
The Ghost of SVR4

The Ghost of SVR4

by Lee

for India

Introduction

This technical guide provides comprehensive documentation of the SVR4 (System V Release 4) i386 kernel architecture and implementation. The documentation is derived from source code analysis and covers the major subsystems that comprise a complete Unix kernel.

Purpose and Scope

The SVR4 kernel represents a mature implementation of Unix system interfaces and internal architecture. This guide examines:

- **Process Management**: How processes are created, scheduled, signaled, and terminated
- **Memory Management**: Virtual memory implementation, paging, and memory allocation strategies
- **File Systems**: The Virtual File System layer and various filesystem implementations
- **Networking**: Network stack architecture including TCP/IP and NFS
- **I/O and Device Management**: Device drivers, interrupt handling, and boot process

Documentation Methodology

Each subsection follows a consistent structure:

1. **Technical Summary**: A detailed explanation of the subsystem's architecture and operation
2. **Code Analysis**: Key code snippets showing critical implementation details
3. **Diagrams**: Visual representations of state transitions, data structures, and flow control

The focus is on core logic and significant design decisions, excluding boilerplate and macro definitions that obscure understanding.

Source Code

The kernel source code analyzed is located in the `svr4-src/uts/i386/` directory, with primary subsystems in:

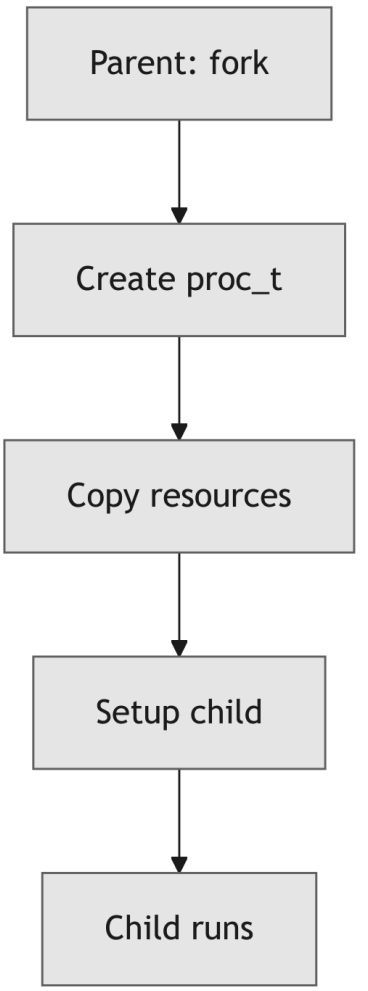
- `os/` - Core operating system functions
- `mem/` - Memory management
- `fs/` - File system implementations
- `net/` - Networking protocols
- `io/` - Device drivers and I/O subsystem

Process Lifecycle

The Genesis and Demise of a Process: A Kernel's Grand Orchestration

The SVR4 i386 kernel, a venerable maestro of multitasking, conducts a perpetual symphony of processes. Each process, from its first breath to its final sigh, embarks on an intricate dance with the kernel's inner workings. To truly grasp the essence of SVR4—its elegant resource management, its disciplined isolation of execution, and its masterful orchestration of concurrent tasks—one must first become intimately acquainted with this grand lifecycle. We peel back the layers, not merely observing, but *feeling* the silicon pulse beneath the C logic.

The Spark of Life: Process Creation



In the dominion of SVR4, a new process doesn't simply *appear*; it is *born*. The primal act of creation is embodied by the `fork()` system call—a moment of digital fission where one process, the stoic parent, begets another, the eager child. Imagine, if you will, the kernel as a meticulous scribe, duplicating the parent's entire universe: its memory segments, its open portals to the filesystem (file descriptors), and even the fleeting thoughts held within its CPU registers.

Upon this miraculous birth, a subtle yet profound distinction emerges. The `fork()` call, like an oracle, whispers the child's unique Process ID (PID) to the parent, while to the child, it merely bestows a humble `0`, a symbol of its nascent identity.

```
// A classic fork() scenario in SVR4
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h> // For pid_t

int main() {
    pid_t pid;
    printf("Before fork, PID is %d\n", getpid()); // Line 6

    pid = fork(); // Line 8

    if (pid < 0) {
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        printf("I am the child! My PID is %d, my parent's PID is %d\n",
getpid(), getppid());
    } else {
        // Parent process
        printf("I am the parent! My PID is %d, my child's PID is %d\n",
getpid(), pid);
    }
    return 0;
}
```

Code Snippet 1.1: The Primal `fork()`

More often than not, this newborn child process hungers for its own destiny, a program distinct from its parent's. This desire is fulfilled by the `exec` family of system calls (e.g., `execve()`, `execl()`). When an `exec` call is invoked, the kernel, with swift precision, incinerates the child's current identity—its code, its data, its very stack and heap—and replaces it with the fresh, crisp image of a new program. Yet, through this metamorphosis, the process's soul, its PID, remains steadfast, a constant identifier in a changing world.

Process Lifecycle - Orchestra

`fork()` vs. `vfork()`: A Tale of Two Births and Kernel Optimization

But SVR4, ever the pragmatist, offered a sibling to `fork()`: the enigmatic `vfork()`. This wasn't merely a naming convention; it was an optimization born from the austere memory landscapes of 1988. Unlike its memory-duplicating cousin, `vfork()` was a gambit. The child, rather than receiving its own pristine copy of the parent's address space, was instead granted temporary stewardship *within the parent's very own address space*.

The parent, a benevolent but watchful guardian, would then pause, suspended in time, awaiting the child's crucial next move: either an `exec()` to shed its shared skin and embrace its own program, or a swift `_exit()` to gracefully depart. This daring optimization elegantly sidestepped the heavy toll of copying an entire address space, a true boon when an `exec()` was imminent.

However, like any daring maneuver, `vfork()` came with its own perilous tightrope walk. Should the child, in its shared dominion, dare to *modify* the parent's address space before its inevitable

`exec()`, chaos could ensue. The kernel, ever vigilant, enforces this strict contract to prevent a child's nascent scribbles from corrupting the parent's pristine canvas.

> **The Ghost of SVR4: Memory Constraints of Yore**
>
> In the dimly lit server rooms of 1988, memory was a precious commodity, measured in megabytes rather than gigabytes. The full duplication performed by `fork()` —especially for large processes—could be a performance bottleneck. `vfork()` was a clever, if slightly perilous, solution to mitigate this. It exploited the common `fork()`-then-`exec()` pattern, avoiding an expensive copy that would immediately be discarded.
>
> **Modern Contrast (2026):** Modern `fork()` on Linux uses Copy-On-Write: parent and child remain runnable and only incur page copies on write. `vfork()` still exists, but it retains its historical contract by suspending the parent until the child calls `exec()` or `_exit()`. The optimization is now mostly transparent, and the explicit `vfork()` dance is used sparingly, but the distinction remains: COW `fork()` preserves parallelism, while `vfork()` trades it for speed.

