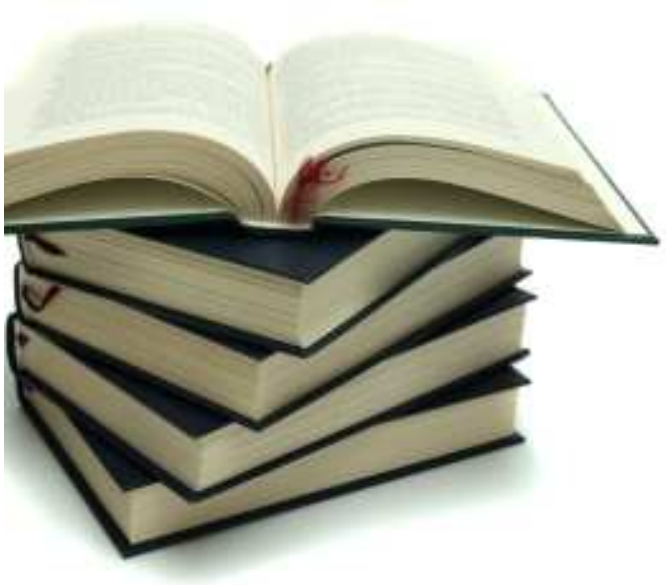
Cursul 11



11

Gestiunea rețelei - Linux

14 mai 2009

Test Curs 10

- Dați un exemplu în care tabelele de pagini nu sunt suficiente pentru a trata un page fault, situație în care sistemul de operare trebuie să acceseze descriptorul spațiului de adresă al procesului.
- Fie a un fișier, b un hard-link către a și c un sof-link către a. Câte inode-uri sunt asociate cu cele trei fișiere?
- Ce operații sunt efectuate la trunchierea unui fișier intr-un sistem de fișiere (asupra inode-ului, dentry-ului, superblock-ului, vectorului ce descrie blocurile libere, etc.)

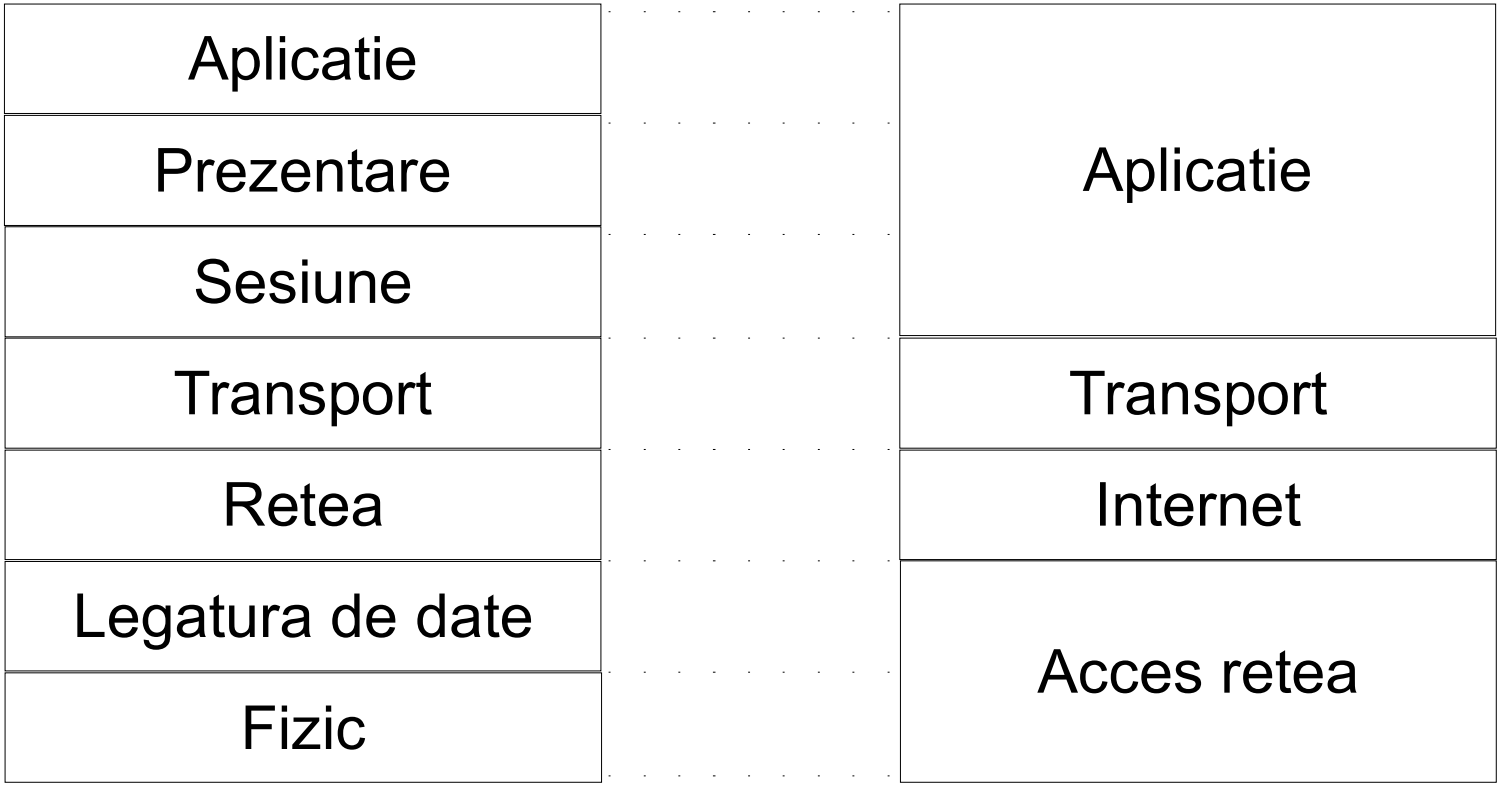
Linux networking

- Implementarea socketilor în Linux
- Rutarea în Linux
- Interfața cu hardware-ul
- Optimizări

