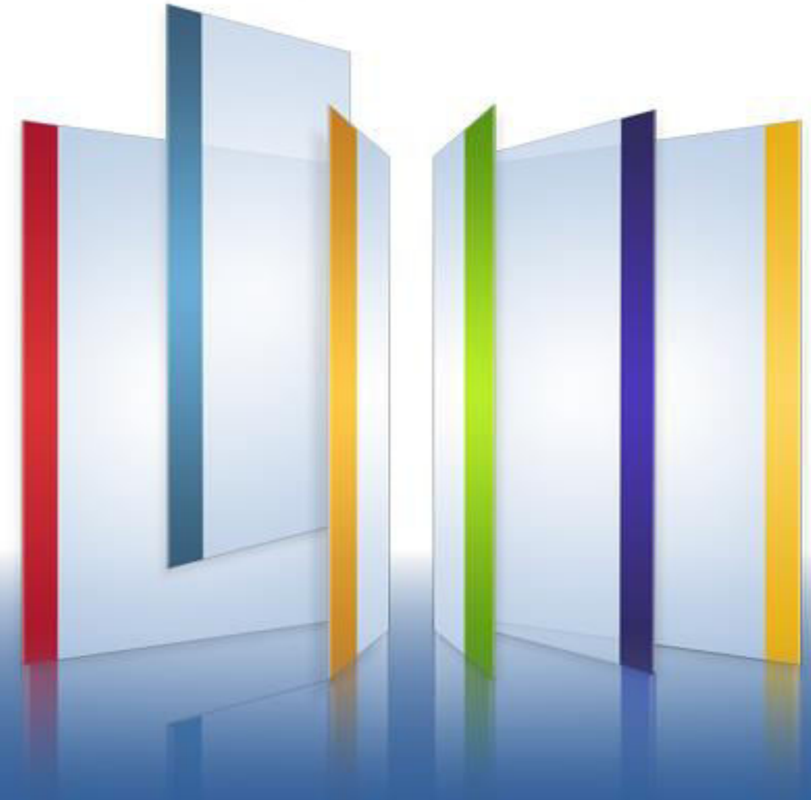
CHECK POINT
INSTITUTE FOR
INFORMATION SECURITY

בית הספר למדעי המחשב על שם בלבטניק
The Blavatnik School of Computer Science

Check Point
SOFTWARE TECHNOLOGIES LTD.



# Lecture 3: Stateless Packet Filtering

# Agenda

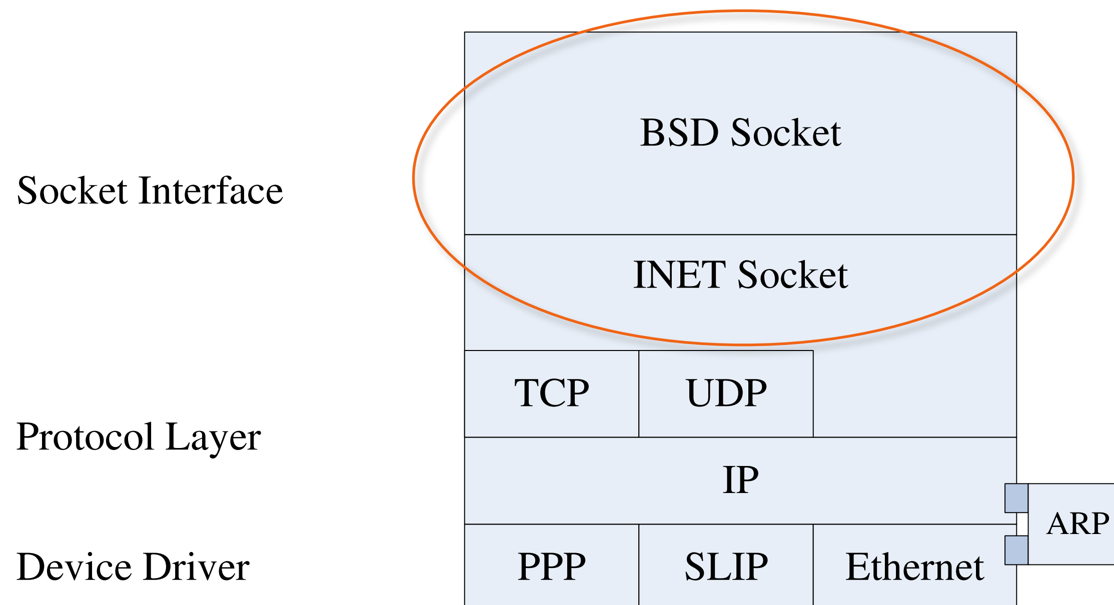
1. Linux file system - networking
2. sk_buff
3. Stateless packet filtering
4. About next assignment

# Agenda

1. **Linux file system - networking**
2. sk_buff
3. Stateless packet filtering
4. About next assignment

# Linux networking

- "Berkeley Sockets" library implemented in 1983 in Unix
- In time, become the standard networking interface in Unix and Windows (posix, winsock)
- First, a little reminder about how it all works