The u-Block: A Glimpse into a Process's Soul (SVR4 Edition)

Central to the SVR4 process's identity, beyond its PID, was the venerable **User Area**, affectionately known as the **u-block**. This kernel-resident data structure, unique to each process, was a veritable treasure trove of context. It housed the process's kernel stack, its per-process open file table, signal handling information, error numbers, and a myriad of other critical runtime details. In SVR4 terms, the u-block and the `proc` table entry together serve as the Process Control Block (PCB): the hardware context lives in the u-block, while system-wide identity and state live in `proc`.

```
/* Excerpted fields from struct user (sys/user.h) */
typedef struct user {
    char        u_stack[KSTKSZ];        /* kernel stack */
    char        u_stack_filler_1[2];
    char        u_fpvalid;              /* fp state valid */
    char        u_weitek;               /* uses weitek */
    struct tss386 *u_tss;               /* pointer to user TSS */
    ushort      u_sztss;                /* size of tss */
    ulong       u_sub;                  /* stack upper bound */
    proc_t *u_procp;            /* pointer to proc structure */
    int u_arg[MAXSYSARGS];      /* args to current syscall */
    label_t     u_qsav;                 /* longjmp label */
    char        u_error;                /* return error code */
    struct rlimit u_rlimit[RLIM_NLIMITS]; /* resource limits */
    k_sigset_t u_sigonstack;    /* signals on alt stack */
    k_sigset_t u_sigrestart;    /* restart syscalls */
    k_sigset_t u_sigmask[MAXSIG];       /* signals held in catcher */
    void        (*u_signal[MAXSIG])();  /* dispositions */
} user_t;
```

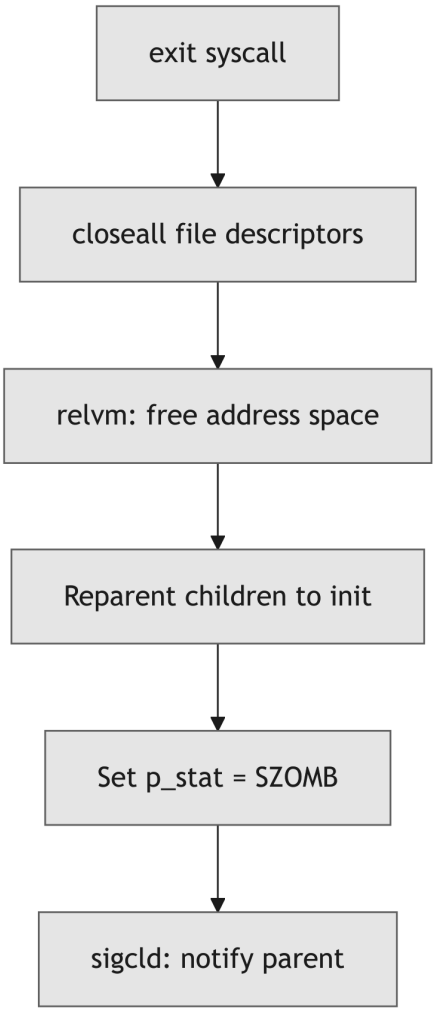
Code Snippet 1.2: The Intimate `u_block` (Excerpted)

The Ghost of SVR4: The u-block's Legacy

The u-block was a cornerstone of SVR4 process management, a compact and efficient way to store per-process kernel context. Its design reflected a time when every byte of memory was carefully accounted for. It served as a critical bridge between the generic `proc` table entry (which held system-wide process information) and the specific, rapidly changing context of a process's kernel execution.

Modern Contrast (2026): In contemporary Linux, the concept of a monolithic u-block has evolved. Its responsibilities are now distributed across various fields within the comprehensive `struct task_struct` and related data structures. While the `task_struct` in Linux is far more extensive and integrated, the spirit of the u-block—that direct, intimate connection to a process's ephemeral state—is still palpable within the modern kernel's design. The SVR4 u-block stands as an elegant predecessor, showing how fundamental information was once encapsulated.

The Final Curtain: Process Termination



Alas, even the most vibrant process must, at some juncture, meet its cessation. This can occur either by graceful self-annihilation or by an unforeseen, often forceful, intervention.

- **Voluntary Departure**: A process, having fulfilled its purpose, may choose to exit gracefully by invoking `exit()` or `_exit()`. The `_exit()` call, a more direct route, bypasses the user-space cleanup routines (like `atexit()` handlers or `stdio` buffer flushing), making it suitable for children destined for an immediate `exec()` or for robust error handling where user-space state is untrustworthy. Returning from the `main()` function in C is, in essence, a call to `exit()`.

- **Involuntary Eviction**: The kernel, or another process, can forcibly terminate a process, most often through the delivery of a signal. A `SIGKILL`, for instance, is the ultimate, unblockable

eviction notice, while a `SIGSEGV` (segmentation fault) marks a catastrophic misstep in memory access, compelling the kernel to intervene.

Upon termination, a process does not vanish instantaneously. Instead, it lingers as a spectral **"zombie"** process. In this enigmatic state, its computational essence is gone, its resources largely reclaimed by the kernel. Yet, a vestige remains: its entry in the kernel's Process Table and its u-block/PCB context, preserved long enough to hold the exit status. This spectral existence serves a vital purpose: it allows the parent process, through the `wait()` or `waitpid()` system calls, to collect the child's final report—its exit status—and only then does the kernel fully expunge the zombie, "reaping" its last kernel resources.

Should a parent process prematurely depart this digital realm before its children, those orphaned processes are not left to wander the wilderness. Instead, they are nobly adopted by the venerable `init` process (always PID 1), the primordial ancestor of all user-space processes. `init`, the steadfast caretaker, assumes the responsibility of patiently `wait()`ing for these adopted children, dutifully reaping their zombie forms when their time comes. This ensures that no process lingers eternally, hogging precious Process Table entries.

The Ghost of SVR4: The Importance of Reaping

Unreaped zombie processes, while consuming minimal resources, can exhaust the finite number of entries in the kernel's Process Table. In the SVR4 era, this was a tangible risk, capable of leading to system instability by preventing new processes from being created. The `wait()` family of calls wasn't just good practice; it was a necessary ritual to maintain kernel hygiene.

