

CS 378 (Spring 2003)

Linux Kernel Programming

Yongguang Zhang

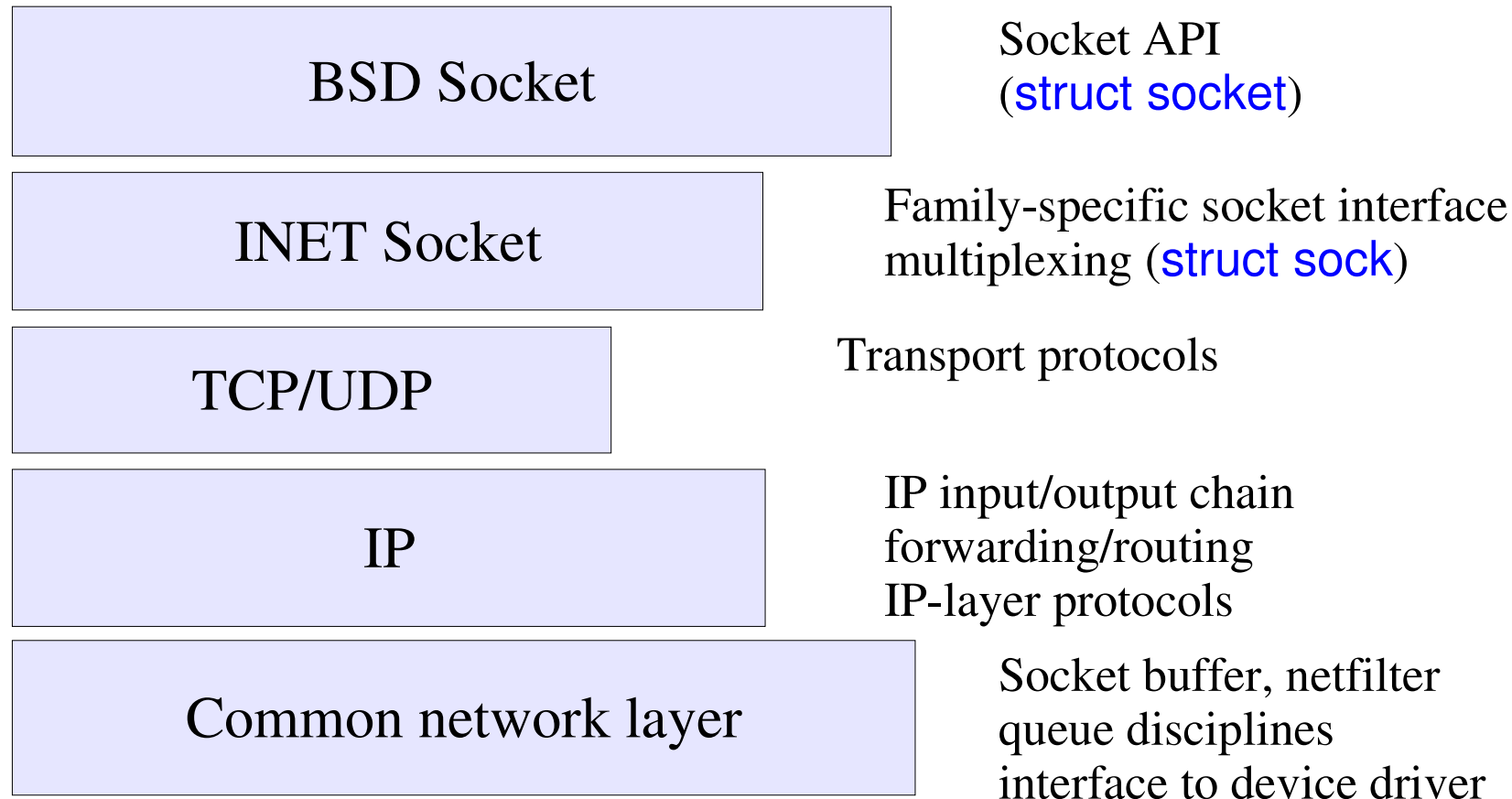
(ygz@cs.utexas.edu)

This Lecture

- Linux Networking
 - Data structures pertaining to socket interface
 - Managing the network families and protocols
 - Socket and file operations
 - Socket buffer `sk_buff` operations
 - Tracing the output chain

- Questions?