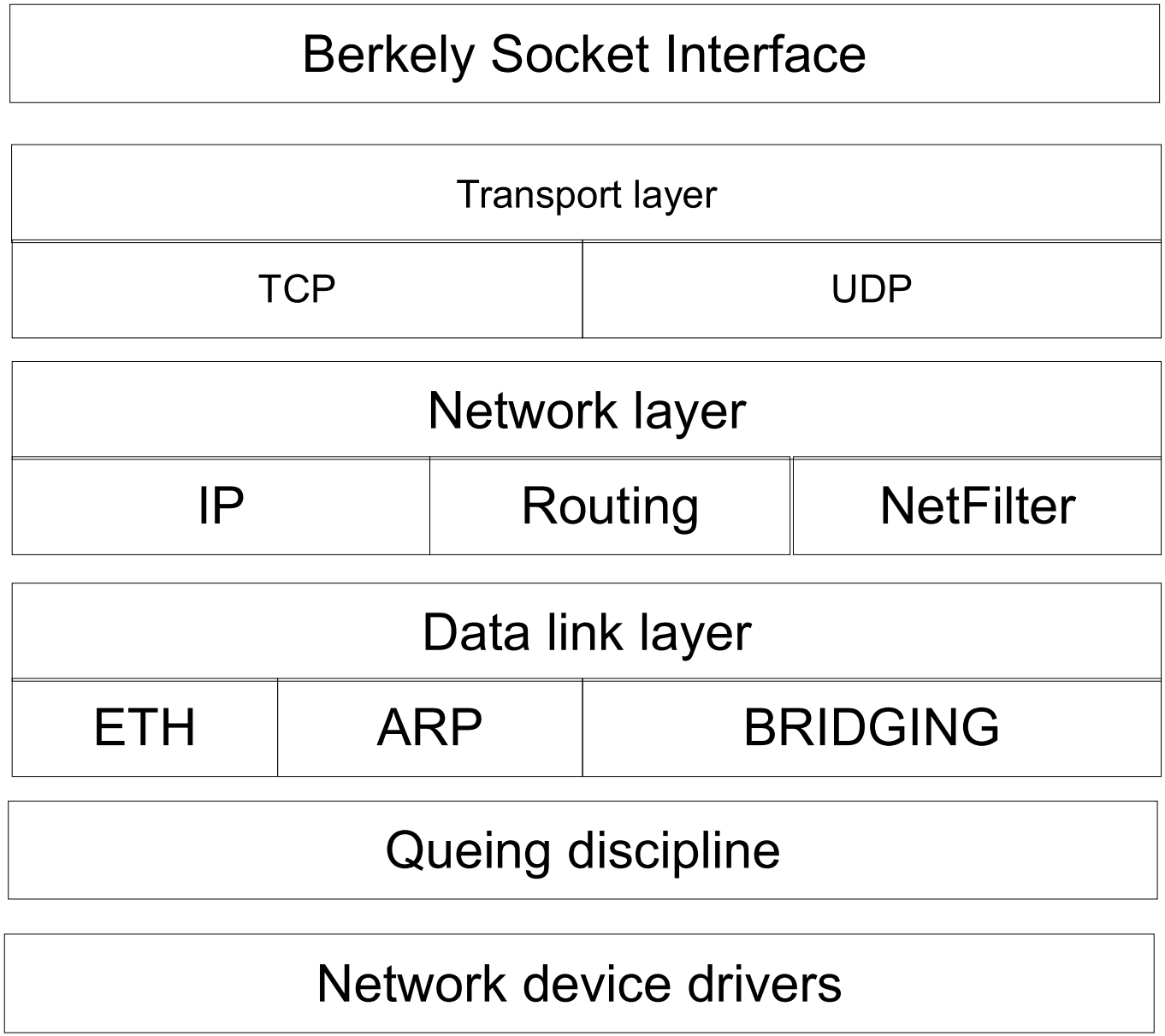
Stive de protocoale

- Retelele sunt controlate prin intermediul unor protocoale, de obicei grupate intr-o stiva de protocoale
- Avantajele unei stive de protocoale: modularitate
  - Reduce complexitate
  - Mai usor de inteles
  - Usor de modificat

