Inside the Black Box:
Kernel Analysis of TCP/IP Flow

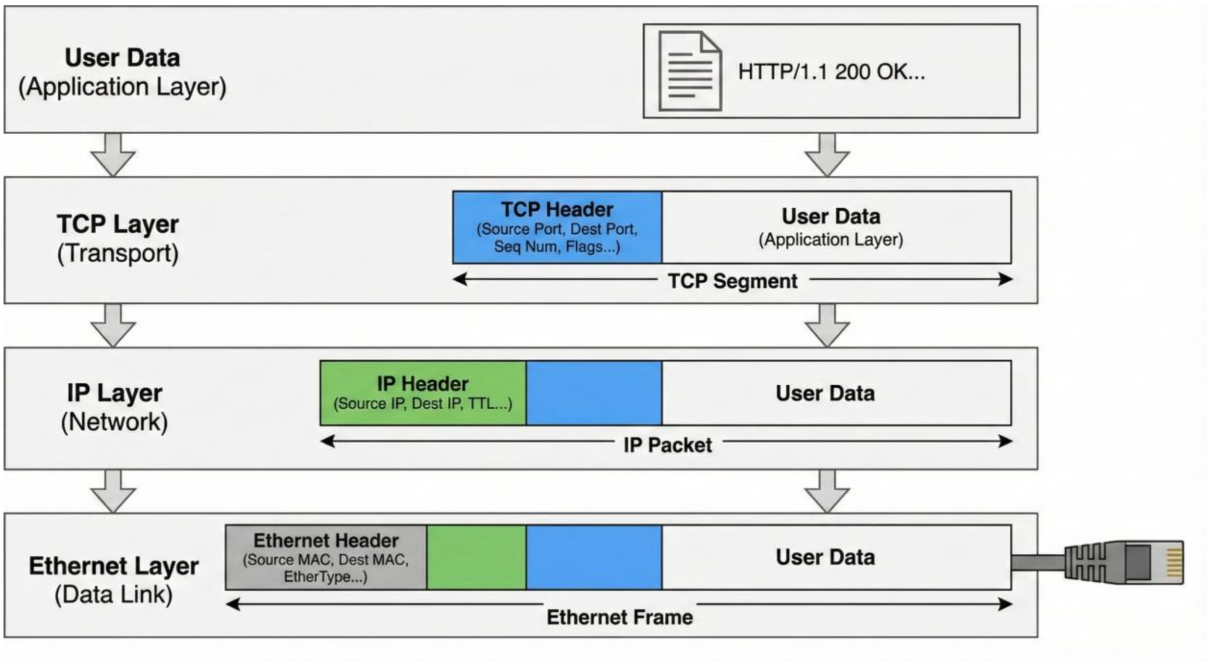
Class	리눅스 시스템 응용 설계 Class #01
Professor	손용석
Team number	Team 6
Team Members	김찬중 (20213780) 장지요 (20222712) 박연우 (20223605) 배영경 (20222343)
Author	김찬중, 장지요, 박연우, 배영경

Project Overview

Kernel Version Selection: Linux 6.12.57. We have selected **Linux Kernel 6.12.57** as the target environment for this analysis. This version was chosen because it represents the most recent **Long Term Support (LTS)** release available. Analyzing the latest LTS version is crucial for systems programming research, as it offers the most up-to-date network stack optimizations and security features while ensuring the stability and long-term relevance widely adopted in modern production environments.

Motivation and Objectives: Bridging the Gap from Backend to Kernel The primary motivation for selecting the **TCP subsystem** stems from our team's strong background in **Web Backend Development**. While we possess extensive experience handling Application Layer (Layer 7) traffic, our understanding of the underlying mechanisms was limited to the theoretical concepts of the IP Layer covered in university "Computer Communication" coursework.

To overcome this abstraction gap, we designed this project to go beyond a superficial review of TCP state transitions. Our objective is to perform a vertical analysis of the Linux networking stack, tracing the data flow **from Layer 4 (Transport Layer) down to Layer 2 (Data Link Layer)**. By investigating the kernel code starting from the `sendmsg` system call, we aim to visualize the practical "encapsulation" process—how a user's payload is wrapped with TCP, IP, and Ethernet headers—and to fully comprehend how Linux orchestrates network communication at the implementation level.



Research part in charge

김찬중: Part 0 ~ Part 1

Part 0 (Generic Socket Layer) examines how the kernel manages the transition from user space to kernel space. It details the mechanism of resolving a file descriptor into an internal socket object, securely importing message metadata into kernel memory, and enforcing mandatory security policies via Linux Security Modules (LSM).

Part 1 (Protocol Family Resolution) investigates the transition from the generic layer to the IPv4 protocol family (AF_INET). It focuses on the necessary preparations, such as socket binding, and analyzes the optimization techniques used to efficiently dispatch the control flow to the specific TCP protocol handler.

장지요: Part 2

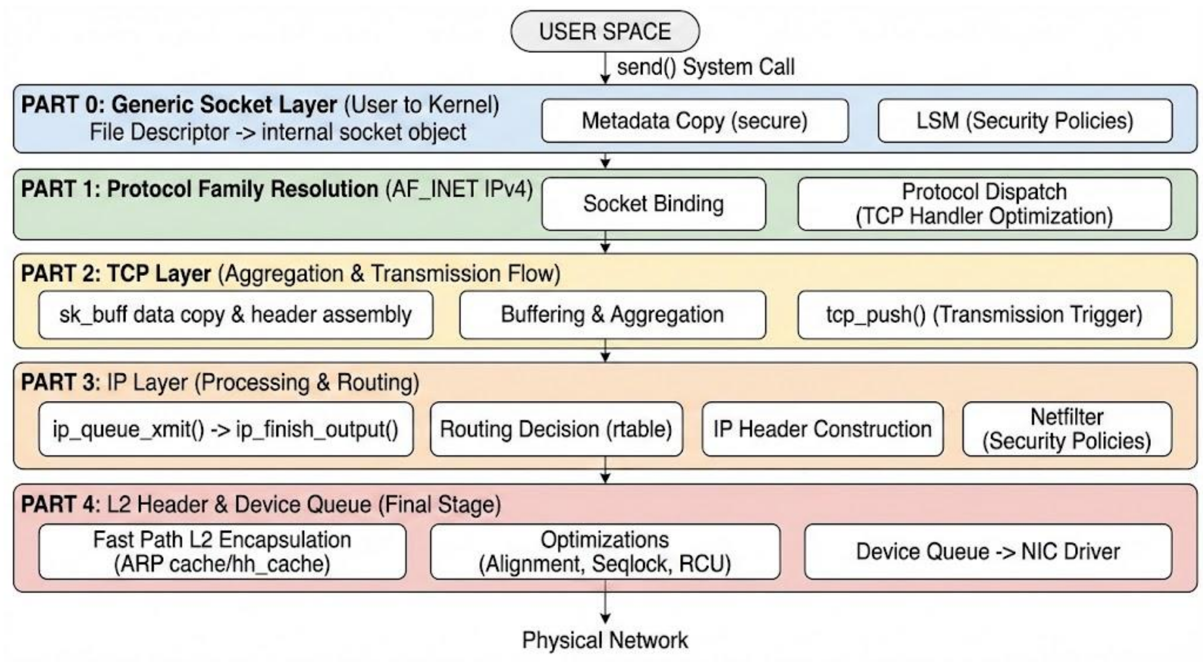
Part 2 (TCP Segment Aggregation & Transmission Flow Analysis) analyzed the entire process of TCP data transmission by tracing it at the Linux kernel level. The flow, from the `send()` system call initiated in user space to the secure copying of data into the kernel's internal socket buffer (`struct sk_buff`), followed by header assembly and the ultimate handoff to the IP layer, is meticulously dissected focusing on functions and structures. Particular emphasis is placed on analyzing the mechanism utilized by TCP for efficient buffering and data aggregation, along with the structure of `tcp_push()` which triggers actual transmission after evaluating specific delivery conditions. This process allows for a comprehensive synthesis of the practical operational principles, the memory processing flow, and the transmission timing determination mechanisms of the TCP send path.

박연우: Part 3

Part 3 (IP Layer Processing & Routing Decision) analyzed the entire process from when the sk_buff transmitted from the TCP layer is converted into a complete IPv4 packet and transmitted to the L2 layer. This part focuses on the IPv4 transmission path from ip_queue_xmit() -> ip_local_out() -> ip_output() -> ip_finish_output(), and analyzed in depth the routing decision performed by the kernel, the physical construction of the IP header (Header Construction), and the process of applying security policies through Netfilter from the perspective of structures (sk_buff, rtable) and memory manipulation.

배영경: Part 4

Part 4 (L2 Header Encapsulation & Device Queue) analyzes the final stages of packet transmission, focusing on the transition from the IP layer to the physical device queue. It details the Fast Path mechanism for efficient L2 header encapsulation, utilizing the ARP cache (hh_cache) to bypass redundant address resolution. Key optimizations analyzed include Space-Time Trade-offs in memory alignment (16-byte copy for 14-byte headers) and Concurrency Control mechanisms (Seqlock for cache integrity, RCU for lock-free transmission), demonstrating how the kernel maximizes throughput in the critical path to the NIC driver.


USER SPACE
send() System Call
PART 0: Generic Socket Layer (User to Kernel)
File Descriptor -> internal socket object
Metadata Copy (secure)
LSM (Security Policies)
PART 1: Protocol Family Resolution (AF_INET IPv4)
Socket Binding
Protocol Dispatch (TCP Handler Optimization)
PART 2: TCP Layer (Aggregation & Transmission Flow)
sk_buff data copy & header assembly
Buffering & Aggregation
tcp_push() (Transmission Trigger)
PART 3: IP Layer (Processing & Routing)
ip_queue_xmit() -> ip_finish_output()
Routing Decision (rtable)
IP Header Construction
Netfilter (Security Policies)
PART 4: L2 Header & Device Queue (Final Stage)
Fast Path L2 Encapsulation (ARP cache/hh_cache)
Optimizations (Alignment, Seqlock, RCU)
Device Queue -> NIC Driver
Physical Network

Part 0. System Call Entry & Generic Socket Abstraction

1) SYSCALL_DEFINE3(sendmsg, ...)

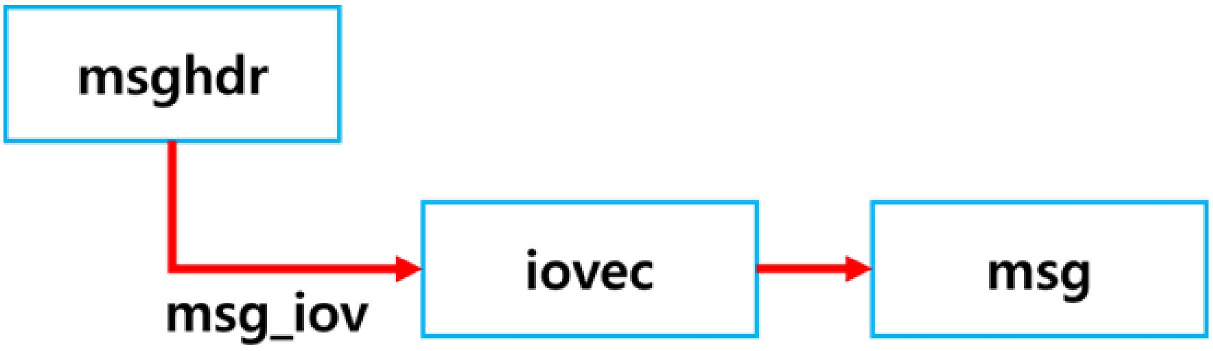
```
SYSCALL_DEFINE3(sendmsg, int, fd, struct user_msghdr __user *, msg, unsigned int, flags)
{
	return __sys_sendmsg(fd, msg, flags, true);
}
```

Detailed Code Logic Analysis

- **SYSCALL_DEFINE3(sendmsg, ...)**
  - **Description: The System Call Macro.**
  - **Mechanism:** This macro expands into the actual function definition (e.g., __x64_sys_sendmsg). The number 3 indicates it accepts three arguments. It handles the context switch overhead and sets up the initial kernel stack frame for this operation.
- **Arguments:**
  - **int fd:** The integer File Descriptor representing the socket. At this stage, it is just a number (index) provided by the user.
  - **struct user_msghdr __user *msg:** A raw pointer pointing to a memory location in **User Space**. It contains the "blueprint" of the message (metadata and data locations). The __user annotation is a safety marker for static analysis tools (sparse) to prevent direct dereferencing.
  - **unsigned int flags:** User-provided control flags (e.g., MSG_OOB, MSG_DONTWAIT).
- **return __sys_sendmsg(fd, msg, flags, true);**
  - **Description: Immediate Delegation.**
  - **Mechanism:** The function does zero processing logic. It simply calls __sys_sendmsg.
  - **true Argument:** This corresponds to the forbid_cmsg_compat parameter. It explicitly tells the kernel **not** to support 32-bit compatibility mode for control messages (CMSG) in this specific call path, assuming a standard 64-bit environment (or matching native environment).

Data Transformation Status

- **Header/Data Status: Raw & External.** No headers (TCP/IP) exist. The data and metadata reside entirely in User Space, and the kernel has not yet verified if the pointers are valid.
- **Encapsulation: None.** This is the starting point of the encapsulation chain.



(msghdr and iovec points msg)

This macro defines the **kernel entry point** for the sendmsg system call. It acts as the interface between User Space and Kernel Space. Its primary responsibility is to handle the architecture-specific **Application Binary Interface (ABI)** details—mapping values from CPU registers into standard C function arguments (fd, msg, flags)—and to immediately delegate execution to the internal handler __sys_sendmsg.

2) __sys_sendmsg()

```
long __sys_sendmsg(int fd, struct user_msghdr __user *msg, unsigned int flags,
                 bool forbid_cmsg_compat)
{
        int fput_needed, err;
        struct msghdr msg_sys;
        struct socket *sock;

        if (forbid_cmsg_compat && (flags & MSG_CMSG_COMPAT))
                return -EINVAL;

        sock = sockfd_lookup_light(fd, &err, &fput_needed);
        if (!sock)
                goto out;

        err = ___sys_sendmsg(sock, msg, &msg_sys, flags, NULL, 0);
        fput_light(sock->file, fput_needed);
out:
        return err;
}
```

Detailed Code Logic Analysis

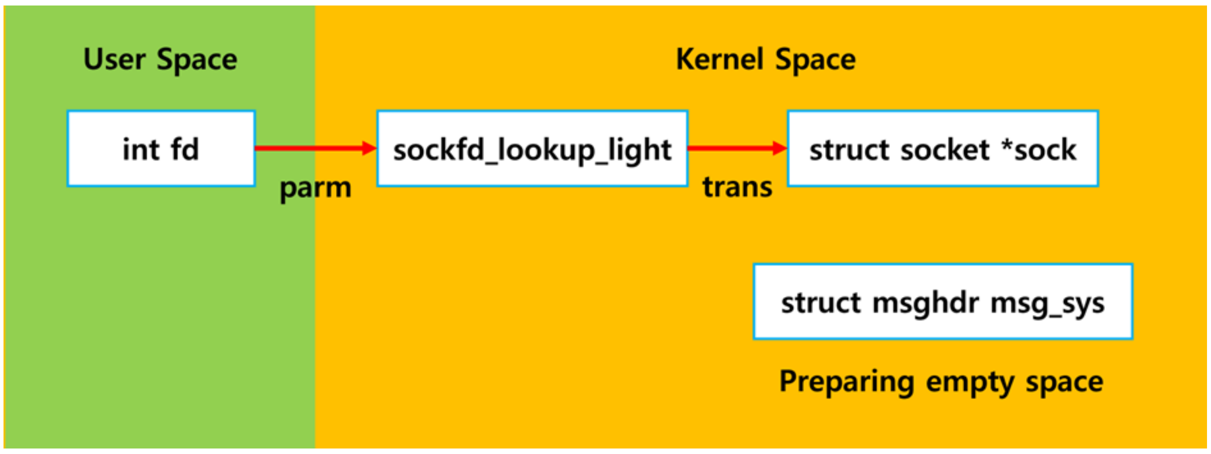
- **struct msghdr msg_sys;**
  - **Description:** A local variable allocated on the kernel stack.
  - **Purpose:** This structure is essentially an empty container at this stage. It is destined to hold the sanitized copy of the user's msghdr. The actual copying

happens inside the called function ___sys_sendmsg (note the three underscores).

- **sock = sockfd_lookup_light(fd, &err, &fput_needed);**
  - **Description:** This is the most critical operation in this function.
  - **Mechanism:** It takes the user-provided integer fd and searches the current process's file descriptor table to find the corresponding struct socket instance.
  - **"Light" optimization:** The suffix _light indicates a performance optimization. It attempts to access the file structure without aggressively incrementing the atomic reference count (refcount), assuming the file won't be closed during this quick operation. fput_needed stores the state of whether a full reference release is needed later.
- **err = ___sys_sendmsg(sock, msg, &msg_sys, flags, NULL, 0);**
  - **Description:** The core logic is delegated to ___sys_sendmsg.
  - **Transformation:** Notice that the first argument has changed from fd (integer) to sock (pointer). The function now passes the resolved socket object and the address of the empty msg_sys container to the next layer.
- **fput_light(sock->file, fput_needed);**
  - **Description:** Cleans up the reference taken during the lookup phase. This ensures resource safety (preventing memory leaks or race conditions) after the sending process is initiated.

Data Transformation Status

- **Identifier Transformation:** int fd -> struct socket *sock.
  - The system no longer relies on the user-provided ID number; it now holds a direct pointer to the internal kernel object representing the connection.
- **Message Status:** The message data is still in user space. The msg_sys structure is allocated on the stack but is currently uninitialized/empty.