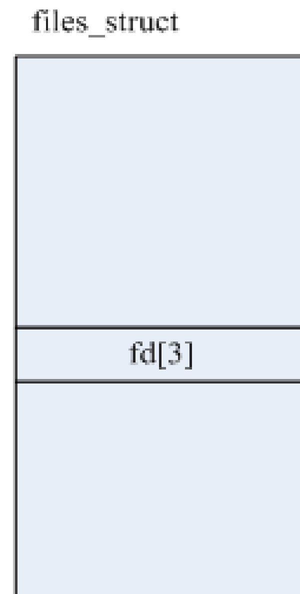
# File Descriptor

- Abstract indicator for accessing a "file"
- Three standard streams:

| Integer value | Name | <unistd.h> symbolic constant[1] | <stdio.h> file stream[2] |
|---|---|---|---|
| 0 | Standard input | STDIN_FILENO | stdin |
| 1 | Standard output | STDOUT_FILENO | stdout |
| 2 | Standard error | STDERR_FILENO | stderr |

- Every stream (fd) described as a file
  - Unified API (kobject,inode,fops) for handle different stream's types
- Defined in linux/file.h

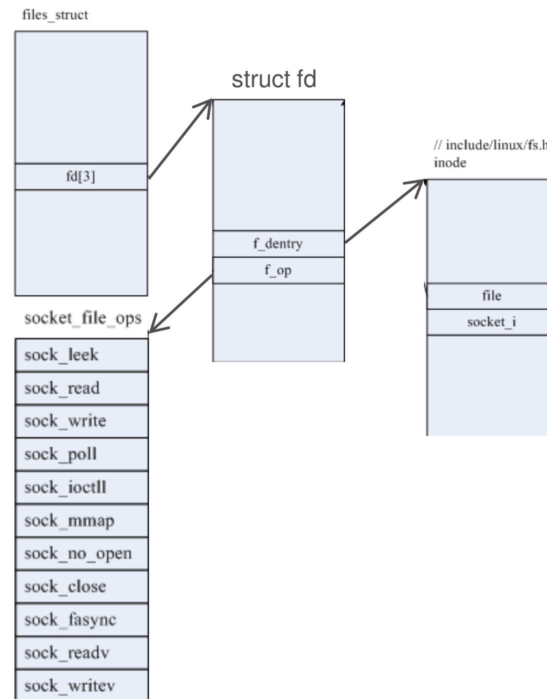
```
struct fd {
   struct file *file;
   unsigned int flags;
};
```



# file (pointed from fd)

- The `file` structure holds all the information about a “file” in linux
- Includes the f_op to handle operations on the file (read,write,etc)
- Holds a pointer to the file’s index node

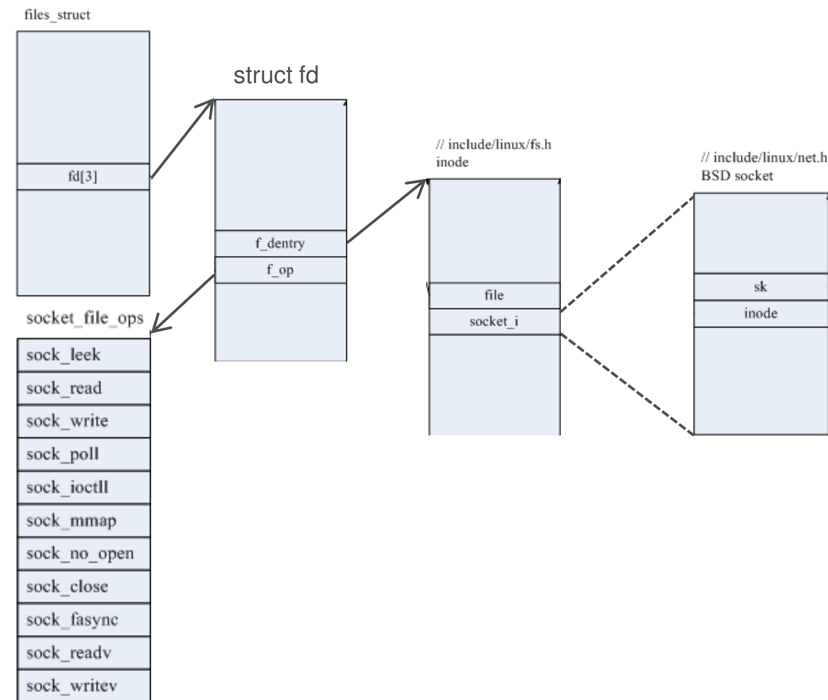
```
struct file {
    …
    const struct file_operations *f_op;
    f_dentry f_path.dentry;
}
```



# Index node - inode

- The `inode` structure used to represent a filesystem object, which can be file, a directory or a socket
- We can use `SOCKET_I` function to get the socket fields from the inode:

```
struct socket
   *SOCKET_I(struct
   inode *inode)
```