Layering in the Networking Stack



Family, Type, and Protocol

- Socket has many “family”
 - PF_INET, ...
- And many types
 - SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- Each family can have many protocols:
 - PF_INET: IPPROTO_TCP, IPPROTO_UDP, ...
- Two type of socket data structures
 - struct socket: for geneirc BSD socket API
 - struct sock: for family/protocol specific information

struct socket

- Kernel data structure for a open socket (for BSD Socket API)
 - Data type: struct socket (in include/linux/net.h)
- Important fields:
 - socket_state state: state of the socket (connected, unconnected, disconnecting, or connecting?)
 - struct proto_ops *ops: socket API methods
 - struct inode *inode, struct file *file: sockfs objects
 - struct sock *sk: network-family-specific socket object
 - wait_queue_head_t wait: for blocking send/receive
 - short type: type of socket (stream, dgram, or raw?)

struct proto_ops

- Methods for BSD socket API
 - Fields:
 - int family: network family (PF_INET, ...)
 - int (*release)(struct socket *sock)
 - int (*bind)(struct socket *sock, ...)
 - int (*connect)(struct socket *sock, ...)
 - int (*accept)(struct socket *sock, ...)
 - ...
- Two sets of methods defined for INET
 - See net/ipv4/af_inet.c:
 - struct proto_ops inet_stream_ops = { ... };
 - struct proto_ops inet_dgram_ops = { ... };

struct sock

- Network family/protocol-specific data structure
 - Data type: struct sock (in include/net/sock.h)
 - Currently kitchen sink for fields mostly used by TCP
 - Need clean up and organized into unions/external data structures pertaining to family, protocol, or each layer
- Important fields
 - struct proto *prot: protocol-specific socket methods
 - union {...} tp_pinfo: transport-protocol-specific fields
 - struct tcp_opt af_tcp;
 - union {...} protinfo: network-protocol-specific fields
 - struct inet_opt af_inet;