(File descriptor is parameterized to sockfd_lookup_light, and it translates into struct socket *sock. Then kernel prepares empty space to store messages to send)

This function serves as the **Context Resolution Layer** within the kernel. While the previous step (SYSCALL_DEFINE3) handled the interface with user space, __sys_sendmsg is responsible for translating the abstract file descriptor (fd) into a concrete kernel object (struct socket). It also allocates a local kernel structure (msg_sys) to hold the message metadata in the subsequent steps, but **no data copying has occurred yet.**

3) ___sys_sendmsg()

```
static int ___sys_sendmsg(struct socket *sock, struct user_msghdr __user *msg,
                       struct msghdr *msg_sys, unsigned int flags,
                       struct used_address *used_address,
                       unsigned int allowed_msghdr_flags)
{
       struct sockaddr_storage address;
       struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;
       ssize_t err;

       msg_sys->msg_name = &address;

       err = sendmsg_copy_msghdr(msg_sys, msg, flags, &iov);

       if (err < 0)
              return err;

       err = ____sys_sendmsg(sock, msg_sys, flags, used_address,
                            allowed_msghdr_flags);

       kfree(iov);

       return err;
}
```

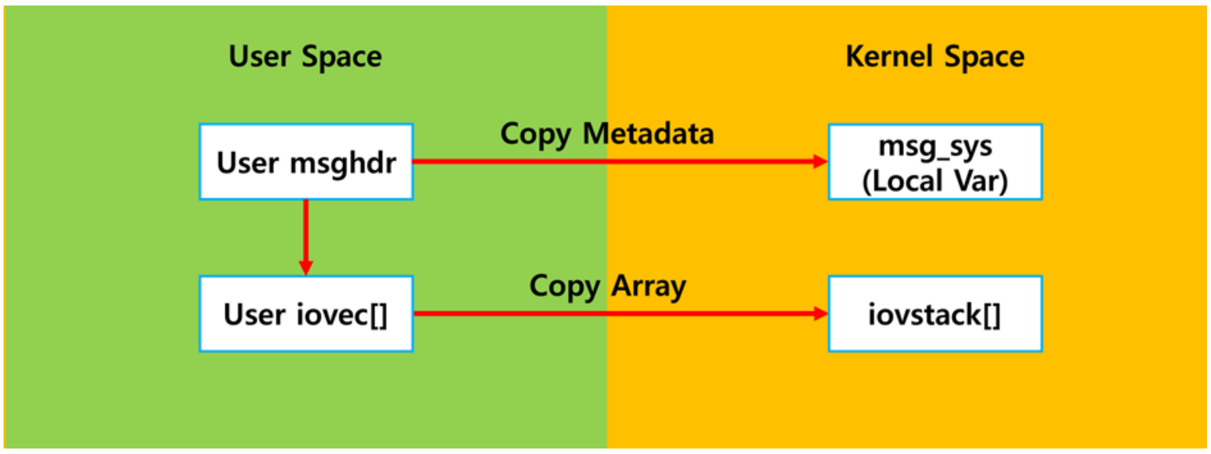
Detailed Code Logic Analysis

- **struct iovec iovstack[UIO_FASTIOV], *iov = iovstack;**
    - **Description:** The **Fast Path Optimization**.
    - **Mechanism:** UIO_FASTIOV is typically defined as 8. The kernel assumes that most sendmsg calls involve fewer than 8 distinct data buffers. By declaring this array on the kernel stack, the function avoids the overhead of dynamic memory allocation. The pointer iov is initialized to point to this stack buffer.
- **msg_sys->msg_name = &address;**
    - **Description:** Address buffer assignment.
    - **Mechanism:** The kernel links the msg_name field of the local msg_sys structure to the local address variable (struct sockaddr_storage). This prepares the container to receive the destination address (if provided) from user space.
- **err = sendmsg_copy_msghdr(msg_sys, msg, flags, &iov);**
    - **Description:** The **Primary Import Routine**.
    - **Mechanism:** This function performs the critical work:

   1. Copies the msghdr fields from user space (msg) to kernel space (msg_sys).
   2. Copies the iovec array (the list of data pointers).
   3. **Logic:** If the user's IO vector fits in UIO_FASTIOV, it uses the stack. If the user provides a large number of buffers (e.g., 100), this function dynamically allocates memory on the heap and updates the iov pointer.
- **err = ____sys_sendmsg(sock, msg_sys, flags, ...);**
   - **Description:** Delegation to the next layer.
   - **Transformation:** The function passes the fully populated kernel-side msg_sys structure and the validated iovec (via the struct) to the next function (____sys_sendmsg - 4 underscores).
- **kfree(iov);**
   - **Description:** Conditional Cleanup.
   - **Mechanism:** If sendmsg_copy_msghdr had to allocate heap memory (Slow Path), this frees it. If the stack buffer (iovstack) was used, kfree handles it gracefully (or the pointer logic ensures no free is attempted).

Data Transformation Status

- **Metadata Status: Copied to Kernel.** The kernel now possesses its own trusted copy of the message headers and the IO vector map.
- **Payload Status: Still in User Space.** The actual data bytes (e.g., "Hello World") have not been moved yet. The kernel only has the *addresses* pointing to where that data lives.



(This diagram illustrates the fast-path optimization in ___sys_sendmsg, where message metadata and IO vectors are securely copied into the kernel stack to avoid dynamic allocation, while the actual data payload remains in user space.)

In this phase, the kernel actively imports the "blueprint" of the message from User Space to Kernel Space. The primary responsibility of ___sys_sendmsg is to safely copy the message metadata (struct msghdr) and the array of data pointers (struct iovec). Notably, it implements **stack-allocation optimization** to avoid expensive heap allocations (kmalloc) for common, small IO operations.

4) ____sys_sendmsg()

```
static int ____sys_sendmsg(struct socket *sock, struct msghdr *msg_sys,
                           unsigned int flags, struct used_address *used_address,
                           unsigned int allowed_msghdr_flags)
{
        unsigned char ctl[sizeof(struct cmsghdr) + 20]
                                __aligned(sizeof(__kernel_size_t));

        /* 20 is size of ipv6_pktinfo */

        unsigned char *ctl_buf = ctl;

        int ctl_len;

        ssize_t err;

        err = -ENOBUFS;

        if (msg_sys->msg_controllen > INT_MAX)
                goto out;

        flags |= (msg_sys->msg_flags & allowed_msghdr_flags);
        ctl_len = msg_sys->msg_controllen;

        if ((MSG_CMSG_COMPAT & flags) && ctl_len) {
                err =
                    cmsghdr_from_user_compat_to_kern(msg_sys, sock->sk, ctl,
                                                     sizeof(ctl));

                if (err)
                        goto out;

                ctl_buf = msg_sys->msg_control;
                ctl_len = msg_sys->msg_controllen;

        } else if (ctl_len) {

                BUILD_BUG_ON(sizeof(struct cmsghdr) !=
                             CMSG_ALIGN(sizeof(struct cmsghdr)));

                if (ctl_len > sizeof(ctl)) {
                        ctl_buf = sock_kmalloc(sock->sk, ctl_len, GFP_KERNEL);

                        if (ctl_buf == NULL)
                                goto out;
                }

                err = -EFAULT;

                if (copy_from_user(ctl_buf, msg_sys->msg_control_user, ctl_len))
                        goto out_freectl;

                msg_sys->msg_control = ctl_buf;
                msg_sys->msg_control_is_user = false;
        }
        flags &= ~MSG_INTERNAL_SENDMSG_FLAGS;
        msg_sys->msg_flags = flags;

        if (sock->file->f_flags & O_NONBLOCK)
                msg_sys->msg_flags |= MSG_DONTWAIT;
        /*
         * If this is sendmmsg() and current destination address is same as
         * previously succeeded address, omit asking LSM's decision.
         * used_address->name_len is initialized to UINT_MAX so that the first
         * destination address never matches.
         */
        if (used_address && msg_sys->msg_name &&
            used_address->name_len == msg_sys->msg_namelen &&
            !memcmp(&used_address->name, msg_sys->msg_name,
                    used_address->name_len)) {

                err = sock_sendmsg_nosec(sock, msg_sys);
                goto out_freectl;
        }

        err = __sock_sendmsg(sock, msg_sys);

        /*
         * If this is sendmmsg() and sending to current destination address was
         * successful, remember it.
         */

        if (used_address && err >= 0) {
                used_address->name_len = msg_sys->msg_namelen;

                if (msg_sys->msg_name)
                        memcpy(&used_address->name, msg_sys->msg_name,
                               used_address->name_len);

        }

out_freectl:
        if (ctl_buf != ctl)

                sock_kfree_s(sock->sk, ctl_buf, ctl_len);

out:
        return err;
}
```

Detailed Code Logic Analysis

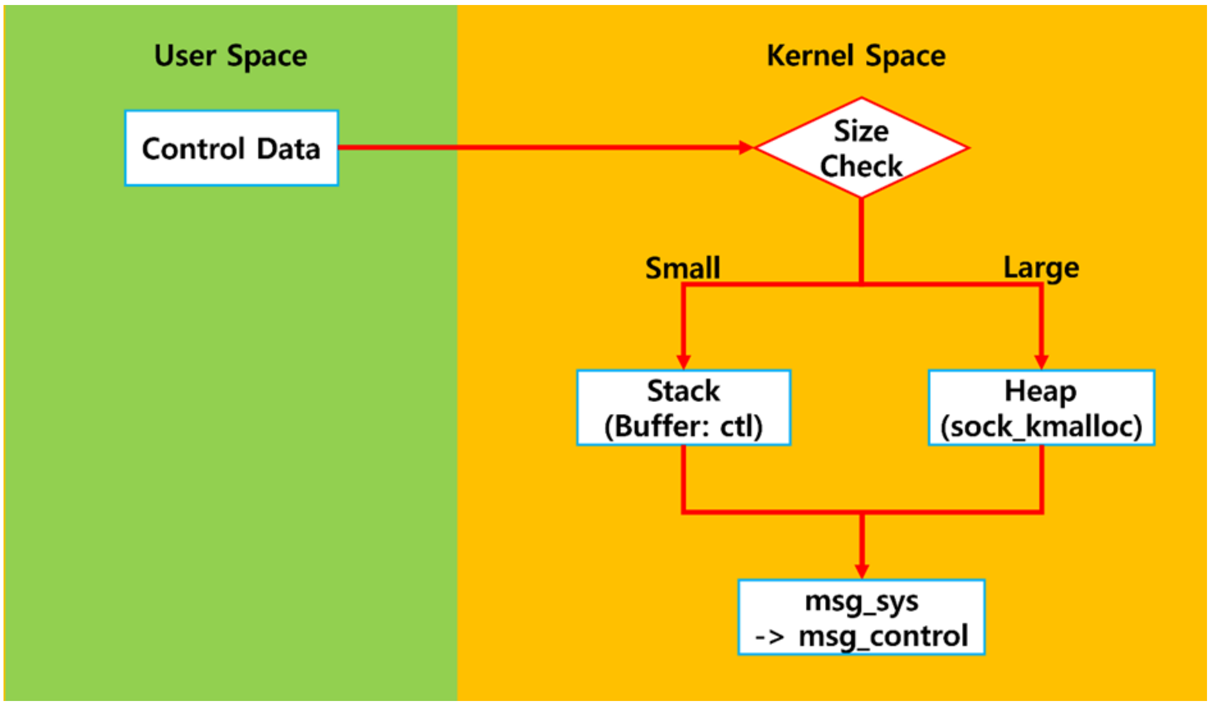
- **unsigned char ctl[sizeof(struct cmsghdr) + 20];**
    - **Description: Stack Buffer Optimization for CMSG.**
    - **Mechanism:** Similar to the iovec optimization in the previous phase, the kernel allocates a small buffer on the stack to hold control messages. The

size 20 allows for common small headers like ipv6_pktinfo without triggering dynamic memory allocation.

- **if (ctl_len > sizeof(ctl)) { ctl_buf = sock_kmalloc(...); }**
    - **Description: Dynamic Allocation Fallback.**
    - **Mechanism:** If the user provides a large amount of control data (exceeding the stack buffer size), the kernel switches to sock_kmalloc to allocate memory from the heap (GFP_KERNEL). This handles complex cases like passing file descriptors (SCM_RIGHTS).
- **copy_from_user(ctl_buf, msg_sys->msg_control_user, ctl_len)**
    - **Description: Secure Data Import.**
    - **Mechanism:** This is where the actual control data (not the payload data) is copied into the kernel. The msg_sys->msg_control pointer is updated to point to this new kernel-resident buffer (ctl_buf).
- **if (sock->file->f_flags & O_NONBLOCK) ...**
    - **Description: Flag Synchronization.**
    - **Mechanism:** It checks the file descriptor's status. If the socket was opened in Non-blocking mode, it forcibly sets the MSG_DONTWAIT flag, ensuring the transport layer knows not to sleep if the buffers are full.
- **if (used_address && ... !memcmp(...))**
    - **Description: LSM (Linux Security Module) Caching.**
    - **Mechanism:** This is an optimization for sendmmsg (sending multiple messages). If the current message is going to the exact same address as the previous one, it calls sock_sendmsg_nosec. This bypasses repetitive security checks (like SELinux or AppArmor), improving performance for bulk transfers.
- **err = __sock_sendmsg(sock, msg_sys);**
    - **Description: Protocol Dispatch.**
    - **Mechanism:** Everything is now ready. The function calls __sock_sendmsg, which will look up the socket's virtual function table (sock->ops->sendmsg) and route the packet to the TCP (or UDP) layer.

Data Transformation Status

- **Control Data (CMSG): Copied to Kernel.** Auxiliary information (like setting TTL or IP options) now resides safely in kernel memory (either stack or heap).
- **Payload Data: Still in User Space.** The actual application data ("Hello World") has *still* not been copied. The kernel only holds the iovec pointers (from Phase 3) and now the msg_control data (from Phase 4).



(This diagram depicts the kernel's optimization strategy of using a fast stack buffer for small control messages and falling back to heap allocation only for larger data.)

This function acts as the **final preparation layer** before handing control over to the generic socket subsystem. Its primary responsibilities are:

1) **Handling Ancillary Data (Control Messages):** It securely copies control messages (CMSG), such as IP options or auxiliary data, from user space to kernel space.
2) **Configuration Sanitization:** It finalizes the flag settings (e.g., Non-blocking mode).
3) **Security Optimization:** It attempts to cache security checks (LSM) for repeated sends to the same destination (sendmsg).
4) **Dispatch:** It calls __sock_sendmsg, which invokes the protocol-specific transport logic.

5) __sock_sendmsg()

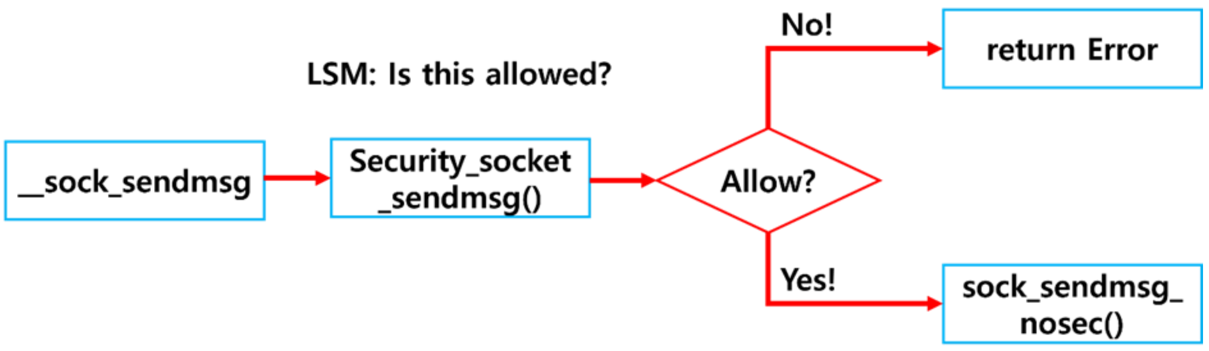
```
static int __sock_sendmsg(struct socket *sock, struct msghdr *msg)
{
	int err = security_socket_sendmsg(sock, msg,
					  msg_data_left(msg));

	return err ?: sock_sendmsg_nosec(sock, msg);
}
```

Detailed Code Logic Analysis

- **int err = security_socket_sendmsg(sock, msg, msg_data_left(msg));**
  - **Description: The LSM Hook.**
  - **Mechanism:** This function triggers the security hooks registered in the kernel.
    - It passes the socket, the message structure, and the total size of the payload (msg_data_left(msg)).
    - Security modules (SELinux, AppArmor, Smack, etc.) inspect this request. For example, they verify if an application restricted to local communication is trying to send data to an external IP.
  - **Return Value:** It returns 0 on success, or a negative error code (e.g., -EACCES or -EPERM) if the security policy denies the action.
- **return err ?: sock_sendmsg_nosec(sock, msg);**
  - **Description: Conditional Dispatch (Elvis Operator).**
  - **Mechanism:** This line uses a C extension (often called the Elvis operator).
    - **If err is non-zero (Failure):** The security check failed. The function immediately returns the error code, aborting the transmission.
    - **If err is zero (Success):** The security check passed. The function proceeds to call sock_sendmsg_nosec ("no security").
  - **Note:** The suffix _nosec implies that the internal function does not need to perform the security check again, as it has just been cleared by this wrapper.

Data Transformation Status

- **Logic Status: Validated.** The operation is now "authorized" by the kernel's security subsystem.
- **Data Status: Unchanged.** No data has moved yet. This step is purely bureaucratic, getting the "stamp of approval" to proceed with the transport.



(This flowchart illustrates the mandatory LSM security checkpoint, where the operation is authorized by modules like SELinux before proceeding to the protocol-specific transmission logic.)

This function acts as the **Mandatory Security Gatekeeper**. Before the kernel attempts to actually transmit any data, it must consult the Linux Security Modules (LSM). This function ensures that the calling process has the necessary permissions (defined by policies like SELinux or AppArmor) to send data through the specific socket instance.

6) sock_sendmsg_nosec()

```
static inline int sock_sendmsg_nosec(struct socket *sock, struct msghdr *msg)
{

	int ret = INDIRECT_CALL_INET(READ_ONCE(sock->ops)->sendmsg, inet6_sendmsg,
inet_sendmsg, sock, msg, msg_data_left(msg));

	BUG_ON(ret == -EIOCBQUEUED);

	if (trace_sock_send_length_enabled())
		call_trace_sock_send_length(sock->sk, ret, 0);

	return ret;
}
```

Detailed Code Logic Analysis

- **INDIRECT_CALL_INET(READ_ONCE(sock->ops)->sendmsg, ...)**
  - **Description: Indirect Call Optimization (Retpoline Mitigation).**
  - **Mechanism:** In modern kernels, calling a function via a pointer (indirect call) is expensive due to CPU mitigations for speculative execution attacks (Spectre v2/Retpoline).
  - **Logic:** This macro acts as a "prediction" cache. It checks:
    1. "Is the target function inet_sendmsg (IPv4)?" -> If yes, call it directly.
    2. "Is the target function inet6_sendmsg (IPv6)?" -> If yes, call it directly.

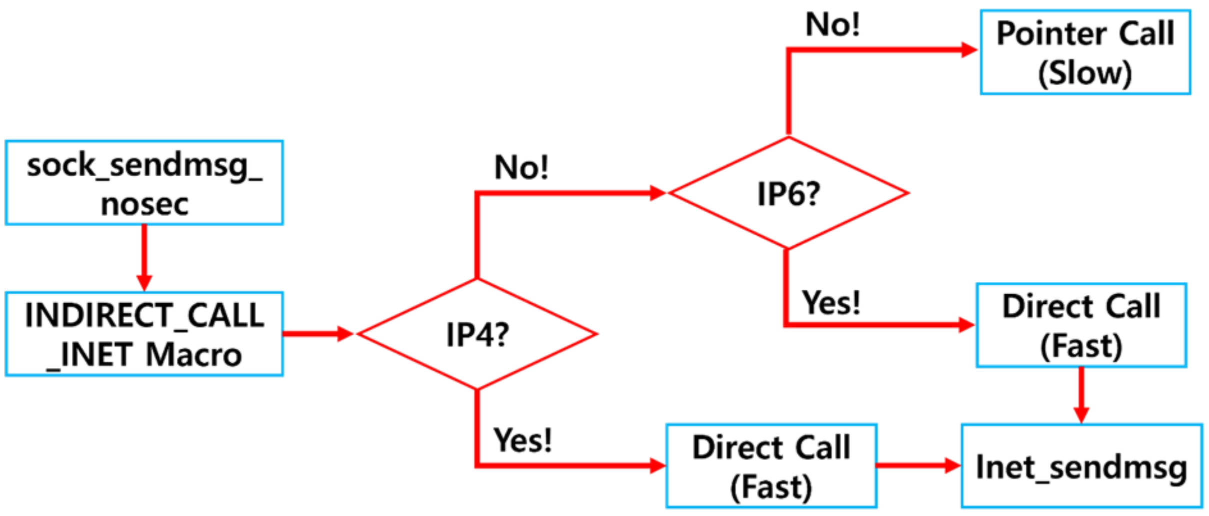
3. **Fallback:** Only if it's neither, use the slower function pointer sock->ops->sendmsg.

   - **Result:** Since most traffic is IPv4/TCP or IPv6/TCP, this turns a slow indirect call into a fast direct call.

- **BUG_ON(ret == -EIOCBQUEUED);**
  - **Description: Sanity Check.**
  - **Mechanism:** This asserts that synchronous sendmsg calls should not return a queued status code intended for asynchronous I/O (AIO). If this happens, it indicates a severe kernel bug, and the system halts to prevent corruption.
- **if (trace_sock_send_length_enabled()) ...**
  - **Description: Observability Hook.**
  - **Mechanism:** If tracing (like ftrace or eBPF) is enabled, it records the length of the data successfully sent. This is useful for performance monitoring tools.