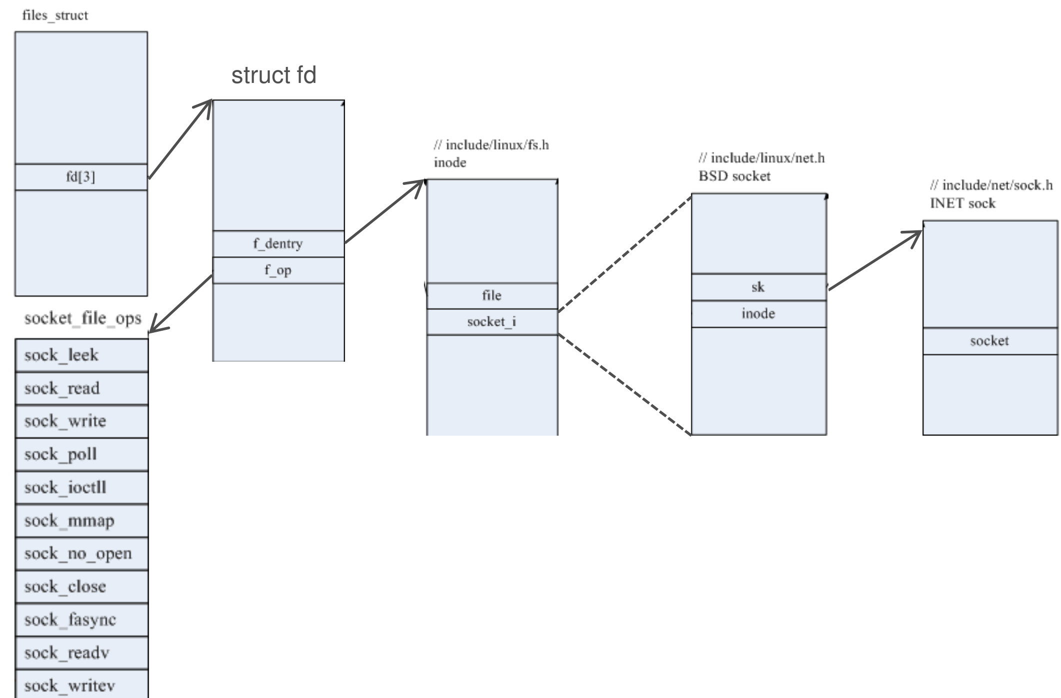


# BSD socket interface and INET sock

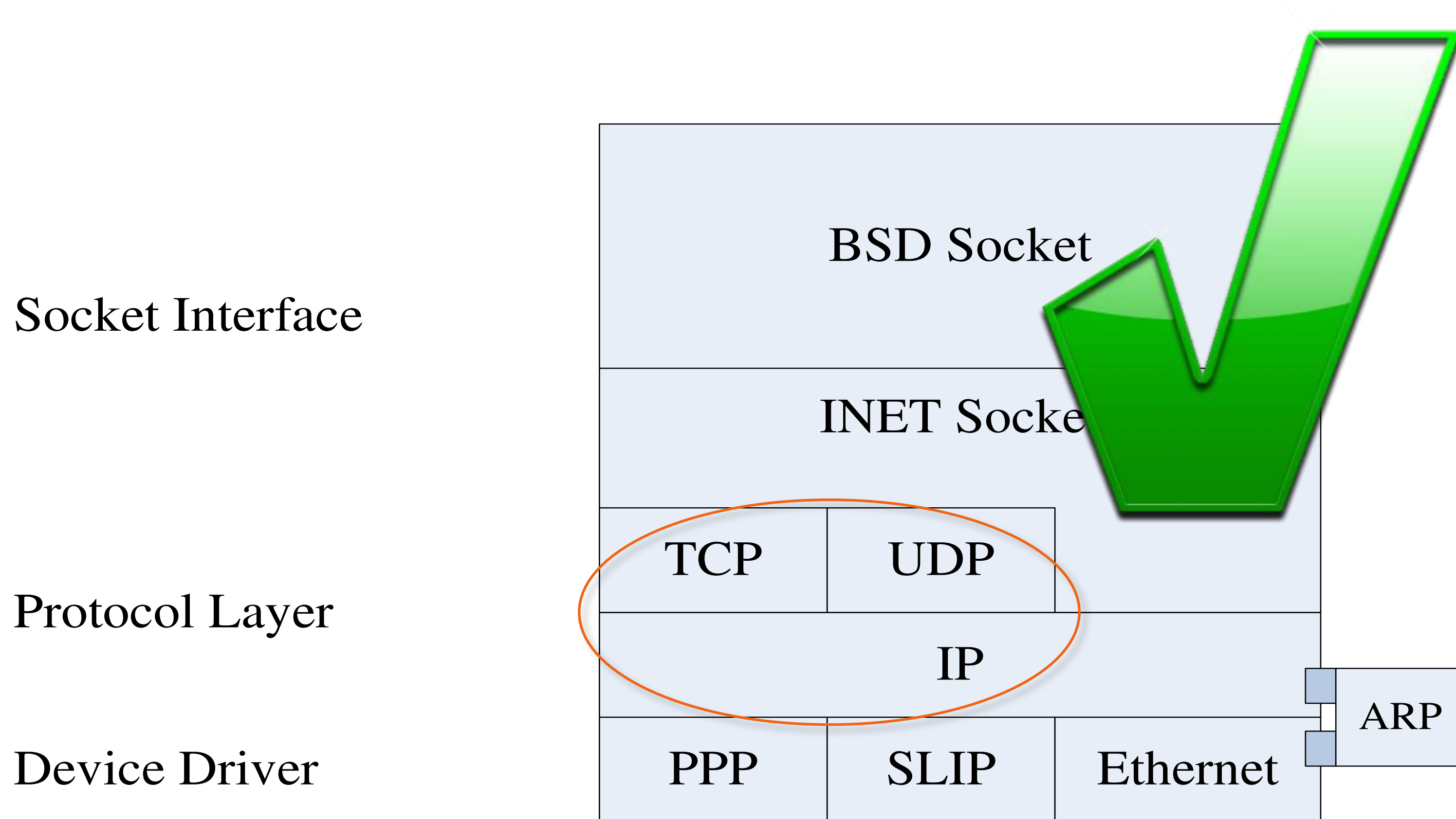
- The `socket` structure holds all the information about the socket
  - connection, type, state, flags, and most important: sock
- `sock` is a huge struct (80-100 variables). Hold the data of the connection, and a pointer the packet structure – sk_buff

```
struct socket {
    ...
    struct file *file;
    struct sock *sk;
    ...
};
```

```
struct sock {
    ...
    struct sk_buff head;
    struct sk_buff next;
    struct sk_buff tail;
    ...
};
```



