet to the upper layer, the packet must be wrapped in a packet structure that the OS uses to understand the packet.
- For example, **sk_buff** of **Linux**, **mbuf** of **BSD-series** kernel, and **NET_BUFFER_LIST** of **Microsoft Windows**.

Ethernet Layer

Ethernet Layer

1. Check whether the packet is valid

Ethernet Layer

1. Check whether the packet is valid
2. De-multiplexes the upper protocol (network protocol)

Ethernet Layer

1. Check whether the packet is valid
2. De-multiplexes the upper protocol (network protocol)
3. It uses the **ethertype value** of Ethernet header.
4. **IPv4** ethertype value is **0x0800**

Ethernet Layer

1. Check whether the packet is valid
2. De-multiplexes the upper protocol (network protocol)
3. It uses the **ethertype value** of Ethernet header.
4. **IPv4** ethertype value is **0x0800**
5. It removes Ethernet header and sends packet to the IP layer.

IP Layer

IP Layer

1. Checks whether packet is valid, using IP header checksum.

IP Layer

1. Checks whether packet is valid, using IP header checksum.
2. Logically determines whether it should perform IP routing and make the local system handle the packet, or send the packet to the other system.

IP Layer

1. Checks whether packet is valid, using IP header checksum.
2. Logically determines whether it should perform IP routing and make the local system handle the packet, or send the packet to the other system.
3. If packet must be handled by the local system, IP layer de-multiplexes the upper protocol (transport protocol) by referring to the **proto value** of IP header.
4. **TCP** proto value is 6.

IP Layer

1. Checks whether packet is valid, using IP header checksum.
2. Logically determines whether it should perform IP routing and make the local system handle the packet, or send the packet to the other system.
3. If packet must be handled by the local system, IP layer de-multiplexes the upper protocol (transport protocol) by referring to the **proto value** of IP header.
4. **TCP** proto value is 6.
5. It removes IP header and then sends the packet to the TCP layer.

TCP Layer

TCP Layer

- Checks whether packet is valid (TCP checksum).
- TCP checksum is computed by NIC, not kernel.

TCP Layer

- Checks whether packet is valid (TCP checksum).
- TCP checksum is computed by NIC, not kernel.
- It searches TCP control block where the packet is connected. Following is used as identifier

```
<source IP, source port, target IP,
   target port>
```

TCP Layer

- Checks whether packet is valid (TCP checksum).
- TCP checksum is computed by NIC, not kernel.
- It searches TCP control block where the packet is connected. Following is used as identifier

```
<source IP, source port, target IP,
   target port>
```

- After searching the connection, it performs the protocol to handle the packet.

TCP Layer

- Checks whether packet is valid (TCP checksum).
- TCP checksum is computed by NIC, not kernel.
- It searches TCP control block where the packet is connected. Following is used as identifier

```
<source IP, source port, target IP,
   target port>
```

- After searching the connection, it performs the protocol to handle the packet.
- If it has received new data, it adds the data to the receive socket buffer.

TCP Layer

- Checks whether packet is valid (TCP checksum).
- TCP checksum is computed by NIC, not kernel.
- It searches TCP control block where the packet is connected. Following is used as identifier

```
<source IP, source port, target IP,
   target port>
```

- After searching the connection, it performs the protocol to handle the packet.
- If it has received new data, it adds the data to the receive socket buffer.
- According to TCP state, it can send a new TCP packet (ex. ACK).

Now

Now, TCP/IP receiving packet handling has completed.

Notes

Notes

- Size of receive socket buffer is TCP receive window.

Notes

- Size of receive socket buffer is TCP receive window.
- To a certain point, TCP throughput increases when receive windows is large.

Notes

- Size of receive socket buffer is TCP receive window.
- To a certain point, TCP throughput increases when receive windows is large.
- In the past, socket buffer size had been adjusted on the application or the OS configuration.

Notes

- Size of receive socket buffer is TCP receive window.
- To a certain point, TCP throughput increases when receive windows is large.
- In the past, socket buffer size had been adjusted on the application or the OS configuration.
- Latest network stack has a function to adjust the receive window automatically.

Application

Application

- When application calls the **read** system call, area is changed to the kernel area and data in the socket buffer is copied to the memory in the user area.

Application

- When application calls the **read** system call, area is changed to the kernel area and data in the socket buffer is copied to the memory in the user area.
- Copied data is removed from the socket buffer.

Application

- When application calls the **read** system call, area is changed to the kernel area and data in the socket buffer is copied to the memory in the user area.
- Copied data is removed from the socket buffer.
- Then, TCP is called.

Application

- When application calls the **read** system call, area is changed to the kernel area and data in the socket buffer is copied to the memory in the user area.
- Copied data is removed from the socket buffer.
- Then, TCP is called.
- TCP increases receive window because there is new space in the socket buffer.

Application

- When application calls the **read** system call, area is changed to the kernel area and data in the socket buffer is copied to the memory in the user area.
- Copied data is removed from the socket buffer.
- Then, TCP is called.
- TCP increases receive window because there is new space in the socket buffer.
- TCP sends a packet according to protocol status. If no packet is transfered, system call is terminated.

Functions of Network Stack Layers

Functions of Network Stack Layers

- Functions described so far are the most basic functions.

Functions of Network Stack Layers

- Functions described so far are the most basic functions.
- Network stack in the early 1990s had few more functions than described.

Functions of Network Stack Layers

- Functions described so far are the most basic functions.
- Network stack in the early 1990s had few more functions than described.
- Latest network stack has many more functions and complexity.

Functions of Network Stack Layers

- Functions described so far are the most basic functions.
- Network stack in the early 1990s had few more functions than described.
- Latest network stack has many more functions and complexity.
- Latest network stack is classified by purpose as follows:
- Packet processing procedure manipulation
- Protocol performance
- Packet processing efficiency

Packet Processing Procedure Manipulation

Packet Processing Procedure Manipulation

- It is a function like Netfilter (firewall, NAT) and traffic control.

Packet Processing Procedure Manipulation

- It is a function like Netfilter (firewall, NAT) and traffic control.
- By inserting the user-controllable code to the basic processing flow, the function can work differently according to user configuration.

Protocol Performance

Protocol Performance

- It aims to improve the throughput, latency, and stability

Protocol Performance

- It aims to improve the throughput, latency, and stability
- Various congestion control algorithms and additional TCP functions can be added

Packet Processing Efficiency

Packet Processing Efficiency

- Aims to improve maximum number of packets that can be processed per second, by reducing
  - CPU cycle
  - memory usage
  - memory accesses

Packet Processing Efficiency

- Aims to improve maximum number of packets that can be processed per second, by reducing
  - CPU cycle
  - memory usage
  - memory accesses
- Attempts include
  - stack parallel processing
  - header prediction
  - zero-copy
  - single-copy
  - checksum offload
  - TSO, LRO, RSS
  - etc.

Control Flow in the Stack

More detailed look at the internal flow of the Linux network stack.

Control Flow in the Stack

More detailed look at the internal flow of the Linux network stack.

- Like a subsystem which is not a network stack, it basically runs as the event-driven way

Control Flow in the Stack

More detailed look at the internal flow of the Linux network stack.

- Like a subsystem which is not a network stack, it basically runs as the event-driven way
- Event-driven way: reacts when the event occurs.