Data Transformation Status

- **Logic Status: Handover.** The control flow transitions from the abstract "Socket" concept to the concrete "Internet Protocol" concept (AF_INET or AF_INET6).
- **Next Destination:** For a standard TCP/IPv4 socket, the flow will jump to **inet_sendmsg**.



(This diagram highlights the INDIRECT_CALL_INET optimization, which bypasses expensive indirect branch predictions by explicitly checking for and directly calling standard handlers like inet_sendmsg.)

This function serves as the **bridge** between the Generic Socket Layer and the specific Protocol Family Layer (e.g., IPv4 or IPv6). Its primary role is to invoke the sendmsg function defined in the socket's operation table (sock->ops). Crucially, it utilizes the INDIRECT_CALL_INET macro to optimize performance by avoiding the overhead of indirect branch prediction where possible (Spectre mitigation).

Part 1. Protocol Family Resolution & Transport Dispatch

1) inet_sendmsg()

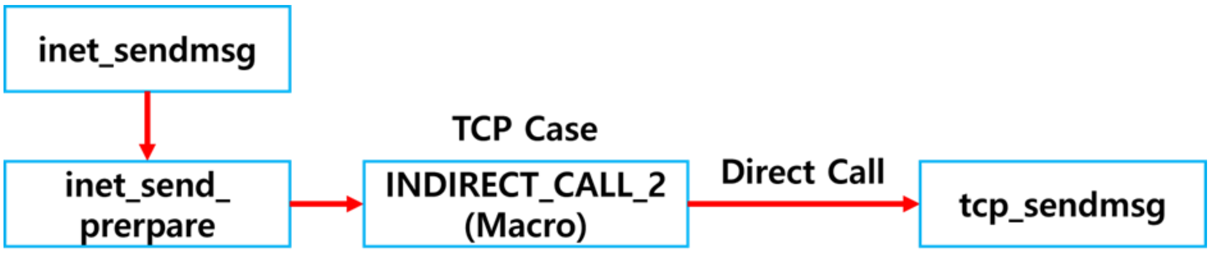
```
int inet_sendmsg(struct socket *sock, struct msghdr *msg, size_t size)
{
	struct sock *sk = sock->sk;

	if (unlikely(inet_send_prepare(sk)))
		return -EAGAIN;

	return INDIRECT_CALL_2(sk->sk_prot->sendmsg, tcp_sendmsg, udp_sendmsg,
			       sk, msg, size);
}
EXPORT_SYMBOL(inet_sendmsg);
```

Detailed Code Logic Analysis

- **struct sock *sk = sock->sk;**
  - **Description: Context Extraction.**
  - **Mechanism:** The function extracts the internal network-layer representation (struct sock) from the BSD-style file-layer socket (struct socket). sk contains all the protocol-specific states (like TCP states, port numbers, etc.).
- **if (unlikely(inet_send_prepare(sk))) return -EAGAIN;**
  - **Description: Auto-Binding (Preparation).**
  - **Mechanism:** inet_send_prepare checks if the socket is bound to a local port and address.
    - If the socket is not yet bound (common in UDP or initial TCP setup), this function assigns an ephemeral port (Auto-bind).
    - **unlikely:** This macro hints to the branch predictor that this preparation step usually succeeds or is already done (especially for established TCP connections), optimizing the CPU instruction pipeline.
- **return INDIRECT_CALL_2(sk->sk_prot->sendmsg, tcp_sendmsg, udp_sendmsg, ...);**
  - **Description: Transport Protocol Dispatch (Retpoline Optimization).**
  - **Mechanism:** This is a specialized version of the indirect call macro seen in the previous phase.
  - **Logic:** It explicitly checks the two most common cases for IPv4:
    1. **Is it TCP?** Checks against tcp_sendmsg. If yes, call directly.
    2. **Is it UDP?** Checks against udp_sendmsg. If yes, call directly.
    3. **Fallback:** Only otherwise, use the function pointer sk->sk_prot->sendmsg.
  - **Significance:** This essentially hard-wires the jump to tcp_sendmsg for your analysis flow, bypassing the overhead of Spectre/Meltdown mitigations.

Data Transformation Status

- **Logic Status: Protocol Identified.** The kernel has confirmed this is an IPv4 operation and successfully routed it to the TCP handler.
- **Data Status: Ready for Transport.** The data is still in User Space. The msg structure and iovec are passed to tcp_sendmsg.
    - **Next Step:** Inside tcp_sendmsg, the data will finally be copied into an sk_buff (Kernel Memory) and encapsulated with TCP headers.



(This diagram represents the final dispatch within the socket layer, where inet_sendmsg prepares the socket and efficiently routes the execution flow to the specific transport protocol handler, such as tcp_sendmsg.)

This function is the specific handler for the **IPv4 Protocol Family (AF_INET)**. It serves as the final gateway within the socket layer before handing control over to the specific Transport Layer protocols (TCP or UDP). Its main duties are to ensure the socket is properly bound (prepared) and to dispatch the call to tcp_sendmsg or udp_sendmsg using high-performance optimization techniques.

Part 2. TCP Sendmsg Internal Flow: Full Stack from syscall to IP Layer

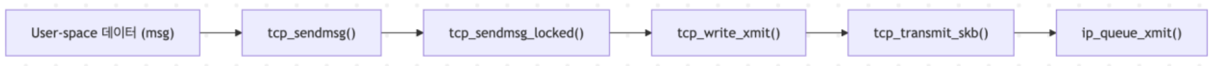
1. Overall Process Overview: TCP Send Flow

TCP data transmission is broadly composed of the following three stages:

- **Data Loading Stage**:
  Data is copied from user-space buffers into kernel-managed sk_buff structures (skb).
  This occurs inside tcp_sendmsg() and its core implementation, tcp_sendmsg_locked().

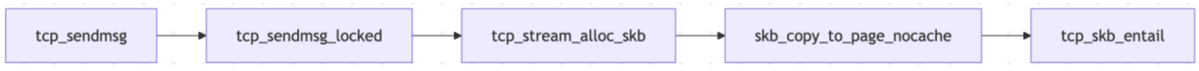
- **Transmission Decision Stage**:
  Based on current network conditions, the kernel decides whether to proceed with actual packet transmission.
  This decision is managed inside tcp_write_xmit().

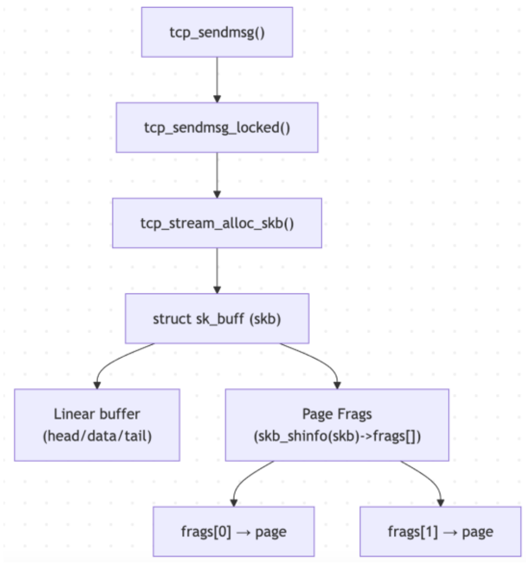
- **Header Assembly and Transmission Stage**:
  The kernel constructs the TCP header, applies necessary TCP options (e.g., Timestamp, SACK), calculates checksums, and finally hands off the packet to the IP layer.
  This final handoff is initiated by tcp_transmit_skb() and followed by ip_queue_xmit().



2. Data Buffering & Enqueuing: From User to sk_buf

The initial step in the TCP transmission process is the **efficient and secure copying** of data from the user space into the kernel memory (specifically, the `struct sk_buff`). This phase begins with the `tcp_sendmsg()` wrapper function, which internally delegates core processing to `tcp_sendmsg_locked()`.





[Initial TCP Data Loading Flow]

2.1 Socket Locking & Delegation (via tcp_sendmsg())

```
/* net/ipv4/tcp.c */
int tcp_sendmsg(struct sock *sk, struct msghdr *msg, size_t size)
{
    int ret;

    lock_sock(sk);                  // Acquire lock: See [Sub Function 01-01] lock_sock
    ret = tcp_sendmsg_locked(sk, msg, size);  // Perform the actual work
    release_sock(sk);               // Release lock and process Backlog

    return ret;
}
```

All TCP transmission starts with tcp_sendmsg() acquiring exclusive ownership of the socket via lock_sock(). While this lock is held:

- **All incoming packets** (e.g. ACKs, FINs) are **deferred** into sk->sk_backlog, not processed immediately.
- **SoftIRQ context is disabled**, ensuring consistency even under preemption or interrupt.

This guarantees that **buffer state**, **send window**, and **congestion window** are updated without interference. Such deferred processing avoids complex reentrancy bugs in the kernel's packet handling path.

[Sub Function 01-01] lock_sock(): Preventing Interleaved State Mutations
The actual locking mechanism is implemented as follows (in net/core/sock.c via lock_sock_nested):

```
/* net/core/sock.c (via lock_sock_nested) */
void lock_sock_nested(struct sock *sk, int subclass)
```

```
{
       // ... (mutex acquire logic)
       spin_lock_bh(&sk->sk_lock.slock); // Acquire spinlock and disable SoftIRQ
       if (sock_owned_by_user_nocheck(sk)) // Check if already owned
              __lock_sock(sk); // If locked, sleep in the wait queue
       sk->sk_lock.owned = 1;  // Mark socket ownership acquired
       spin_unlock_bh(&sk->sk_lock.slock); // Release spinlock and re-enable SoftIRQ
}
```

- lock_sock() ensures that the current process has **exclusive access** to the socket (struct sock) by setting the sk->sk_lock.owned flag.

- While the lock is held, **all incoming ACK packets** are queued into the socket's backlog (sk->sk_backlog), rather than being processed immediately.

- This deferral prevents race conditions and deadlocks, particularly those related to transmission window updates.

- When the lock is released (via release_sock()), all queued packets in the backlog are processed **in batch**, ensuring safe and efficient state transitions.

2-2. Transmission Preparation & Aggregation (via tcp_sendmsg_locked())

```
/* net/ipv4/tcp.c */
int tcp_sendmsg_locked(struct sock *sk, struct msghdr *msg, size_t size)
{
    // ... (ZeroCopy, FastOpen processing omitted) ...

restart:
    // Calculate the target size considering TSO: [Sub Function 02-01]
    mss_now = tcp_send_mss(sk, &size_goal, flags);
    // ... (Main data copying and transmission loop)
```

All data transmission after socket locking is managed within tcp_sendmsg_locked().
This function is responsible for:

- **Preparing the transmission environment**:
  Checks and enables features such as Zero-Copy, Splice, and TCP Fast Open before entering the main loop.

- **Dynamic segment size & aggregation**:
  At every restart: (which is triggered after events like memory wait), the function recalculates the current **Maximum Segment Size (MSS)** and **aggregation target (size_goal)** by calling tcp_send_mss(). This adapts transmission to changes in Path MTU, TCP options, and window size.

- **Efficient data copying & aggregation**:
  Copies user data into sk_buffs, aggregating up to size_goal.

When TSO (TCP Segmentation Offload) is available, this enables batching of multiple MSS-sized chunks (often up to 64KB) in one skb, maximizing throughput.

- **Atomic, consistent operation**:
  All steps occur under the socket lock, ensuring that state changes are atomic and race conditions are prevented.

[Sub Function 02-01] tcp_send_mss()

```
/* net/ipv4/tcp_output.c */
int tcp_send_mss(struct sock *sk, int *size_goal, int flags)
{
    int mss_now;

    // Calculate MSS: See [Sub Function 02-02]
    mss_now = tcp_current_mss(sk);

    // Calculate size_goal: See [Sub Function 02-04]
    *size_goal = tcp_xmit_size_goal(sk, mss_now, !(flags & MSG_OOB));

    return mss_now;
}
```

tcp_send_mss() computes the effective MSS (Maximum Segment Size) by considering the current Path MTU and TCP option length, and then determines the size_goal, which is the number of bytes to be aggregated into a single skb before sending.

- **mss_now:** The upper limit for the data payload in a single segment, reflecting the latest Path MTU and TCP header/option size.
- **size_goal:** The number of bytes to aggregate into a single skb for transmission. When TSO is enabled, this value can be several times the MSS, typically up to 64KB.

[Sub Function 02-02] tcp_current_mss()

Compute Maximum Segment Size.

```
/* net/ipv4/tcp_output.c */
unsigned int tcp_current_mss(struct sock *sk)
{
    const struct tcp_sock *tp = tcp_sk(sk);
    const struct dst_entry *dst = __sk_dst_get(sk);
    u32 mss_now;
    unsigned int header_len;
    struct tcp_out_options opts;
    struct tcp_key key;

    // 1. Start with cached MSS
    mss_now = tp->mss_cache;

    // 2. If Path MTU has changed, resync MSS
    if (dst) {
        u32 mtu = dst_mtu(dst);
        // Recalculate if it is different from previously recorded PMTUs
        if (mtu != inet_csk(sk)->icsk_pmtu_cookie)
            mss_now = tcp_sync_mss(sk, mtu);
    }

    // 3. Calculate current TCP header (including options)
    tcp_get_current_key(sk, &key);
    header_len = tcp_established_options(sk, NULL, &opts, &key) +
                 sizeof(struct tcphdr);  // 합산 -> 실제 TCP 헤더 길이 결정

    // 4. Adjust MSS if header length changed
    if (header_len != tp->tcp_header_len) {
        int delta = (int) header_len - tp->tcp_header_len;
        mss_now -= delta;
    }

    return mss_now;
}
```

- Uses the cached MSS by default.
- If the Path MTU has changed, recalculates MSS for the new value.
- Computes the actual TCP header length (including options) and adjusts the MSS accordingly.

[Sub Function 02-03] tcp_xmit_size_goal()

```
static unsigned int tcp_xmit_size_goal(struct sock *sk, u32 mss_now, int large_allowed)
{
       struct tcp_sock *tp = tcp_sk(sk);
       u32 new_size_goal, size_goal;

       // If large packet aggregation is not allowed (TSO/GSO off), return MSS
       if (!large_allowed)
              return mss_now;

       // 1. new_size_goal: Compute the upper bound for segment size based on
window/buffer
       new_size_goal = tcp_bound_to_half_wnd(tp, sk->sk_gso_max_size);

       // 2. Predict target size: gso_segs * mss_now (number of segments * MSS)
       size_goal = tp->gso_segs * mss_now;

       // 3. Adjust if predicted goal is out of acceptable range
       if (unlikely(new_size_goal < size_goal || new_size_goal >= size_goal + mss_now))
{
              tp->gso_segs = min_t(u16, new_size_goal / mss_now,
                                   sk->sk_gso_max_segs);
              size_goal = tp->gso_segs * mss_now;
       }

       // Always return at least one MS
       return max(size_goal, mss_now);
}
```

Determines the aggregation target (size_goal) for a single skb.
If TSO is unavailable, the target is simply MSS; if TSO is available, aggregates multiple MSS segments into one skb for maximum throughput.

- If GSO/TSO is unavailable, size_goal = MSS.
- Otherwise, size_goal = mss_now * gso_segs (up to window/buffer/driver limits).

A. Data Aggregation Loop & skb Management (via main send loop in tcp_sendmsg_locked())

All user data destined for transmission is processed in the **main data loop**, which continues until the application's data has been fully enqueued into the TCP send buffer.

```
 while (msg_data_left(msg)) {
        int copy = 0;
        skb = tcp_write_queue_tail(sk);         // Get the last skb in the send queue
        if (skb)
             copy = size_goal - skb->len;       // Space left in current skb

        // If no space left or cannot append, allocate new skb
        if (copy <= 0 || !tcp_skb_can_collapse_to(skb)) {
             bool first_skb;

new_segment:
        // Check if send buffer is full
        if (!sk_stream_memory_free(sk))
            goto wait_for_space;            // Block until memory is available

        // Backlog management: periodically release the lock to process incoming packets
        if (unlikely(process_backlog >= 16)) {
            process_backlog = 0;
            if (sk_flush_backlog(sk))
                goto restart;
        }

        // Determine if this is the first packet in the queue
        first_skb = tcp_rtx_and_write_queues_empty(sk);

        // Allocate a new skb for outgoing data
        skb = tcp_stream_alloc_skb(sk, sk->sk_allocation, first_skb);
        if (!skb)
            goto wait_for_space;            // Memory allocation failed

        process_backlog++;

        // (Omitted: skb decryption flag setting, if applicable)

        // Enqueue the new skb to the send queue
        tcp_skb_entail(sk, skb);

        copy = size_goal;                   // Aim to fill up to size_goal in new skb
        // (Omitted: repair mode bookkeeping)
    }
    // ... (Copy user data into skb, advance pointers, continue loop)
}
```

- **Attempt to Merge with Last skb:**
  First, the function tries to append user data to the last skb (packet buffer) in the TCP send queue, if there's still space left (i.e., skb->len < size_goal) and the skb is eligible for merging.

- **Allocate a New skb (new_segment):**
  If merging is not possible (no space or not allowed), or the queue is empty, the function proceeds to allocate a new skb for the next data chunk.

  - **Memory Limit Check:**
    Before allocation, it checks if the socket send buffer is full (sk_stream_memory_free(sk)); if not enough space, it enters a wait state (wait_for_space) until memory is freed.

  - **Backlog Handling for Fairness:**
    To avoid holding the socket lock for too long (starving receive-side packet processing), the loop periodically processes any pending backlog (process_backlog >= 16 triggers sk_flush_backlog(sk)), potentially restarting the main loop if needed.

  - **First Packet Determination:**
    Determines if this is the first packet to be sent (impacts header settings and allocation).

  - **skb Allocation and Queueing:**
    Allocates a new skb (tcp_stream_alloc_skb) with appropriate headroom.
    Enqueues this skb to the send queue using tcp_skb_entail(), which initializes its TCP state and updates memory accounting.
    Sets the copy size to the aggregation target (size_goal) for efficient filling.

[Sub Function 02-04] tcp_skb_entail()

```
/* net/ipv4/tcp.c */
void tcp_skb_entail(struct sock *sk, struct sk_buff *skb)
{

       struct tcp_sock *tp = tcp_sk(sk);
       struct tcp_skb_cb *tcb = TCP_SKB_CB(skb);
       // 1. Initialize sequence numbers
       tcb->seq      = tcb->end_seq = tp->write_seq;
       // 2. Default ACK flag
       tcb->tcp_flags = TCPHDR_ACK;
       // 3. Reset skb header state
       __skb_header_release(skb);
       // 4. Enqueue to TCP send queue
       tcp_add_write_queue_tail(sk, skb);
```

```
        // 5. Update socket memory accounting
        sk_wmem_queued_add(sk, skb->truesize);
        sk_mem_charge(sk, skb->truesize);
        // 6. Clear forced push if previously set
        if (tp->nonagle & TCP_NAGLE_PUSH)
                tp->nonagle &= ~TCP_NAGLE_PUSH;
        // 7. Check for slow start after idle
        tcp_slow_start_after_idle_check(sk);
}
```

- **Sequence Number Assignment:**
  Sets the starting and ending sequence numbers for the new skb, based on the socket's current write sequence.
- **ACK Flag Setup:**
  Sets the default TCP header flags (usually includes ACK).
- **Queue Integration:**
  Adds the new skb to the end of the TCP send queue (write queue), making it available for congestion control, retransmission, and packet scheduling.
- **Memory Accounting:**
  Updates both per-socket and global memory usage statistics for the newly allocated skb.
- **Nagle/Idle Handling:**
  Resets push flags and checks whether slow start should be triggered after an idle period.