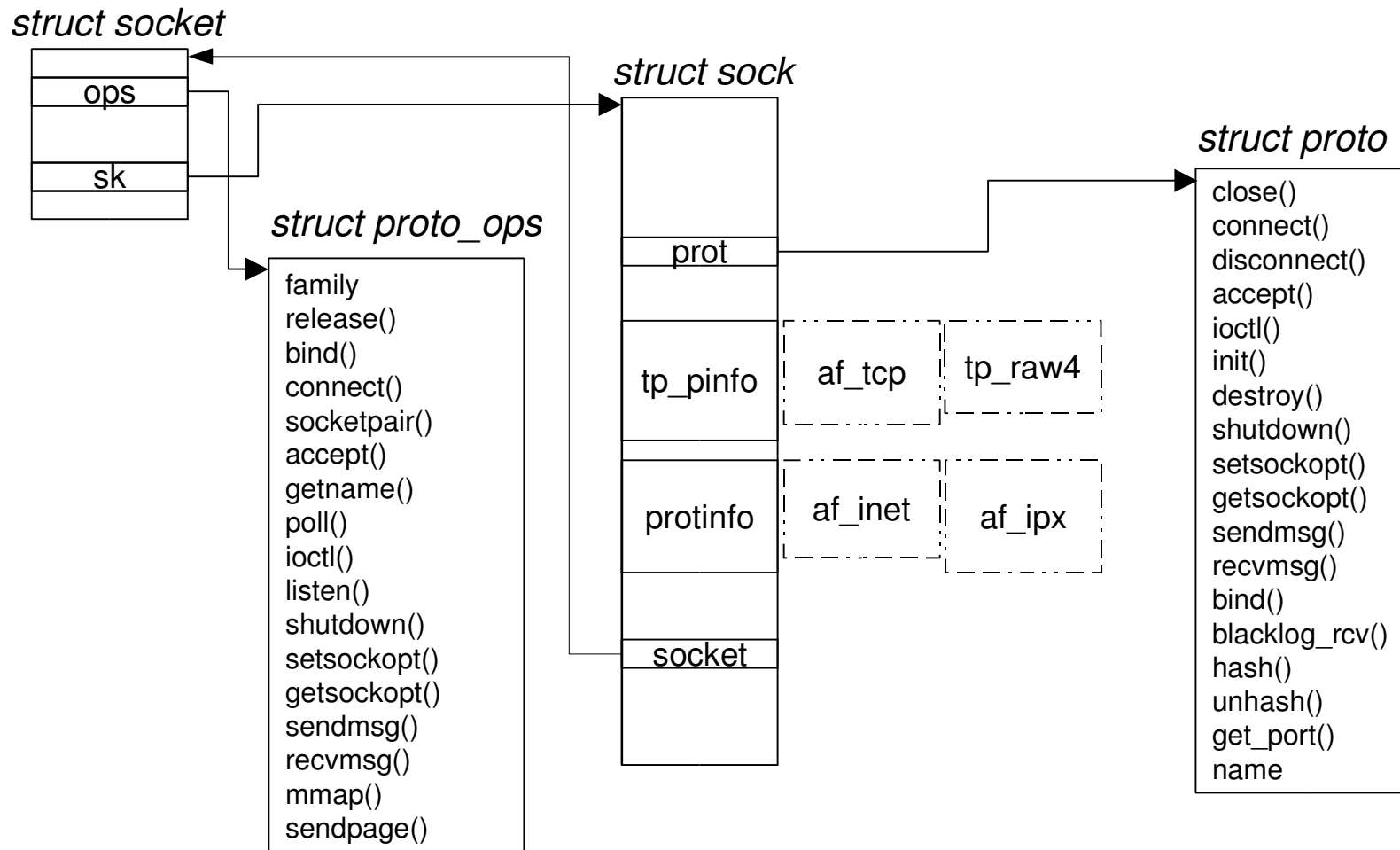
More fields of struct sock

- Many more fields (kitchen sink)
 - Pointers for lists and hashes: next, prev, pprev, ...
 - Addresses and ports: daddr, rcv_saddr, dport, ...
 - States and flags: state, zapped, shutdown, refcnt, ...
 - Destination and routing: dst_cache
 - Packet queues: receive_queue, write_queue, ...
 - Buffer, memory, locks: rcvbuf, lock, dst_lock, ...
 - Wait queue for blocking send/receive: sleep
 - Time stamp and timers: timer, stamp, rcvtimeo, ...
 - A set of functions: state_change(), data_ready(), ...
 - ...

struct proto

- Socket methods pertaining to specific protocol
 - Fields:
 - int (*close)(struct sock *sk, ...)
 - int (*connect)(struct sock *sk, ...)
 - int (*disconnect)(struct sock *sk, ...)
 - int (*accept)(struct sock *sk, ...)
 - ...
 - char name[32]: name of this protocol
- Example objects:
 - In net/ipv4/udp.c: struct proto udp_prot = { ... };
 - In net/ipv4/tcp_ipv4.c: struct proto tcp_prot = { ... };

Two Socket Structures Illustrated



Keeping Track of Families/Protocols

- Kernel data structure for network family
 - struct net_proto_family
- Two types of kernel data structure for protocol
 - For socket operations (multiplexing): struct inet_protosw
 - For receive operation (demultiplexing): struct inet_protocol
- Kernel keep three lists for above data structures
 - One for each type
 - A protocol implementation must register to the lists

Network Protocol Family

- Kernel data structure for a network family
 - Data type: `struct net_proto_family`
 - Each family must define this structure
- Kernel keeps a list of network protocol families
 - As an array of pointers to `struct net_proto_family`
 - Array name: `net_families[]` (in `net/socket.c`)
 - Each family must register by `sock_register(ops)`
 - Registration function `sock_register(ops)` essentially `net_families[ops->family]=ops;`

INET Family Example

- In net/ipv4/af_net.c:

```
struct net_proto_family inet_family_ops = {
    family:    PF_INET,
    create:    inet_create
};

...

static int __init inet_init(void)
{
    ...
    (void) sock_register(&inet_family_ops);
    ...
}
```

INET Protocols for Socket API

- Data structure for protocols supported by socket
 - struct `inet_protosw` (`include/net/protocol.h`)
- Fields
 - `type`: stream, dgram, or raw?
 - `protocol`: transport protocol number
 - struct `proto *prot`: transport-specific methods
 - struct `proto_ops *ops`: general BSD socket methods
- Kernel keeps a list of objects of this type
 - As an array `inetsw[]` (in `net/ipv4/af_inet.c`)
 - To add to list: call `inet_register_protosw(p)`