Control Flow in the Stack

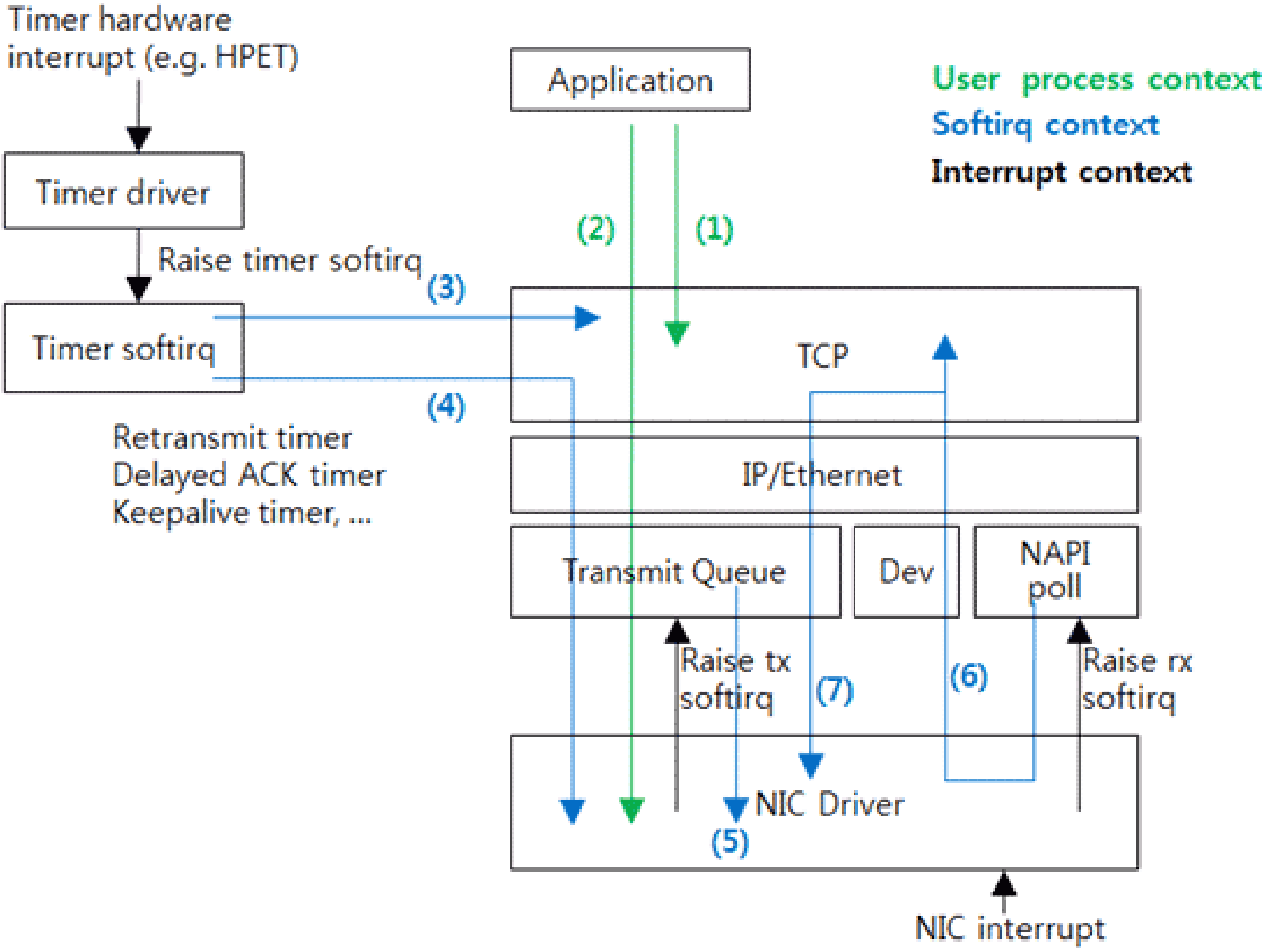
More detailed look at the internal flow of the Linux network stack.

- Like a subsystem which is not a network stack, it basically runs as the event-driven way
- Event-driven way: reacts when the event occurs.
- Therefore, there is no separated thread to execute the stack.

Control Flow in the Stack

More detailed look at the internal flow of the Linux network stack.

- Like a subsystem which is not a network stack, it basically runs as the event-driven way
- Event-driven way: reacts when the event occurs.
- Therefore, there is no separated thread to execute the stack.
- Previous figures were simplified diagrams of control flow.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

Flow 1

Flow 1

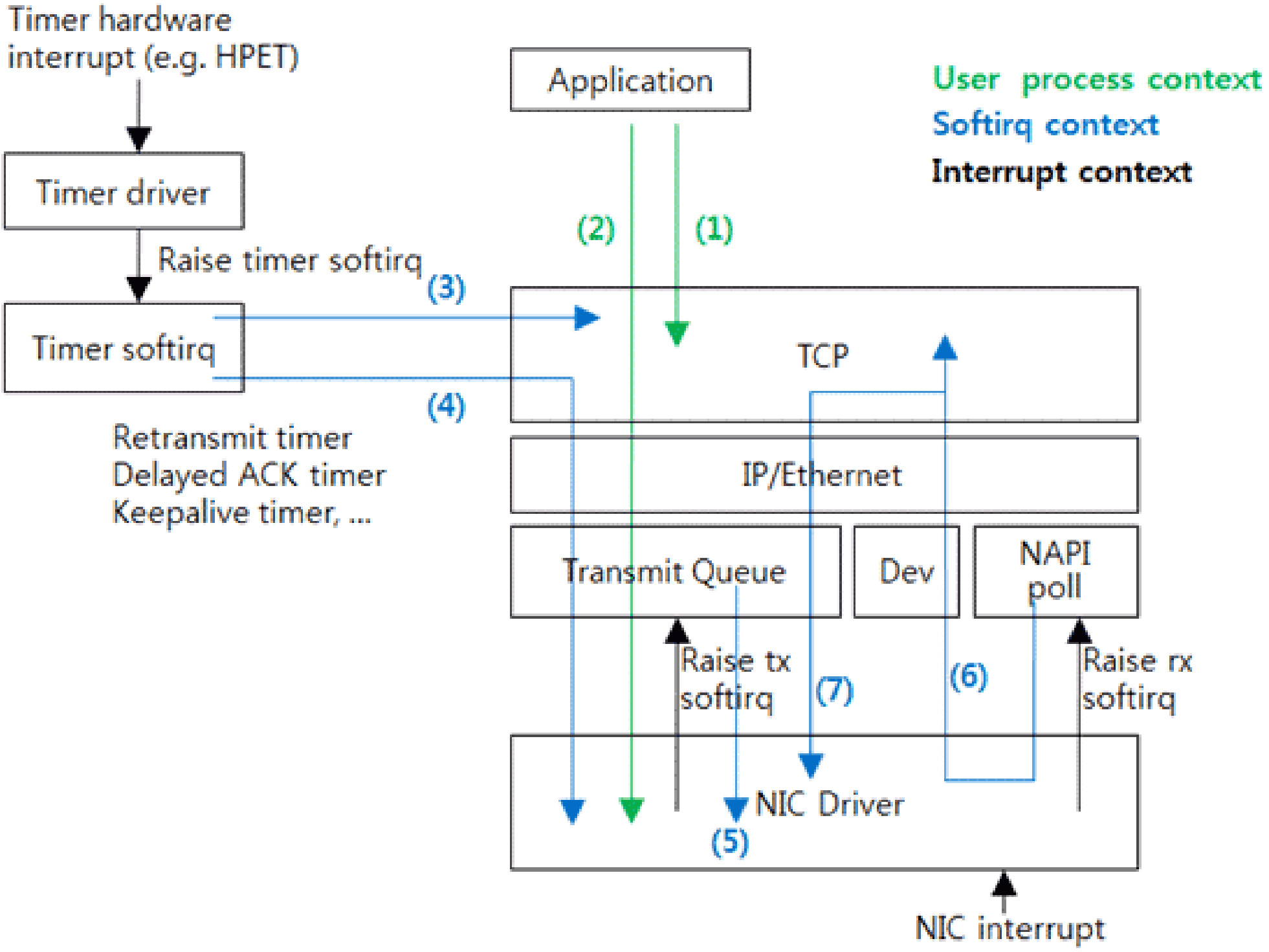
- Application calls a system call to execute (use) TCP.

Flow 1

- Application calls a system call to execute (use) TCP.
- For example, calls the read system call and the write system call and then executes TCP.

Flow 1

- Application calls a system call to execute (use) TCP.
- For example, calls the read system call and the write system call and then executes TCP.
- However, there is no packet transmission.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

Flow 2

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.
- A queue is in front of the driver.

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.
- A queue is in front of the driver.
- The packet comes into the queue first, and then the queue implementation structure decides the time to send the packet to the driver.