B. Data Copy: User Buffer to skb (Copy to Kernel, Fragment Management)

The most common data path for TCP transmission involves copying user-space data into kernel skb buffers, managing memory at the granularity of page fragments for efficiency and scalability.

```
if (copy > msg_data_left(msg))
                copy = msg_data_left(msg);
if (zc == 0) {
                        bool merge = true;
                        int i = skb_shinfo(skb)->nr_frags;
                        struct page_frag *pfrag = sk_page_frag(sk);
                        // 1. ensure sufficient page fragment space (see [Sub Function 02-
05])
                        if (!sk_page_frag_refill(sk, pfrag))
                                goto wait_for_space;
                        // 2. Decide if we can merge data with the previous fragment
                        if (!skb_can_coalesce(skb, i, pfrag->page, pfrag->offset)) {
                                if (i >= READ_ONCE(net_hotdata.sysctl_max_skb_frags)) {
                                                tcp_mark_push(tp, skb);
                                                goto new_segment;
                                }
                                merge = false;
                        }
                        // 3. Limit copy to available page fragment size
                        copy = min_t(int, copy, pfrag->size - pfrag->offset);
                        // 4. Handle zcopy fallback if needed (omitted for brevity)
                        if (unlikely(skb_zcopy_pure(skb) || skb_zcopy_managed(skb))) {
                                if (tcp_downgrade_zcopy_pure(sk, skb))
                                                goto wait_for_space;
                                skb_zcopy_downgrade_managed(skb);
                        }

                        // 5. Reserve send buffer memory for this copy
                        copy = tcp_wmem_schedule(sk, copy);
                        if (!copy)
                                goto wait_for_space;

                        // 6. Copy user data into the kernel skb fragment
                        err = skb_copy_to_page_nocache(sk, &msg->msg_iter, skb,
                                                       pfrag->page,
                                                       pfrag->offset,
                                                       copy);
                        if (err)
                                goto do_error;

                        // 7. Update skb metadata and fragment info
                        if (merge) {
                                skb_frag_size_add(&skb_shinfo(skb)->frags[i - 1], copy);
                        } else {
                                skb_fill_page_desc(skb, i, pfrag->page,
                                                   pfrag->offset, copy);
                                page_ref_inc(pfrag->page);
                        }
                        pfrag->offset += copy;
                }
                // ... (ZeroCopy, Splice 처리 생략) ...
```

- **Page Fragment Reservation:**
  Use sk_page_frag_refill() to ensure that enough memory (page fragment) is available for the copy operation. If allocation fails, the kernel enters a wait state for buffer availability.

- **Fragment Merge Decision:**
  Attempt to coalesce (merge) the new data into the last fragment if possible, minimizing fragment count. If merging isn't possible (due to memory limits or fragment rules), allocate a new fragment.
  If the fragment limit (max_skb_frags) is reached, push the current skb and start a new segment.

- **Memory Reservation:**
  Before copying, check and reserve send buffer space for the requested copy size using tcp_wmem_schedule(). If there is insufficient memory, wait for space.

- **Actual Data Copy:**
  Copy user data from the application's buffer to the kernel page using

skb_copy_to_page_nocache().
Any error in copying aborts the process and triggers error handling.

- **Metadata Update:**
  After copying, update the skb's fragment array, adjust the fragment's length or create a new descriptor, and increment the page reference count if needed.
  Advance the page fragment offset to reflect the newly written data.

- **Optional Push:**
  If the skb is now full or meets push conditions, initiate sending via __tcp_push_pending_frames().

[Sub Function 02-05] sk_page_frag_refill()

```
bool sk_page_frag_refill(struct sock *sk, struct page_frag *pfrag)
{
    if (likely(skb_page_frag_refill(32U, pfrag, sk->sk_allocation)))
        return true;
    sk_enter_memory_pressure(sk);         // Mark the socket as memory-pressured
    sk_stream_moderate_sndbuf(sk);        // Attempt to shrink sndbuf for memory relief
    return false;
}
```

- **Ensures Sufficient Fragment Capacity:**
  Checks if the current socket page fragment has at least 32 bytes available. If so, returns true for immediate use.

- **Handles Allocation Failure:**
  If not enough space is available, the function:
  - Notifies the kernel that the socket is under memory pressure (sk_enter_memory_pressure()).
  - Attempts to shrink the send buffer size to relieve overall memory load.
  - Returns false, signaling the caller to wait for space.

C. State Update

After each successful data copy, TCP updates key internal state variables to ensure reliable transmission and proper sequencing

```
if (!copied)                               // If this is the first packet of this
sendmsg() call
    TCP_SKB_CB(skb)->tcp_flags &= ~TCPHDR_PSH; // Temporarily clear PSH flag

WRITE_ONCE(tp->write_seq, tp->write_seq + copy); // Advance global write sequence by
'copy' bytes
TCP_SKB_CB(skb)->end_seq += copy;          // Advance end sequence for the current skb
tcp_skb_pcount_set(skb, 0);                // Update processed byte count (not always
used here)
```

```
copied += copy;                        // Accumulate total copied bytes for this
call
```

- **First Packet Check:**
  If this is the first skb filled during this sendmsg() call, the PSH flag is cleared to delay pushing, allowing further aggregation.
- **Sequence Advancement:**
  The write sequence (tp->write_seq) and the end sequence in the skb are both incremented by the number of bytes copied, ensuring correct tracking for retransmission and ACKs.
- **Byte Count Tracking:**
  The copied variable accumulates the total number of bytes successfully enqueued during this call.

D. Transmission Decision

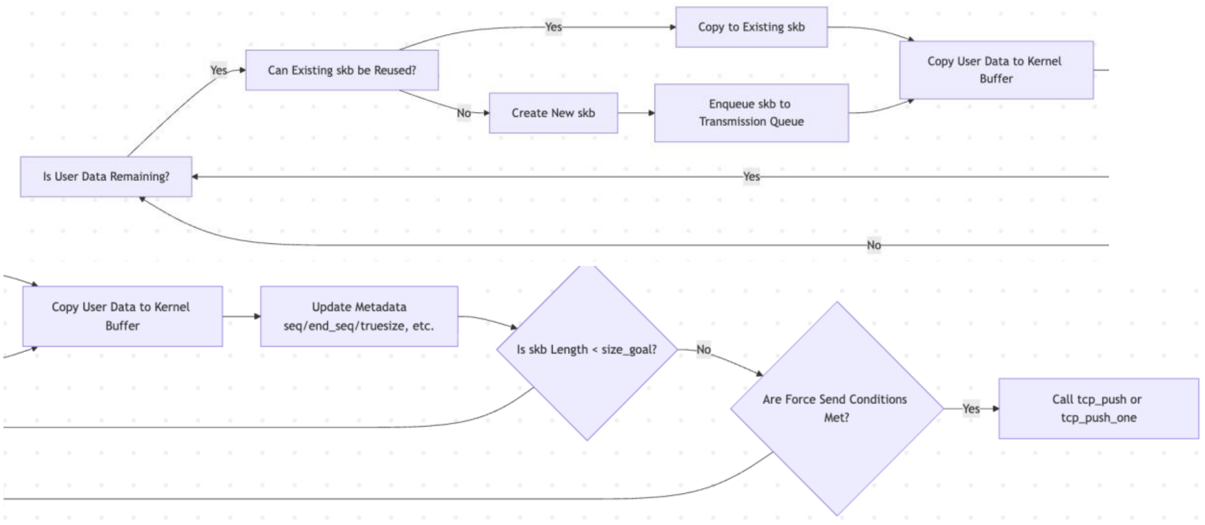
```
// 1. All user data has been enqueued: time to exit
if (!msg_data_left(msg)) {
    if (unlikely(flags & MSG_EOR))
        TCP_SKB_CB(skb)->eor = 1;       // Mark end-of-record if requested
    goto out;                           // Exit main loop and proceed to send completion
}

// 2. Not enough data yet: keep filling
if (skb->len < size_goal || (flags & MSG_OOB) || unlikely(tp->repair))
    continue;                           // Continue to next loop iteration (aggregate
more data)

// 3. Forced push conditions met
if (forced_push(tp)) {
    tcp_mark_push(tp, skb);             // Set PSH flag for immediate transmission
    __tcp_push_pending_frames(sk, mss_now, TCP_NAGLE_PUSH); // Trigger transmission
} else if (skb == tcp_send_head(sk)) {
    tcp_push_one(sk, mss_now);          // If this skb is at the head, send one packet
}
continue;
```

- **Loop Exit (All Data Sent):**
  If there is no user data left (msg_data_left(msg) == 0), the function marks end-of-record if needed and breaks out of the main send loop for post-processing.
- **Continue Aggregating:**
  If the current skb is not yet filled up to the aggregation target (size_goal), or if special flags (MSG_OOB, repair mode) are set, the function continues to accumulate more data before sending.
- **Immediate/Forced Transmission:**
  If forced push criteria are met (forced_push(tp) is true)—such as reaching certain queue or time thresholds—the kernel marks the PSH flag and pushes all pending

frames for transmission.
Alternatively, if the current skb is the first in the send queue, tcp_push_one() is called to send it immediately.

2.3 Wait-for-Space Handling & Push to Network

When the kernel's send buffer is full during data transmission, TCP must pause further copying and wait for memory to become available.


Is User Data Remaining?
Yes
Can Existing skb be Reused?
Yes
Copy to Existing skb
No
Create New skb
Enqueue skb to Transmission Queue
Copy User Data to Kernel Buffer
Yes
No
Copy User Data to Kernel Buffer
Update Metadata seq/end_seq/truesize, etc.
Is skb Length < size_goal?
No
Are Force Send Conditions Met?
Yes
Call tcp_push or tcp_push_one

```
wait_for_space:
    // 1. Mark NOSPACE status (socket out of space)
    set_bit(SOCK_NOSPACE, &sk->sk_socket->flags);

    // 2. Remove any empty skb at the tail (housekeeping)
    tcp_remove_empty_skb(sk);

    // 3. Try to send any data already copied, if any
    if (copied)
        tcp_push(sk, flags & ~MSG_MORE, mss_now, TCP_NAGLE_PUSH, size_goal);

    // 4. Sleep until space is available
    err = sk_stream_wait_memory(sk, &timeo);
    if (err != 0)
        goto do_error;

    // 5. On wakeup: Recalculate MSS/aggregation in case network conditions changed
    mss_now = tcp_send_mss(sk, &size_goal, flags);
```

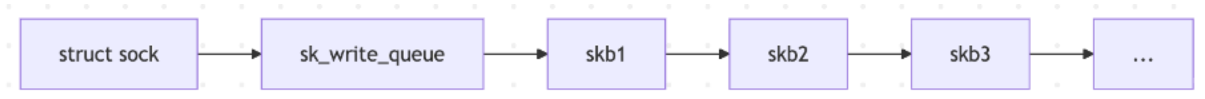
- **Mark NOSPACE:**
  Signals to the kernel that the socket has run out of send buffer space.
- **Cleanup:**
  Removes any empty skb from the send queue tail to maximize available space.

- **Intermediate Send:**
  If data has already been copied to the send buffer, attempts a partial send via tcp_push() before sleeping.

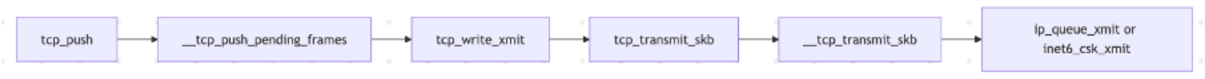
- **Sleep:**
  Uses sk_stream_wait_memory() to block the process until more send buffer memory is available.

- **Resume:**
  Upon waking, re-evaluates MSS and size_goal (since MTU/window might have changed), then continues the main loop

[Sub Function 02-06] tcp_push(): The Transmission Controller



The tcp_push() function acts as the control tower, deciding **when** and **how** to trigger the actual transmission of data queued in the TCP write queue.



```
void tcp_push(struct sock *sk, int flags, int mss_now, int nonagle, int size_goal)
{
    struct tcp_sock *tp = tcp_sk(sk);
    struct sk_buff *skb;

    // 1. Identify the skb to transmit (last in the queue)
    skb = tcp_write_queue_tail(sk);
    if (!skb)
        return;

    // 2. Set PSH (push) flag if ready to send now
    if (!(flags & MSG_MORE) || forced_push(tp))
        tcp_mark_push(tp, skb);

    // 3. Handle urgent (URG) flag if set
    tcp_mark_urg(tp, flags);

    // 4. Apply autocork (auto-delay) if needed to batch small packets
    if (tcp_should_autocork(sk, skb, size_goal)) {
        if (!test_bit(TSQ_THROTTLED, &sk->sk_tsq_flags)) {
            NET_INC_STATS(sock_net(sk), LINUX_MIB_TCPAUTOCORKING);
            set_bit(TSQ_THROTTLED, &sk->sk_tsq_flags);
            smp_mb__after_atomic();
        }
        if (refcount_read(&sk->sk_wmem_alloc) > skb->truesize)
            return; // Still too full, skip send for now
    }

    // 5. Select corking mode if MSG_MORE is set
    if (flags & MSG_MORE)
```

```
        nonagle = TCP_NAGLE_CORK;

    // 6. Finally, trigger transmission
    __tcp_push_pending_frames(sk, mss_now, nonagle);
}
```

- **Target skb Selection:**
  Identifies the skb to send (usually the last in the write queue).
- **Push/URG Flag Setting:**
  Sets the PSH or URG flags if immediate transmission is required.
- **Autocork (Automatic Delay):**
  May delay sending small packets for batching efficiency, unless the send buffer is about to overflow.
- **Actual Send Trigger:**
  Calls __tcp_push_pending_frames() to send out queued packets, depending on current cork/nagle settings.

[Sub Function 02-07] __tcp_push_pending_frames(): Starting Real Transmission

It acts as a transmission starting point for actual transmitting sk_buff.

```
void __tcp_push_pending_frames(struct sock *sk, unsigned int cur_mss, int nonagle)
{
    // 1. Only send if socket is not closed
    if (unlikely(sk->sk_state == TCP_CLOSE))
        return;

    // 2. Call tcp_write_xmit() to handle real data transmission
    if (tcp_write_xmit(sk, cur_mss, nonagle, 0, sk_gfp_mask(sk, GFP_ATOMIC)))
        tcp_check_probe_timer(sk); // If zero window detected, start probe timer
}
```

- **Ensures Socket is Open:**
  No data is sent if the socket is closed.
- **Triggers tcp_write_xmit():**
  Moves packets from the write queue to the actual network stack for transmission, handling congestion, pacing, and window logic.
- **Zero Window Handling:**
  If the receiver's window is full, starts a probe timer to periodically check when sending can resume.

[Sub Function 02-08] tcp_write_xmit(): Actual Data Output

tcp_write_xmit() is responsible for pulling packets from the TCP write queue and transmitting them to the IP layer, ensuring that all congestion control, pacing, and memory constraints are satisfied.

1. Preparation & Path MTU Probing

```
/* net/ipv4/tcp_output.c */
static bool tcp_write_xmit(struct sock *sk, unsigned int mss_now, int nonagle,
                           int push_one, gfp_t gfp)
{
        struct tcp_sock *tp = tcp_sk(sk);
        struct sk_buff *skb;
        unsigned int tso_segs, sent_pkts;
        u32 cwnd_quota, max_segs;
        int result;
        bool is_cwnd_limited = false, is_rwnd_limited = false;

        sent_pkts = 0;
        // Refresh the transmission timestamp
        tcp_mstamp_refresh(tp);
        if (!push_one) {
                result = tcp_mtu_probe(sk);  // Probe for current Path MTU
                if (!result) {               // Cannot send (PMTU discovery failed)
                        return false;
                } else if (result > 0) {
                        sent_pkts = 1;        // MTU probe sent one packet
                }
        }
```

- **MTU Probing:**
  Checks the maximum packet size allowed on the path, ensuring segments are not unnecessarily fragmented.
- **Early Exit:**
  If Path MTU probing blocks sending, the function exits immediately.

2. Main Transmission Loop

Processes each skb (send buffer) at the head of the TCP write queue as long as conditions allow.

```
        max_segs = tcp_tso_segs(sk, mss_now);  // Maximum TSO segments allowed
        // 2. Main Transmission Loop
        while ((skb = tcp_send_head(sk))) {
                unsigned int limit;
                int missing_bytes;

                // TCP repair mode only updates without actual transmission
                if (unlikely(tp->repair) && tp->repair_queue == TCP_SEND_QUEUE) {
                        tp->tcp_wstamp_ns = tp->tcp_clock_cache;
                        skb_set_delivery_time(skb, tp->tcp_wstamp_ns, SKB_CLOCK_MONOTONIC);
                        list_move_tail(&skb->tcp_tsorted_anchor, &tp->tsorted_sent_queue);
                        tcp_init_tso_segs(skb, mss_now);
                        goto repair; /* Skip network transmission */
                }
                // 2-1. Pacing Check
                if (tcp_pacing_check(sk))
                        break;
                // 2-2. Congestion Window (CWND) Check
                cwnd_quota = tcp_cwnd_test(tp);  // Congestion window
                if (!cwnd_quota) {
                        if (push_one == 2)
                                cwnd_quota = 1;
                        else
                                break;
                }
                cwnd_quota = min(cwnd_quota, max_segs);
                // Attempt to Expand skb
                missing_bytes = cwnd_quota * mss_now - skb->len;
                if (missing_bytes > 0)
                        tcp_grow_skb(sk, skb, missing_bytes);
                // 2-3. Receive Window (RWND) Check
                tso_segs = tcp_set_skb_tso_segs(skb, mss_now);

                if (unlikely(!tcp_snd_wnd_test(tp, skb, mss_now))) {
                        is_rwnd_limited = true;
                        break;
                }
                // 2-4. Efficiency Tests (Nagle/TSO)
                if (tso_segs == 1) {
                        if (unlikely(!tcp_nagle_test(tp, skb, mss_now,
                                                     (tcp_skb_is_last(sk, skb) ?
                                                      nonagle : TCP_NAGLE_PUSH))))
                                break;
                } else {
                        if (!push_one &&
                            tcp_tso_should_defer(sk, skb, &is_cwnd_limited,
                                                 &is_rwnd_limited, max_segs))
                                break;
                }
```

A. Pacing Check: If packet pacing is enabled, may temporarily defer sending to limit transmission rate.

B. Congestion Window (CWND) Check: Only sends as many packets as allowed by the current congestion window.

C. Receive Window (RWND) Check: Stops sending if the receiver cannot accept more data.

D. Efficiency Test (Nagle/TSO): If sending small packets, checks Nagle's algorithm (tcp_nagle_test()). If sending with TSO, may defer sending for batching (tcp_tso_should_defer()).

3. Packet Preparation & Splitting (Segmentation)

```
limit = mss_now;
if (tso_segs > 1 && !tcp_urg_mode(tp))
    limit = tcp_mss_split_point(sk, skb, mss_now, cwnd_quota, nonagle);

if (skb->len > limit && unlikely(tso_fragment(sk, skb, limit, mss_now, gfp)))
    break; // Split skb if it exceeds the allowed segment size
```

If the skb is larger than the current transmission limit, it is split into smaller segments for transmission.

4. Final Transmission

```
		if (tcp_small_queue_check(sk, skb, 0))
			break;

		if (TCP_SKB_CB(skb)->end_seq == TCP_SKB_CB(skb)->seq)
			break;
		// Attach TCP header/options and send to IP layer [Sub Function 02-09]
		if (unlikely(tcp_transmit_skb(sk, skb, 1, gfp)))
			break;
```

- **Pre-Checks:**
  Confirms the packet is not too small to send and actually contains new data.

- **Header Attachment & Transmission:**
  Adds TCP header/options, then passes the skb to the network stack (IP layer). If transmission fails, stops the loop.