INET Protocols for Demultiplexing

- Data structure for protocols used in input chain
 - struct `inet_protocol` (`include/net/protocol.h`)
- Major fields
 - `handler()`, `err_handler()`: incoming packet handlers
 - `protocol`: transport protocol number
- Kernel keeps a list of objects of this type
 - Variable `inet_protos[]` (in `net/ipv4/protocol.c`)
 - To add to list: call `inet_add_protocol(prot)`
- May not have one-to-one mapping with the list of protocols supported by socket API

Builtin Protocols in 2.4 Kernel

- Transport protocols supported by socket API:
 - TCP (stream type) and UDP (dgram type)
 - RAW (not really a protocol, but a way to bypass the transport layer)
 - Look for `inetsw_array[]` in `net/ipv4/af_inet.c`:
- Demultiplexing protocols included:
 - 4 built-in: TCP, UDP, ICMP, IGMP
 - 3 loaded by modules: IPIP, GRE, PIM
 - Look for `igmp_protocol`, `tcp_protocol`, `udp_protocol`, and `icmp_protocol` in `net/ipv4/protocol.c`

Families and Protocols

net_families[]

| <i>family</i> | <i>create</i> |
|---------------|---------------|
| PF_INET | inet_create |

inet_protos[]

| |
|-------------------------------|
| tcp_rcv IPPROTO_TCP ... |
| udp_rcv IPPROTO_UDP ... |

inetsw[]

| |
|--|
| SOCK_STREAM IPPROTO_TCP &tcp_prot &inet_stream_ops ... |
| SOCK_DGRAM IPPROTO_UDP &udp_prot &inet_dgram_ops ... |
| SOCK_RAW IPPROTO_IP &raw_prot &inet_dgram_ops ... |

inet_stream_ops

PF_INET,
inet_release
inet_bind
inet_stream_connect
sock_no_socketpair
inet_accept
...

inet_dgram_ops

PF_INET,
inet_release
inet_bind
inet_dgram_connect
sock_no_socketpair
sock_no_accept
...

tcp_prot

tcp_close
tcp_v4_connect
tcp_disconnect
tcp_accept
tcp_ioctl
tcp_v4_init_sock
...
"TCP"

udp_prot

udp_close
udp_connect
udp_disconnect
NULL
udp_ioctl
NULL
...
"UDP"

Networking Stack Initialization

- Function `void sock_init(void)` in `net/socket.c`
 - Called by `do_basic_setup()` in `init/main.c`
 - Initialize the data structures and slab caches
 - Register the sockfs file system (will explain later)
- Each network family initializes when loading
 - Example: init function `int __init inet_init(void)` in `inet/ipv4/af_net.c` (for INET module)
 - Register the INET family (explained before)
 - Add all built-in protocols (to both lists)
 - Initialize other modules: arp, ip, tcp, icmp, ...

Socket and Inode

- An open socket is an open file in sockfs filesystem
- One-to-one mapping between socket and inode
 - In fact, socket object is embedded in the inode object:

```
struct inode {  
    ...  
    union {  
        struct minix_inode_info    minix_i;  
        struct ext2_inode_info     ext2_i;  
        ...  
        struct socket              socket_i;  
        ...  
    } u;  
};
```

Socket Filesystem

- Implementation in `net/socket.c`
 - Every data structures and functions for a file system

```
struct super_operations sockfs_ops = ...
struct dentry_operations sockfs_dentry_operations = ...
struct file_operations socket_file_ops = {
    llseek:    no_llseek,
    read:     sock_read,
    write:    sock_write,
    ... };
DECLARE_FSTYPE(sock_fs_type, "sockfs",
    sockfs_read_super, FS_NOMOUNT);
```
 - Register this file system in `sock_init()`:

```
register_filesystem(&sock_fs_type);
sock_mnt = kern_mount(&sock_fs_type);
```

Operations on a sockfs fd

- `sock_alloc()`: to allocate a struct socket object
 - Allocate an inode object instead
- `sock_map_fd()`: to allocate an open file object for the socket
 - Create and setup open file and dentry (including the links and methods)
- File operations on the task's fd: delegated to the corresponding socket operations
 - `sock_read()` calls `sock_recvmsg()`
 - `sock_write()` calls `sock_sendmsg()`

Allocate a struct socket

- Allocate an inode object instead, and establish the mapping

- struct socket *sock_alloc(void):

- inode = get_empty_inode();

- ...

- inode->i_sb = sock_mnt->mnt_sb;

- sock = socki_lookup(inode);

- ...