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.
- A queue is in front of the driver.
- The packet comes into the queue first, and then the queue implementation structure decides the time to send the packet to the driver.
- This is queue discipline **qdisc** of Linux

Flow 2

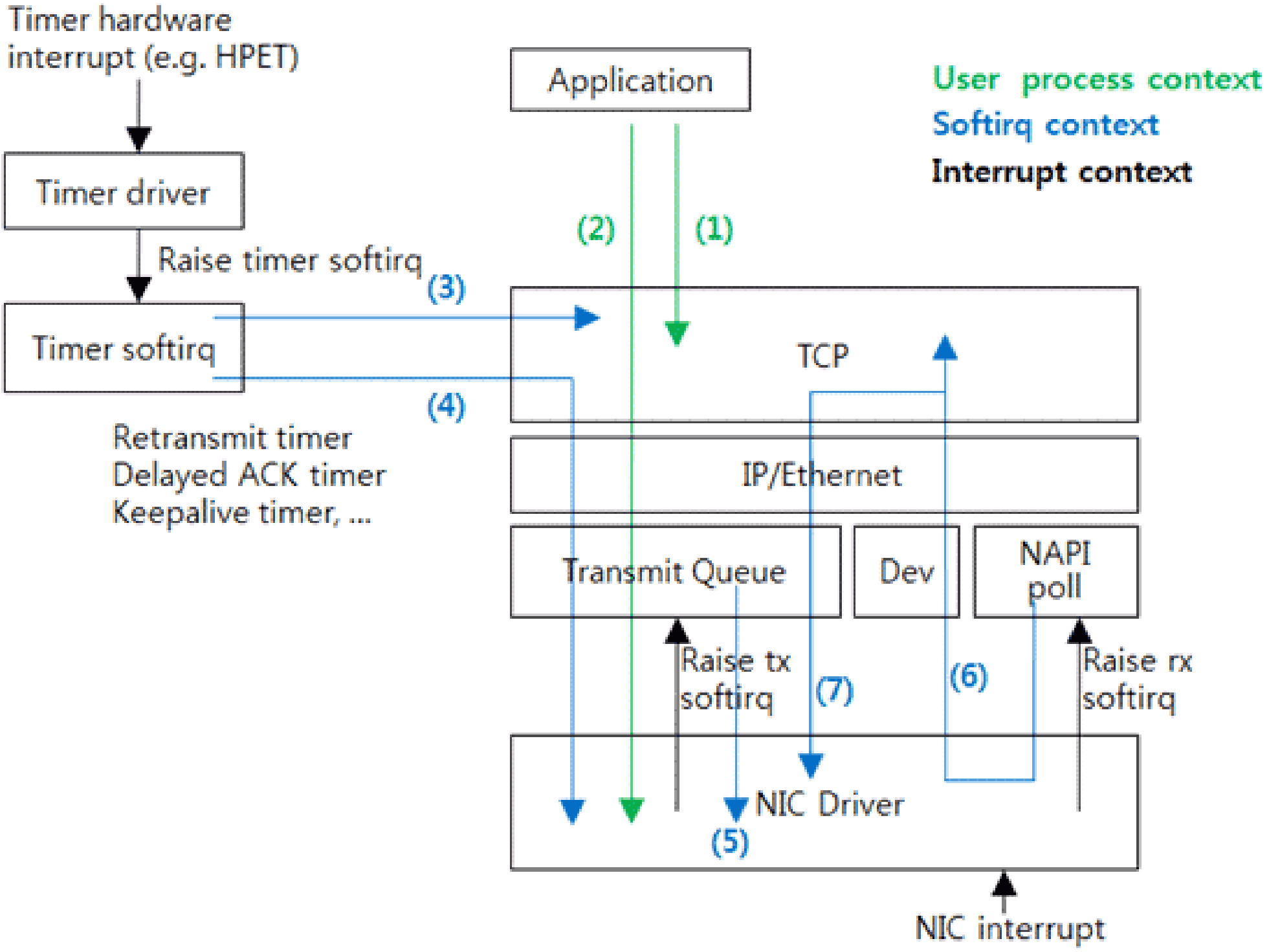
- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.
- A queue is in front of the driver.
- The packet comes into the queue first, and then the queue implementation structure decides the time to send the packet to the driver.
- This is queue discipline **qdisc** of Linux
- Function of Linux traffic control is to manipulate **qdisc**. FIFO is the default.

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.
- A queue is in front of the driver.
- The packet comes into the queue first, and then the queue implementation structure decides the time to send the packet to the driver.
- This is queue discipline **qdisc** of Linux
- Function of Linux traffic control is to manipulate **qdisc**. FIFO is the default.
- Using another **qdisc**, operators can achieve various effects such as artificial packet loss, packet delay, transmission rate limit, etc.

Flow 2

- Same as Flow 1 if it requires packet transmission after executing TCP.
- It creates a packet and sends down the packet to the driver.
- A queue is in front of the driver.
- The packet comes into the queue first, and then the queue implementation structure decides the time to send the packet to the driver.
- This is queue discipline **qdisc** of Linux
- Function of Linux traffic control is to manipulate **qdisc**. FIFO is the default.
- Using another **qdisc**, operators can achieve various effects such as artificial packet loss, packet delay, transmission rate limit, etc.
- At Flow 1 and Flow 2, the process thread of the application also executes the driver.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

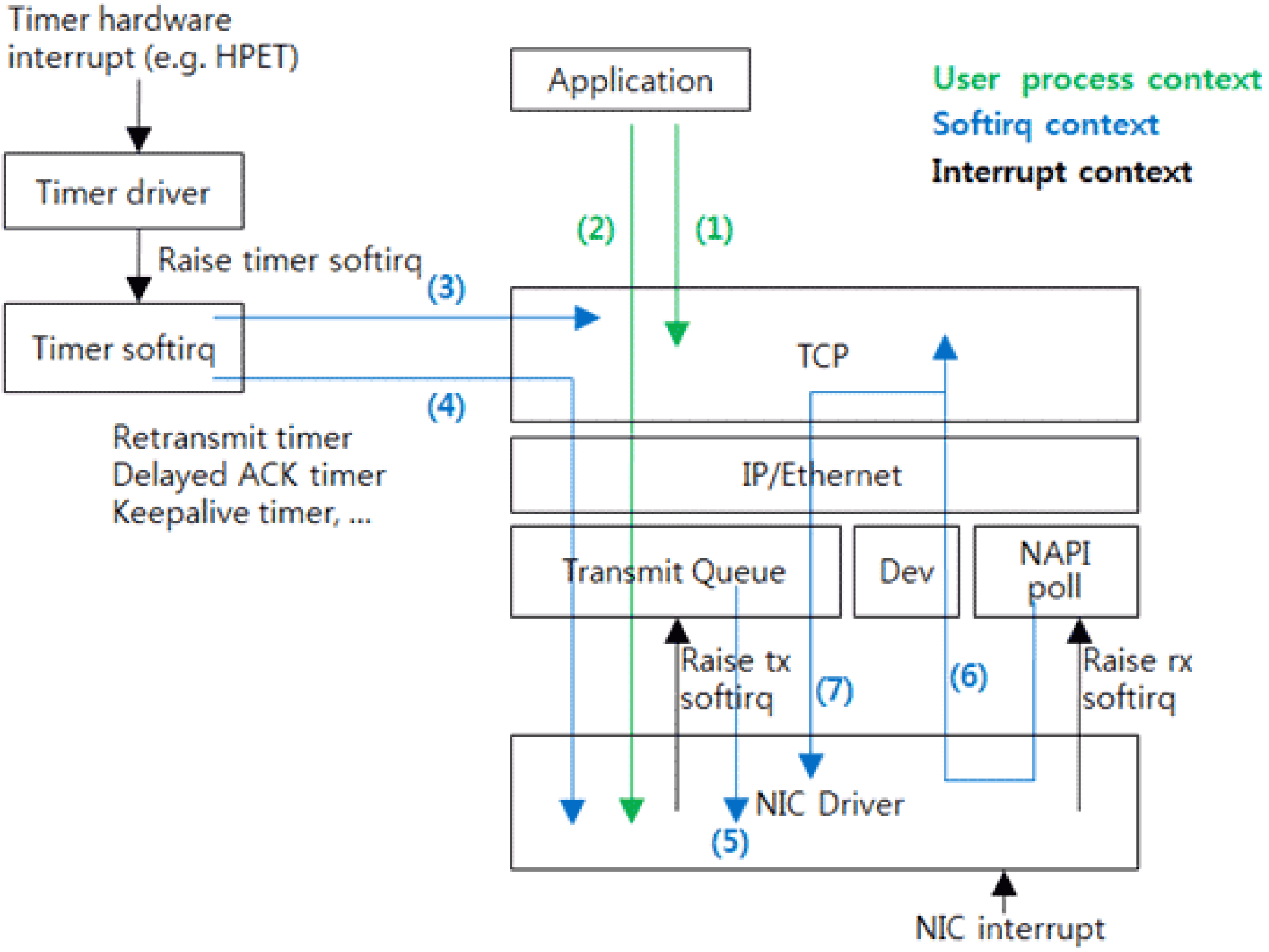
Flow 3

Flow 3

- Shows the case in which timer used by TCP has expired.