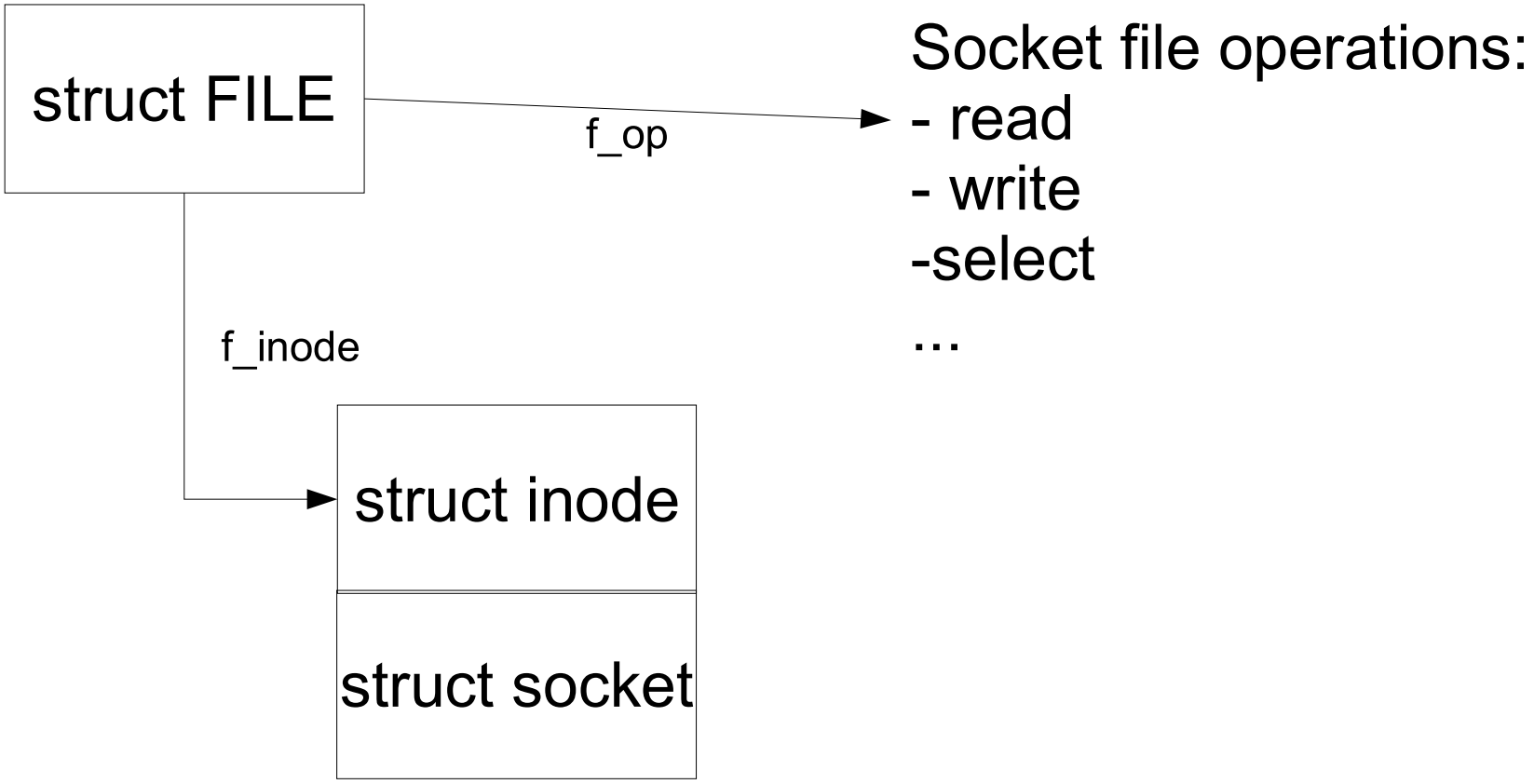
ISO OSI vs TCP/IP



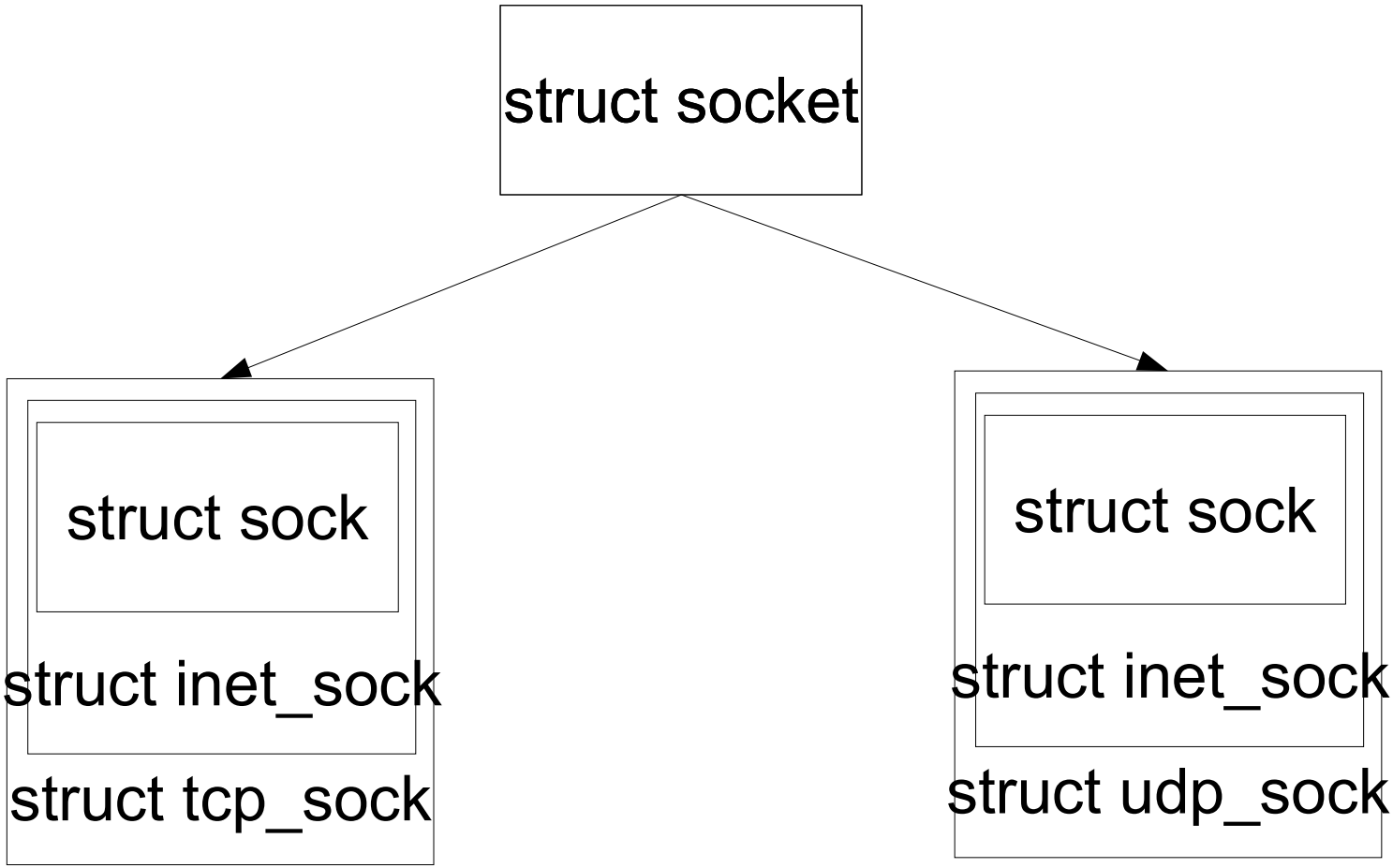
Componente ale stivei de rețea în Linux



Implementarea socketilor în Linux



Implementarea socketilor în Linux (2)



Fluxul pachetelor intr-un ruter

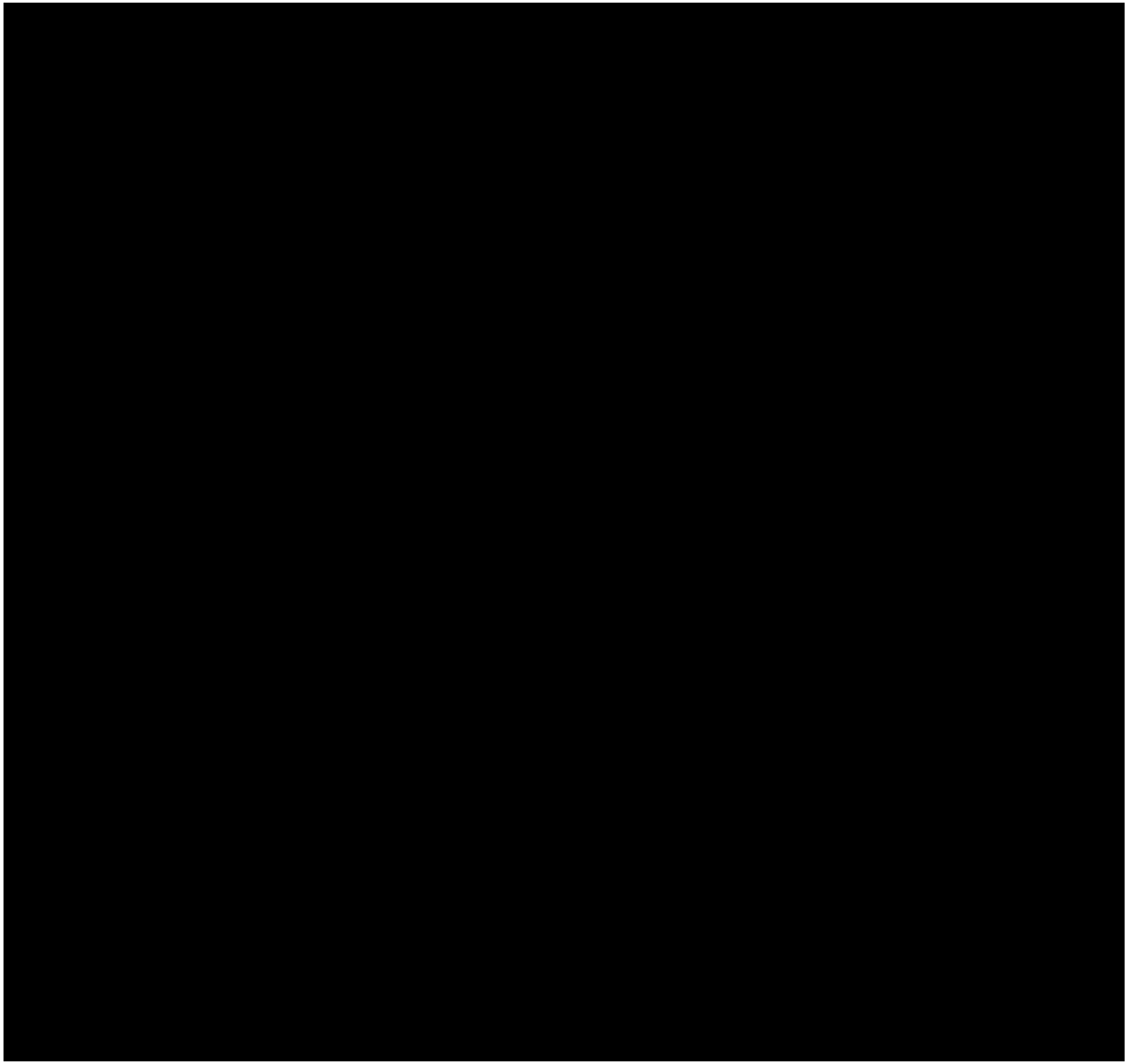


Tabela de rutare

```
[tavi@tropaila tavi]$ /sbin/route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.168.0.0     *               255.255.255.0   U     0      0        0 eth0
172.16.1.0      192.168.58.1    255.255.255.0   UG    0      0        0 vmnet1
172.16.125.0    *               255.255.255.0   U     0      0        0 vmnet8
192.168.58.0    *               255.255.255.0   U     0      0        0 vmnet1
127.0.0.0       *               255.0.0.0       U     0      0        0 lo
```