Modern Contrast (2026): While modern kernels have larger process tables, the principle remains. Zombie processes, if accumulated in large numbers, can still signify application bugs (e.g., parent processes not `wait()`ing for their children) and can indeed consume resources, albeit mostly Process Table entries, which are still finite. The `init` adoption mechanism remains a cornerstone of UNIX-like systems, a testament to the foresight of early designers.

The Dance of Existence: Process States

Throughout its ephemeral existence, a process whirls through a ballet of states, each signifying its current relationship with the CPU and the kernel's resources. The SVR4 kernel, with its meticulous oversight, maintains a **Process Table**—a grand ledger of `proc` structures, each a detailed dossier on a single process. This `proc` structure is the central repository, containing its PID, its current state, its security credentials, and a web of pointers to other critical kernel data, such as our familiar u-block.

The primary states in this grand choreography include:

- **Running**: The process is, at this very instant, clutching the CPU, executing its instructions with fervor.
- **Ready (Runnable)**: Poised and eager, the process is perfectly capable of running but is momentarily sidelined, patiently awaiting its turn on the CPU, a hopeful contender in the scheduler's queue.
- **Sleeping (Waiting)**: The process has temporarily retired from active computation, having voluntarily surrendered the CPU. It slumbers, awaiting a specific event—perhaps the completion of an I/O operation, the arrival of a signal, or the tick of a timer. It's in a state of suspended animation, ready to awaken when its desired event materializes.
- **Zombie**: As discussed, this is the spectral aftermath of a terminated process, awaiting its parent's final rites of `wait()` or `waitpid()`.
- **Stopped**: A process temporarily frozen in time, usually by an external force—a `SIGSTOP` or `SIGTSTP` signal—often seen in the graceful mechanics of shell job control. It can be resumed by a `SIGCONT` signal.

The Ghost of SVR4: Process Groups and Sessions for Order

The SVR4 kernel introduced sophisticated mechanisms like **process groups** and **sessions** not merely as abstract concepts, but as fundamental tools for imposing order on the chaos of multiple processes.

Process Groups are collections of related processes, typically created by a shell pipeline (e.g., `command1 | command2`). Signals, such as `SIGINT` (Ctrl+C), are often delivered to an entire process group, allowing for collective control. This was crucial for shell job control,

enabling a user to suspend (`SIGTSTP`), resume (`SIGCONT`), or terminate an entire pipeline of commands with a single keystroke.

Sessions elevate this concept further, encapsulating one or more process groups. A session typically corresponds to a login session or a terminal, acting as an insulating layer. When a terminal closes, a `SIGHUP` signal is often sent to the session leader, which can then propagate it to its process groups, gracefully terminating the applications associated with that session. This structured hierarchy was vital for managing interactive user environments and background jobs reliably.

The Kernel's Hand-Off: Context Switching

The very illusion of concurrency—of many processes seemingly running simultaneously on a single CPU—is conjured by the kernel's exquisite mastery of **context switching**. This is the heart of multitasking: the instantaneous, surgical act of preserving the entire operational state of the currently executing process, and then, with equal precision, reinstating the state of another.

When the scheduler, having made its momentous decision, dictates a change, the kernel embarks on a critical, low-level ballet. The CPU's fleeting memories are meticulously archived:

- **General-Purpose Registers**: The immediate workspace of the CPU, holding operands and results.
- **Program Counter (Instruction Pointer)**: The CPU's bookmark, indicating the next instruction to execute.
- **Stack Pointer**: The current top of the process's stack.
- **Process Status Word (PSW)**: A register reflecting the CPU's current operational mode, flags, and interrupt status.
- **Memory Management Unit (MMU) State**: Crucially, the base register pointing to the process's page tables (e.g., `CR3` on x86), which defines its unique virtual address space.

This meticulously saved "context" is stored within the process's `proc` structure (or its u-block, or a combination thereof), a snapshot awaiting the process's eventual return to the CPU.

At the core of this resurrection, particularly in SVR4, lies the often-unsung hero: the `resume()` function.

The `resume()` Function: Awakening a Slumbering Giant

The `resume()` function, a highly optimized, architecture-dependent assembly routine, is the incantation that breathes life back into a scheduled process. It is the final, atomic act of the context switch. Its mission: to load the previously saved CPU state of the chosen `newproc` into the physical CPU registers. On i386 this logic lives in `ml/misc.s` (ml/misc.s:1339-1370).

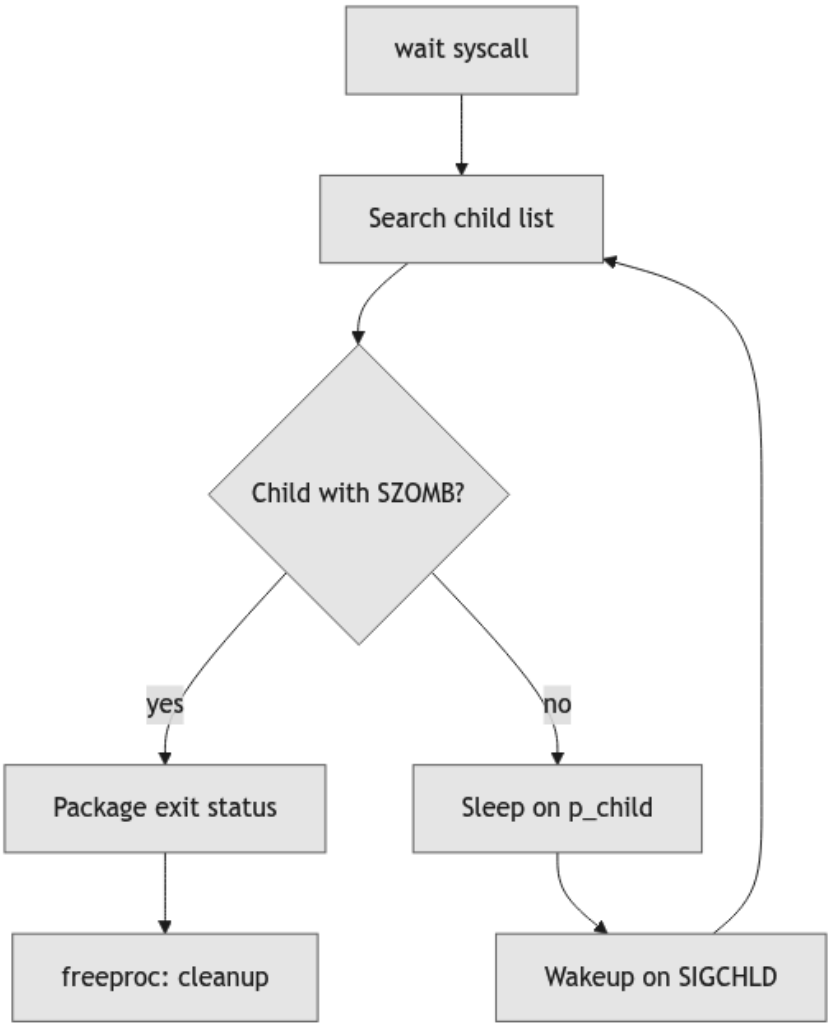
In practice, `resume()` performs the inverse operation of context saving:

1. It receives pointers to the `proc` structures (or their relevant context saving areas) of the `oldproc` (the process being switched *from*) and `newproc` (the process being switched *to*) to save the current CPU state (registers, stack pointer, program counter) of `oldproc`.
2. It updates the kernel's internal pointers to reflect the currently executing process.
3. **Crucially**, it restores the saved CPU state of `newproc`, including its `CR3` register to point to `newproc`'s page tables, effectively switching the active virtual address space.
4. It then "returns" into the `newproc`'s context, making it appear as if `newproc` was simply suspended and is now continuing from where it left off.

```
// Schematic outline for resume() (x86 specific)
resume(oldproc, newproc):
    ; Save oldproc's context (often done by the caller or an interrupt
handler)
    ; ...

    ; Switch current process pointers (conceptual)
    mov EAX, newproc_ptr
    mov [current_proc_ptr], EAX ; Update global current process pointer

    ; Restore newproc's MMU context (CR3 register)
    mov EAX, [newproc_page_table_base]
    mov CR3, EAX ; This is where the virtual address space magic happens!

    ; Restore newproc's general purpose registers, stack pointer, and
instruction pointer
    pop EBP
    pop ESI
    pop EDI
    pop EDX
    pop ECX
    pop EBX
    pop EAX

    mov ESP, [newproc_kernel_stack_pointer] ; Load new kernel stack

    ; Finally, return to the new process's execution flow
    ret_from_interrupt_or_call ; Jumps to newproc's saved EIP/CS
```

Code Snippet 1.3: The `resume()` Function's Orchestration (Schematic)

The `resume()` function is an exquisite piece of engineering, often residing at the very boundary between C code and the raw power of assembly. It operates with surgical precision, ensuring that the transition between processes is seamless and efficient, a blink-and-you-miss-it moment that underpins the entire multiprocessing paradigm.


wait syscall
Search child list
Child with SZOMB?
yes
no
Package exit status
Sleep on p_child
freeproc: cleanup
Wakeup on SIGCHLD

Process Scheduling

The Time-Keeper’s Art: Process Scheduling

Having explored the fascinating genesis and eventual cessation of a process in SVR4, let us now turn our gaze to the ballet master itself: the **Process Scheduler**. This unsung hero within the kernel is the ultimate arbiter of CPU time, meticulously deciding which hungry contender gets to taste the precious silicon next. In a world teeming with processes, each clamoring for attention, the scheduler’s role is akin to a benevolent, yet firm, traffic controller, ensuring that the CPU’s finite resources are distributed with purpose, policy, and a touch of SVR4 elegance.

The Grand Conductor: SVR4 Scheduling Classes

SVR4, with its characteristic architectural foresight, did not merely present a monolithic scheduling policy. Instead, it unveiled a highly flexible, multi-tiered framework—the **Scheduling Classes**. This innovative design allowed the kernel to employ distinct algorithms tailored to the very nature of the processes they governed. No longer would a time-critical sensor process be treated with the same casual indifference as a background compilation job. Each had its class, its policy, and its priority.

The primary dramatis personae in this scheduling theater were:

- **Real-Time (RT)**: These are the prima donnas of the kernel, demanding and receiving the highest accolades of priority. An RT process, once runnable, seizes the CPU with an almost tyrannical grip, running uninterrupted until it either voluntarily yields (blocks) or explicitly steps aside. Their existence is predicated on absolute predictability, making them indispensable for applications where temporal guarantees are not merely desirable, but mission-critical—think industrial control systems or specialized multimedia streams.

- **System (SYS)**: The backbone of the kernel itself, this class is reserved for the essential gears and levers of the operating system. Core kernel threads, interrupt handlers, and critical system daemons reside here. SYS processes operate at fixed, elevated priorities, ensuring that the kernel, the very heart of the system, remains responsive, vigilant, and unburdened by the whims of user-space frivolity.

- **Time-Sharing (TS)**: This is the bustling marketplace of user processes, the common folk of the SVR4 kingdom. TS processes are scheduled with an emphasis on fairness and equitable CPU distribution. They dance to a **round-robin** tune, each receiving a quantum of CPU time (a “time slice”), after which they are politely, but firmly, ushered back into the run queue. Their priorities are not static decrees but dynamic whispers, constantly adjusted by the kernel’s keen observation of their CPU appetites and periods of rest. A process that has recently been CPU-hungry might find its priority gently nudged downwards, while one that has patiently slept might receive a boost, ensuring that no single process hoards the CPU indefinitely.

The Ghost of SVR4: The Genesis of Modular Scheduling

The SVR4 scheduling classes were a remarkably prescient design choice in their era. Before this, many UNIX-like systems employed simpler, more monolithic scheduling algorithms that struggled to balance the demands of real-time responsiveness with general-purpose time-sharing. The modularity introduced by SVR4 allowed for greater flexibility and extensibility, foreshadowing the plugin-based scheduling architectures seen in modern kernels.

Modern Contrast (2026): Modern Linux kernels, while not explicitly using the "SVR4 classes" nomenclature, adopt similar multi-policy approaches. The Completely Fair Scheduler (CFS) handles general time-sharing, while `SCHED_FIFO` and `SCHED_RR` cater to real-time needs, and a `SCHED_DEADLINE` class provides even stronger temporal guarantees. The SVR4 design laid crucial groundwork for separating policy from mechanism in kernel scheduling, a principle that endures to this day.