Flow 3

- Shows the case in which timer used by TCP has expired.
- Example, when **TIME_WAIT** timer has expired, TCP is called to delete the connection.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

Flow 4

Flow 4

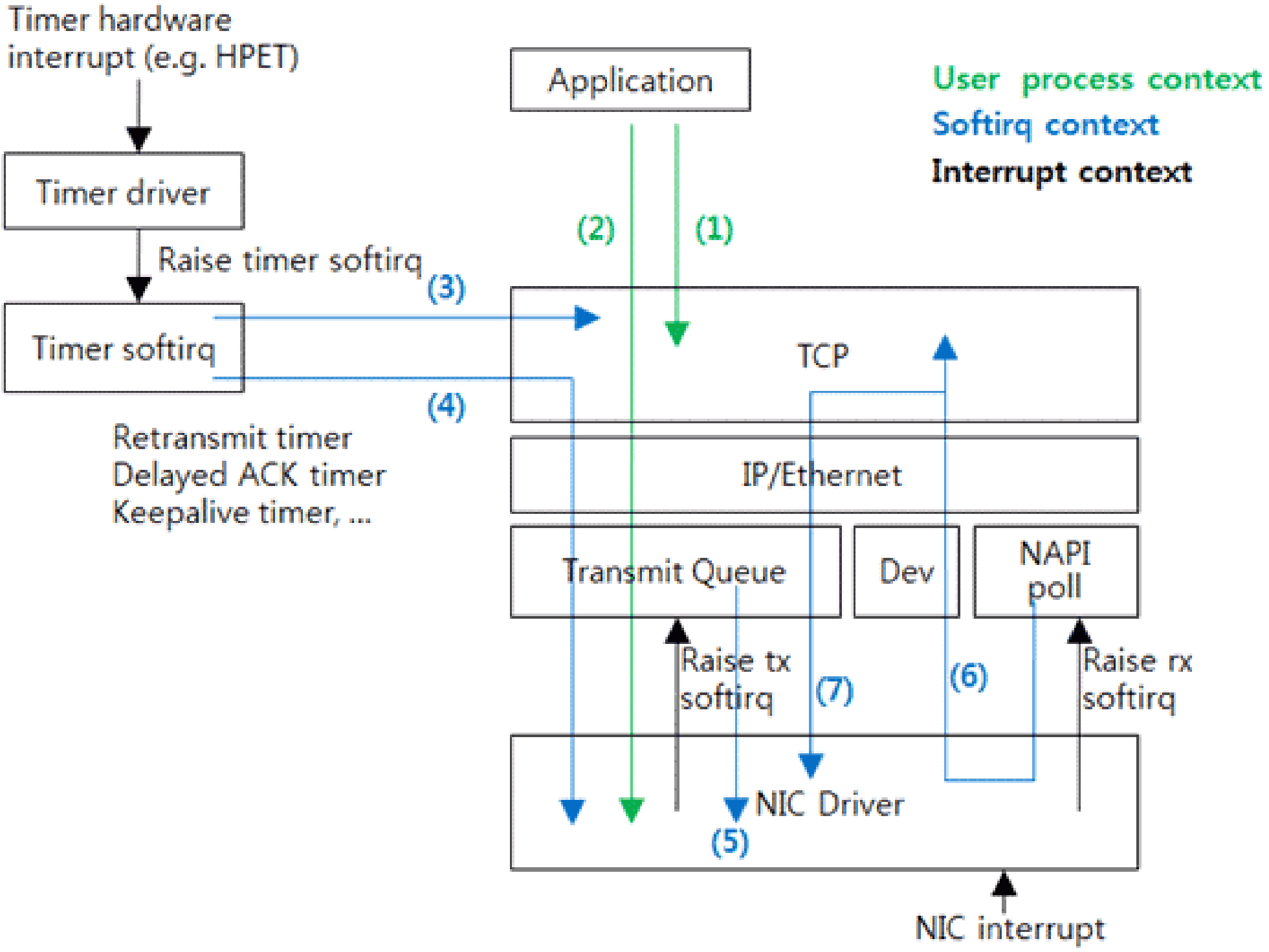
- It is the case in which the timer used by TCP has expired and TCP execution result packet should be transmitted.

Flow 4

- It is the case in which the timer used by TCP has expired and TCP execution result packet should be transmitted.
- Example, when the retransmit timer has expired, packet of which ACK has not been received is transmitted.

Flow 4

- It is the case in which the timer used by TCP has expired and TCP execution result packet should be transmitted.
- Example, when the retransmit timer has expired, packet of which ACK has not been received is transmitted.
- **Flow 3** and **Flow 4** show the procedure of executing the timer softirq that has processed the timer interrupt.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

Flow 5

Flow 5

- When NIC driver receives an interrupt, it frees the transmitted packet.

Flow 5

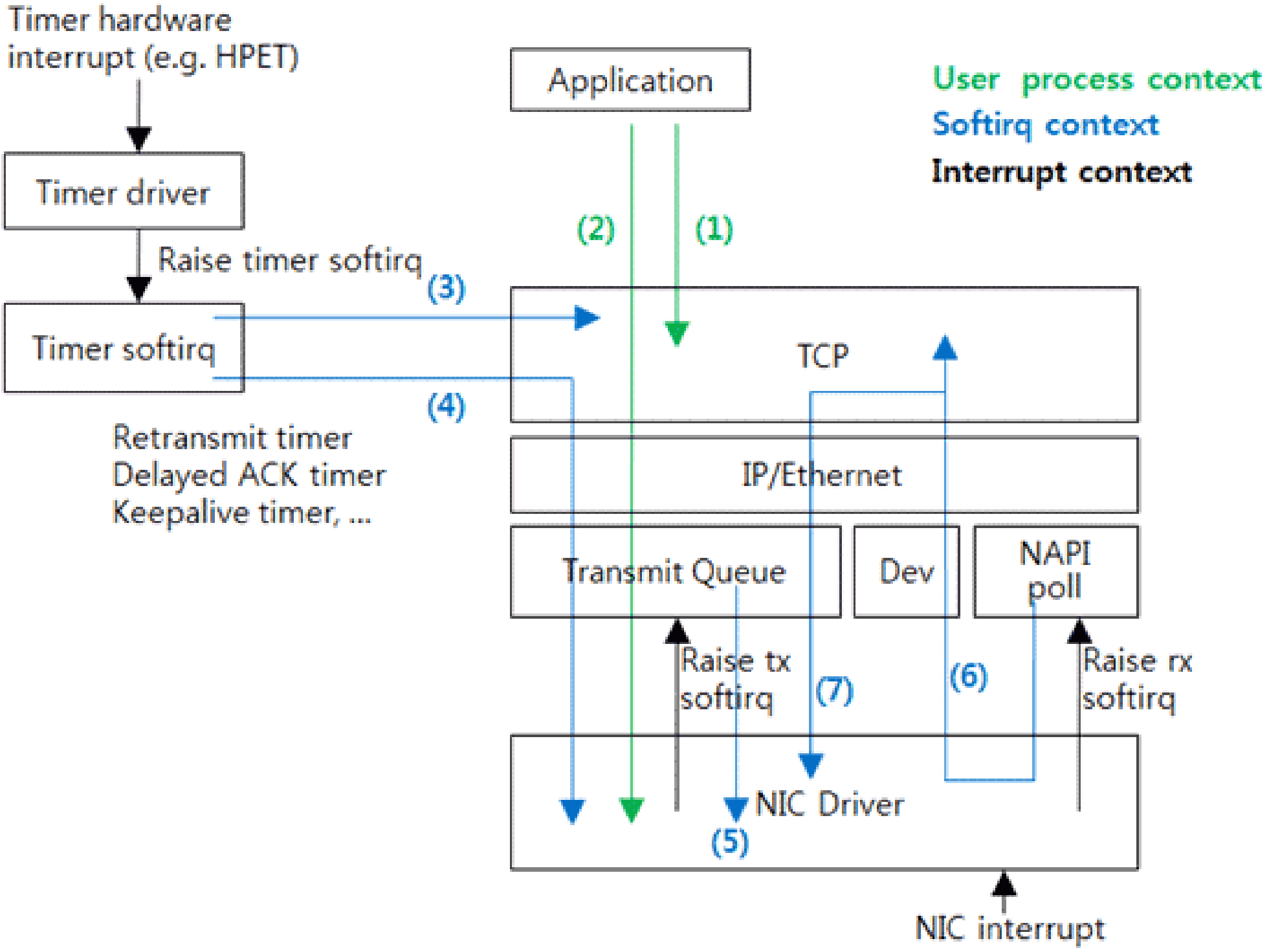
- When NIC driver receives an interrupt, it frees the transmitted packet.
- In most cases, execution of the driver is terminated here.