- Contine informatii necesare rutarii unui pachet pentru o retea cunoscuta
  - Adresa retelei
  - Masca retelei
  - Interfata sau next hop-ul pe unde trebuie trimis pachetul
  - Metrica

Operatii necesare rutarii

- Pentru pachetele primite se verifica daca adresa destinatie coincide cu vreo adresa locala
  - Daca da, se trimite pachetul local
- Daca nu, se ruteaza pachetul
  - Se determina
    - Interfata pe care trebuie trimis pachetul
    - Adresa de nivel 2 a next hop-ului
    - Adresa de nivel 3 a next hop-ului
  - daca este necesar se fragmenteaza / defragmenteaza pachetul
  - se suprascrie header-ul de nivel 2 si se transmite pachetul de interfata determinata

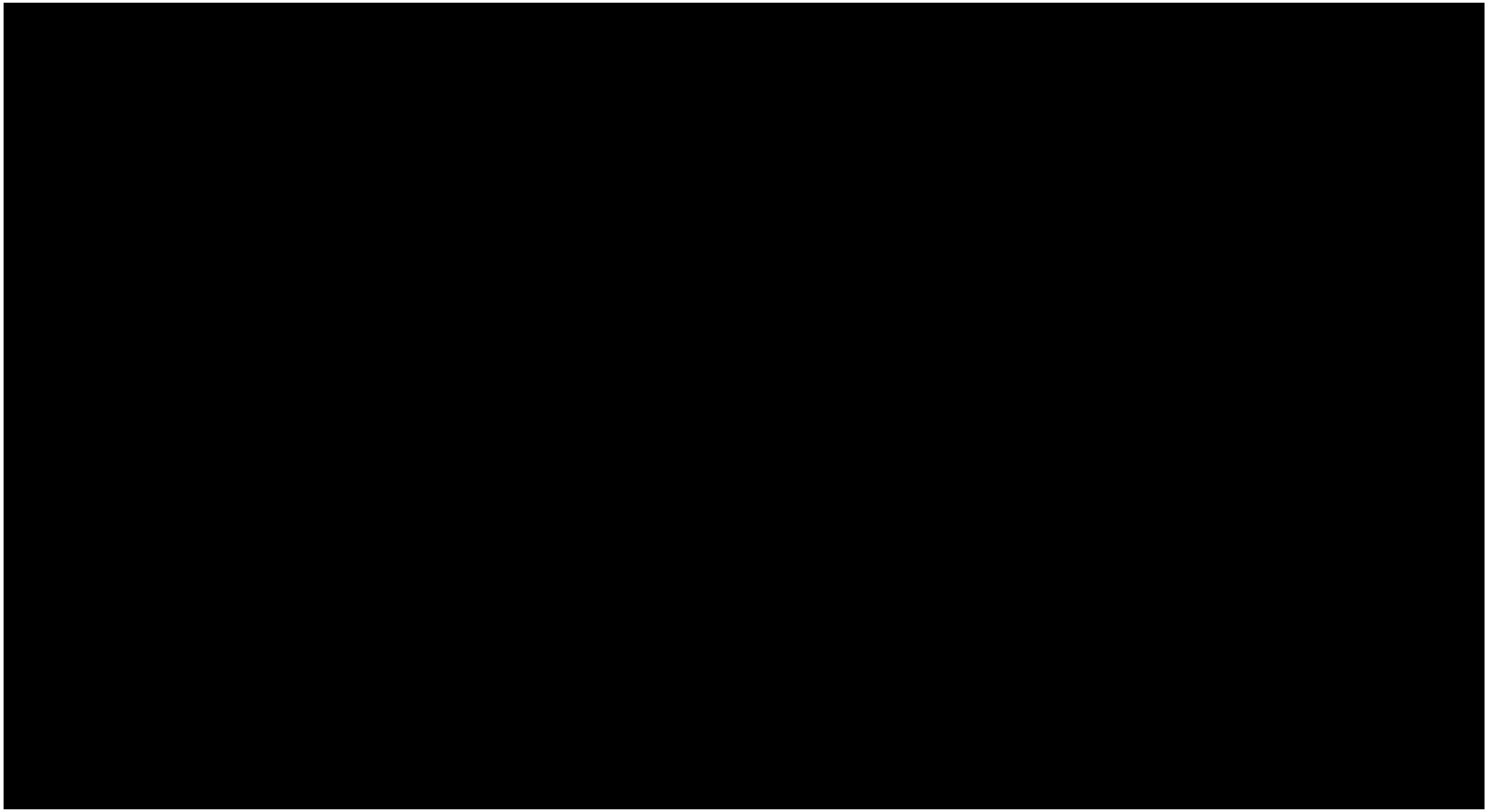
Implementarea rutarii in Linux

- Procesul de rutare este initiat din bottom-half handler-ul asociat gestiunii retelei
- Pentru determinarea informatiilor de rutare se consulta in ordine
  - Cache-ul de rute
  - Forwarding Information Database (FIDB)