5. Post-Transmission Handling

```
	if (is_rwnd_limited)
		tcp_chrono_start(sk, TCP_CHRONO_RWND_LIMITED);
	else
		tcp_chrono_stop(sk, TCP_CHRONO_RWND_LIMITED);

	is_cwnd_limited |= (tcp_packets_in_flight(tp) >= tcp_snd_cwnd(tp));
	if (likely(sent_pkts || is_cwnd_limited))
		tcp_cwnd_validate(sk, is_cwnd_limited);

	if (likely(sent_pkts)) {
		if (tcp_in_cwnd_reduction(sk))
			tp->prr_out += sent_pkts;

		if (push_one != 2)
			tcp_schedule_loss_probe(sk, false);
		return false;
	}
	return !tp->packets_out && !tcp_write_queue_empty(sk);
}
```

- **Chrono/State Management:**
  Updates TCP state for receive window blocking or congestion.

- **Congestion Control Update:**
  Validates congestion window status and possibly triggers loss detection logic or timers.

- **Return:**
  Returns false if packets were sent, or if more work remains, true otherwise.

[Sub Function 02-09] tcp_transmit_skb(): Transmission Initiation

```
/* net/ipv4/tcp_output.c */
static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
                                  gfp_t gfp_mask)
{
       // Internal implementation [Sub Function 02-10]
       return __tcp_transmit_skb(sk, skb, clone_it, gfp_mask,
                                       tcp_sk(sk)->rcv_nxt);
}
```

This function acts as a wrapper for the actual transmission logic in __tcp_transmit_skb(). It prepares all necessary arguments, especially the ACK number (using tcp_sk(sk)->rcv_nxt), and passes the skb (socket buffer) for final header construction and handoff to the network stack.

[Sub Function 02-10] __tcp_transmit_skb(): Final TCP Header Construction and Transmission

This is the core function for preparing, formatting, and pushing a TCP segment to the IP layer.

A. skb Cloning (if needed)

```
/* net/ipv4/tcp_output.c */
static int __tcp_transmit_skb(struct sock *sk, struct sk_buff *skb,
                              int clone_it, gfp_t gfp_mask, u32 rcv_nxt)
{
       const struct inet_connection_sock *icsk = inet_csk(sk);
       struct inet_sock *inet;
       struct tcp_sock *tp;
       struct tcp_skb_cb *tcb;
       struct tcp_out_options opts;
       unsigned int tcp_options_size, tcp_header_size;
       struct sk_buff *oskb = NULL;
       struct tcp_key key;
       struct tcphdr *th;
       u64 prior_wstamp;
       int err;

       BUG_ON(!skb || !tcp_skb_pcount(skb));
       tp = tcp_sk(sk);
       prior_wstamp = tp->tcp_wstamp_ns;
       tp->tcp_wstamp_ns = max(tp->tcp_wstamp_ns, tp->tcp_clock_cache);
       skb_set_delivery_time(skb, tp->tcp_wstamp_ns, SKB_CLOCK_MONOTONIC);
       // 1. Cloning Process
       if (clone_it) {
              oskb = skb;    // The original skb that will remain in the retransmission
queue

              tcp_skb_tsorted_save(oskb) {
                     if (unlikely(skb_cloned(oskb)))
                            skb = pskb_copy(oskb, gfp_mask);     // Deep copy of
actual data
                     else
```

```
                                     skb = skb_clone(oskb, gfp_mask);     // Shallow copy
(pointer only)
               } tcp_skb_tsorted_restore(oskb);

               if (unlikely(!skb))
                      return -ENOBUFS;
               skb->dev = NULL;      // Reset device information as this is a new packet
        }

        inet = inet_sk(sk);
        tcb = TCP_SKB_CB(skb);
        memset(&opts, 0, sizeof(opts));   // Initialize TCP options
}
```

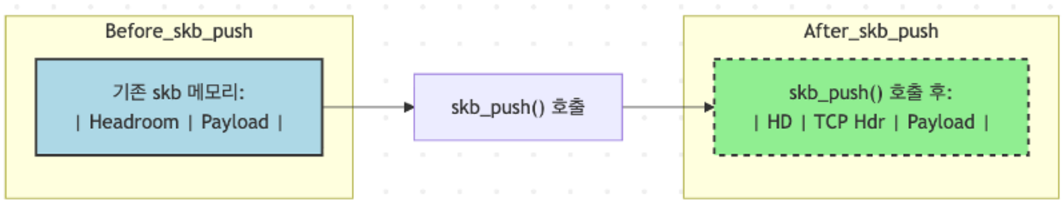
If retransmission or zero-copy is in use, the function may need to **clone** the skb so the original remains in the retransmission queue, while a duplicate is handed off for transmission.

- If clone_it is true, the skb is duplicated (via skb_clone() or pskb_copy() depending on the state).
- The clone is detached from any device and its metadata reset.

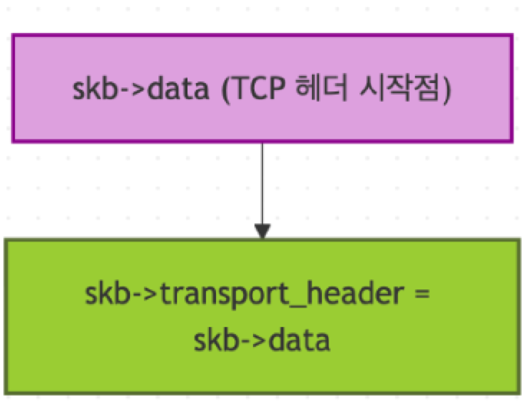
B. TCP Options Calculation and Header Space Allocation

```
    // 2. Option Calculation and Space Reservation
    tcp_get_current_key(sk, &key);
    if (unlikely(tcb->tcp_flags & TCPHDR_SYN)) {
        tcp_options_size = tcp_syn_options(sk, skb, &opts, &key);
    } else {
        tcp_options_size = tcp_established_options(sk, skb, &opts, &key);

        if (tcp_skb_pcount(skb) > 1)
            tcb->tcp_flags |= TCPHDR_PSH;
    }
    tcp_header_size = tcp_options_size + sizeof(struct tcphdr);

    skb->ooo_okay = sk_wmem_alloc_get(sk) < SKB_TRUESIZE(1) ||
                    tcp_rtx_queue_empty(sk);

    skb->pfmemalloc = 0;

    skb_push(skb, tcp_header_size);       // Move data pointer forward to reserve header
space
    skb_reset_transport_header(skb);      // Mark the start of the TCP header
```

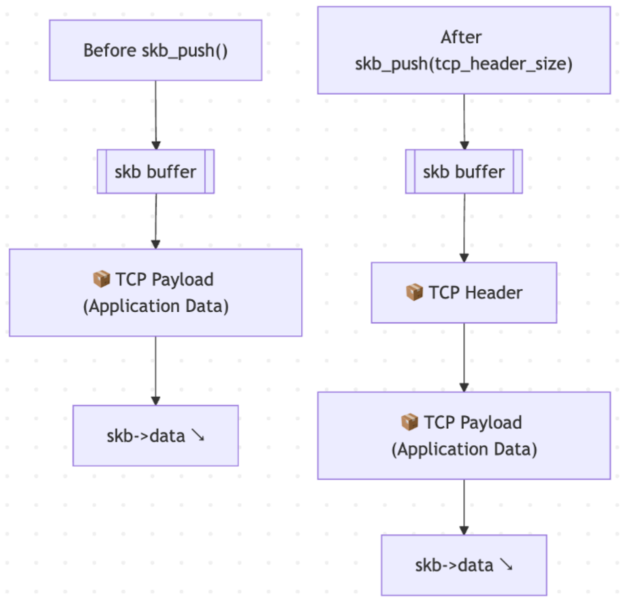
- Computes the size of TCP options (SACK, Timestamp, Window Scale, etc.) to be included after the basic header.



- Uses skb_push() to reserve space at the front of the skb for the entire TCP header (options + struct tcphdr).


skb->data (TCP 헤더 시작점)
skb->transport_header = skb->data

- The header start is marked with skb_reset_transport_header().


Before skb_push()
skb buffer
📦 TCP Payload (Application Data)
skb->data ↘
After skb_push(tcp_header_size)
skb buffer
📦 TCP Header
📦 TCP Payload (Application Data)
skb->data ↘

C. skb Ownership and Orphan Handling

```
        skb_orphan(skb);
        skb->sk = sk;
        skb->destructor = skb_is_tcp_pure_ack(skb) ? __sock_wfree : tcp_wfree;
        refcount_add(skb->truesize, &sk->sk_wmem_alloc);
```

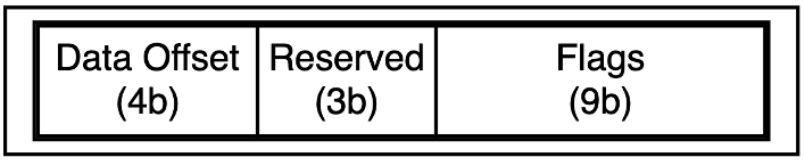
```
    skb_set_dst_pending_confirm(skb, READ_ONCE(sk->sk_dst_pending_confirm))
```

- Calls skb_orphan(skb) to drop any previous owner (as clones are now detached).
- Sets skb->sk = sk so the buffer is associated with the correct socket.
- Sets the appropriate destructor (tcp_wfree or __sock_wfree) for memory management when transmission completes.
- Increments the socket's send buffer accounting (sk->sk_wmem_alloc).

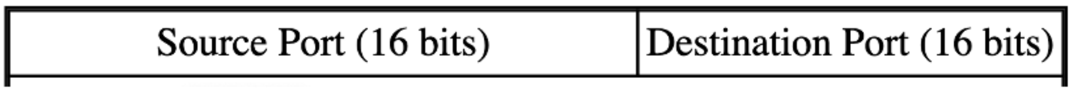
D. TCP Header Construction

```
// 4-1. Set TCP Pointer
th = (struct tcphdr *)skb->data;
// 4-2. Write Main Fields
th->source = inet->inet_sport;          // Source Port
th->dest = inet->inet_dport;            // Destination Port
th->seq = htonl(tcb->seq);              // Sequence Number (Big Endian Conversion)
th->ack_seq = htonl(rcv_nxt);           // ACK Number (Piggybacking)
// Set Header Length and Flags
*(((__be16 *)th) + 6) = htons(((tcp_header_size >> 2) << 12) |
                                tcb->tcp_flags);

th->check = 0;
th->urg_ptr = 0;

if (unlikely(tcp_urg_mode(tp) && before(tcb->seq, tp->snd_up))) {
    if (before(tp->snd_up, tcb->seq + 0x10000)) {
        th->urg_ptr = htons(tp->snd_up - tcb->seq);
        th->urg = 1;
    } else if (after(tcb->seq + 0xFFFF, tp->snd_nxt)) {
        th->urg_ptr = htons(0xFFFF);
        th->urg = 1;
    }
}

skb_shinfo(skb)->gso_type = sk->sk_gso_type;
```

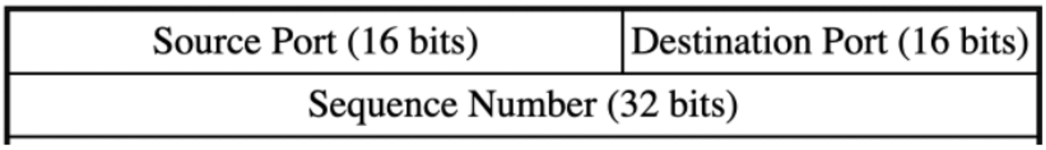
Directly writes all critical fields in the TCP header:



- **Source/Destination Port**: Uses stored values from inet_sock.



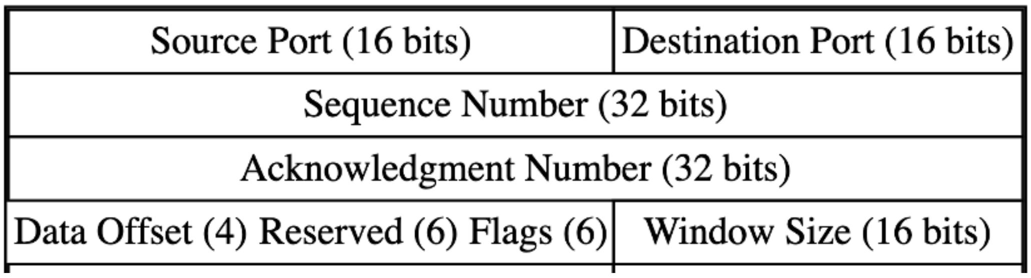
- **Sequence/Acknowledgment Numbers**: Sets based on connection state and current transmission.



Source Port (16 bits)	Destination Port (16 bits)
Sequence Number (32 bits)	

- **Header Length & Flags**: Combines options size and active TCP flags (SYN, ACK, PSH, FIN, etc.).

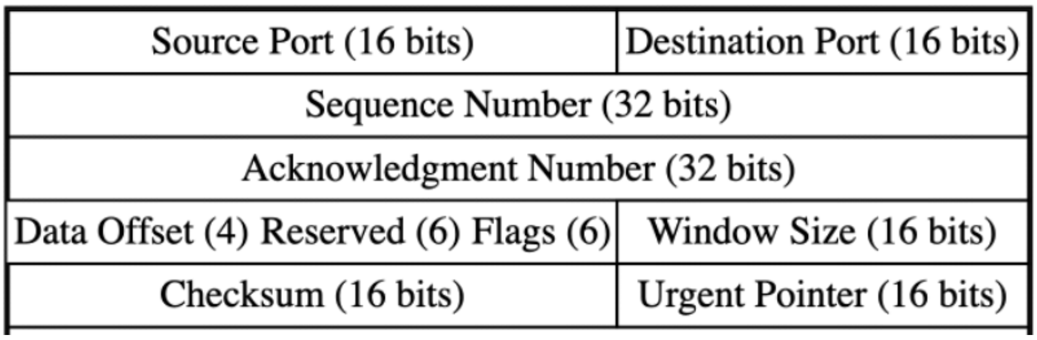
- **Window Size**: Advertises available receive buffer space (RWND).



Source Port (16 bits)	Destination Port (16 bits)
Sequence Number (32 bits)	
Acknowledgment Number (32 bits)	
Data Offset (4) Reserved (6) Flags (6)	Window Size (16 bits)

- **Checksum**: Set to zero here, will be computed by the IP layer.

- **URG Pointer**: Only set if Out-of-Band data is being sent.



Source Port (16 bits)	Destination Port (16 bits)
Sequence Number (32 bits)	
Acknowledgment Number (32 bits)	
Data Offset (4) Reserved (6) Flags (6)	Window Size (16 bits)
Checksum (16 bits)	Urgent Pointer (16 bits)

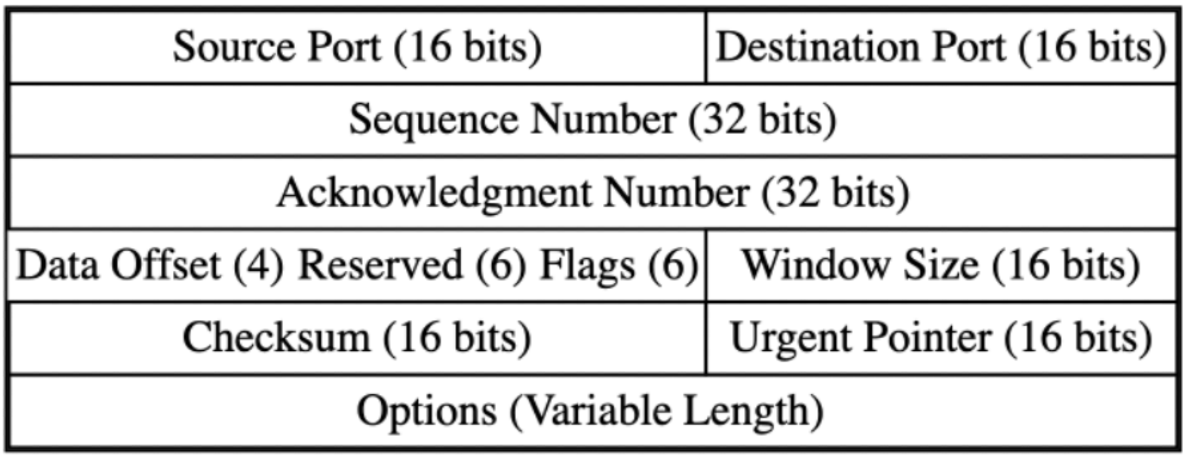
E. Attach Window Size and Options

```
    if (likely(!(tcb->tcp_flags & TCPHDR_SYN))) {
        th->window = htons(tcp_select_window(sk));  // Calculate currently available
receive space
        tcp_ecn_send(sk, skb, th, tcp_header_size);     // Set ECN (Congestion
Notification) bits
    } else {
        th->window = htons(min(tp->rcv_wnd, 65535U));
    }
    // Attach TCP options (See [Sub Function 02-11])
    tcp_options_write(th, tp, NULL, &opts, &key);  // Write the previously calculated
options after the header

    if (tcp_key_is_md5(&key)) {
#ifdef CONFIG_TCP_MD5SIG
        sk_gso_disable(sk);
        tp->af_specific->calc_md5_hash(opts.hash_location,
                                       key.md5_key, sk, skb);
#endif
    } else if (tcp_key_is_ao(&key)) {
        int err;
        err = tcp_ao_transmit_skb(sk, skb, key.ao_key, th,
                                  opts.hash_location);
        if (err) {
            kfree_skb_reason(skb, SKB_DROP_REASON_NOT_SPECIFIED);
            return -ENOMEM;
        }
    }
}
```

- If the packet is not a SYN, sets the receive window size using tcp_select_window().
- Handles ECN (Explicit Congestion Notification) flags as needed.
- Copies the previously-calculated options block into the header using tcp_options_write().
- If MD5 or AO (TCP Authentication Option) is configured for this connection, appends the corresponding authentication signature.

[Sub Function 02-11] tcp_options_write(): Constructing TCP Option Fields

tcp_options_write() assembles and writes all enabled TCP options directly after the main TCP header, based on connection state and negotiated features. This function ensures that advanced TCP features (security, performance, compatibility) are correctly signaled and utilized.


Source Port (16 bits)
Destination Port (16 bits)
Sequence Number (32 bits)
Acknowledgment Number (32 bits)
Data Offset (4) Reserved (6) Flags (6)
Window Size (16 bits)
Checksum (16 bits)
Urgent Pointer (16 bits)
Options (Variable Length)

```
 /* net/ipv4/tcp_output.c */
static void tcp_options_write(struct tcphdr *th, struct tcp_sock *tp,
                              const struct tcp_request_sock *tcprsk,
                              struct tcp_out_options *opts,
                              struct tcp_key *key)
{
        // A. Set up working pointer (starts right after the fixed TCP header)
        __be32 *ptr = (__be32 *)(th + 1);
        u16 options = opts->options;

        // B. Authentication Options (MD5/TCP-AO)
        if (tcp_key_is_md5(key)) {
                *ptr++ = htonl((TCPOPT_NOP << 24) | (TCPOPT_NOP << 16) |
                               (TCPOPT_MD5SIG << 8) | TCPOLEN_MD5SIG);
                /* overload cookie hash location */
                opts->hash_location = (__u8 *)ptr;
                ptr += 4;
        } else if (tcp_key_is_ao(key)) {
                ptr = process_tcp_ao_options(tp, tcprsk, opts, key, ptr);
        }
        // C. MSS (Maximum Segment Size)
        if (unlikely(opts->mss)) {
                *ptr++ = htonl((TCPOPT_MSS << 24) |
                               (TCPOLEN_MSS << 16) |
                               opts->mss);
        }

        // D. Timestamps (TS)
        if (likely(OPTION_TS & options)) {
                if (unlikely(OPTION_SACK_ADVERTISE & options)) {
                        *ptr++ = htonl((TCPOPT_SACK_PERM << 24) |
                                       (TCPOLEN_SACK_PERM << 16) |
                                       (TCPOPT_TIMESTAMP << 8) |
                                       TCPOLEN_TIMESTAMP);
                        options &= ~OPTION_SACK_ADVERTISE;
                } else {
                        *ptr++ = htonl((TCPOPT_NOP << 24) |
                                       (TCPOPT_NOP << 16) |
                                       (TCPOPT_TIMESTAMP << 8) |
                                       TCPOLEN_TIMESTAMP);
                }
                *ptr++ = htonl(opts->tsval); // My current timestamp value
                *ptr++ = htonl(opts->tsecr); // The most recent timestamp I received from
the peer
        }

        // E. SACK Permitted
        if (unlikely(OPTION_SACK_ADVERTISE & options)) {
                *ptr++ = htonl((TCPOPT_NOP << 24) |
                               (TCPOPT_NOP << 16) |
                               (TCPOPT_SACK_PERM << 8) |
                               TCPOLEN_SACK_PERM);
        }
}
```

A. Workspace Setup

- TCP options are written immediately after the 20-byte base header.
- The pointer is cast to __be32 * for aligned, efficient option insertion.