# Linux TCP/IP Networking



# Agenda With Highlight

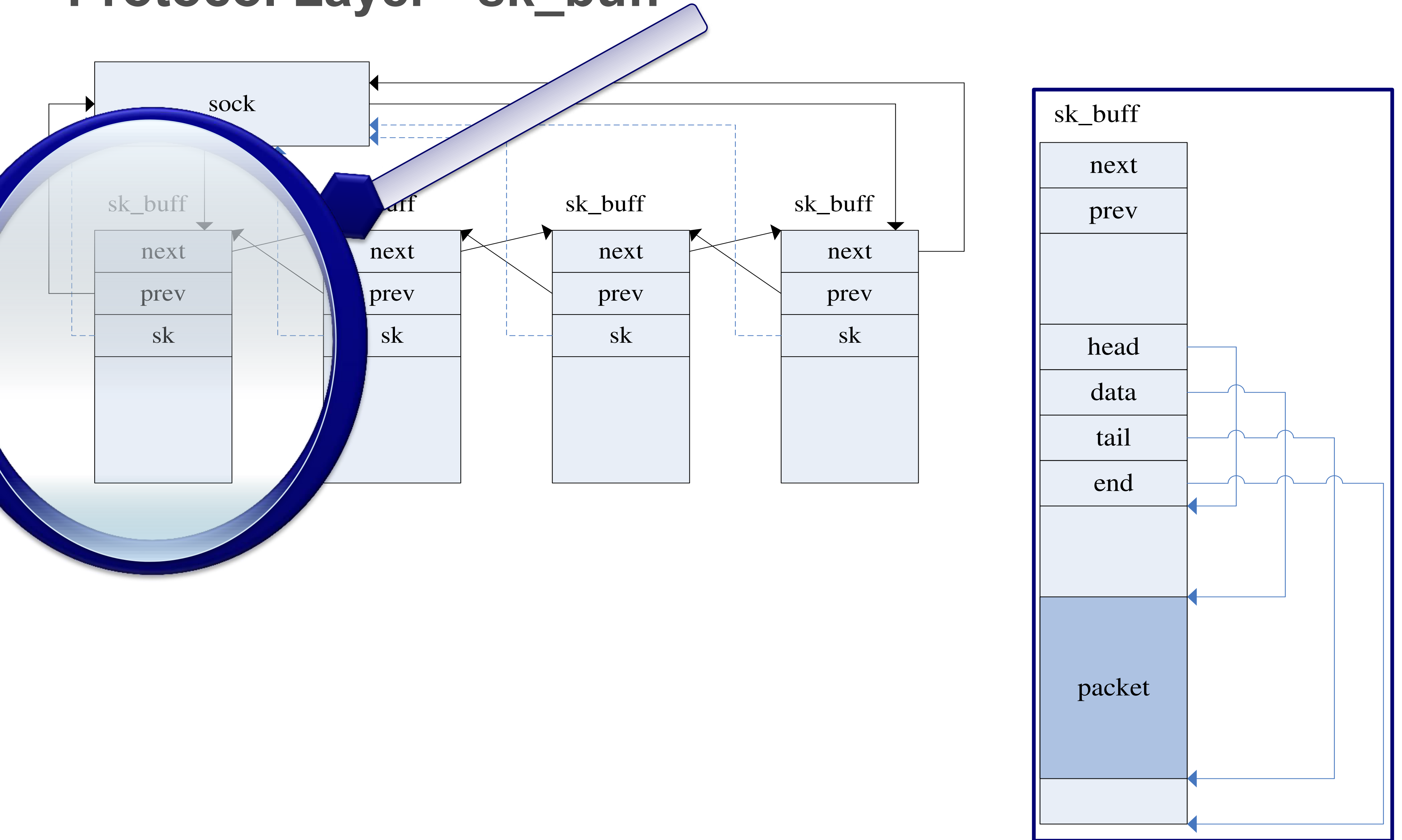
1. Linux file system - networking
2. **sk_buff**
3. Stateless packet filtering
4. About next assignment