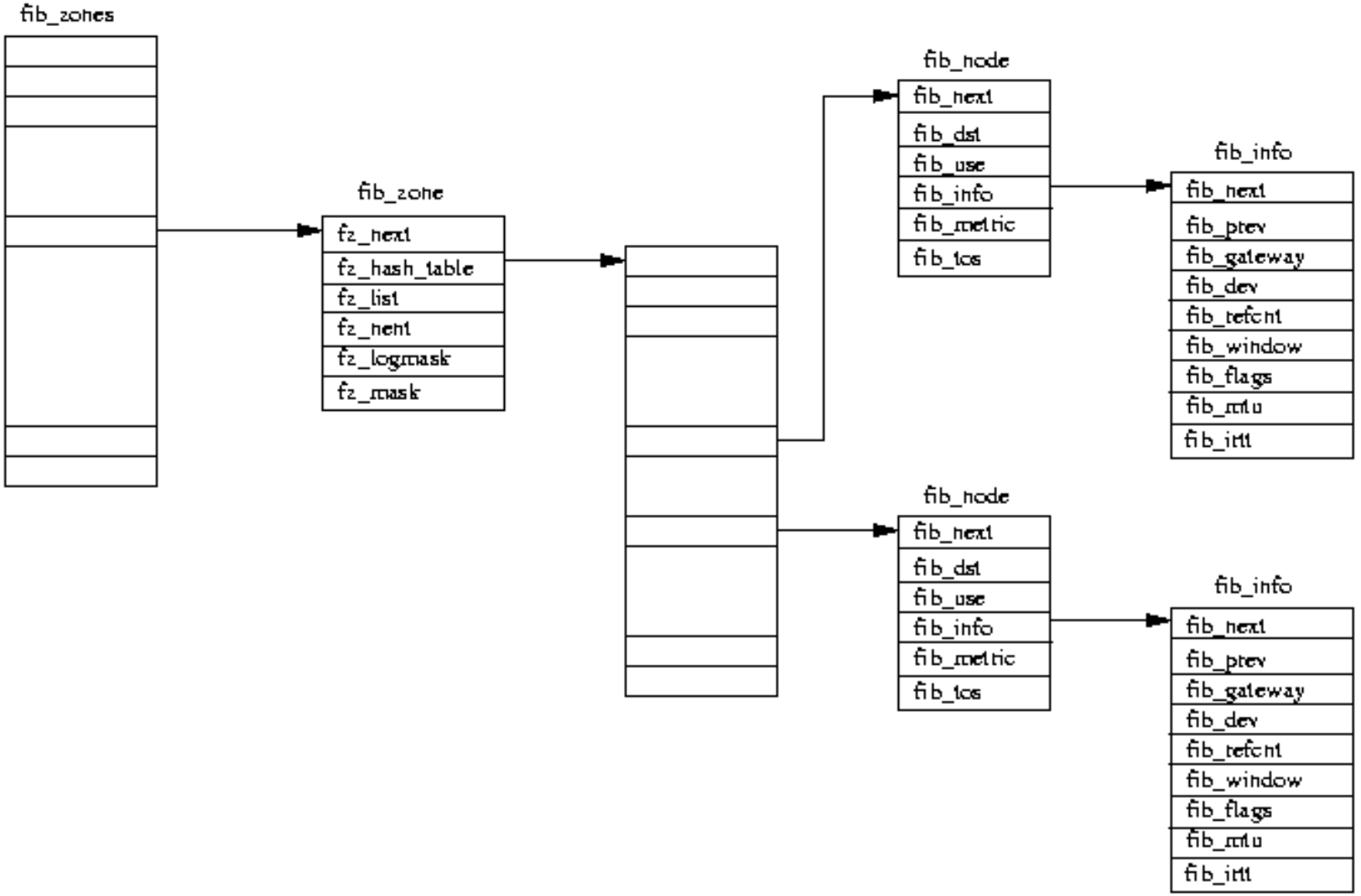
Forwarding Information Database

- Structura de date ce contine informatii necesare rutarii
- O tabela cu 32 de intrari, cate una pentru fiecare masca de retea posibila
- O intrare in tabela contine un pointer catre o lista de structuri ce descriu rute cu aceasi masca de retea

Forwarding Information Database (2)



Forwarding Information Database (3)



Rutarea prin FIB

- Se consulta FIB incepand cu zona asociata celei mai specifice masti (/32)
- Se inspecteaza hash table-ul fiecarei zone
  - Pentru a mari viteza de cautare se folosesc contoare pentru a indica cat de des a fost folosita o ruta; in cadrul unei liste din hash table nodurile se tin ordonate descrescator dupa acest contor
- Daca nu se face match in hash table-ul unei zone, se trece la urmatoarea
- Daca nu se face match pe nici o ruta se face drop pe pachet

Cache-ul de rute

- Rutarea fiecarui pachet prin FIB este extrem de lenta atunci cand tabela de rutare nu este foarte mica
- Pentru a mari viteza se foloseste un cache de rute
  - Un hash table cu 256 de intrari
  - Fiecare intrare contine o lista cu noduri ce descriu
    - un triplet <adresa sursa, adresa destinatie, TOS>
    - informatii necesare de rutere al acestui tip de pachet: adresele de nivel 2 si 3, interfata de iesire
  - Nodurile sunt ordonate descrescator dupa un contor ce specifica cat de des a fost folosit acesta

Cache-ul de rute (2)

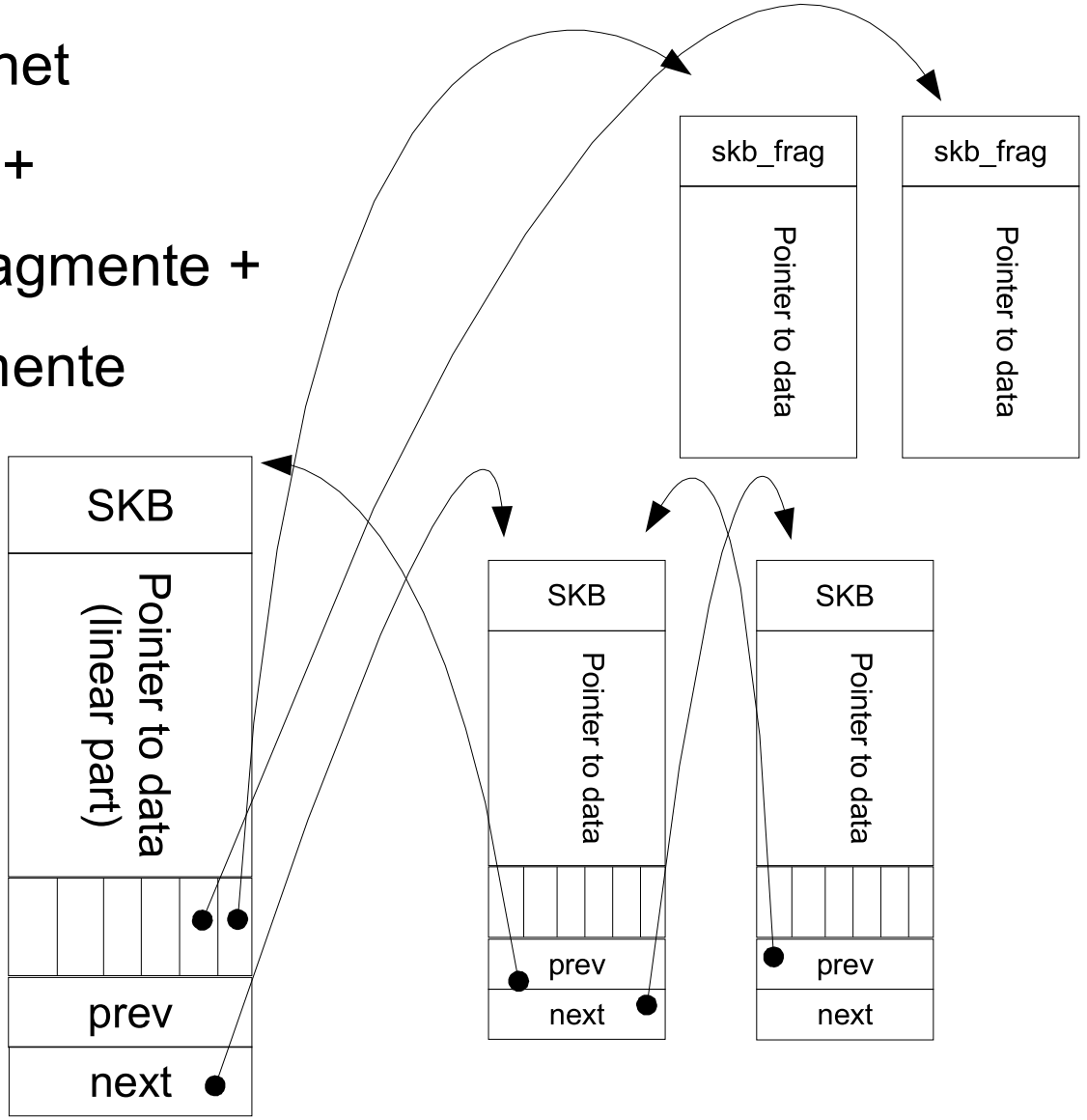
Rutarea prin cache-ul de rute

- Se calculeaza indexul pentru hash table
  - O combinatie intre adresa sursa, adresa destinatie si TOS
- Se parcurge lista asociata indexului
- Daca nu se gaseste nici o potrivire se ruteaza prin FIB
  - Dupa determinarea informatiilor necesare rutarii se introduce o intrare in cache-ul de rute
- Daca se gaseste o potrivire se actualizeaza contorul nodului, si se reordoneaza lista
- Intrarile din cache-ul de rute sunt invalidate periodic