- sock->inode = inode;

- ...

- return sock;

- struct socket *socki_lookup(struct inode *inode):

- return &inode->u.socket_i;

Allocate fd for a Socket

- Create and setup open file and dentry objects

- int sock_map_fd(struct socket *sock):

- fd = get_unused_fd();

- if (fd >= 0) {

- ...

- file->f_dentry = d_alloc(...);

- file->f_dentry->d_op = &socket_dentry_operations;

- d_add(file->f_dentry, sock->inode);

- ...

- sock->file = file;

- file->f_op = sock->inode->i_fop = &socket_file_ops;

- ...

- fd_install(fd, file);

- }

- return fd;

File Operations on a Socket fd

- Translate into the corresponding socket operations
 - According to the `socket_file_ops` method table
 - Exmple: `ssize_t sock_read(file, ubuf, size, ppos)`

```
sock = socki_lookup(file->f_dentry->d_inode);  
...  
msg.... = ...  
...  
return sock_recvmsg(sock, &msg, size, flags);
```

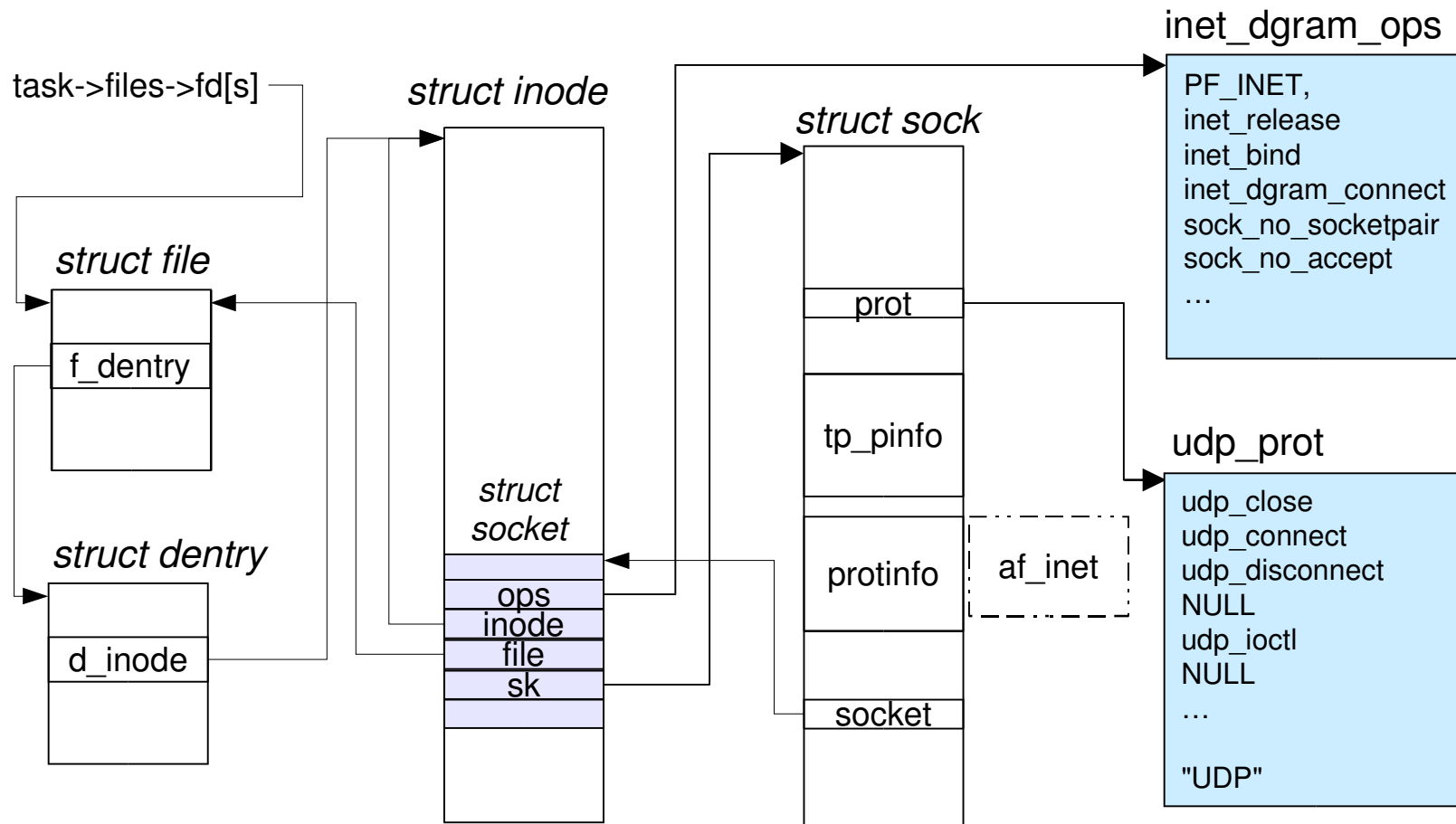
How is a Socket Created?