Flow 5

- When NIC driver receives an interrupt, it frees the transmitted packet.
- In most cases, execution of the driver is terminated here.
- **Flow 5** is the case of packet accumulation in the transmit queue.

Flow 5

- When NIC driver receives an interrupt, it frees the transmitted packet.
- In most cases, execution of the driver is terminated here.
- **Flow 5** is the case of packet accumulation in the transmit queue.
- The driver requests softirq and the softirq handler executes the transmit queue to send the accumulated packet to the driver.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

NAPI

NAPI

- When NIC driver receives an interrupt and finds a newly received packet, it requests softirq.

NAPI

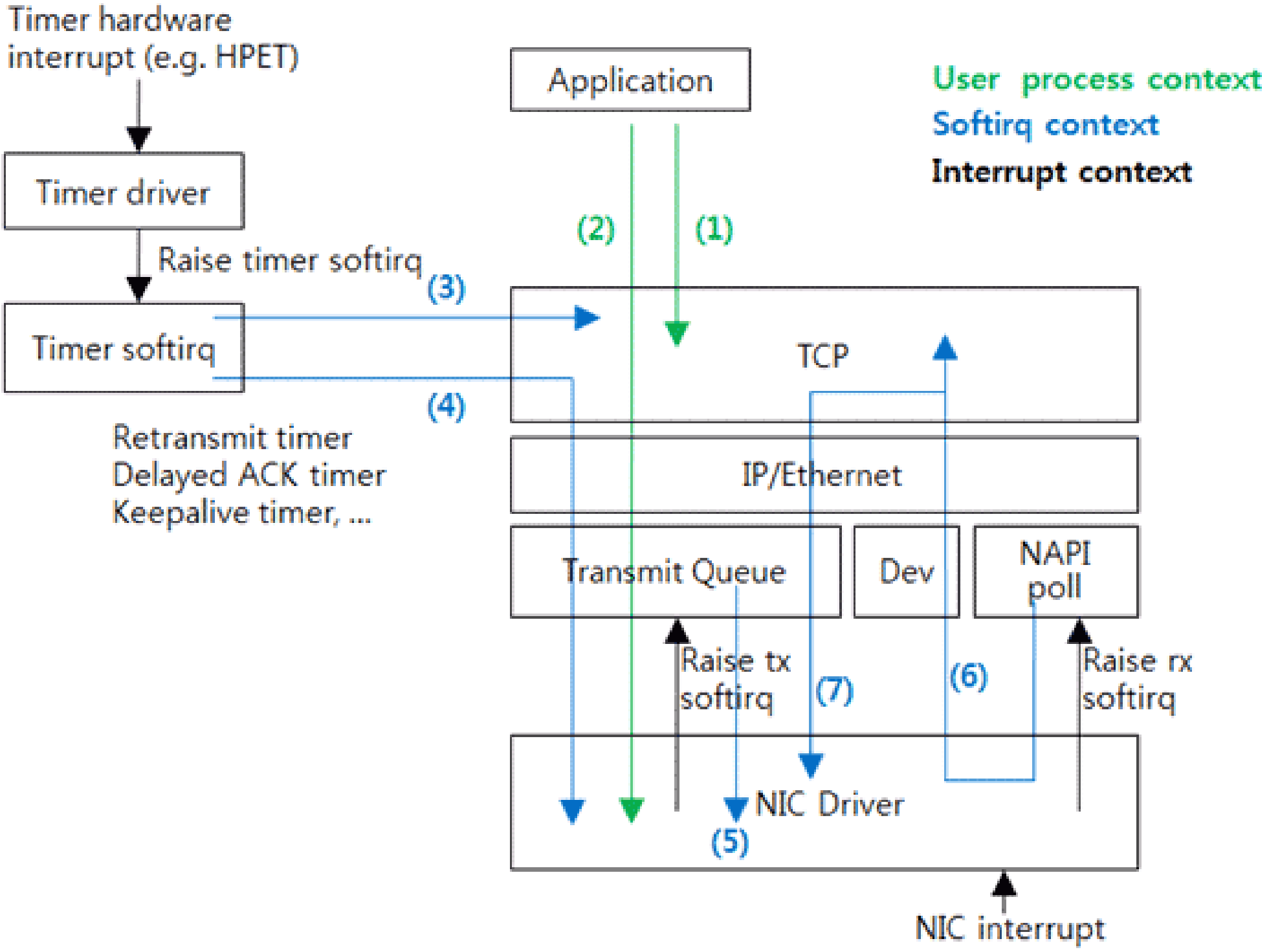
- When NIC driver receives an interrupt and finds a newly received packet, it requests softirq.
- softirq that processes the received packet, calls the driver, and transmits the received packet to upper layer.

NAPI

- When NIC driver receives an interrupt and finds a newly received packet, it requests softirq.
- softirq that processes the received packet, calls the driver, and transmits the received packet to upper layer.
- In Linux, processing the received packet this way is called **New API (NAPI)**.