B. Authentication Options (MD5, TCP-AO)

- For security-aware connections (e.g., BGP sessions), attaches the MD5 or TCP-AO signature field.
- Updates opts->hash_location so later routines can fill in the correct hash.

C. MSS (Maximum Segment Size) Option

- Included only in SYN packets, this field advertises the sender's supported MSS.
- Ensures path compatibility and avoids fragmentation.

D. Timestamp Option (RFC 1323)

- Widely used for RTT estimation (for RTO calculation) and PAWS (Protection Against Wrapped Sequence numbers).
- Writes 12 bytes (with 2 NOPs for alignment): Kind, Len, TSval (local timestamp), TSecr (echo of remote timestamp).

E. SACK Permitted

- Indicates whether SACK (Selective Acknowledgment) is allowed, required for robust retransmission.
- Present only in SYN exchange.

```
        // F. Window Scale
        if (unlikely(OPTION_WSCALE & options)) {
               *ptr++ = htonl((TCPOPT_NOP << 24) |
                              (TCPOPT_WINDOW << 16) |
                              (TCPOLEN_WINDOW << 8) |
                              opts->ws);
        }

        // G. SACK (Selective Acknowledgement)
        if (unlikely(opts->num_sack_blocks)) {
               struct tcp_sack_block *sp = tp->rx_opt.dsack ?
                      tp->duplicate_sack : tp->selective_acks;
               int this_sack;

               *ptr++ = htonl((TCPOPT_NOP  << 24) |
                              (TCPOPT_NOP  << 16) |
                              (TCPOPT_SACK <<  8) |
                              (TCPOLEN_SACK_BASE + (opts->num_sack_blocks *
                                                   TCPOLEN_SACK_PERBLOCK)));

               for (this_sack = 0; this_sack < opts->num_sack_blocks;
                    ++this_sack) {
                      *ptr++ = htonl(sp[this_sack].start_seq);    // Start sequence
number of the block
                      *ptr++ = htonl(sp[this_sack].end_seq);      // End sequence number
of the block
               }

               tp->rx_opt.dsack = 0;
        }

        // H. Extension Features (TCP Fast Open - TFO)
        if (unlikely(OPTION_FAST_OPEN_COOKIE & options)) {
```

```
		struct tcp_fastopen_cookie *foc = opts->fastopen_cookie;
		u8 *p = (u8 *)ptr;
		u32 len;

		if (foc->exp) {
			len = TCPOLEN_EXP_FASTOPEN_BASE + foc->len;
			*ptr = htonl((TCPOPT_EXP << 24) | (len << 16) |
				     TCPOPT_FASTOPEN_MAGIC);
			p += TCPOLEN_EXP_FASTOPEN_BASE;
		} else {
			len = TCPOLEN_FASTOPEN_BASE + foc->len;
			*p++ = TCPOPT_FASTOPEN;
			*p++ = len;
		}

		memcpy(p, foc->val, foc->len);
		if ((len & 3) == 2) {
			p[foc->len] = TCPOPT_NOP;
			p[foc->len + 1] = TCPOPT_NOP;
		}
		ptr += (len + 3) >> 2;
	}

	smc_options_write(ptr, &options);

	mptcp_options_write(th, ptr, tp, opts);
}
```

F. Window Scale

- Enables scaling of the window size field beyond 64KB (RFC 7323), critical for high-bandwidth links.
- Like MSS, only present in SYN/SYN-ACK.

G. SACK Blocks

- If retransmissions are needed, includes up to four SACK blocks specifying the start and end sequence numbers of correctly received but out-of-order data segments

H. Fast Open, MPTCP, SMC, and Other Advanced Options

- TCP Fast Open: Allows sending data in the SYN, speeding up initial data delivery.
- Multipath TCP (MPTCP): Allows a single connection to use multiple paths (e.g., WiFi + LTE).
- SMC (Shared Memory Communications): Advanced IBM extension for in-memory transfer.

```
    // I. Checksum and GSO Configuration
    bpf_skops_write_hdr_opt(sk, skb, NULL, NULL, 0, &opts);

    // Checksum Calculation (Can be delegated to hardware offload)
    INDIRECT_CALL_INET(icsk->icsk_af_ops->send_check,
                       tcp_v6_send_check, tcp_v4_send_check,
                       sk, skb);

    if (likely(tcb->tcp_flags & TCPHDR_ACK))
        tcp_event_ack_sent(sk, rcv_nxt);

    if (skb->len != tcp_header_size) {
        tcp_event_data_sent(tp, sk);
        tp->data_segs_out += tcp_skb_pcount(skb);
        tp->bytes_sent += skb->len - tcp_header_size;
    }

    if (after(tcb->end_seq, tp->snd_nxt) || tcb->seq == tcb->end_seq)
        TCP_ADD_STATS(sock_net(sk), TCP_MIB_OUTSEGS,
                      tcp_skb_pcount(skb));

    tp->segs_out += tcp_skb_pcount(skb);
    skb_set_hash_from_sk(skb, sk);

    // GSO Configuration
    skb_shinfo(skb)->gso_segs = tcp_skb_pcount(skb);
    skb_shinfo(skb)->gso_size = tcp_skb_mss(skb);
```

I. Checksum & GSO (Giant Segment Offload) Preparation

- Calls out to eBPF or kernel helper routines for header options.
- Sets up the TCP checksum (typically offloaded to hardware/NIC).
- Updates GSO metadata fields so the NIC can split large frames as needed.

```
    // 7. Handoff to IP Layer
    memset(skb->cb, 0, max(sizeof(struct inet_skb_parm),
                           sizeof(struct inet6_skb_parm)));

    tcp_add_tx_delay(skb, tp);

    err = INDIRECT_CALL_INET(icsk->icsk_af_ops->queue_xmit,
                             inet6_csk_xmit, ip_queue_xmit,
                             sk, skb, &inet->cork.fl);

    if (unlikely(err > 0)) {
        tcp_enter_cwr(sk);
        err = net_xmit_eval(err);
    }
    if (!err && oskb) {
        tcp_update_skb_after_send(sk, oskb, prior_wstamp);
        tcp_rate_skb_sent(sk, oskb);
    }
    return err;
}
```

J. Handoff to IP Layer

- Zeroes the control block (skb->cb) to ensure clean transition to IPv4 or IPv6.
- Schedules the skb for transmission by calling the relevant IP layer queueing function (ip_queue_xmit for IPv4, inet6_csk_xmit for IPv6).
- Handles errors and adjusts congestion state if needed.

2-4. Finalization & Transmission (out label)

After all data copying and buffer aggregation is complete, the TCP stack performs final operations before returning from tcp_sendmsg_locked(). This ensures any remaining data in the transmission queue is sent out.

```
out:
        if (copied) {
                tcp_tx_timestamp(sk, sockc.tsflags);
                tcp_push(sk, flags, mss_now, tp->nonagle, size_goal);
        }
```

- **Transmit Timestamping:**
  tcp_tx_timestamp() records the send time for the data, which may be used for diagnostics, RTT estimation, or socket-level timestamping features.
- **Final Data Push:**
  If any data has been buffered (copied > 0), the function calls tcp_push(). This triggers actual transmission of accumulated skb(s) in the write queue, ensuring no data is left pending.
- **Return Value:**
  The function returns the total number of bytes successfully sent (copied). This signals completion of the send operation to the user application.

3. Release socket and after services (via release_sock())

After a TCP send operation (or any critical socket work) is complete, the kernel must release its exclusive ownership of the socket. This is handled by release_sock(), which not only unlocks the socket, but also performs essential deferred processing and cleanup.

```
/* net/core/sock.c */
void release_sock(struct sock *sk)
{
        // 1. Acquire spinlock and disable interrupts
        spin_lock_bh(&sk->sk_lock.slock);
        // 2. Process Backlog
        if (sk->sk_backlog.tail)
```

```
            __release_sock(sk);
        // 3. Protocol-specific callback handling
        if (sk->sk_prot->release_cb)
                INDIRECT_CALL_INET_1(sk->sk_prot->release_cb,
                                                    tcp_release_cb, sk);
        // 4. Release ownership
        sock_release_ownership(sk);
        // 5. Wake up waiters
        if (waitqueue_active(&sk->sk_lock.wq))
                wake_up(&sk->sk_lock.wq);
        spin_unlock_bh(&sk->sk_lock.slock);
}
```

- **Acquire Spinlock (SoftIRQ Protection):**
  The function begins by acquiring a spinlock, ensuring exclusive access and blocking SoftIRQ handlers to avoid race conditions.

- **Backlog Processing:**
  Any packets that arrived while the socket was locked are processed via __release_sock(). This ensures ACKs, retransmission queue updates, and receive window adjustments are handled *after* critical state changes, avoiding reentrancy bugs.

- **Protocol-Specific Cleanup:**
  If the protocol (e.g., TCP) has a cleanup callback (release_cb), it is invoked to perform protocol-specific deferred tasks. For TCP, this may include handling Tail Loss Probe, sending delayed ACKs, or cleaning up timers.

- **Release Socket Ownership:**
  The current thread relinquishes exclusive access to the socket, allowing other threads or processes to acquire it.

- **Wake Up Waiters:**
  If other processes or threads are waiting for the socket lock, they are woken up so they can proceed.

- **Release Spinlock:**
  Finally, the spinlock is released and SoftIRQ is re-enabled, fully restoring concurrency on the socket.

Part 3. IP Layer Processing & Routing Decision

ip_queue_xmit

Step 1: Check if the SKB is already routed

```
/* Skip all of this if the packet is already routed,
* f.e. by something like SCTP.
*/
rcu_read_lock();
inet_opt = rcu_dereference(inet->inet_opt);
fl4 = &fl->u.ip4;
rt = skb_rtable(skb);
if (rt)
       goto packet_routed;
```

Detailed Code Logic Analysis

- **rcu_read_lock()**
  - **Description:** Read-side critical section entered to read RCU (Read-Copy-Update) protected data
  - **Mechanism:** This macro expands to an actual function call, which configures the environment so that the RCU reader can safely read data by disabling Preemption (task replacement by the scheduler) and incrementing the RCU read counter.
- **rcu_dereference()**
  - **Description:** Safely reads the IPv4 options structure (inet_opt) set on a socket using RCU.
  - **Mechanism**: Ensures that the most up-to-date/consistent data is read within the RCU critical section, including a memory barrier (smp_read_barrier_depends) that prevents compiler/CPU memory relocation.
- **skb_rtable()**
  - **Description:** Check if the SKB already has routing information (dst_entry/rtable) to see if it is an SKB that has been pre-routed by another layer.
  - **Mechanism:** This macro casts skb_dst(skb) to rtable.

Step 2: Create a new route if no routing information exists

```
/* Make sure we can route this packet. */
rt = dst_rtable(__sk_dst_check(sk, 0));
if (!rt) {
	inet_sk_init_flowi4(inet, fl4);

	/* sctp_v4_xmit() uses its own DSCP value */
	fl4->flowi4_tos = tos & INET_DSCP_MASK;

	/* If this fails, retransmit mechanism of transport layer will
	 * keep trying until route appears or the connection times
	 * itself out.
	 */
	rt = ip_route_output_flow(net, fl4, sk);
	if (IS_ERR(rt))
		goto no_route;

	sk_setup_caps(sk, &rt->dst);
}
skb_dst_set_noref(skb, &rt->dst);
```

Detailed Code Logic Analysis

- **rt = dst_rtable(__sk_dst_check(sk, 0))**
  - **Description:** Check if the current socket (sk) has previously stored valid routing information.
  - **Mechanism:** Compares the dst_entry stored in the socket with the current routing generation ID of the system, and if there is no network status change (link down, IP change, etc.), returns the cached route immediately.
- **`inet_sk_init_flowi4(inet, fl4);`**
  - **Description:** If the cache is missing or invalid (requiring new routing), the flow key (flowi4) is initialized as a search condition for a new route search.
  - **Mechanism:** Copy the destination IP, source IP, TOS (Type of Service), OIF (Output Interface), etc. stored in the socket structure (inet_sock) to the fl4 structure to prepare a query object for searching the FIB (Forwarding Information Base).
- **`rt = ip_route_output_flow(net, fl4, sk);`**
  - **Description:** It actually queries the kernel's main routing table (FIB) to find the optimal route.
  - **Mechanism:** Based on the prepared fl4 key, a matching routing entry is found using a Trie algorithm or similar. If successful, the result is returned in the form of a struct rtable, which contains the outgoing interface (dev) and dst_entry, which contains information for resolving L2 addresses.
- **`sk_setup_caps(sk, &rt->dst);`**
  - **Description:** Adjusts the socket's transfer capabilities to match the hardware characteristics of the newly discovered path.

    - **Mechanism:** Check if the network interface (NIC) of the found path supports features (e.g., TSO - TCP Segmentation Offload, Checksum Offload) and synchronize flags so that the socket can utilize these features. This reduces CPU load and prepares the socket for hardware acceleration.
- `skb_dst_set_noref(skb, &rt->dst);`
    - **Description:** Connect the determined routing information (rt->dst) to the packet (skb).
    - **Mechanism:** The skb->_skb_refdst field is assigned a pointer to the routing entry. The _noref suffix indicates that the reference count of the corresponding dst_entry will not be incremented for performance reasons (this is done safely within the RCU lock). This ensures that the packet has a clear destination, indicating where it should go.

Data Transformation Status

- **Before:** A mass of data that doesn't know where to go
- **After:** Attaching dst_entry. The routing decision determines which interface (NIC) and next hop (Gateway) the packet should go to, which is recorded in the skb->dst pointer. Consequently, the packet's "destination path" is determined.

Step 3: Create IP header

```
/* OK, we know where to send it, allocate and build IP header. */
skb_push(skb, sizeof(struct iphdr) + (inet_opt ? inet_opt->opt.optlen : 0));
skb_reset_network_header(skb);
iph = ip_hdr(skb);
*((__be16 *)iph) = htons((4 << 12) | (5 << 8) | (tos & 0xff));
```

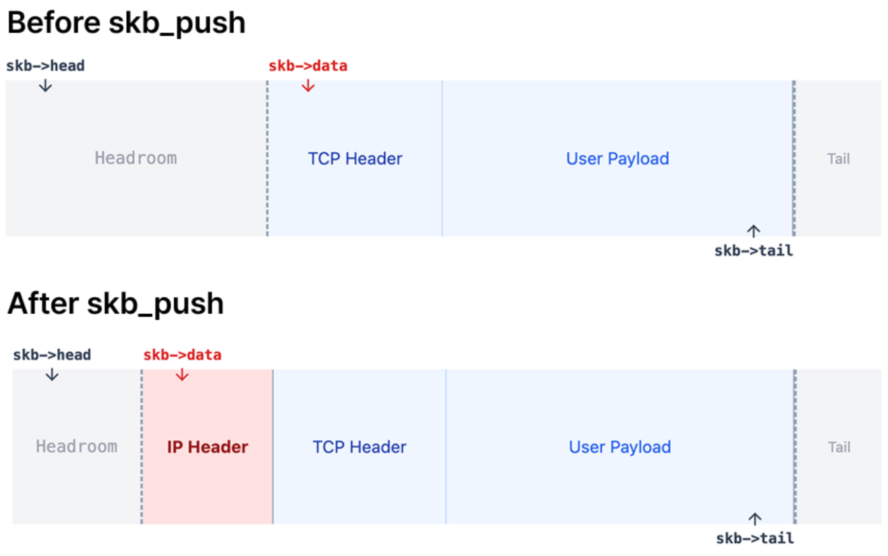
Detailed Code Logic Analysis

- `skb_push(skb, len)`

```
void *skb_push(struct sk_buff *skb, unsigned int len)
{
       skb->data -= len;
       skb->len  += len;
       if (unlikely(skb->data < skb->head))
              skb_under_panic(skb, len, __builtin_return_address(0));
       return skb->data;
}
```

    - **Description:** Reserves space for the IP header at the beginning of the TCP payload (data).
    - **Mechanism:**
        - sk_buff manipulates pointers instead of copying and moving data for efficiency.

        - **`skb->data -= len`**: Decrease the starting address of the data (data) by len (move towards a lower address in memory) and incorporate the headroom in front into the valid data area.
        - **`skb->len += len`**: Increases the total length of the packet by the header size.
        - **Panic Check**: If skb->data becomes smaller than the start of the allocated buffer (skb->head), i.e., if there is insufficient headroom, skb_under_panic is called to trigger a kernel panic. (Usually, the TCP layer reserves sufficient space in advance.)



- **`skb_reset_network_header(skb)`**
    - **Description:** We record in the sk_buff structure that the current location of skb->data is the start of the L3 (Network Layer) header.
    - **Mechanism:** Stores the offset of the current data in the skb->network_header field.
- **ip_hdr()**
    - **Description:** Obtain an iphdr pointer pointing to the IP header based on the network_header location set above.
    - **Mechanism:** You can find the exact IP header location by referencing the offset stored in the skb->network_header field.

Data Transformation Status

- **Transformation:** When skb_push(skb, len) is called, the skb->data pointer moves to a lower address (front) in memory.
- **Status:** The empty space (headroom) preceding the existing TCP data is incorporated into the valid data area. The total length (len) of the packet now becomes the IP Header Size + TCP Segment Size.

- **Marker:** Using skb_reset_network_header(), we mark the sk_buff structure to indicate that the starting point of the space just acquired is the starting point of the IP header.

Step 4: Fragmentation settings

```
if (ip_dont_fragment(sk, &rt->dst) && !skb->ignore_df)
       iph->frag_off = htons(IP_DF);
else
       iph->frag_off = 0;
```

Detailed Code Logic Analysis

- **ip_dont_fragment(sk, &rt->dst)**
    - **Description:** Check if the Path MTU (PMTU) Discovery feature is enabled, taking into account the current socket settings and characteristics of the routing path.
    - **Mechanism:** PMTU Discovery is a technology that prevents packet fragmentation for network efficiency. This function checks socket options (such as IP_PMTUDISC_DO) and returns true if fragmentation should be prevented.
- **iph->frag_off = htons(IP_DF)**
    - **Description:** Sets the DF (Don't Fragment) bit in the Flags field of the IP header.
    - **Mechanism:** IP_DF is a constant value of 0x4000 (bit 14). It is converted to network byte order (Big Endian) using htons() and written to the header.
        - If a packet with this bit set encounters a router with a smaller MTU than its own, the router drops the packet instead of fragmenting it and sends an ICMP (Packet Too Big) message to the sender. This forces the sender to adjust the packet size.

Data Transformation Status

- **DF field settings**: The DF (Don't Fragment) flag is set in the frag_off field, physically embedding the routing policy "do not fragment this packet" into the header.