- System call service routine `sys_socket()`
 - First call `sock_create()`, which does

```
...
sock = sock_alloc()
sock->type = type;
net_families[family]->create(sock, protocol)
...
```
 - Then call `sock_map_fd()`, to allocate open file and fd
- In the INET family, `create` is `inet_create()`
 - Essentially, allocate a `struct sock` object and set up socket methods for the requested type/protocol pair:

```
sock->ops = ...->ops; sk->prot = ...->prot;
```

A UDP Socket Example



Socket Buffer

- Very important data structure for a packet
 - Carry packet payload through the input/output chains
 - Minimize copying
 - Memory allocated when the packet is first created (by socket or by network device driver) and released when it is done (by socket or network device driver)
- Data type: `struct sk_buff`
 - Defined in `include/linux/skbuff.h`

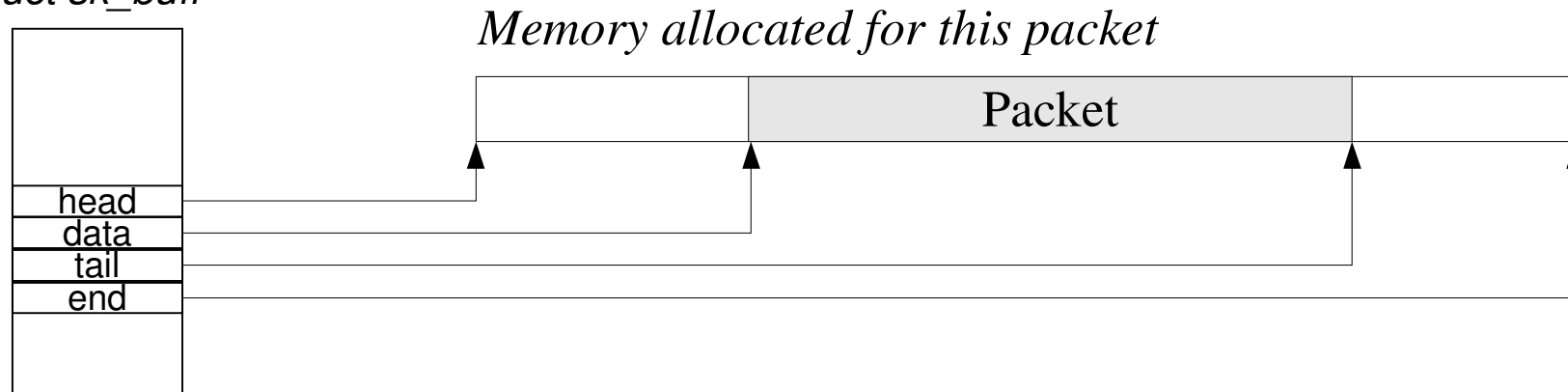
sk_buff Fields

- Pointers used to form various lists: next, prev, list
- Which socket it belongs to: sk
- Keeping track of time: stamp
- Incoming or outgoing network device: dev
- Pointer to transport layer header: union { ... } h
- Pointer to network layer header: union { ... } nh
- Pointer to link layer headers: union { ... } mac
- Information of this packet: len, csum, ...
- Pointers to packet data: head, data, tail, end

Buffer Handling

- head, end: beginning and end of allocated space
- data, tail: beginning and end of the packet area
- Space between head and data is for headers
 - Can shrink and grow as headers are added or removed

struct sk_buff



sk_buff Operations

- In `include/linux/skbuff.h` and `net/core/socket.c`
- `struct sk_buff *alloc_skb(size, gfp_mask)`
 - Create a `sk_buff` object and allocate memory for `size`
`skb->data = skb->tail = skb->head`
`skb->end = skb->head + size`
`skb->len = 0`
- `struct sk_buff *kfree_skb(skb)`
 - Free the `sk_buff` and deallocate packet memory
- `void skb_reserve(skb, len)`
 - Reserve headroom space before storing packet
`skb->data += len; skb->tail += len;`