# Protocol Layer - sk_buff

- Defined in *include/linux/skbuff.h*
- The structure which de facto contains the packet (and its data)
- We will use it to access the packet data and meta data (layers' headers) to determine the verdict of the packet according to our stateless rules and/or stateful inspect
- The netfilter give us, as an input to its hook functions, a pointer to the current packet's sk_buff structure

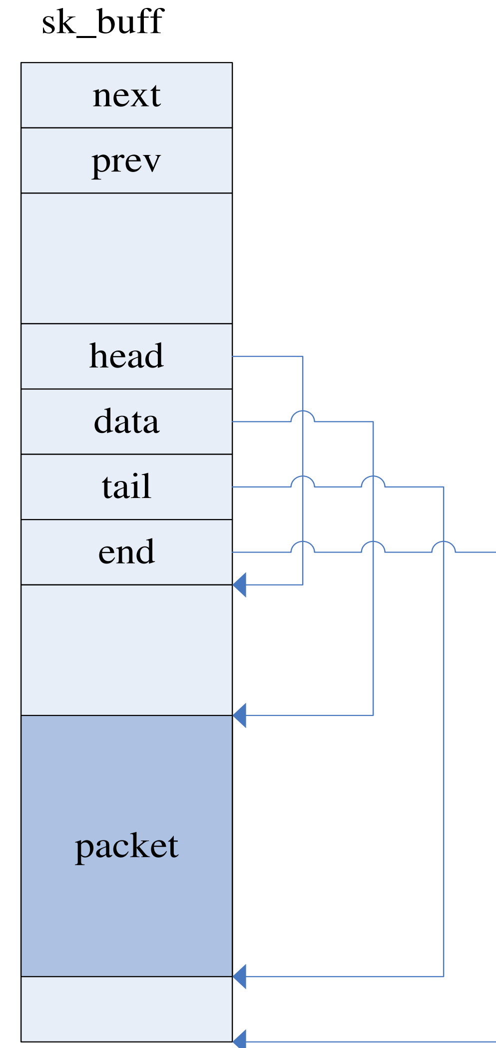
```
hook_func(unsigned int hooknum, struct sk_buff *skb, const struct net_device *in,
          const struct net_device *out, int (*okfn)(struct sk_buff *))
```

# Protocol Layer - sk_buff



# Protocol Layer - sk_buff

```
struct sk_buff {
   /* These two members must be first. */
   struct sk_buff *next;
   struct sk_buff *prev;
   ...
   __u16 transport_header;
   __u16 network_header;
   __u16 mac_header;
   /* These elements must be at the end...
   sk_buff_data_t tail;
   sk_buff_data_t end;
   unsigned char *head, *data;
   ...
};
```



# sk_buff – access the headers

- How can we access directly to the ip/tcp header?
- We can access directly from skb->data
  - For example:

```
protocol = *((__u8 *) (buffer->data + 9));
source_ip = *((unsigned int *)(buffer->data + 12));
dest_ip = *((unsigned int *)(buffer->data + 16));
source_port = ((__be16 *)((unsigned char*) (&(buffer->data[4 *
   ((((unsigned char*)(buffer->data))[0]) & 0b00001111)]))))[0];
```

- We can also use preset functions and Linux structures to get directly to important fields

# sk_buff – access the headers

- For example

```
protocol = iphdr->protocol;
source_ip = iphdr->saddr;
dest_ip = iphdr->daddr;
source_port = tcphdr->source;
```

- Choose your method wisely

```
protocol = *((__u8 *) (buffer->data + 9));
source_ip = *((unsigned int *)(buffer->data + 12));
dest_ip = *((unsigned int *)(buffer->data + 16));
source_port = ((__be16 *)((unsigned char*) (&(buffer->data[4 *
   ((((unsigned char*)(buffer->data))[0]) & 0b00001111)]))))[0];
```

# Access the headers

- So how do we set those headers?