Process Scheduling - Restaurant Kitchen

The SVR4 Scheduler: A Priority-Driven Maestro

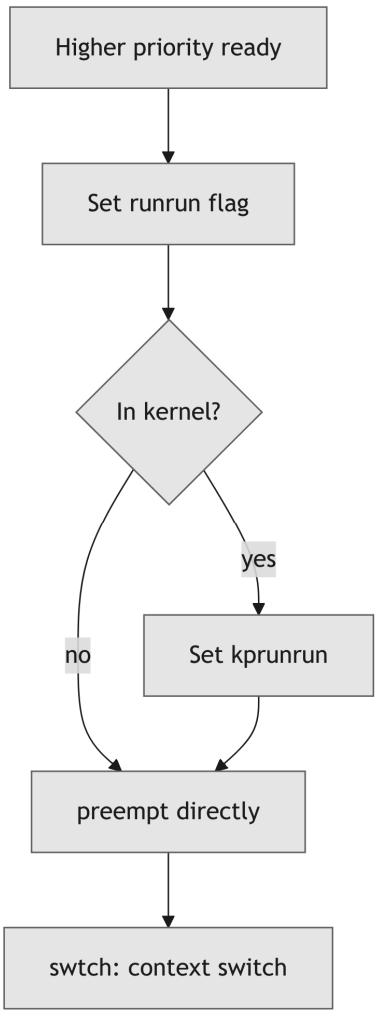
At the operational core of the SVR4 scheduler lies a finely tuned priority system, managed through a series of **run queues**. Imagine a grand antechamber with many doors, each labeled with a priority level within a specific scheduling class. When a process becomes runnable—awakening from a sleep, or finishing an I/O operation—it is meticulously placed onto its appropriate run queue.

When a CPU, that insatiable hunger for instructions, becomes available (perhaps because a running process decided to block, or its allotted time slice expired), the scheduler springs into action. Its directive is absolute: **always select the highest-priority runnable process from the highest-priority non-empty scheduling class.**

Within the `Time-Sharing` class, this mechanism becomes particularly nuanced. The scheduler isn't merely a static gatekeeper; it's an active observer. Processes that tirelessly grind away,

consuming vast tracts of CPU time, will find their dynamic priorities gently, but persistently, decreasing. Conversely, those processes that have patiently waited, perhaps slumbering for an I/O event, will experience a compensatory increase in their priority. This astute mechanism, often termed **“priority aging”**, is the scheduler’s subtle art of preventing CPU starvation, ensuring that even the most humble Time-Sharing process will eventually receive its moment in the silicon sun.

The Interruption: Preemption



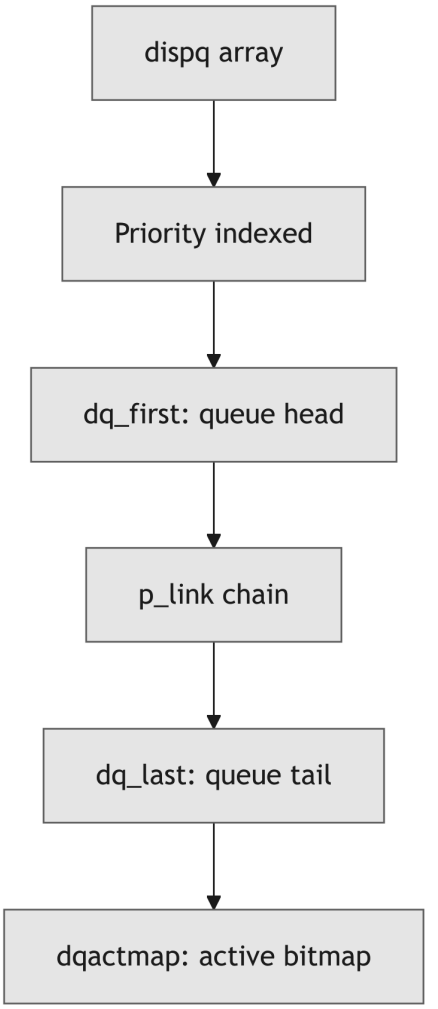
SVR4, a marvel of its time, was a **preemptive kernel**. This means that the CPU’s current tenant—the running process—could be summarily interrupted and forced to relinquish its hold, even if it

hadn’t completed its quantum of time or explicitly paused. Preemption is the very essence of responsive multitasking, ensuring that critical events or higher-priority demands are never ignored.

Preemption could be triggered by several compelling forces:

- **The Ascent of a Monarch**: Should a process of unequivocally higher priority (an `RT` process, for example) suddenly become runnable, the currently executing, lower-priority process is immediately—and without ceremony—ejected from the CPU. The new monarch reigns supreme.
- **The Tyranny of the Time Slice**: For `Time-Sharing` processes, the CPU is not an endless buffet but a carefully rationed meal. Once a process consumes its entire allotted **time slice**, the scheduler, with clockwork precision, preempts it. The process is then returned to the run queue, often with a subtly adjusted (read: reduced) priority, awaiting its next turn.
- **The Call of the Hardware**: External **interrupts**, those urgent cries from the hardware (a disk signaling completion, a network packet arriving), can also trigger preemption. The CPU immediately diverts its attention to the interrupt handler. Upon completion, the scheduler re-evaluates the landscape, often leading to a context switch if a higher-priority task is now runnable.

Dispatch Queues: The Scheduler’s Ledger



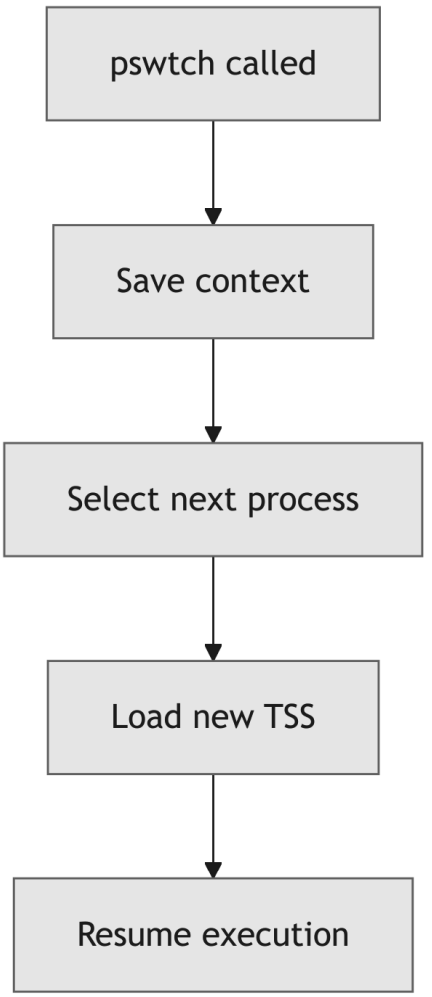
To manage this dynamic ebb and flow of processes, the SVR4 kernel employs **dispatch queues**. In practice, these are explicit data structures, often linked lists, each corresponding to a distinct priority level. When a process transitions from a blocked state to a runnable state, it is meticulously inserted into the appropriate dispatch queue for its current priority.