NAPI

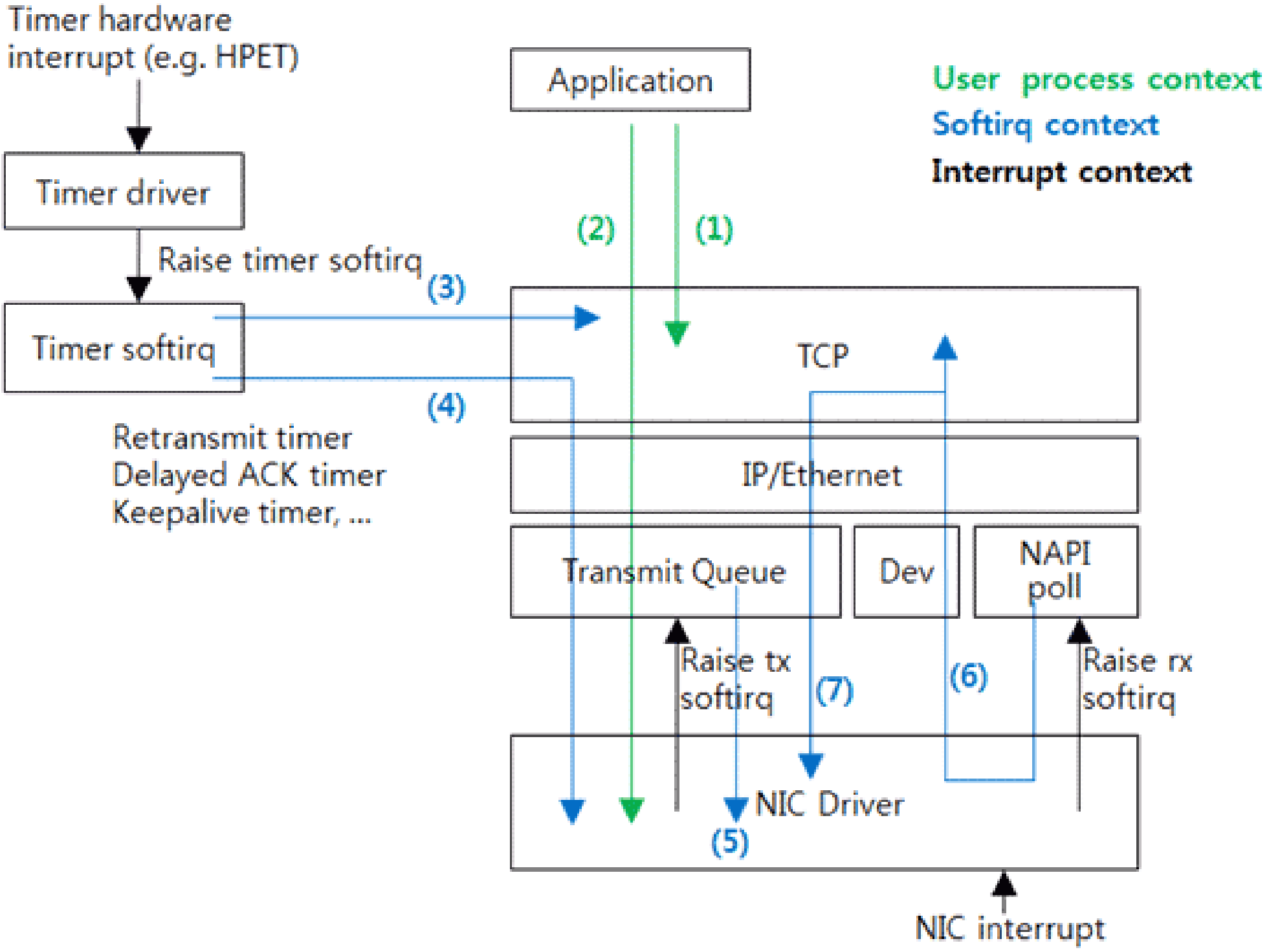
- When NIC driver receives an interrupt and finds a newly received packet, it requests softirq.
- softirq that processes the received packet, calls the driver, and transmits the received packet to upper layer.
- In Linux, processing the received packet this way is called **New API (NAPI)**.
- It is similar to polling because the driver does not directly transmit the packet to the upper layer, but the upper layer directly gets the packet.
- The actual code is called NAPI poll or poll


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

Flow 6

Flow 6

- Shows the case that completes execution of TCP.


Timer hardware interrupt (e.g. HPET)
Timer driver
Raise timer softirq
Timer softirq
Retransmit timer
Delayed ACK timer
Keepalive timer, ...
Application
User process context
Softirq context
Interrupt context
(1)
(2)
(3)
(4)
(5)
(6)
(7)
TCP
IP/Ethernet
Transmit Queue
Dev
NAPI poll
Raise tx softirq
Raise rx softirq
NIC Driver
NIC interrupt

Flow 7

Flow 7

- Shows the case that requires additional packet transmission.

Flow 7

- Shows the case that requires additional packet transmission.
- All of **Flow (5), (6), and (7)** are executed by the **softirq** which has processed the NIC interrupt.

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

- Assume CPU 0 is executing an application program (user program)

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

- Assume CPU 0 is executing an application program (user program)
- At this time, NIC receives a packet and generates an interrupt for the CPU 0.

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

- Assume CPU 0 is executing an application program (user program)
- At this time, NIC receives a packet and generates an interrupt for the CPU 0.
- CPU executes the kernel interrupt (called irq) handler.

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

- Assume CPU 0 is executing an application program (user program)
- At this time, NIC receives a packet and generates an interrupt for the CPU 0.
- CPU executes the kernel interrupt (called irq) handler.
- This handler refers to the interrupt number and then calls the driver interrupt handler.

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

- Assume CPU 0 is executing an application program (user program)
- At this time, NIC receives a packet and generates an interrupt for the CPU 0.
- CPU executes the kernel interrupt (called irq) handler.
- This handler refers to the interrupt number and then calls the driver interrupt handler.
- Driver frees the packet transmitted and then calls the **napi_schedule()** function to process the received packet.

How to Process Interrupt and Received Packet - 1

Next slide, shows the procedure of processing an interrupt

- Assume CPU 0 is executing an application program (user program)
- At this time, NIC receives a packet and generates an interrupt for the CPU 0.
- CPU executes the kernel interrupt (called irq) handler.
- This handler refers to the interrupt number and then calls the driver interrupt handler.
- Driver frees the packet transmitted and then calls the **napi_schedule()** function to process the received packet.
- This function requests the softirq (software interrupt).

How to Process Interrupt and Received Packet - 2

How to Process Interrupt and Received Packet - 2

- After execution of the driver interrupt handler has been terminated, the control is passed to the kernel handler.

How to Process Interrupt and Received Packet - 2

- After execution of the driver interrupt handler has been terminated, the control is passed to the kernel handler.
- The kernel handler executes the interrupt handler for the softirq.

How to Process Interrupt and Received Packet - 2

- After execution of the driver interrupt handler has been terminated, the control is passed to the kernel handler.
- The kernel handler executes the interrupt handler for the softirq.
- After the interrupt context has been executed, the softirq context will be executed.
- **Note:** Interrupt context and softirq context are executed by identical thread, however they use different stacks.
- **Note:** Interrupt context blocks hardware interrupts, however softirq context allows for hardware interrupts.

How to Process Interrupt and Received Packet - 3

How to Process Interrupt and Received Packet - 3

- softirq handler that processes the received packet is the **net_rx_action()** function.

How to Process Interrupt and Received Packet - 3

- softirq handler that processes the received packet is the **net_rx_action()** function.
- This function calls the **poll()** function of the driver.

How to Process Interrupt and Received Packet - 3

- softirq handler that processes the received packet is the **net_rx_action()** function.
- This function calls the **poll()** function of the driver.
- The **poll()** function calls the **netif_receive_skb()** function and then sends the received packets one by one to the upper layer.

How to Process Interrupt and Received Packet - 3

- softirq handler that processes the received packet is the **net_rx_action()** function.
- This function calls the **poll()** function of the driver.
- The **poll()** function calls the **netif_receive_skb()** function and then sends the received packets one by one to the upper layer.
- After processing the softirq, the application restarts execution from the stopped point in order to request a system call.

How to Process Interrupt and Received Packet - 4

How to Process Interrupt and Received Packet - 4

- Therefore, the CPU that has received the interrupt processes the received packets from the firts to the last.

How to Process Interrupt and Received Packet - 4

- Therefore, the CPU that has received the interrupt processes the received packets from the firts to the last.
- In Linux, BSD, and Microsoft Windows, the processing procedure is basically the same on this wise.

How to Process Interrupt and Received Packet - 4

- Therefore, the CPU that has received the interrupt processes the received packets from the firts to the last.
- In Linux, BSD, and Microsoft Windows, the processing procedure is basically the same on this wise.
- When you check the server CPU utilization, sometimes you can check that only one CPU executes the softirq hard among the server CPUs.

How to Process Interrupt and Received Packet - 4

- Therefore, the CPU that has received the interrupt processes the received packets from the firts to the last.
- In Linux, BSD, and Microsoft Windows, the processing procedure is basically the same on this wise.
- When you check the server CPU utilization, sometimes you can check that only one CPU executes the softirq hard among the server CPUs.
- The phenomenon occurs due to the way of processing received packets explained so far.