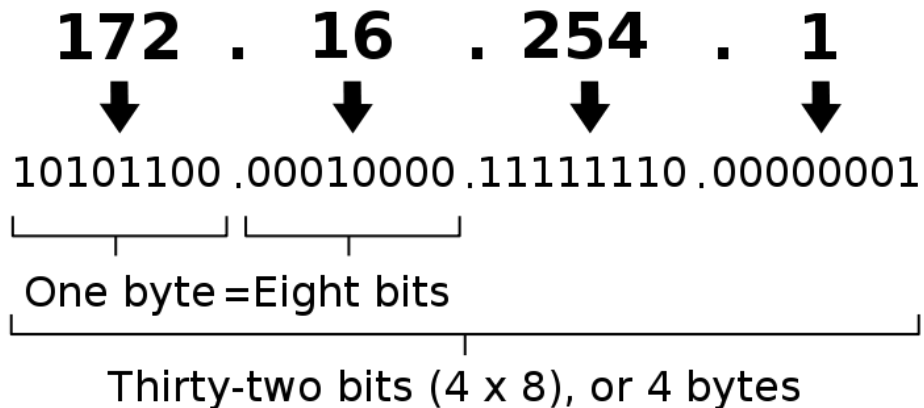
```
//from include/linux/ip.h
static inline struct iphdr *ip_hdr(const struct sk_buff *skb) {
   return (struct iphdr *)skb_network_header(skb);
}
//from include/linux/skbuff.h
static inline unsigned char *skb_network_header(const struct sk_buff *skb) {
   return skb->head + skb->network_header;
}
```

- Same for tcphdr
- Each layer got the same macro with the same variable

```
__u16 transport_header;
__u16 network_header;
__u16 mac_header;
```

# IP address structure

- IP address as we know it: 172.16.254.1
- What does it means?




- When we realize what it is, we can easily convert and compare it to whatever we want.

# Subnet mask

- IP address is logically divided to two parts:
  - Network prefix.
  - Host identifier.
- All host in the same network have the same network prefix

| | Binary form | Dot-decimal notation |
|---|---|---|
| IP address | 11000000.10101000.00000101.10000010 | 192.168.5.130 |
| Subnet mask | 11111111.11111111.11111111.00000000 | 255.255.255.0 |
| Network prefix | 11000000.10101000.00000101.00000000 | 192.168.5.0 |
| Host part | 00000000.00000000.00000000.10000010 | 0.0.0.130 |

# Network prefix length

- Usually instead of writing the IP address and subnet mask we will use shorten version of the two of them, using network prefix length
- For example, instead of: 192.168.5.130 and 255.255.255.0 we can write 192.168.5.130/24
- /24 represent 24 bits of network. As we saw earlier:
  - 255.255.255.0 = 11111111 11111111 11111111 00000000
  - We can just take the first 24 bits of the IP address and see the network part:
    - 192.168.5.0
    - This network can support $2^{(32-network\ prefix\ length)}$ hosts

# Network prefix length translation

| Binary Mask | Prefix Length | Subnet Mask |
|---|---|---|
| 11111111 00000000 00000000 00000000 | /8 | 255.0.0.0 |
| 11111111 10000000 00000000 00000000 | /9 | 255.128.0.0 |
| 11111111 11000000 00000000 00000000 | /10 | 255.192.0.0 |
| 11111111 11100000 00000000 00000000 | /11 | 255.224.0.0 |
| 11111111 11110000 00000000 00000000 | /12 | 255.240.0.0 |
| 11111111 11111000 00000000 00000000 | /13 | 255.248.0.0 |
| 11111111 11111100 00000000 00000000 | /14 | 255.252.0.0 |
| 11111111 11111110 00000000 00000000 | /15 | 255.254.0.0 |
| 11111111 11111111 00000000 00000000 | /16 | 255.255.0.0 |
| 11111111 11111111 10000000 00000000 | /17 | 255.255.128.0 |
| 11111111 11111111 11000000 00000000 | /18 | 255.255.192.0 |
| 11111111 11111111 11100000 00000000 | /19 | 255.255.224.0 |
| 11111111 11111111 11110000 00000000 | /20 | 255.255.240.0 |
| 11111111 11111111 11111000 00000000 | /21 | 255.255.248.0 |
| 11111111 11111111 11111100 00000000 | /22 | 255.255.252.0 |
| 11111111 11111111 11111110 00000000 | /23 | 255.255.254.0 |
| 11111111 11111111 11111111 00000000 | /24 | 255.255.255.0 |
| 11111111 11111111 11111111 10000000 | /25 | 255.255.255.128 |
| 11111111 11111111 11111111 11000000 | /26 | 255.255.255.192 |
| 11111111 11111111 11111111 11100000 | /27 | 255.255.255.224 |
| 11111111 11111111 11111111 11110000 | /28 | 255.255.255.240 |
| 11111111 11111111 11111111 11111000 | /29 | 255.255.255.248 |
| 11111111 11111111 11111111 11111100 | /30 | 255.255.255.252 |
| 11111111 11111111 11111111 11111110 | /31 | 255.255.255.254 |
| 11111111 11111111 11111111 11111111 | /32 | 255.255.255.255 |

# IP address as integer

- In our system, the IP is represented as an integer.
- For example: 185.127.10.42 == 3112110634
- How to calculate?