The scheduler’s central loop, a relentless cycle of decision and execution, can be distilled into these fundamental steps:

1. **Survey the Landscape**: Continuously examine the dispatch queues for any runnable processes.
2. **Identify the Chosen One**: From the highest-priority non-empty dispatch queue, pluck the next process slated for execution.

3. **The Great Switch**: Initiate a **context switch** to the selected process, involving the intricate dance of saving the old state and restoring the new state, as we discussed with the `resume()` function.
4. **Repeat Ad Infinitum**: The cycle perpetuates, an eternal vigil ensuring the CPU is always serving the highest demand.

This sophisticated yet elegant dispatching mechanism is how SVR4 maintained both responsiveness for critical tasks and fairness for general computation, a testament to the enduring principles of efficient operating system design.

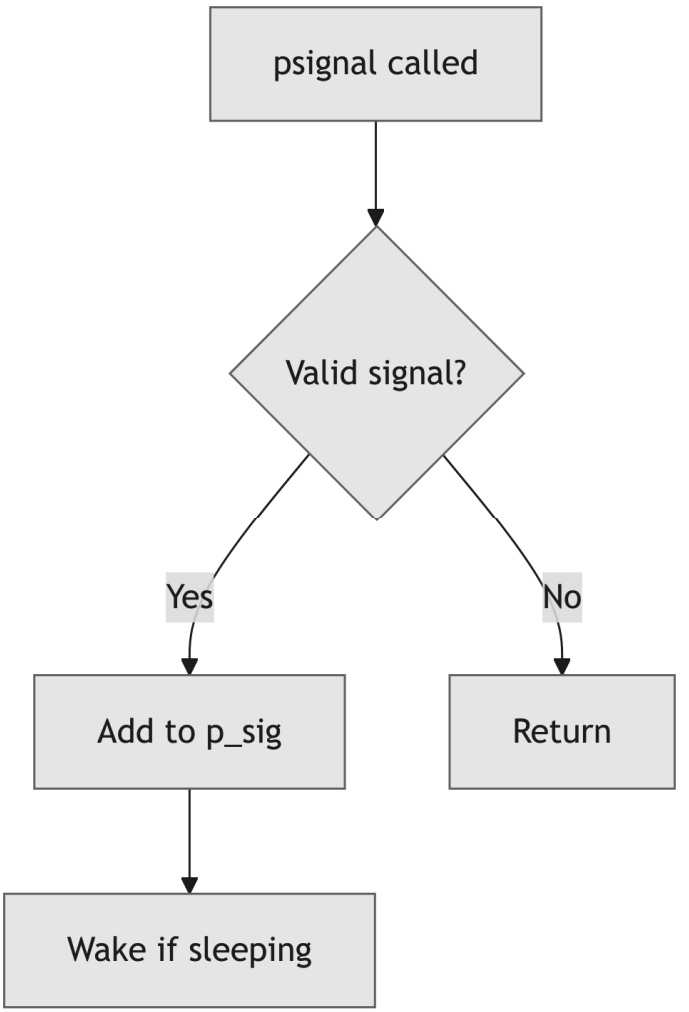


Signal Handling

Whispers from the Kernel: The Art of Signal Handling

In the cacophony of a busy kernel, where processes dance to the CPU's tune, there exists a delicate system of asynchronous communication: **signals**. These are not polite invitations, but urgent whispers, sharp nudges, or even outright shouts from the kernel, hardware, or other processes, designed to notify a process of an event that demands its attention. From an illegal memory access to the tap on the shoulder from a user pressing `Ctrl+C`, signals are the kernel's primitive yet powerful mechanism for event-driven interaction. To truly master SVR4, one must understand this "dark art"—how signals are born, how they travel, and how a process, or indeed the kernel, chooses to respond.

The Genesis of a Whisper: Signal Posting



A signal's journey begins with its **posting**. This is the act of marking a target process as having a pending signal. Whether triggered by a hardware fault (like a `SIGSEGV`), a user's command (`kill -9 PID`), or a kernel event (like a child process terminating, sending a `SIGCHLD`), the kernel's `psignal()` function (often starting its work around `sig.c:103` in the SVR4 source) acts as the signal's dispatcher.

The dispatch mechanism is surprisingly nuanced, especially for job control signals, which carry specific implications for process states. For instance, a `SIGCONT` (continue signal) is a powerful command:

```
/* Excerpt from sigtoproc() in sig.c:112-169 */
if (sig == SIGCONT) {
    if (p->p_sig & sigmask(SIGSTOP))
        sigdelq(p, SIGSTOP);
    if (p->p_sig & sigmask(SIGTSTP))
        sigdelq(p, SIGTSTP);
    if (p->p_sig & sigmask(SIGTTOU))
        sigdelq(p, SIGTTOU);
    if (p->p_sig & sigmask(SIGTTIN))
        sigdelq(p, SIGTTIN);
    sigdiffset(&p->p_sig, &stopdefault);
    if (p->p_stat == SSTOP && p->p_whystop == PR_JOBCONTROL) {
        p->p_flag |= SXSTART;
        setrun(p);
    }
} else if (sigismember(&stopdefault, sig)) {
    sigdelq(p, SIGCONT);
    sigdelset(&p->p_sig, SIGCONT);
}

if (!tracing(p, sig) && sigismember(&p->p_ignore, sig))
    return;

sigaddset(&p->p_sig, sig);

if (p->p_stat == SSLEEP) {
    if ((p->p_flag & SNWAKE)
      || (sigismember(&p->p_hold, sig) && !EV_ISTRAP(p)))
        return;
    setrun(p);
} else if (p->p_stat == SSTOP) {
    if (sig == SIGKILL) {
        p->p_flag |= SXSTART|SPSTART;
        setrun(p);
    } else if (p->p_wchan && ((p->p_flag & SNWAKE) == 0))
        unsleep(p);
} else if (p == curproc) {
    u.u_sigevpend = 1;
}
```

Code Snippet 1.4: Job Control Signal Posting Logic (Excerpt)

This snippet illustrates the delicate dance between `SIGCONT` and stop signals (`SIGSTOP` , `SIGTSTP` , etc.). Posting a `SIGCONT` implicitly cancels any pending stop signals and, crucially, can awaken a process that was previously `SSTOP` (stopped). Conversely, attempting to stop a process

will clear any pending `SIGCONT`. This mutual exclusivity is the kernel's way of enforcing consistent job control semantics, preventing paradoxical states.