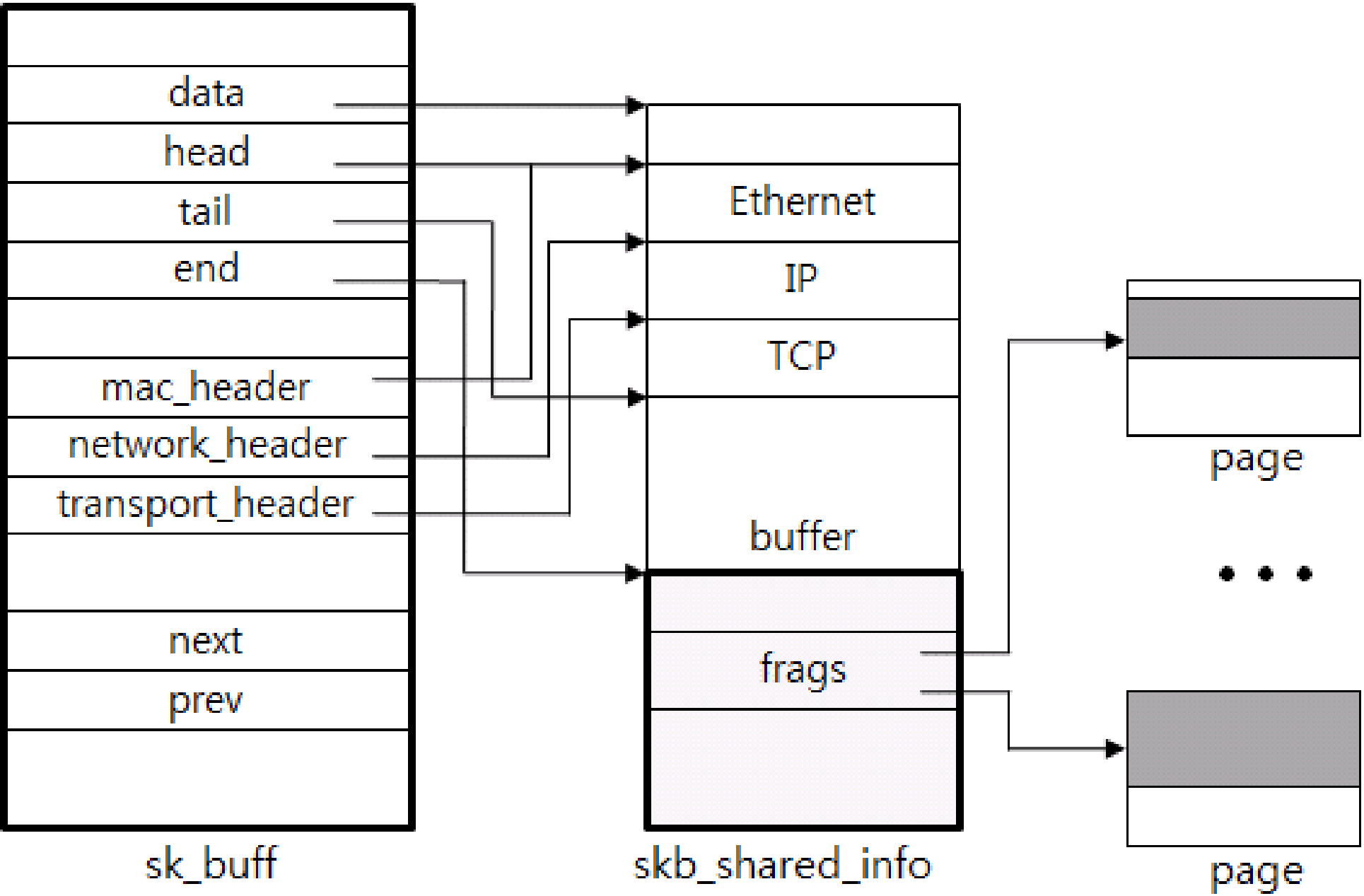
How to Process Interrupt and Received Packet - 4

- Therefore, the CPU that has received the interrupt processes the received packets from the firts to the last.
- In Linux, BSD, and Microsoft Windows, the processing procedure is basically the same on this wise.
- When you check the server CPU utilization, sometimes you can check that only one CPU executes the softirq hard among the server CPUs.
- The phenomenon occurs due to the way of processing received packets explained so far.
- To solve the problem, multi-queue NIC, RSS, and RPS have been developed.

Some Data Structures

The followings are some key data structures:

- sk_buff structure
- TCP Control Block
- net_device structure (NIC - skipped)
- TCP Connection Lookup Table


data
head
tail
end
mac_header
network_header
transport_header
next
prev
sk_buff
Ethernet
IP
TCP
buffer
frags
skb_shared_info
page
page

sk_buff

sk_buff

- **skb** structure that means a **packet**.

sk_buff

- **skb** structure that means a **packet**.
- Basic functions are:
  - Including Packet Data and *meta data*
  - How to Add or Delete a Header
  - How to Combine and Divide Packet
  - Quick Allocation and Free

skb - Including Packet Data and *meta data*

skb - Including Packet Data and *meta data*

- Structure directly includes packet data or refers to it by using a pointer.

skb - Including Packet Data and *meta data*

- Structure directly includes packet data or refers to it by using a pointer.
- In Figure, some of the packets refer to using the data pointer and the additional data (frags) refer to the actual page.

skb - Including Packet Data and *meta data*

- Structure directly includes packet data or refers to it by using a pointer.
- In Figure, some of the packets refer to using the data pointer and the additional data (frags) refer to the actual page.
- Necessary information, such as header and payload length is saved in the *meta data* area.

skb - Including Packet Data and *meta data*

- Structure directly includes packet data or refers to it by using a pointer.
- In Figure, some of the packets refer to using the data pointer and the additional data (frags) refer to the actual page.
- Necessary information, such as header and payload length is saved in the *meta data* area.
- In Figure, mac_header, network_header, transport_header have corresponding pointer data.

skb - Including Packet Data and *meta data*

- Structure directly includes packet data or refers to it by using a pointer.
- In Figure, some of the packets refer to using the data pointer and the additional data (frags) refer to the actual page.
- Necessary information, such as header and payload length is saved in the *meta data* area.
- In Figure, mac_header, network_header, transport_header have corresponding pointer data.
- Pointers point to the starting position of Ethernet header, IP header, and TCP header respectively.

skb - Including Packet Data and *meta data*

- Structure directly includes packet data or refers to it by using a pointer.
- In Figure, some of the packets refer to using the data pointer and the additional data (frags) refer to the actual page.
- Necessary information, such as header and payload length is saved in the *meta data* area.
- In Figure, mac_header, network_header, transport_header have corresponding pointer data.
- Pointers point to the starting position of Ethernet header, IP header, and TCP header respectively.
- This way makes TCP protocol processing easy.

How to Add or Delete a Header

How to Add or Delete a Header

- Header is added or deleted as up and down each layer of the network stack.

How to Add or Delete a Header

- Header is added or deleted as up and down each layer of the network stack.
- Pointers are used for more efficient processing.

How to Add or Delete a Header

- Header is added or deleted as up and down each layer of the network stack.
- Pointers are used for more efficient processing.
- For example, to remove the Ethernet header, just increase the head pointer.

How to Combine and Divide Packet

How to Combine and Divide Packet

- Linked-List is used for efficient execution of tasks such as adding or deleting packet payload data to the socket buffer, or packet chain.

How to Combine and Divide Packet

- Linked-List is used for efficient execution of tasks such as adding or deleting packet payload data to the socket buffer, or packet chain.
- The next pointer and pre pointer are used for this purpose.

Quick Allocation and Free

Quick Allocation and Free

- As a structure is allocated whenever creating a packet, the quick allocator is used.

Quick Allocation and Free

- As a structure is allocated whenever creating a packet, the quick allocator is used.
- For example, if data is transmitted at the speed of 10-Gigabit Ethernet, more than one million packets per second must be created and deleted.