Host (LSB):

3103784960 + 8323072 + 2560 + 42 = 3112110634

$* 256^3 + * 256^2 + * 256 + * 1$

185.127.10.42

$* 1 + * 256 + * 256^2 + * 256^3$

Network (MSB):

185 + 32512 + 655360 + 704643072 = 705331129

# Endianness

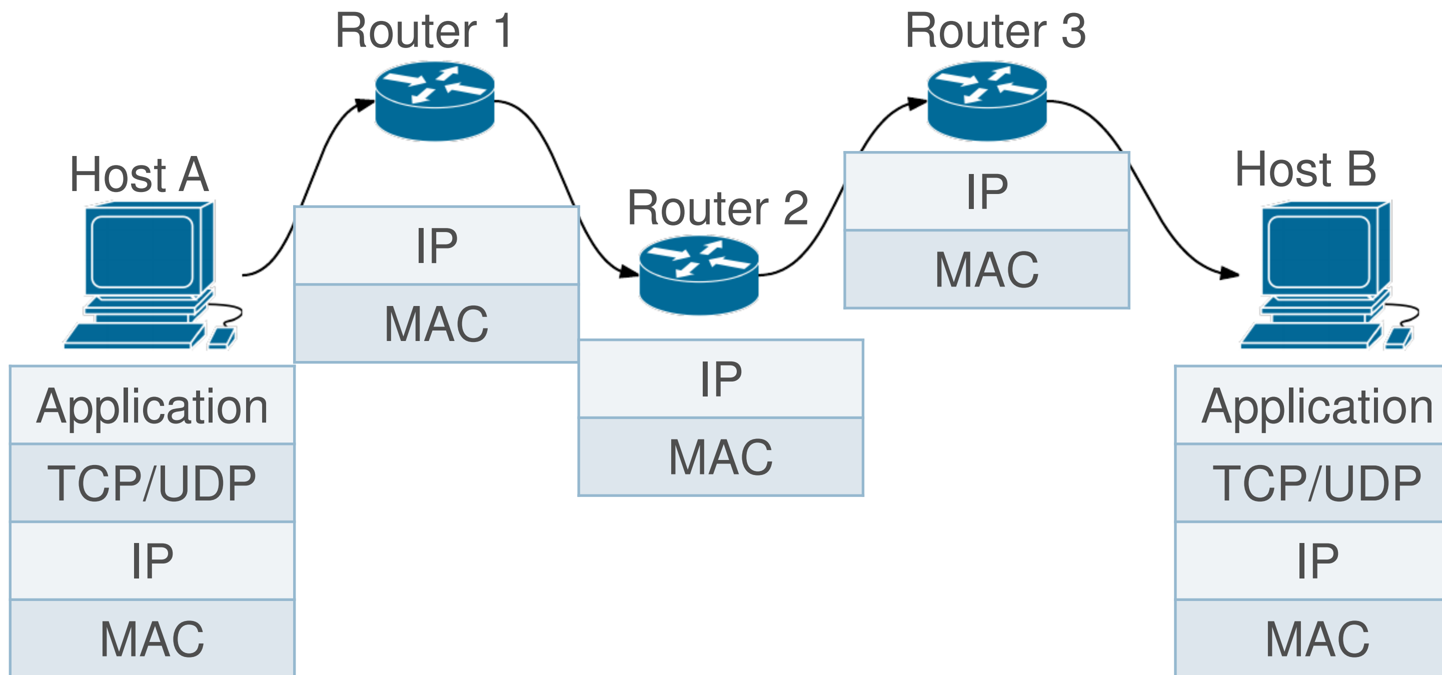
- In different architectures , there's a different order between the LSB (least significant byte) and the MSB (most significant byte)
- In particular, network structures use Big-Endian order, and x86 machines (like our VMs) use Little-Endian order.
- In order to be able to transfer data between the network and the machine, we will use the following functions:
  - `u_long htonl(u_long); // host to network long (32 bits)`
  - `u_short htons(u_short); // host to network short (16 bits)`
  - `u_long ntohl(u_long); // network to host long (32 bits)`
  - `u_short ntohs(u_short); // network to host short (16 bits)`