Once these job control nuances are handled, the signal is added to the process's **pending signal set** (`p->p_sig`). If the target process is currently in an interruptible sleep (`SSLEEP` without the `SNWAKE` flag, meaning it can be awakened by a signal), the `setrun()` function is invoked to stir it from its slumber, making it runnable. However, the venerable `SIGKILL` is a brute-force exception: it will always awaken even a `SSTOP` process, as termination is its ultimate, undeniable decree.

For the currently executing process, a special flag (`u.u_sigevpend`) is set. This ensures that any signals posted to `self` (even from an interrupt context) are noticed and handled *before* the kernel finally relinquishes control and returns to user mode. This separation of "posting" (the asynchronous event) from "delivery" (the synchronous handling upon returning to user space) is a cornerstone of UNIX signal reliability.

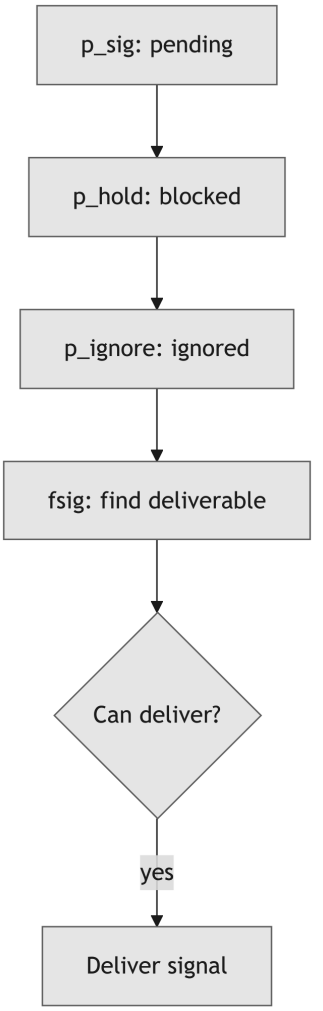
The Ghost of SVR4: Reliability in a Preemptive World

The careful separation of signal posting from delivery was a design marvel for its time, ensuring consistency even when signals could arrive asynchronously from various sources (hardware, other processes). This two-phase approach prevented many race conditions that could plague simpler signal implementations. The kernel explicitly waits until a safe point (return to user mode) to process signals.

Modern Contrast (2026): While the fundamental principles remain, modern kernels often employ more sophisticated, per-thread signal queues and more granular control over signal delivery to individual threads within a multi-threaded process. However, the core idea of signals as asynchronous notifications, processed at specific, safe points, traces its lineage directly back to robust designs like SVR4's.

Signal Handling - Whispers and Nudges

The Sentry's Gate: Signal Masks and Sets



A process is not merely a passive recipient of signals; it possesses a sophisticated array of defenses and preferences, managed by the kernel through various **signal bitmasks** and predefined **signal sets**. These act as filters, allowing a process to selectively block, ignore, or trace specific signals, thereby controlling its susceptibility to external interruptions.

The kernel maintains several critical bitmasks for each process (conceptually depicted in Figure 1.3.3):

- **`p_sig` (Pending Signals)**: This bitmask holds the signals that have been posted to the process but have not yet been delivered. Think of it as a process's "inbox" for incoming notifications.

- **`p_hold` (Blocked Signals)**: Signals whose corresponding bit is set in `p_hold` are *blocked*. They will not be delivered to the process even if they are pending (`p_sig` has their bit set). They effectively wait in `p_sig` until they are unblocked. This is crucial for critical sections of code where a process needs to avoid asynchronous interruptions.
- **`p_ignore` (Ignored Signals)**: Signals in this set are simply discarded upon delivery. The process explicitly tells the kernel, "Don't bother me with these; I don't care."
- **`p_sigmask` (Traced Signals)**: Primarily used by debuggers, signals in this set cause the process to stop and notify its tracing parent when they are about to be delivered. This allows a debugger to intercept and potentially alter the signal's fate.

Beyond these per-process masks, SVR4 defines several global `k_sigset_t` (kernel signal set type) bitmasks that encode fundamental, unchangeable behaviors (found in `sig.c:72-89`):

```
// Excerpt from sig.c:72-89 - Immutable Signal Properties
k_sigset_t cantmask = (sigmask(SIGKILL)|sigmask(SIGSTOP));
k_sigset_t cantreset = (sigmask(SIGILL)|sigmask(SIGTRAP)|sigmask(SIGPWR));
k_sigset_t stopdefault = (sigmask(SIGSTOP)|sigmask(SIGTSTP)
            |sigmask(SIGTTOU)|sigmask(SIGTTIN));
k_sigset_t coredefault =
(sigmask(SIGQUIT)|sigmask(SIGILL)|sigmask(SIGTRAP)
            |sigmask(SIGIOT)|sigmask(SIGEMT)|sigmask(SIGFPE)
            |sigmask(SIGBUS)|sigmask(SIGSEGV)|sigmask(SIGSYS)
            |sigmask(SIGXCPU)|sigmask(SIGXFSZ));
```

Code Snippet 1.5: Predefined Kernel Signal Sets

These predefined sets are the kernel's immutable laws regarding signals:

- **`cantmask`** : This set, containing `SIGKILL` and `SIGSTOP` , represents signals that cannot be blocked, ignored, or caught. `SIGKILL` ensures a process can always be forcibly terminated, while `SIGSTOP` guarantees it can always be halted by the system. There are no safe words against these.
- **`cantreset`** : Signals like `SIGILL` (illegal instruction) and `SIGTRAP` (trap instruction) cannot be reset to their default handlers by `SA_RESETHAND` . This prevents scenarios where a handler for a CPU exception might accidentally re-enable an infinite loop, crashing the system.
- **`stopdefault`** : These are the signals whose default action is to stop the process (e.g., `SIGTSTP` from `Ctrl+Z`).

- **`coredefault`** : Signals in this set (e.g., `SIGSEGV` for segmentation fault, `SIGQUIT` for quit) will, by default, cause the process to terminate and dump a core file for post-mortem debugging.

Understanding these masks and sets is paramount, for they define the very boundaries of a process's control over its own execution flow in the face of asynchronous events.