Netfilter

- Infrastructura folosită în Linux pentru implementarea facilităților de firewall & NAT
- Bazat pe hook-uri:
  - NF_IP_PRE_ROUTING
  - NF_IP_LOCAL_IN
  - NF_IP_FORWARD
  - NF_IP_LOCAL_OUT
  - NF_IP_POST_ROUTING
  - NF_IP_NUMHOOKS

- Descrie un pachet
- O parte lineară +
- Un vector de fragmente +
- O listă de fragmente



Skb (2)

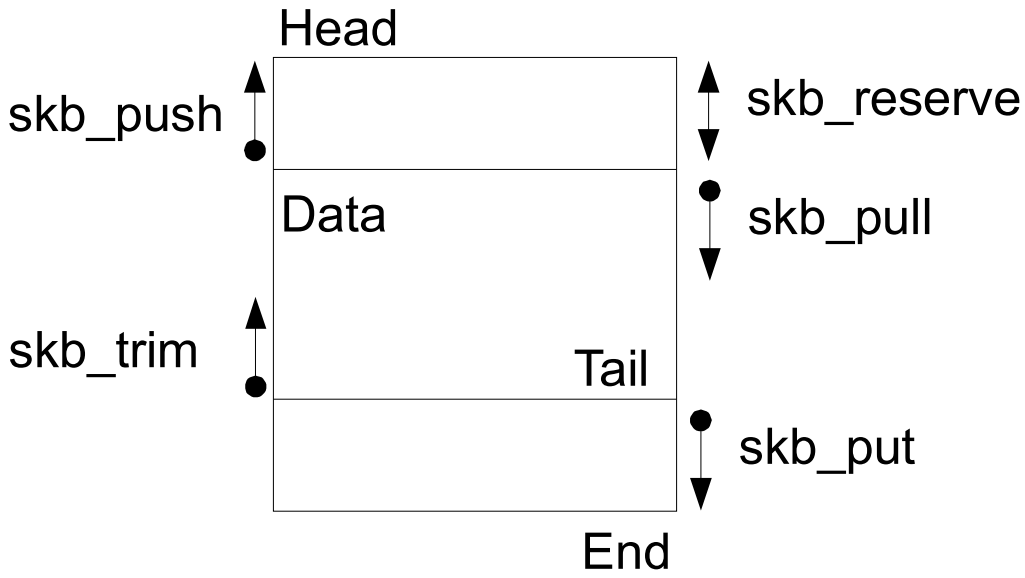
```
struct sk_buff {
        struct sk_buff          *next;
        struct sk_buff          *prev;

        struct sock             *sk;
        ktime_t                 tstamp;
        struct net_device       *dev;
        char                    cb[48];

        unsigned int            len,
                                data_len;
        __u16                   mac_len,
                                hdr_len;

        void                    (*destructor)(struct sk_buff *skb);

        sk_buff_data_t          transport_header;
        sk_buff_data_t          network_header;
        sk_buff_data_t          mac_header;
        sk_buff_data_t          tail;
        sk_buff_data_t          end;

        unsigned char           *head,
                                *data;
        unsigned int            truesize;
        atomic_t                users;
};
```

Operații comune

```
unsigned char *skb_transport_header(const struct sk_buff *skb);

void skb_reset_transport_header(struct sk_buff *skb);

void skb_set_transport_header(struct sk_buff *skb, const int offset);

unsigned char *skb_network_header(const struct sk_buff *skb);

void skb_reset_network_header(struct sk_buff *skb);

void skb_set_network_header(struct sk_buff *skb, const int offset);

unsigned char *skb_mac_header(const struct sk_buff *skb);

int skb_mac_header_was_set(const struct sk_buff *skb);

void skb_reset_mac_header(struct sk_buff *skb);

void skb_set_mac_header(struct sk_buff *skb, const int offset);
```

Operații comune (2)

```
/* reserve head room */
void skb_reserve(struct sk_buff *skb, int len);
/* add data to the end */
unsigned char *skb_put(struct sk_buff *skb, unsigned int len);
/* add data to the top */
unsigned char *skb_push(struct sk_buff *skb, unsigned int len);
/* discard data at the top */
unsigned char *skb_pull(struct sk_buff *skb, unsigned int len);
/* discard data at the end */
unsigned char *skb_trim(struct sk_buff *skb, unsigned int len);
```