# Agenda With Highlight

1. Linux file system - networking
2. sk_buff
3. **Stateless packet filtering**
4. About next assignment

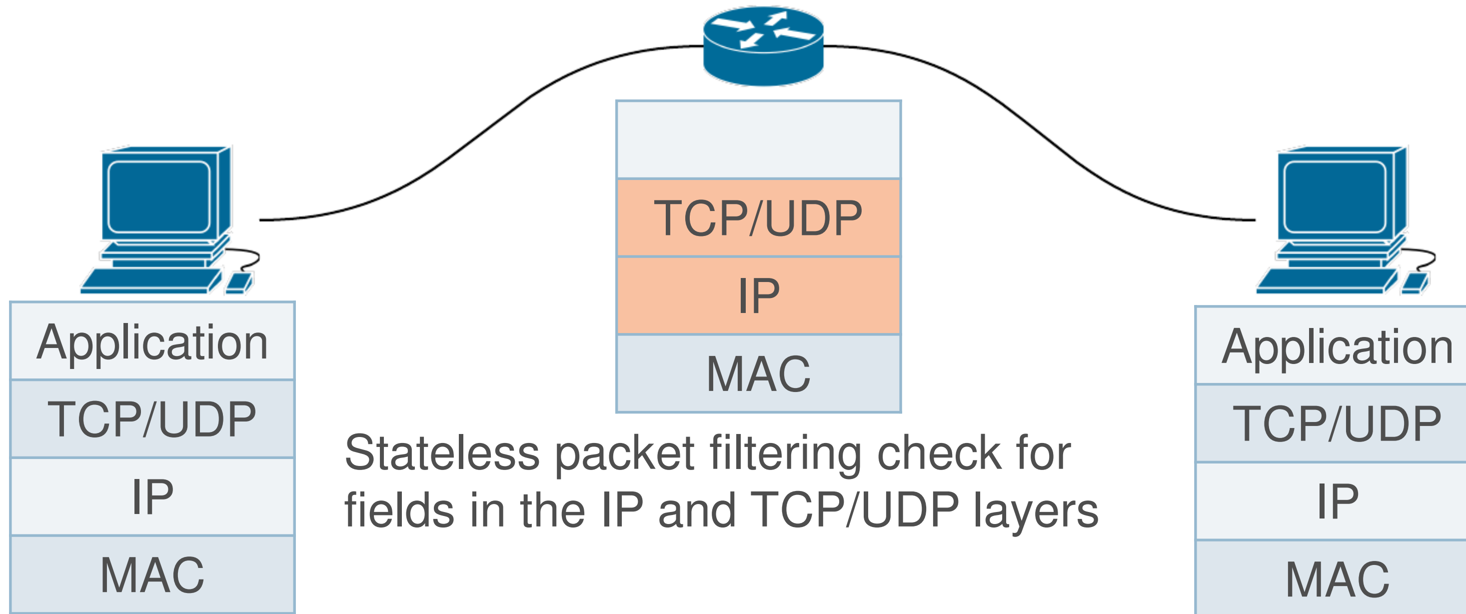
# Stateless packet filtering

- How packets are transferred in the web?



# Stateless packet filtering

- The idea behind stateless packet filtering



# Stateless packet filtering

- Stateless packet filtering scans each packet which go through it
  - Decides if accept it or drop it.
  - The verdict is based on a static rule table.
- The network administrator will write rules in accordance to the company policy.
- Includes the fields:
  - Source and destination address
  - Protocol
  - Source and destination ports
  - Ack bit
  - Direction
  - verdict

IP header

TCP/UDP header

TCP header

# Stateless packet filtering

- Rules for example

| Rule | Diraction | Source Addr | Dest. Addr | Protocol | Source Port | Dest. Port | Ack | Action |
|---|---|---|---|---|---|---|---|---|
| Spoof1 | In | Internal | Any | Any | Any | Any | Any | Drop |
| Spoof2 | Out | External | Any | Any | Any | Any | Any | Drop |
| Telnet1 | Out | Internal | Any | TCP | >1023 | 23 | Any | Accept |
| Telnet2 | In | Any | Internal | TCP | 23 | >1023 | Yes | Accept |
| Default | Any | Any | Any | Any | Any | Any | Any | Drop |

# The rule table

- Stateless packet filtering check each packet against the rule table.
- The packet’s internet and transport layers are being examined
  - Checking rules from top to bottom
  - Each packet is individual
- The packet acts according to the first matching rule (accept or drop)
- A default rule drops/accepts all packets

# The ACK bit

- Only relevant for TCP
- In each TCP session, the first packet is the one who initiate the session and it has the ACK set to 0
  - All the following packets in the session has the Ack bit set to 1
- Therefore, a packet with ACK = 0 is trying to initiate connection.
- A common policy is to drop incoming packets with ACK = 0
  - Prevent outside computers from initiating connections to the private network

# Agenda With Highlight

1. Linux file system - networking
2. sk_buff
3. Stateless packet filtering
4. **About next assignment**

# About assignment 3

- In this assignment you would start to create your true firewall
- The first step would be to think how to enforce rules
  - Make sure you can access the data and have a way to reach the important fields
  - Check the Internet for examples.
- You need to make sure you can work with an interface
  - For the Rules device: create and use Sysfs devices which would handle the rules table.
  - For the Log device: due to large amount of data, Sysfs can’t be used for this device. Therefore you should use fileop struct and implement the relevant functions (.open, .read/write etc.)
- Start the assignment early, it could take some time to get use to handle packets
- We’re here for everything you need