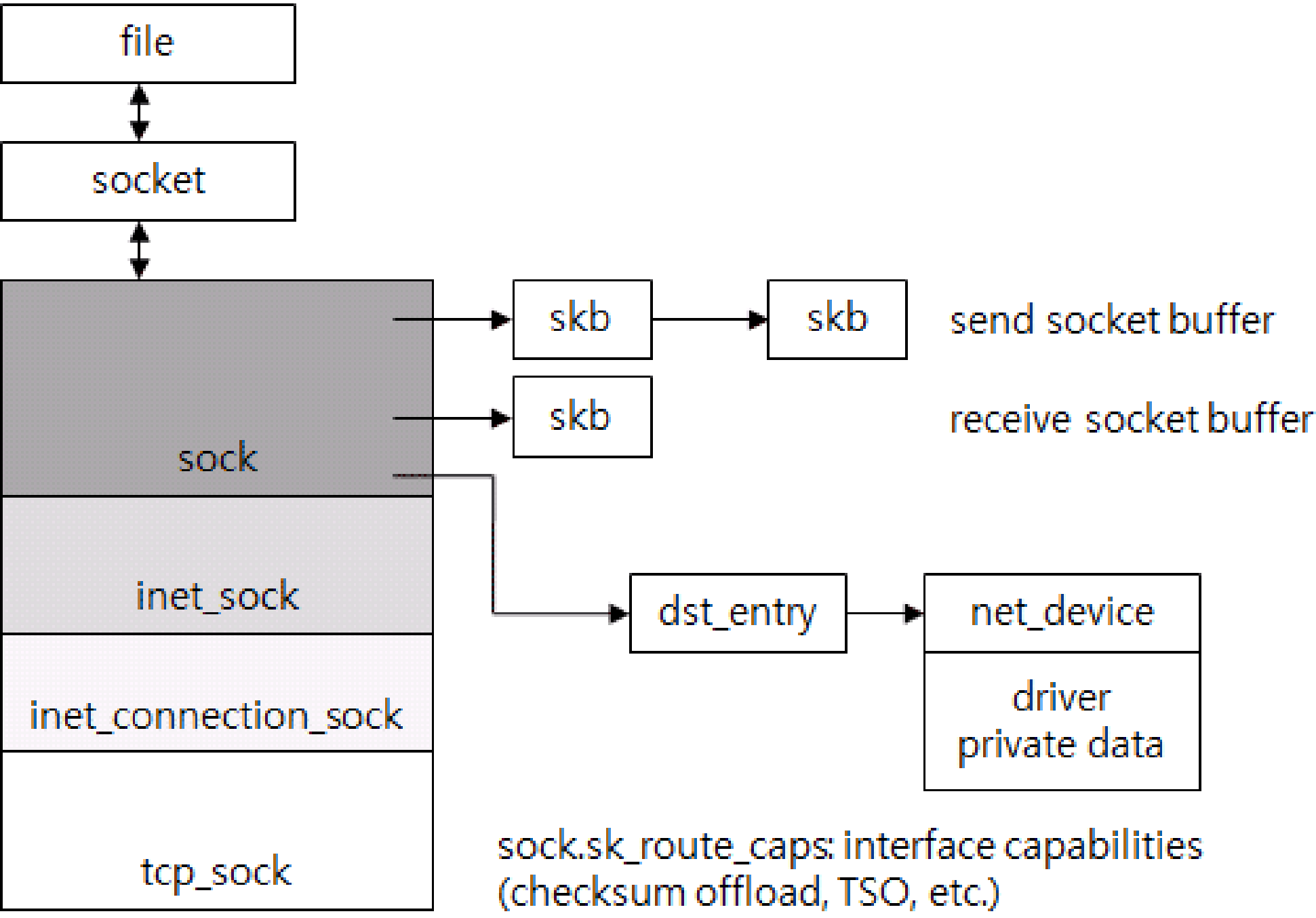
TCP Control Block - 1

TCP Control Block - 1

- Linux uses tcp_sock to represent the TCP connection.

TCP Control Block - 1

- Linux uses tcp_sock to represent the TCP connection.
- Here is the relationship among the file, the socket, and the tcp_sock.


file
socket
sock
inet_sock
inet_connection_sock
tcp_sock
skb
skb
skb
send socket buffer
receive socket buffer
dst_entry
net_device
driver
private data
sock.sk_route_caps: interface capabilities
(checksum offload, TSO, etc.)

TCP Control Block - 2

TCP Control Block - 2

- When a system call has occurred, it searches the file in the file descriptor used by the application that has caled the system call.

TCP Control Block - 2

- When a system call has occurred, it searches the file in the file descriptor used by the application that has caled the system call.
- For the Unix-series OS, the socket, the file, and the device for general file system for storage are abstracted to a file.

TCP Control Block - 2

- When a system call has occurred, it searches the file in the file descriptor used by the application that has caled the system call.
- For the Unix-series OS, the socket, the file, and the device for general file system for storage are abstracted to a file.
- Therefore, the file structure includes the least information.

TCP Control Block - 2

- When a system call has occurred, it searches the file in the file descriptor used by the application that has caled the system call.
- For the Unix-series OS, the socket, the file, and the device for general file system for storage are abstracted to a file.
- Therefore, the file structure includes the least information.
- For a socket, a separate socket structure saves the socket-related information and the file refers to the socket as a pointer.

TCP Control Block - 2

- When a system call has occurred, it searches the file in the file descriptor used by the application that has caled the system call.
- For the Unix-series OS, the socket, the file, and the device for general file system for storage are abstracted to a file.
- Therefore, the file structure includes the least information.
- For a socket, a separate socket structure saves the socket-related information and the file refers to the socket as a pointer.
- The socket refers to the **tcp_sock** again.

TCP Control Block - 2

- When a system call has occurred, it searches the file in the file descriptor used by the application that has caled the system call.
- For the Unix-series OS, the socket, the file, and the device for general file system for storage are abstracted to a file.
- Therefore, the file structure includes the least information.
- For a socket, a separate socket structure saves the socket-related information and the file refers to the socket as a pointer.
- The socket refers to the **tcp_sock** again.
- **tcp_sock** is classified into **sock, inet_sock**, etc. to support various protocols except TCP (kind of polymorphism).

TCP Control Block - 3

TCP Control Block - 3

- All status information used by TCP protocol is saved in **tcp_sock** (sequence number, receive window, etc.).

TCP Control Block - 3

- All status information used by TCP protocol is saved in **tcp_sock** (sequence number, receive window, etc.).
- send socket buffer and receive socket buffer are the **sk_buff** lists and they include the **tcp_sock**.

TCP Control Block - 3

- All status information used by TCP protocol is saved in **tcp_sock** (sequence number, receive window, etc.).
- send socket buffer and receive socket buffer are the **sk_buff** lists and they include the **tcp_sock**.
- **dst_entry** (the IP routing result) is referred to in order to avoid frequent routing.

TCP Control Block - 3

- All status information used by TCP protocol is saved in **tcp_sock** (sequence number, receive window, etc.).
- send socket buffer and receive socket buffer are the **sk_buff** lists and they include the **tcp_sock**.
- **dst_entry** (the IP routing result) is referred to in order to avoid frequent routing.
- **dst_entry** allows for easy search of the ARP result, i.e., the destination MAC address.

TCP Control Block - 3

- All status information used by TCP protocol is saved in **tcp_sock** (sequence number, receive window, etc.).
- send socket buffer and receive socket buffer are the **sk_buff** lists and they include the **tcp_sock**.
- **dst_entry** (the IP routing result) is referred to in order to avoid frequent routing.
- **dst_entry** allows for easy search of the ARP result, i.e., the destination MAC address.
- **dst_entry** is part of the routing table.

TCP Control Block - 3

- All status information used by TCP protocol is saved in **tcp_sock** (sequence number, receive window, etc.).
- send socket buffer and receive socket buffer are the **sk_buff** lists and they include the **tcp_sock**.
- **dst_entry** (the IP routing result) is referred to in order to avoid frequent routing.
- **dst_entry** allows for easy search of the ARP result, i.e., the destination MAC address.
- **dst_entry** is part of the routing table.
- NIC is expressed as the **net_device** structure.

TCP Connection Lookup Table

TCP Connection Lookup Table

- Hash table used to search TCP connection where the received data belongs.

TCP Connection Lookup Table

- Hash table used to search TCP connection where the received data belongs.
- Hash is calculated by using Jenkins hash algorithm of the input data of:

TCP Connection Lookup Table

- Hash table used to search TCP connection where the received data belongs.
- Hash is calculated by using Jenkins hash algorithm of the input data of:
- source IP, target IP, source port, target port

TCP Connection Lookup Table

- Hash table used to search TCP connection where the received data belongs.
- Hash is calculated by using Jenkins hash algorithm of the input data of:
- source IP, target IP, source port, target port
- Hash function has been selected by considering defense against attacks to the hash table.

Next Week InchALLAH

Next Lecture

- Following Code
- Lab setup
- Basic *nix Networking commands
- Socket Programming in C++

Next Lab

- **Continue** Review TCP/IP Suite (Solve selected Qs)
- Qs can be found at `http://technologyeye.weebly.com/uploads/9/6/1/4/9614102/interview_question_networkingrajkumar.pdf`

Homework

- Study this lecture (very well)
- Prepare for the Next Lecture
- Check some Kernel code (mainly Networking Subsystem)
- Study the Lab file
- Check the github Course repository


YOU GO
NOW!