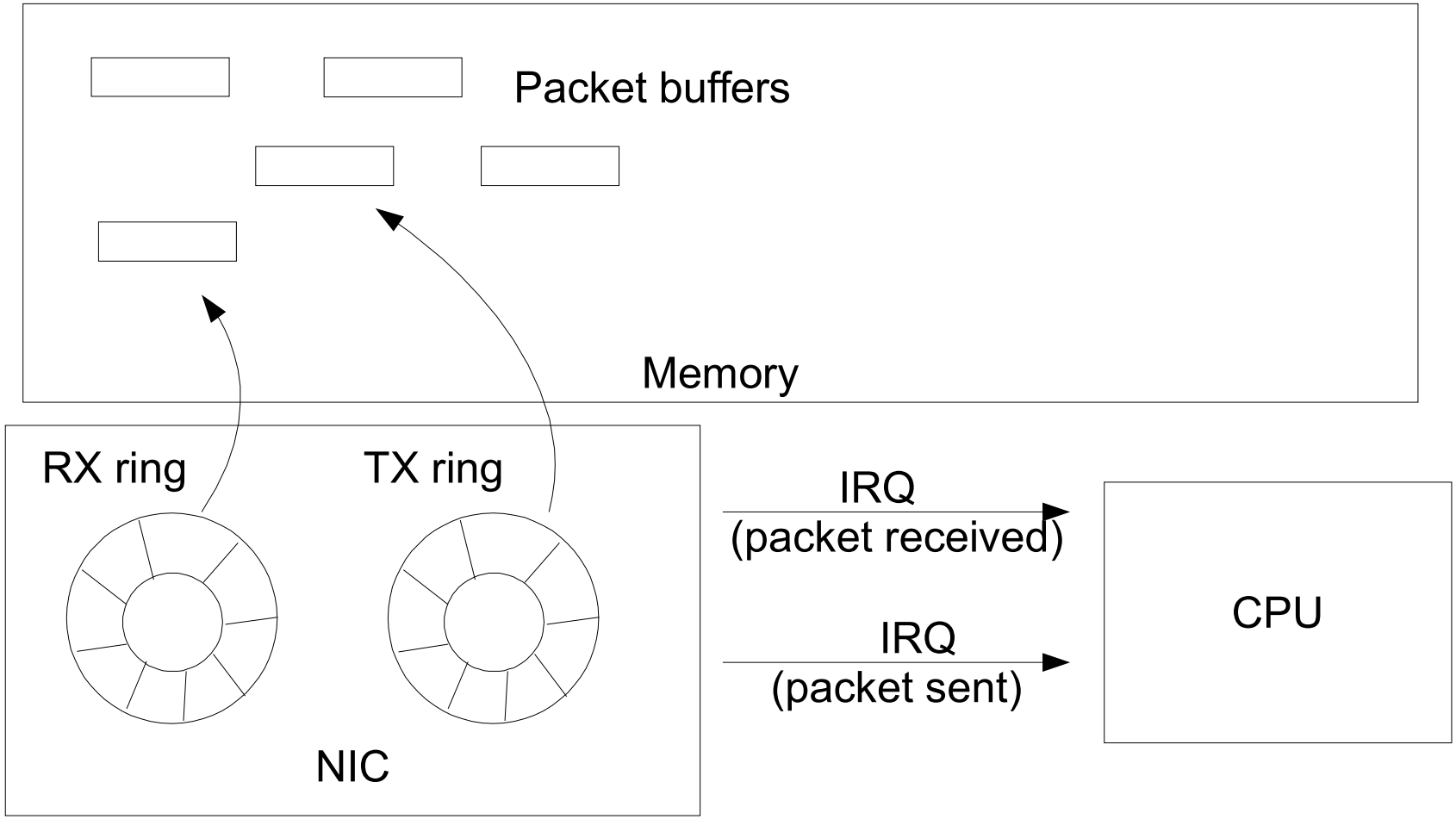
Network device drivers

- Operații
  - `int hard_start_xmit(struct sk_buff *skb, struct net_device *dev);`
  - `int open(struct net_device *dev);`
  - `int stop(struct net_device *dev);`
  - `int do_ioctl(struct net_device *dev, struc tifreq *ifr, int cmd);`
  - `int change_mtu(struct net_device *dev, int new_mtu);`
  - `int create(struct sk_buff *skb, struct net_device *dev, unsigned short type, const void *daddr, const void *saddr, unsigned len);`
  - `int rebuild(struct sk_buff *skb);`
- **Recepția pacheteleor se face din intrerupere**
  - `int netif_rx(struct sk_buff *skb);`
  - `int netif_receive_skb(struct sk_buff skb); /* NAPI */`
- Flags
  - **IFF_UP, IFF_BRODCAST, IFF_RUNNING, etc.**
- Features
  - **NETIF_F_SG, NETIF_F_CSUM, NETIF_F_GSO, NETIF_F_TSO, etc.**

Înregistrarea unui network device driver

```
dev_ixianet = alloc_netdev(sizeof(struct net_device_stats),
                           "ixint0",
                           ixnam_llm_probe);

if(dev_ixianet)
{
    SET_MODULE_OWNER(dev_ixianet);

    ret = register_netdev(dev_ixianet);
}
else
{
    ret = -ENOMEM;
}
```

Intefața hardware



Interfața hardware (2)

- RX ring – umplut de SO cu buffere pentru pachetele recepționate
- TX ring – actualizat de SO pentru fiecare pachet de trimis
- Transferul pachetelor din/în memorie se face de către placa de rețea
- Procesorul este întrerupt doar cand
  - S-a recepționat un pachet
  - Place de rețea confirmă trimiterea unui pachet

Procesare pachetelor

- Se face în context softirq
- În context IRQ
  - La RX: se face doar extragerea din ringul RX si plasarea pachetului în coada de procesare a socket-ului
  - La TX: se face (eventual) doar eliberarea memoriei pentru un pachet trimis

Interrupt coalescing

- Se aşteaptă un număr de pachete sau un timeout înainte de a trimite întreruperea
- Se reduce costul procesării pe RX atunci când pachetele vin în burst
- Creşte latența şi jitter-ul

NAPI

- Acelaşi scop ca şi interrupt coalescing, dar fără a influența jitter-ul sau latența
- Nu necesită suport hardware
- În momentul generării unei întreruperi
  - Se dezactivează întreruperea plăcii de rețea
  - Se procesează toate pachetele
  - Abia apoi se activează întreruperea la loc

Scatter-gather

- Un pachet poate fi format din zone ne-contigue de memorie
- Descriptorii de TX/RX mai au un bit: end-of-packet
- Necesar pentru zero-copying

Checksum offloading

- Checksum-ul pe pachetele TCP/UDP se efectuează de către placa de rețea
- Place de rețea Primeşte pseudo-header checksum-ul şi poziția checksum-ului în hedear-ul TCP
- Necesar pentru zero-copying