Step 5: TTL, Protocol, Address (Source/Dest) Record

```
iph->ttl      =ip_select_ttl(inet, &rt->dst);
iph->protocol = sk->sk_protocol;
ip_copy_addrs(iph,fl4);
```

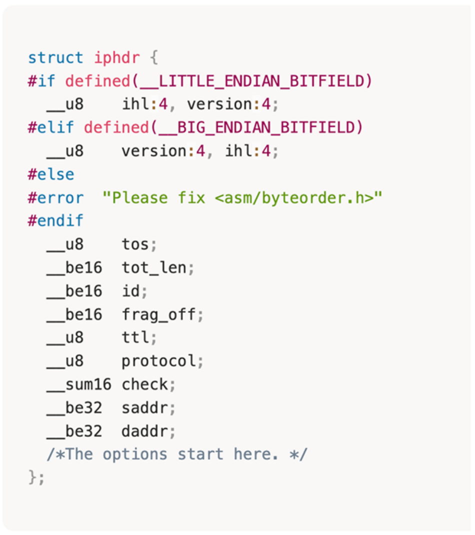
Detailed Code Logic Analysis

- **iph->ttl = ip_select_ttl(inet, &rt->dst)**
    - **Description:** Set a TTL (Time To Live) value to prevent packets from looping infinitely on the network.

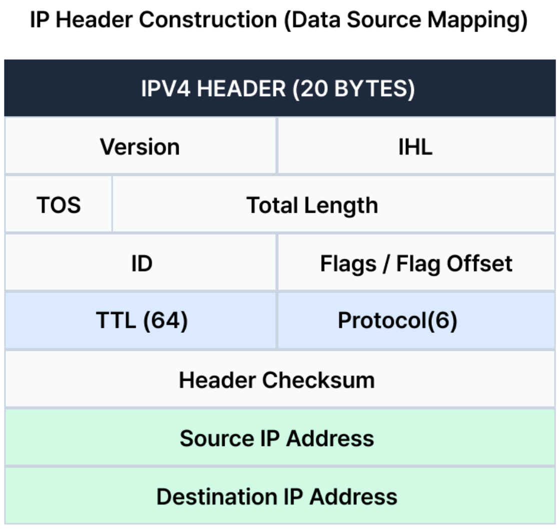
    - **Mechanism:** The TTL value set on the socket (inet->uc_ttl) is used first, or if not set, the kernel default (usually 64) is used. This value decreases by 1 for each router it passes through, and if it reaches 0, the packet is dropped.
- **iph->protocol = sk->sk_protocol**
    - **Description:** Records an ID that identifies which upper layer protocol the data contained in the IP payload is from.
    - **Mechanism:** Obtain the protocol number specified when creating the socket. Since this is a TCP socket, this value is 6 (TCP). The receiving IP layer sees this value and passes the packet on to the TCP processing routine (demultiplexing).
- **ip_copy_addrs(iph, fl4)**
    - **Description:** Copy the Source IP and Destination IP addresses into the header.
    - **Mechanism:**
        - **iph->saddr = fl4->saddr**: Record the source IP determined in the routing decision step (fl4).
        - **iph->daddr = fl4->daddr**: Likewise, record the destination IP.
        - Although this process is a simple memory copy (assign), it is the most crucial step in establishing the logical connection (end-to-end connectivity) of packets.

Data Transformation Status

- **TTL**: The number of hops a packet can survive on the network is recorded.
- **Protocol**: It is specified that the contents are TCP(6).
- **Addrs:** The most important source IP and destination IP are recorded to determine the 'sender and receiver' of the packet.



```
struct iphdr {
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u8    ihl:4, version:4;
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u8    version:4, ihl:4;
#else
#error  "Please fix <asm/byteorder.h>"
#endif
    __u8    tos;
    __be16  tot_len;
    __be16  id;
    __be16  frag_off;
    __u8    ttl;
    __u8    protocol;
    __sum16 check;
    __be32  saddr;
    __be32  daddr;
    /*The options start here. */
};
```



Step 6: IP Options Build

```
/* Transport layer set skb->h.foo itself. */

if (inet_opt && inet_opt->opt.optlen) {
       iph->ihl += inet_opt->opt.optlen >> 2;
       ip_options_build(skb, &inet_opt->opt, inet->inet_daddr, rt);
}
```

Detailed Code Logic Analysis

- **iph->ihl += inet_opt->opt.optlen >> 2**
  - **Description:** Increases the length field (IHL) of the IP header by the size of the option.
  - **Mechanism:** IHL represents the length in 4-byte (32-bit) word units. Therefore, to divide the option length (optlen) by 4, perform a right shift (>> 2) operation and add it to the IHL value. (e.g. 4 bytes of option -> IHL +1).
- **ip_options_build(skb, ...)**
  - **Description:** The actual option data is appended immediately after the IP header.
  - **Mechanism:** Copy the optional bytes from the SKB data area to the space after the standard header. If necessary, add padding to align them to a 4-byte boundary. This completes the variable portion of the IP header.

Data Transformation Status

- **extension**: If there are IP options, the header length increases (IHL > 5) and additional option data is appended to the header.

Step 7: IP Identification Generation

```
ip_select_ident_segs(net, skb, sk,
                   skb_shinfo(skb)->gso_segs ?: 1);
```

Detailed Code Logic Analysis

- **`ip_select_ident_segs(...)`**
  - **Description:** Fill in the id field (16 bits) of the IP header.
  - **Mechanism:**
    - Generate a unique ID using a global counter or hash-based generator inside the kernel.
    - **GSO (Generic Segmentation Offload):** If the NIC supports large segmentation and gso_segs is greater than 1, this function handles assigning consecutive IDs to consecutive packets so that the hardware can later segment the packets with the correct IDs.

Data Transformation Status

- **Unique:** A unique number is generated and entered in the ID field. This unique identifier allows the packet to be reassembled later if fragmented.

Step 8: Setting priority/mark values

```
/* TODO : should we use skb->sk here instead of sk ? */
skb->priority = READ_ONCE(sk->sk_priority);
skb->mark = READ_ONCE(sk->sk_mark);
```

Detailed Code Logic Analysis

- **`skb->priority`**
  - **Description:** Used to determine queuing priorities in the Linux Traffic Control (TC) subsystem.
- **`skb->mark`**
  - **Description:** A 32-bit marking value used by netfilter or policy routing to identify and filter specific packets.
- **`READ_ONCE`**
  - **Description:** In a multi-threaded environment, even if the socket settings change, the values are read atomically to prevent data race conditions.

Data Transformation Status

- **Management Tags**: The priority and mark values are copied to the sk_buff object itself (skb), not the packet header (iph). These are "internal tags" that indicate how the packet should be treated within the kernel (QoS, firewall).

Step 9: Finish assembling the IP header and move on to the next step.

```
res = ip_local_out(net, sk, skb);
rcu_read_unlock();
return res;
```

Detailed Code Logic Analysis

- **ip_local_out(net, sk, skb)**
  - **Description:** It is the final gateway of the IP layer that sends the completed packet out of the network.
  - **Mechanism:**
    - **Netfilter Hook (NF_INET_LOCAL_OUT):** Check your firewall rules. This may be causing packets to be dropped or altered.

                ■ **dst_output(skb):** When it passes through Netfilter, it calls the output function (usually ip_finish_output -> dev_queue_xmit) registered in the routing information (rt->dst) to pass the packet to the L2 layer (Ethernet).

- **rcu_read_unlock()**
    - **Description:** Release the RCU Read Lock that was held at the beginning of the function.
    - **Mechanism:** Now we signal the end of the critical section that references routing information, etc., and allow the kernel scheduler to do a context switch if necessary.

Data Transformation Status

- **Handover:** The ip_local_out function is called and control leaves the IP layer.

Step 10: Error Handling (Handling routing failure)

```
no_route:
        rcu_read_unlock();
        IP_INC_STATS(net, IPSTATS_MIB_OUTNOROUTES);
        kfree_skb_reason(skb, SKB_DROP_REASON_IP_OUTNOROUTES);
        return -EHOSTUNREACH;
```

Detailed Code Logic Analysis

- **`IP_INC_STATS`**
    - **Description:** Increments the No Route statistics counter (MIB), allowing administrators to check the frequency of errors using netstat, etc.
- **`kfree_skb_reason`:**
    - **Description:** Frees the memory allocated to sk_buff. This doesn't simply free the memory; it also leaves a drop reason (DROP_REASON_IP_OUTNOROUTES) in the trace system (eBPF, etc.) to aid debugging.
- **`return -EHOSTUNREACH`**
    - **Description:** Returns the Host Unreachable error code to the user program that called the system call.

__ip_local_out

```
int __ip_local_out(struct net *net, struct sock *sk, struct sk_buff *skb)
{
        struct iphdr *iph = ip_hdr(skb);

        IP_INC_STATS(net, IPSTATS_MIB_OUTREQUESTS);
        iph_set_totlen(iph, skb->len);
        ip_send_check(iph);

        /* if egress device is enslaved to an L3 master device pass the
         * skb to its handler for processing
         */
        skb = l3mdev_ip_out(sk, skb);
        if (unlikely(!skb))
                return 0;

        skb->protocol = htons(ETH_P_IP);

        return nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT,
                       net, sk, skb, NULL, skb_dst_dev(skb),
                       dst_output);
}
```

Detailed Code Logic Analysis

- **`iph_set_totlen(iph, skb->len)`**
  - **Description:** Sets the tot_len (Total Length) field in the IP header.
  - **Mechanism:** Since the header and payload have been merged together through skb_push and other methods, skb->len is now the exact total packet length. This is recorded in the header.
- **`ip_send_check(iph)`**
  - **Description:** Calculates and records the checksum of the IP header.
  - **Mechanism:** Since all fields such as tot_len, ttl, protocol, saddr, daddr, etc. are confirmed, we 'seal' the 1's complement sum of the 16-bit words in the header to ensure integrity.
- **`l3mdev_ip_out(sk, skb)`**
  - **Description:** Ensure that the interface through which the packet is going out belongs to an L3 Master Device such as VRF (Virtual Routing and Forwarding).
  - **Mechanism:** If it does, it passes the packet to the handler of that master device for further encapsulation or processing.
- **`skb->protocol = htons(ETH_P_IP)`**
  - **Description:** Specifies the L3 protocol type of this packet in the sk_buff structure.
  - **Mechanism:** When the L2 layer (such as the Ethernet driver) receives this packet, it sets a kernel internal identifier (ETH_P_IP) to indicate that the payload is IPv4.
- **`nf_hook(NFPROTO_IPV4, NF_INET_LOCAL_OUT, ...)`**
  - **Description:** Passes through the LOCAL_OUT point of Netfilter, a Linux firewall framework.
  - **Mechanism:**
    - OUTPUT chain rules registered in iptables or nftables are executed here.

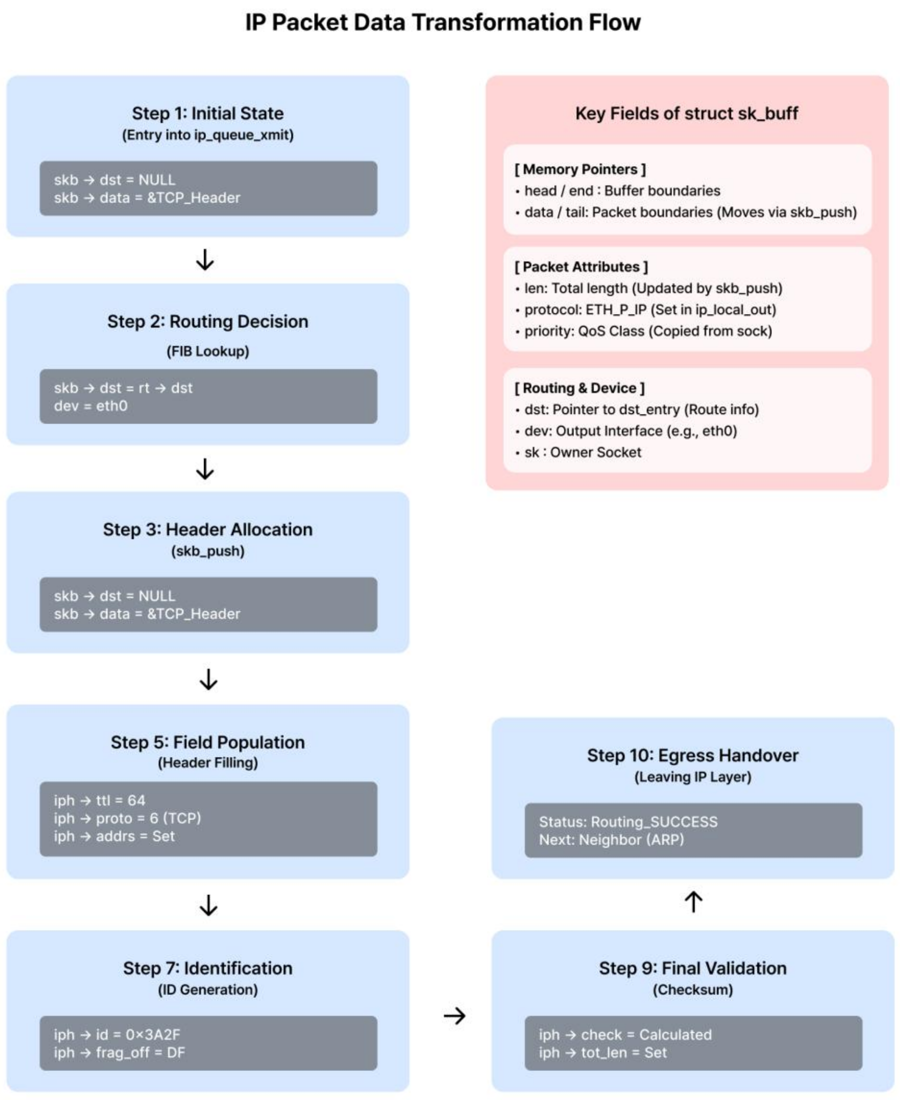
- According to the user-defined policy, actions such as DROP (discard), ACCEPT (pass), DNAT (change destination), and MARK (tagging) are performed.
- **Callback**: If the rule is successfully passed, the dst_output function passed as the last argument is called, moving on to the next step.

Data Transformation Status

- **Transformation:** The tot_len and check fields in the header are changed from 0 (or any value) to valid values.
- **Status (Validation):** The packet is now "Integrity Assured" and enters the "Security Check" phase, where it must survive to be able to continue on the network.

Summary: IP Packet Data Transformation Flow

Here is a quick overview of the order in which the memory structure (skb), header filling, routing, and verification occur until the sk_buff from the TCP layer is completed as an IP packet.



ip_local_out

```
int ip_local_out(struct net *net, struct sock *sk, struct sk_buff *skb)
{
	int err;

	err = __ip_local_out(net, sk, skb);
	if (likely(err == 1))
		err = dst_output(net, sk, skb);

	return err;
}
EXPORT_SYMBOL_GPL(ip_local_out);
```

Detailed Code Logic Analysis

- `err = __ip_local_out(...)`
  - **Description:** Executes the header completion and Netfilter hooks described above.
  - **Mechanism:** Receive the return value of nf_hook.
- `if (likely(err == 1)) err = dst_output(...)`
  - **Description:** If the Netfilter check passes, the following destination output routine is executed.
  - **Mechanism:**
    - A return value of 1 indicates that Netfilter accepted the packet and delegated further processing to the caller.
    - **dst_output**: Calls the output function pointer registered in the routing table (dst_entry). For typical unicast packets, this function pointer points to ip_finish_output, which is where L2 (Data Link Layer) processing (ARP Resolution, Neighbor System, etc.) begins.
    - **likely**: Optimizes branch prediction performance by assuming that most packets will pass through the firewall.

Data Transformation Status

- **Handover (Context Switch):** The IP layer's Generation and Validation roles are now complete.
- **Next Phase:** Control passes to the Neighbor Subsystem (ARP/L2) and Device Driver realms via dst_output. The packet now enters the world of MAC addresses, not IP addresses.

dst_output

```
/* Output packet to network from transport.  */
static inline int dst_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
	return INDIRECT_CALL_INET(READ_ONCE(skb_dst(skb)->output),
				  ip6_output, ip_output,
				  net, sk, skb);
}
```

Detailed Code Logic Analysis

- **`INDIRECT_CALL_INET(...)`**
  - **Description:** Calls the function pointer (output) registered in dst_entry, but induces a direct call instead of an indirect call for performance optimization.
  - **Mechanism:** Due to the Spectre security vulnerability patch (Retpoline), calls via function pointers incur significant overhead. This macro improves branch prediction performance by hinting the compiler, "If the output function is ip_output, call it directly."

ip_output

```
int ip_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
	struct net_device *dev, *indev = skb->dev;
	int ret_val;

	rcu_read_lock();
	dev = skb_dst_dev_rcu(skb);
	skb->dev = dev;
	skb->protocol = htons(ETH_P_IP);

	ret_val = NF_HOOK_COND(NFPROTO_IPV4, NF_INET_POST_ROUTING,
				net, sk, skb, indev, dev,
				ip_finish_output,
				!(IPCB(skb)->flags & IPSKB_REROUTED));
	rcu_read_unlock();
	return ret_val;
}
EXPORT_SYMBOL(ip_output);
```

Detailed Code Logic Analysis

- **`dev = skb_dst_dev_rcu(skb); skb->dev = dev;`**

    - **Description:** Explicitly associates the network interface card (NIC) object that the packet will actually go out to with sk_buff.
    - **Mechanism:** Get the net_device pointed to by the routing table (dst). This is safely referenced in a multicore environment via an RCU lock.
- **`NF_HOOK_COND(..., NF_INET_POST_ROUTING, ...)`**
    - **Description:** Executes the POST_ROUTING chain, the final checkpoint in Linux networking.
    - **Mechanism:** iptables' NAT (SNAT, Masquerade) is primarily performed here. This step converts private IP addresses to public IP addresses, and packets may be dropped depending on policy. Upon passing through, ip_finish_output is called.

Data Transformation Status

- **Logical:** The packet's status transitions from "Local Out Processing" to "Post-Routing Processing".
- **Potential Mutation:** If NAT is set up, the Source IP address or Port number may change at this step.

ip_finish_output

```
static int __ip_finish_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
	unsigned int mtu;

#if defined(CONFIG_NETFILTER) && defined(CONFIG_XFRM)
	/* Policy lookup after SNAT yielded a new policy */
	if (skb_dst(skb)->xfrm) {
		IPCB(skb)->flags |= IPSKB_REROUTED;
		return dst_output(net, sk, skb);
	}
#endif
	mtu = ip_skb_dst_mtu(sk, skb);
	if (skb_is_gso(skb))
		return ip_finish_output_gso(net, sk, skb, mtu);

	if (skb->len > mtu || IPCB(skb)->frag_max_size)
		return ip_fragment(net, sk, skb, mtu, ip_finish_output2);

	return ip_finish_output2(net, sk, skb);
}

static int ip_finish_output(struct net *net, struct sock *sk, struct sk_buff *skb)
{
	int ret;

	ret = BPF_CGROUP_RUN_PROG_INET_EGRESS(sk, skb);
	switch (ret) {
	case NET_XMIT_SUCCESS:
		return __ip_finish_output(net, sk, skb);
	case NET_XMIT_CN:
		return __ip_finish_output(net, sk, skb) ? : ret;
	default:
		kfree_skb_reason(skb, SKB_DROP_REASON_BPF_CGROUP_EGRESS);
		return ret;
	}
}
```

Detailed Code Logic Analysis

- **`BPF_CGROUP_RUN_PROG_INET_EGRESS(...)`**
    - **Description:** Controls outgoing packets by container or process group by executing a cgroup-based eBPF program.
    - **Mechanism:** This hook is crucial in recent cloud-native environments (e.g., Kubernetes), allowing packet monitoring or blocking.
- **`if (skb_is_gso(skb))`**
    - **Description:** Determines whether a packet is a large segment (GSO).
    - **Mechanism:** If the NIC supports TCP Segmentation Offload (TSO), the kernel passes the large segment to the NIC as is (ip_finish_output_gso) without fragmenting it in software. This significantly reduces CPU load.
- **`ip_fragment(...)`**
    - **Description:** When the packet size is larger than the MTU and GSO is not available, the packet is fragmented into multiple smaller IP packets.
    - **Mechanism:** The original sk_buff is copied and split into multiple new sk_buffs, and the Offset field in the IP header is adjusted before each is sent to ip_finish_output2.