More sk_buff Operations

- unsigned char *skb_put(skb, len)
 - Allow appending len bytes to tail, return previous tail
skb->tail += len; skb->len += len;
- unsigned char *skb_push(skb, len)
 - Allow adding len bytes before data, return new head
skb->data -= len; skb->len += len;
- unsigned char *skb_pull(skb, len)
 - Remove len bytes from data, return new head
skb->data += len; skb->len -= len;
- Be careful – kernel panic if data or tail moved out of bound

Protocol Headers

- Transport header: union h in sk_buff:

```
union {
    struct tcphdr    *th;
    struct udphdr    *uh;
    struct icmphdr   *icmph;
    ...
} h;
```

- Network header: union nh in sk_buff:

```
union {
    struct iphdr     *iph;
    struct ipv6hdr   *ipv6h;
    struct arphdr    *arph;
    ...
} nh;
```

Getting to the Protocol Headers

- Through the header pointers and header types
 - Example: UDP header type in include/linux/udp.h:

```
struct udphdr {  
    __u16    source;  
    __u16    dest;  
    __u16    len;  
    __u16    check;  
};
```
- Examples: get to all header fields from `sk_buff`:
 - UDP dest port number: `skb->h.uh->dest`
 - Dest IP address: `skb->nh.iph->daddr`

Tracing the Output Chain

- User-mode process sends a message with UDP
 - `sendto(fd, buff, len, flags, addr, addr_len)`
- System call service routine `sys_sendto()`
 - Look up struct socket from fd and call the socket operation to send message:
 - `sock = sockfd_lookup(fd, &err)`
 - `sock_sendmsg(sock, &msg, len)`
 - Here `sockfd_lookup(fd, err)` is essentially
 - `sock = socki_lookup(fget(fd) ->f_dentry->d_inode)`
 - And `sock_sendmsg(sock, msg, size)` is essentially
 - `sock->ops->sendmsg(sock, msg, size, &scm)`

Multiplexing in Transport Layer

- Socket method under INET family and datagram type (referred in `inet_dgram_ops`)
 - `sock->ops->sendmsg` → `inet_sendmsg()`
- `inet_sendmsg()`
 - Delegate to the protocol's `sendmsg` method:
 - `sk = sock->sk;`
 - `return sk->prot->sendmsg(sk, msg, size);`
- Under UDP protocol (referred in `udp_prot`):
 - `sk->prot->sendmsg` → `udp_sendmsg()`

Protocol Layer

- `udp_sendmsg()` in `net/ipv4/udp.c`
 - Verify the addresses
 - Find a route for this packet (call `ip_route_output()`)
 - Build the UDP header, etc.
 - Call `ip_build_xmit()` to build and transmit the packet
- Why do we need a route (`rt`) here?
 - Cached in struct `sock` (in case of connected socket)
 - Pass down as an argument to `ip_build_xmit()`
- `tcp_sendmsg()` in `net/ipv4/tcp.c`
 - Much more complicated

IP Output: ip_build_xmit()

- Source code in net/ipv4/ip_output.c
- Check for slow path
 - Use ip_build_xmit_slow() if need fragmentation
 - Need to know path MTU, that's why we need rt early
- Allocate socket buffer (sk_buff)
 - skb = sock_alloc_send_skb(...)
 - Which eventually calls alloc_skb()
- Reverse headroom for link layer header
 - skb_reserve(skb, hh_len);
 - Again, we need to know which type of device from rt

ip_build_xmit() continued

- Fill the packet in sk_buff for the packet
 - Expand skb payload to include IP header and message
iph = (struct iphdr *)skb_put(skb, length)
 - Build the IP header (we need rt to know source addr)
 - Get the fragment from user space (by udp_getfrag() function, passed as getfrag from udp_sendmsg())
getfrag(frag, (char *)iph)+iph->ihl*4, 0, length-iph->ihl*4);
- Pass through the netfilter chain
 - Netfilter: will cover later
 - Eventually reach the network device driver

Summary

- Linux Networking
 - LKP: §8
 - ULK: §18
- Next lecture: Linux Networking