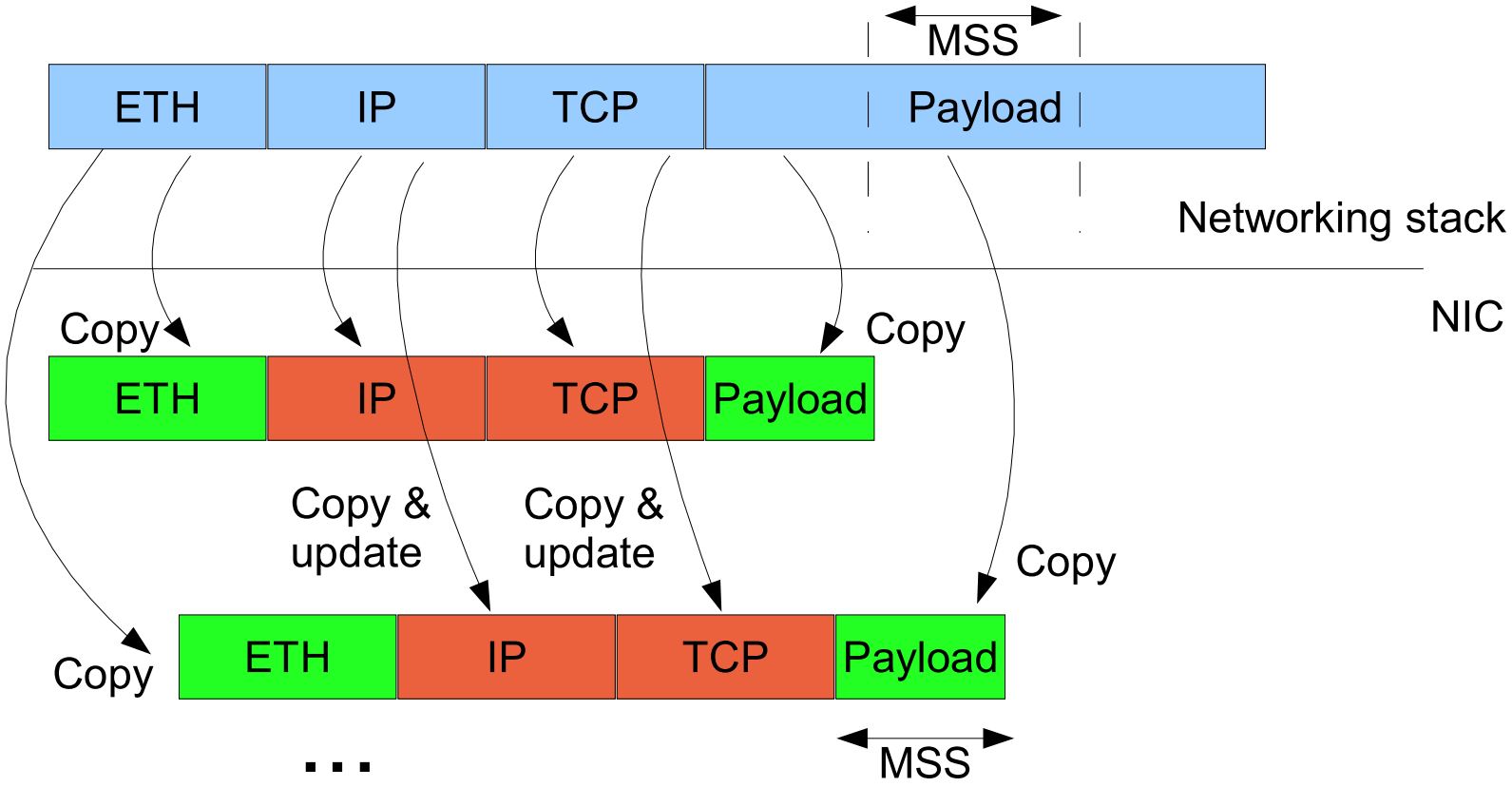
TCP offload

- Full offload - Implementarea stivei TCP/IP direct în placa de rețea
- Probleme:
  - Scalabilitate
  - Securitate
  - Conformanță
- Performanța stivei de rețea este direct proporționala cu numarul de pachete procesate pe secundă (doar headerele sunt procesate)

TCP offload (2)

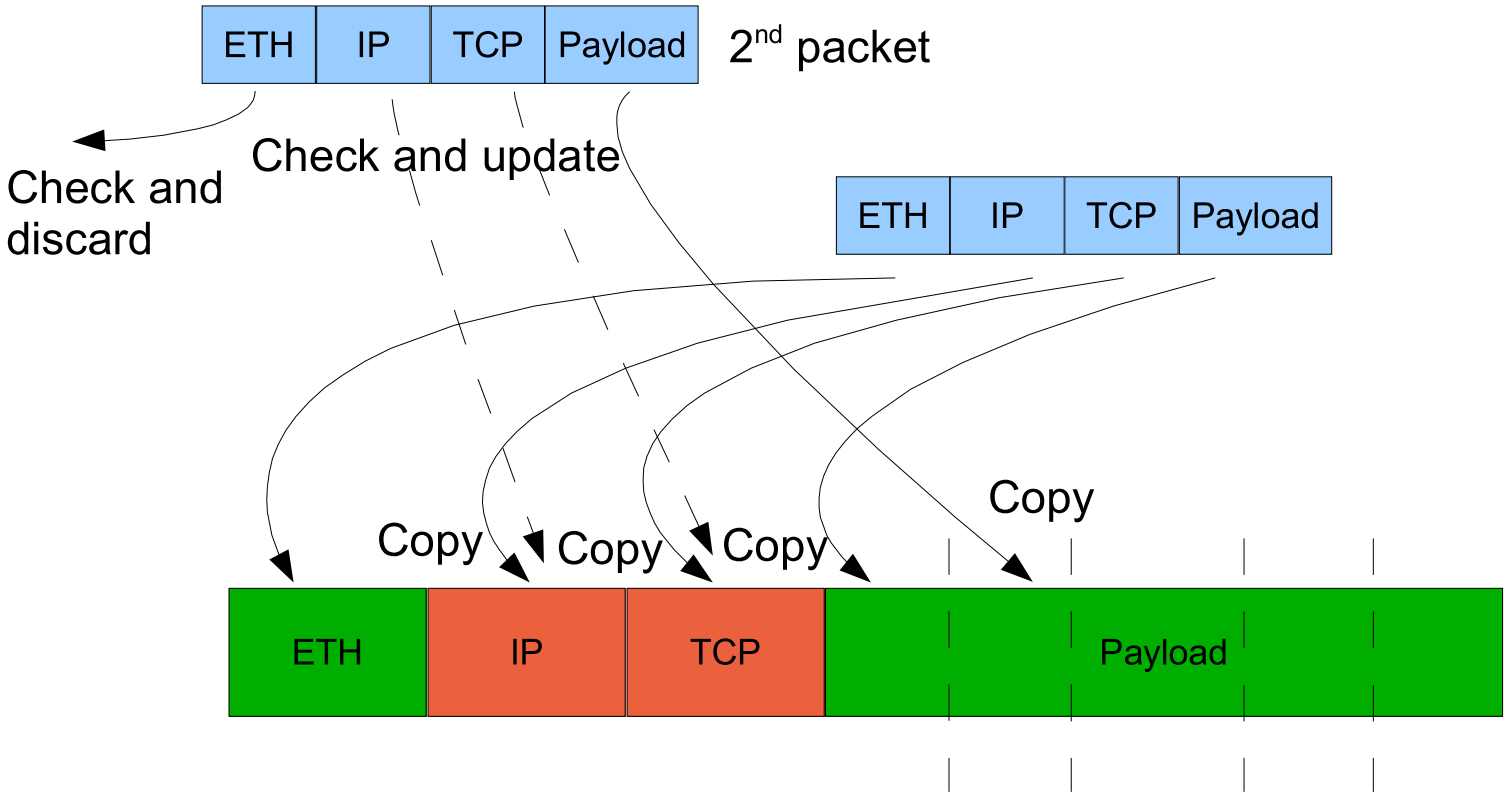
- Exemplu: daca un end-point poate procesa 60K pachete pe secundă, atunci pentru următoarele dimensiuni de pachete obținem:
  - 1538 -> 738Mbps
  - 2038 -> 978Mbps
  - 9038 -> 4.3Gbps
  - 20738 -> 9.9Gbps
- Stateless offload
  - Stiva folosește pachete mari
  - TSO (Transmit/TCP Segmentation Offload): hardware-ul primește pachete mari și le sparge în pachete mai mici pe care apoi le trimite pe fir
  - LRO (Large Receive Offload): hardware-ul agreghează pachetele mici în pachete mari pe care le trimite apoi la stivă

Transmit Segmentation offload



- În hardware – de către placa de rețea
- În software – înainte de trimiterea către device driver (GSO – generic segmentation offload)

Large receive offload



- Dacă se recepționează pachete ale aceluaşi segment în ordine ele se vor agrega şi către stiva se va trimite un pachet mare (până la 64KB)
  - Agregarea se poate face în hardware sau software
  - Pentru a reduce latența, în momentul în care se iesi din bulca NAPI se face flush

splice

- `long splice(int fd_in, loff_t *off_in, int fd_out, loff_t *off_out, size_t len, unsigned int flags);`
- Se mută date de la fd_in la fd_out, fără a se face copierea dacă este posibil
- Unul din file descriptori trebuie să fie neapărat un pipe
- Offset-ul pentru pipe trebuie să fie NULL
- Linux 2.6.25 oferă support complet pentru fişiere, socketi şi memorie (via vmsplice)

Intel I/OAT

- Controller pentru copierea din memorie în memorie via DMA
- Folosit la copierea din/în user-space a datelor socketilor

Intrebări

?