Data Transformation Status

- **Physical:** If the packet size exceeds the MTU, one sk_buff is fragmented into N sk_buffs.
- **Logical:** The final step of packet validation is complete.

ip_finish_output2

```
static int ip_finish_output2(struct net *net, struct sock *sk, struct sk_buff *skb)
{
	struct dst_entry *dst = skb_dst(skb);
	struct rtable *rt = dst_rtable(dst);
	struct net_device *dev = dst_dev(dst);
	unsigned int hh_len = LL_RESERVED_SPACE(dev);
	struct neighbour *neigh;
	bool is_v6gw = false;

	if (rt->rt_type == RTN_MULTICAST) {
		IP_UPD_PO_STATS(net, IPSTATS_MIB_OUTMCAST, skb->len);
	} else if (rt->rt_type == RTN_BROADCAST)
		IP_UPD_PO_STATS(net, IPSTATS_MIB_OUTBCAST, skb->len);

	/* OUTOCTETS should be counted after fragment */
	IP_UPD_PO_STATS(net, IPSTATS_MIB_OUT, skb->len);

	if (unlikely(skb_headroom(skb) < hh_len && dev->header_ops)) {
		skb = skb_expand_head(skb, hh_len);
		if (!skb)
			return -ENOMEM;
	}

	if (lwtunnel_xmit_redirect(dst->lwtstate)) {
		int res = lwtunnel_xmit(skb);

		if (res != LWTUNNEL_XMIT_CONTINUE)
			return res;
	}

	rcu_read_lock();
	neigh = ip_neigh_for_gw(rt, skb, &is_v6gw);
	if (!IS_ERR(neigh)) {
		int res;

		sock_confirm_neigh(skb, neigh);
		/* if crossing protocols, can not use the cached header */
		res = neigh_output(neigh, skb, is_v6gw);
		rcu_read_unlock();
		return res;
	}
	rcu_read_unlock();

	net_dbg_ratelimited("%s: No header cache and no neighbour!\n",
			    __func__);
	kfree_skb_reason(skb, SKB_DROP_REASON_NEIGH_CREATEFAIL);
	return PTR_ERR(neigh);
}
```

Detailed Code Logic Analysis

- **`skb_expand_head(skb, ...)`**
    - **Description:** Check if there is enough space (headroom) to attach an Ethernet header (typically 14 bytes) to the beginning of the sk_buff. If not, increase the headroom.
    - **Mechanism:** If there is insufficient space for the skb_push, a new memory area is allocated and the data is copied (reallocated). This has a significant performance cost, so the initial allocation is usually generous.
- **`neigh = ip_neigh_for_gw(rt, skb, ...)`**
    - **Description:** Find the MAC address information (Neighbor Entry) of the next device to receive this packet.
    - **Mechanism:**

        - The routing information (rt) determines whether the destination is a local network (Direct Connected) or a gateway.
        - The ARP table (Neighbor Table) is looked up using the corresponding IP address (destination or gateway) as a key.
        - If the information is not in the ARP table, an ARP request is generated.
- **`neigh_output(neigh, skb, ...)`**
    - **Description:** Call the output function of the found Neighbor object to pass the packet to the L2 layer.
    - **Mechanism:**
        - If the neighbor's status is NUD_CONNECTED (the MAC address is known), it takes the fast path to dev_queue_xmit.
        - Otherwise, it takes the slow path, waiting for or verifying an ARP response.

Data Transformation Status

- **Status:** The role of the IP protocol ends here.
- **Context:** With the neigh_output call, control of the packet is completely transferred from the IP Subsystem to the Neighbor Subsystem (L2).

Part 4. TCP/IP Header Encapsulation & Device Queue (Layer 2)

In this section, we analyze the process where a packet (skb), having finished IP layer processing, attaches a Layer 2 (Ethernet) header and is delivered to the **Transmission Queue (Tx Queue)** of the actual Network Interface Card.

We conducted the analysis based on the **Fast Path** where the ARP cache exists.

1. ip_finish_output2() : Checking L2 Header Space Allocation

File Path: net/ipv4/ip_output.c

This is the final gateway of the IP layer, serving as the stage to ensure **memory space (Headroom)** for attaching the L2 header.

```
static int ip_finish_output2(struct net *net, struct sock *sk, struct sk_buff *skb)
{
     /* ... (Variable declarations omitted) ... */
     /* [1] Get the L2 header length required by the device (usually 14B + align). */
     unsigned int hh_len = LL_RESERVED_SPACE(dev);

     /*
     * [2] Check Headroom
     * Checks if there is enough empty space (Headroom) at the front of the current
packet (skb)
     * to attach the L2 header.
     * unlikely(): Since space is usually sufficient (Fast Path), branch prediction is
optimized
     * to improve performance.
     */
     if (unlikely(skb_headroom(skb) < hh_len && dev->header_ops)) {
         /* [3] If space is insufficient, call skb_expand_head() to reallocate (expand)
memory. */
         skb = skb_expand_head(skb, hh_len);
         if (!skb)
             return -ENOMEM;
     }

     /* ... (After Neighbor Lookup, call neigh_output) ... */
}
```

The skb_headroom function checks the space where the header will be placed. Since a lack of space is rare, the unlikely macro is used to increase CPU pipeline efficiency.

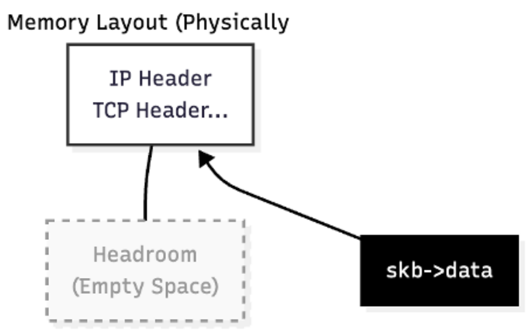
Headroom Space Check:

- LL_RESERVED_SPACE(dev): Returns the header length required by the network device (Ethernet). It calculates not just the Ethernet header length (14 bytes), but the size including padding for memory alignment (Alignment) for performance optimization (usually 16 bytes or more).

- skb_headroom(skb) < hh_len: Checks if there is sufficient empty space at the front of the current packet (skb) data to attach the L2 header.

Space Expansion (skb_expand_head):

- If space is insufficient, it calls skb_expand_head to reallocate memory and secure space.
- unlikely(...): Since space is sufficient in most cases (Fast Path), it informs the compiler that the probability of this branch executing is low, allowing for optimization.



(**Figure 1** Memory Layout at entry of ip_finish_output2. Since it is before the L2 header is attached, skb->data points to the IP header.)

2. neigh_output() : Checking Fast Path Branch

File Path: include/net/neighbour.h

Checks the destination MAC address information (Neighbor) to branch between Fast Path and Slow Path.

```
static inline int neigh_output(struct neighbour *n, struct sk_buff *skb, bool skip_cache)
{
     const struct hh_cache *hh = &n->hh;

     /*
     * [Fast Path Condition]
     * 1. NUD_CONNECTED: Checks if the ARP process is complete and the destination MAC
address
     * is already known (Connected).
     * 2. hh_len: Checks if the cached hardware header length is valid.
     * If these conditions are met, it enters the path to immediately attach the header
     * without complex ARP processes.
     */
     if (!skip_cache &&
         (READ_ONCE(n->nud_state) & NUD_CONNECTED) &&
         READ_ONCE(hh->hh_len))
         return neigh_hh_output(hh, skb); /* [Fast Path] Immediately attach header */

     /* [Slow Path] If ARP request (Solicit) is needed, execute the n->output function
pointer. */
```

```
    return READ_ONCE(n->output)(n, skb);
}
```

It checks the NUD_CONNECTED state bit to distinguish between cases where ARP learning is already finished (neigh_hh_output) and where an ARP request is needed (n->output). This analysis follows the **Fast Path**.

Fast Path Entry Condition (NUD_CONNECTED):

- n->nud_state & NUD_CONNECTED: Checks if the connection state with the destination (Neighbor) is valid. That is, the **ARP procedure is complete, and the peer's MAC address is already known**.
- If this condition is met, neigh_hh_output is called to attach the header immediately (**Fast Path**).

Slow Path (n->output):

- If the MAC address is unknown or the state is unstable, processes such as ARP requests (arp_solicit) are performed via the n->output function pointer.

3. neigh_hh_output() : L2 Header Encapsulation (Core Analysis)

File Path: include/net/neighbour.h

This is the stage where the actual L2 header (MAC address) is copied and packet pointers are manipulated.

```
static inline int neigh_hh_output(const struct hh_cache *hh, struct sk_buff *skb)
{
    unsigned int hh_alen = 0;
    unsigned int seq;
    unsigned int hh_len;

    do {
        /* [1] Seqlock: Checks sequence numbers to prevent data corruption during ARP
updates
           in a multi-core environment. */
        seq = read_seqbegin(&hh->hh_lock);
        hh_len = READ_ONCE(hh->hh_len);
        if (likely(hh_len <= HH_DATA_MOD)) {
            hh_alen = HH_DATA_MOD;
            /* 16 Bytes Alignment */
            /*
             * [2] Memory Alignment Optimization
             * The Ethernet header is logically 14 bytes, but for CPU processing
efficiency (Cache Line),
             * it copies in 16-byte (HH_DATA_MOD) units.
             * This is a design choice (Trade-off) accepting 2 bytes of overhead
(Padding) for speed.
```

```
            */
            if (likely(skb_headroom(skb) >= HH_DATA_MOD)) {
                memcpy(skb->data - HH_DATA_MOD, hh->hh_data, HH_DATA_MOD);
            }
        }
        /* ... (else logic omitted) ... */
    } while (read_seqretry(&hh->hh_lock, seq)); /* Retry if data mismatch */

    /* * [3] Update Pointer (__skb_push)
    * memcpy only wrote values to memory.
    * __skb_push moves the skb->data pointer forward by the actual header length (14B).
    * The packet is now a complete L2 frame.
    */
    __skb_push(skb, hh_len);
    return dev_queue_xmit(skb); /* Transmit to device queue */
}
```

Concurrency Control (Seqlock):

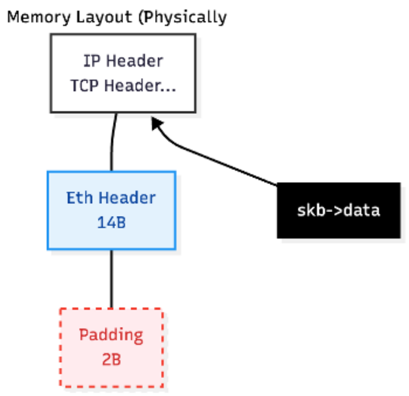
- read_seqbegin / read_seqretry: hh_cache is a resource shared by multiple CPUs. If another CPU updates ARP information while copying the header, data corruption may occur.
- To prevent this, a **Lock-free** style Seqlock technique is used, looping until the sequence numbers before and after the copy match to ensure integrity.

Memory Alignment Optimization:

- The Ethernet header is logically 14 bytes. However, for CPU processing speed (Cache Line efficiency), it is aligned and copied in **16-byte (HH_DATA_MOD)** units.
- memcpy(..., 16): Although the Ethernet header is 14 bytes, 16 bytes (HH_DATA_MOD) are copied entirely for CPU processing efficiency.
- Since the pointer only moves 14 bytes later, the front 2 bytes are discarded (Padding) without being included in the actual packet; this demonstrates the kernel's important design philosophy (Trade-off) of "choosing CPU operation speed even if it wastes a little memory space (2B)".

Pointer Movement (__skb_push):

- memcpy only wrote data to the area before skb->data (Headroom).
- __skb_push(skb, hh_len): Moves the skb->data pointer forward by the actual header length (14 bytes) and increases skb->len.
- At this point, the packet becomes a "complete frame including the Ethernet header".



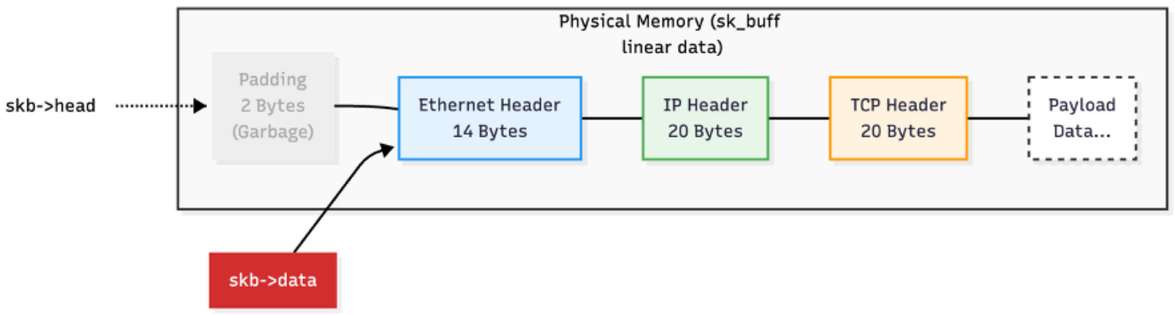
(Figure 2 Placeholder: Result of neigh_hh_output execution. Although 16 bytes were copied, the pointer (skb->data) only moved 14 bytes, leaving 2 bytes of padding in the headroom area.)

4. dev_queue_xmit() : Delivery to Driver Transmission Queue

File Path: net/core/dev.c

Entry point to deliver the completed packet to the network driver.

```
static int __dev_queue_xmit(struct sk_buff *skb, struct net_device *sb_dev)
{
     struct net_device *dev = skb->dev;
     struct netdev_queue *txq;
     struct Qdisc *q;
     /* ... */

     /* [1] RCU Lock: Since transmission paths involve frequent read operations,
        use lightweight RCU instead of Mutex to secure synchronization performance. */
     rcu_read_lock_bh();
     /* [2] Record current skb->data position as the official MAC header start point. */
     skb_reset_mac_header(skb);

     /* [3] Pick Tx Queue: For multi-queue NICs, select appropriate Tx queue (txq)
        based on packet hash, etc. */
     if (!txq)
         txq = netdev_core_pick_tx(dev, skb, sb_dev);

     /* Get the Qdisc (Packet Queue Manager) of the selected queue. */
     q = rcu_dereference_bh(txq->qdisc);

     /* [4] Enqueue: If Qdisc exists (most physical NICs), insert packet into queue for
scheduling. */
     if (q->enqueue) {
         rc = __dev_xmit_skb(skb, q, dev, txq);
         goto out;
     }
     /* ... */
}
```

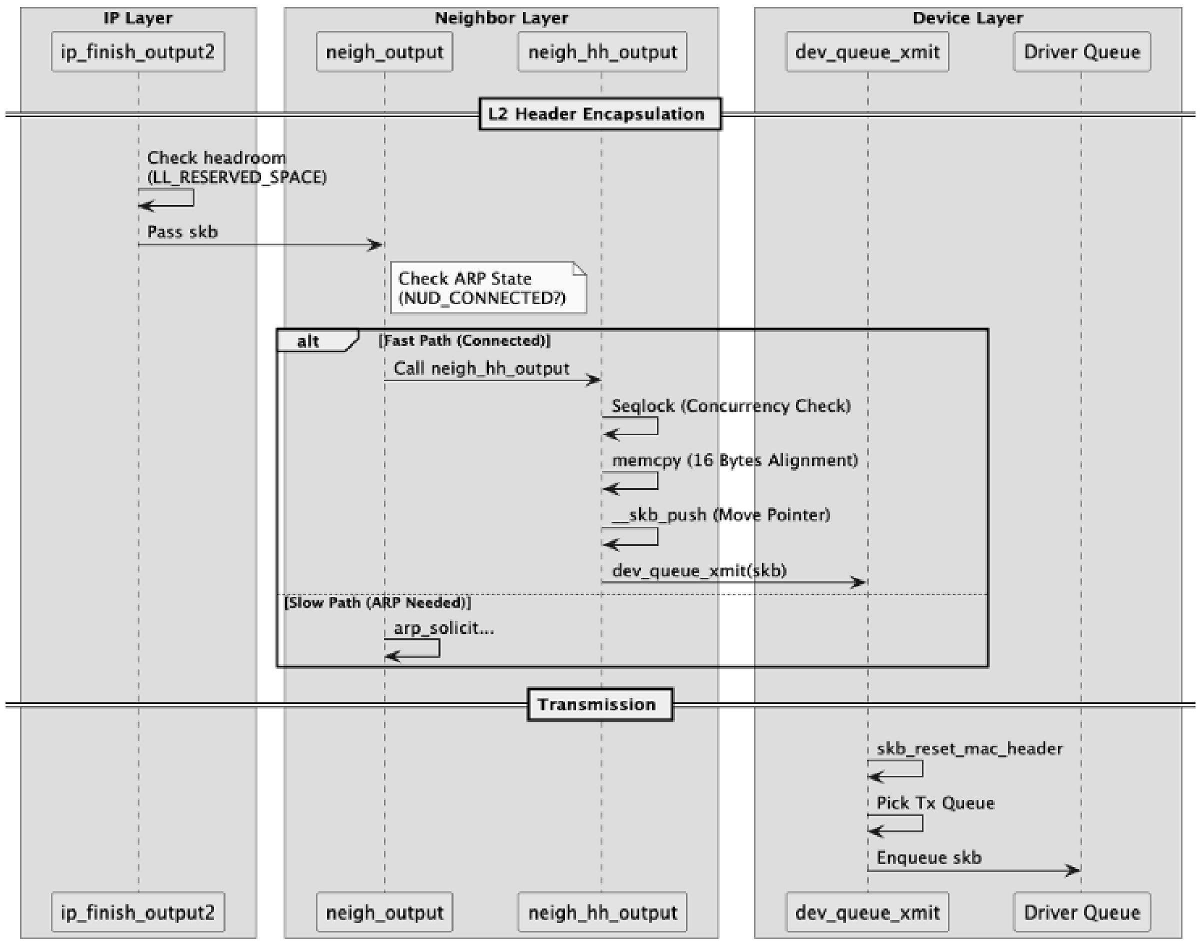


(Figure 3 Placeholder: Physical structure of sk_buff finally delivered to device queue. The Ethernet header has been successfully encapsulated.)

RCU Usage: Unlike the previous stage (neigh_hh_output), since packet transmission (reading) is overwhelmingly more frequent than configuration changes, **RCU (rcu_read_lock_bh)**, which has zero lock cost, was used.

Queue Selection and Enqueue: It selects a queue with netdev_core_pick_tx and puts the packet into the waiting queue (Qdisc) via q->enqueue before driver processing, completing transmission preparation.

Conclusion: Through this process, the TCP/IP packet is encapsulated with an L2 header and is fully prepared to exit to the physical layer.



IP Layer
ip_finish_output2
Neighbor Layer
neigh_output
neigh_hh_output
Device Layer
dev_queue_xmit
Driver Queue
L2 Header Encapsulation
Check headroom (LL_RESERVED_SPACE)
Pass skb
Check ARP State (NUD_CONNECTED?)
alt
[Fast Path (Connected)]
Call neigh_hh_output
Seqlock (Concurrency Check)
memcpy (16 Bytes Alignment)
__skb_push (Move Pointer)
dev_queue_xmit(skb)
[Slow Path (ARP Needed)]
arp_solicit...
Transmission
skb_reset_mac_header
Pick Tx Queue
Enqueue skb