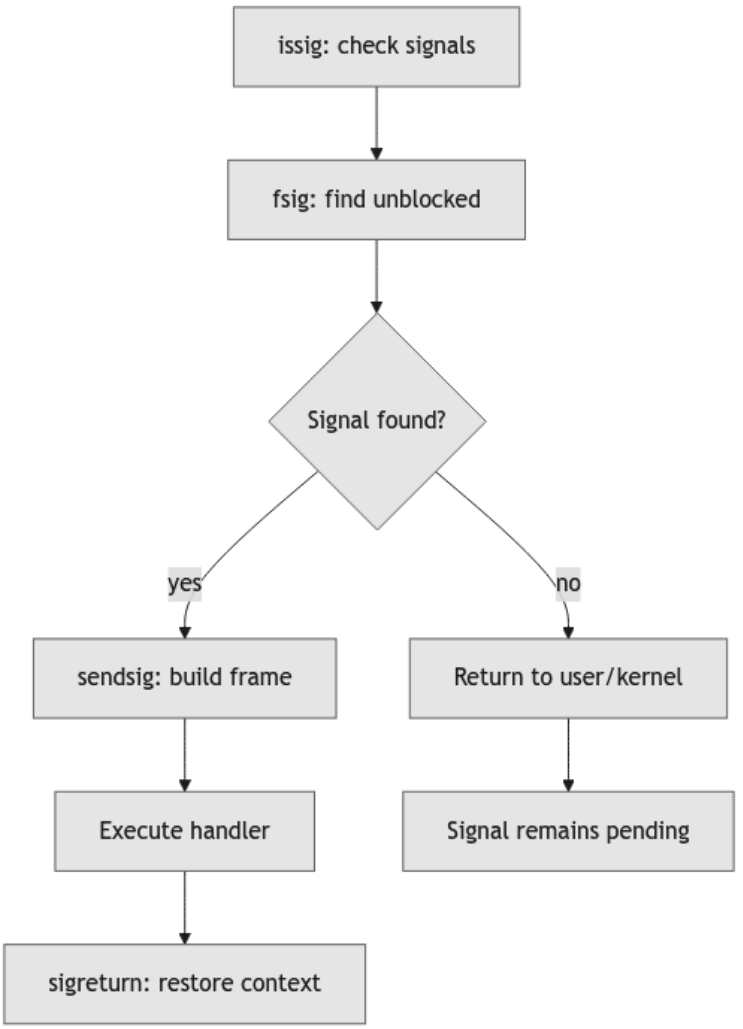
The Grand Interrogation: Signal Delivery

A signal, once posted and nestled in `p_sig`, does not immediately disrupt the process's user-mode reverie. The SVR4 kernel is far more discerning. Signal delivery is typically a synchronous event, carefully orchestrated to occur at safe points—specifically, when a process is about to transition from kernel mode back into user mode. This is the moment for the `issig()` function (`sig.c:224`) to perform its grand interrogation.

Think of `issig()` as the gatekeeper, constantly scanning the process's pending signals (via the `fsig()` helper function, which dutifully checks `p->p_sig` for the first unblocked, unignored signal). Its loop is relentless, ensuring no deliverable signal is overlooked:

```
/* Excerpt from issig() in sig.c:224-324 */
for (;;) {
    if (why == FORREAL
      && (p->p_flag & SPRSTOP)
      && stop(p, PR_REQUESTED, 0, 0))
        swtch();

    if ((sig = p->p_cursig) != 0) {
        p->p_cursig = 0;
        if (why == JUSTLOOKING
          || (p->p_flag & SPTRX)
          || (!sigismember(&p->p_ignore, sig)
            && !isjobstop(sig)))
            return p->p_cursig = (char)sig;
        if (p->p_cursig == 0 && p->p_curinfo != NULL) {
            kmem_free((caddr_t)p->p_curinfo,
              sizeof(*p->p_curinfo));
            p->p_curinfo = NULL;
        }
        continue;
    }

    for (;;) {
        if ((sig = fsig(p)) == 0)
            return 0;
        if (tracing(p, sig)
          || !sigismember(&p->p_ignore, sig)) {
            if (why == JUSTLOOKING)
                return sig;
            break;
        }
        sigdelset(&p->p_sig, sig);
        sigdelq(p, sig);
    }

    sigdelset(&p->p_sig, sig);
    p->p_cursig = (char)sig;
    ASSERT(p->p_curinfo == NULL);
    sigdeq(p, sig, &p->p_curinfo);
}
```

Code Snippet 1.6: The `issig()` Signal Delivery Loop (Excerpt)



The remaining portion handles tracing stops and debugger release (`stop()` and `procxmt()`), then loops to honor requested stops before returning (sig.c:325-360).

The `issig()` function follows a clear hierarchy of handling:

1. **Ignored Signals**: If `fsig()` discovers a signal that is marked in `p->p_ignore` , it's summarily discarded (`sigdelset` , `sigdelq`), and `issig()` continues its scan, as if the signal never existed.
2. **Traced Signals**: For processes being debugged (`tracing(p, sig)`), a deliverable signal causes the process to stop via `stop()` (`sig.c:343`). This notifies the tracing parent (the debugger), allowing it to inspect the process's state and even potentially alter the signal or its action. The process enters an `SSTOP` state and yields the CPU (`swtch()`), patiently awaiting the debugger's command.

3. **Job Control Stop Signals**: If a `stopdefault` signal (`SIGSTOP` , `SIGTSTP` , `SIGTTIN` , `SIGTTOU`) is found and its default action is active (i.e., not ignored or caught), the `isjobstop()` function intervenes (`sig.c:177`). This again stops the process, setting its state to `SSTOP` , and notifies its process group leader (and potentially the parent) via `sigcld()`). This is the fundamental mechanism behind shell job control, allowing you to suspend a foreground process with `Ctrl+Z` .

Only after passing these gauntlets is a signal promoted to `p_cursig` , indicating it's ready for its ultimate **action**.

The Process's Reply: Signal Action and `sigaction`

Once `issig()` has crowned a signal as `p_cursig` , the `psig()` function (`sig.c:420`) steps in to determine the process's final response. SVR4, adhering to the POSIX standard, provides a robust framework for specifying how a process reacts to a signal, primarily through the `sigaction()` system call. This system call is the process's declaration of intent, a detailed instruction set for its signal handling strategy.

There are three fundamental actions a process can take:

1. **Ignore**: As discussed, if a signal is in `p->p_ignore` , it's silently discarded. This is the simplest form of dismissal.
2. **Default**: Each signal has a predefined default action by the kernel. This can range from:
    - **Terminate**: The process simply exits (`exit()`). Signals like `SIGHUP` , `SIGINT` , `SIGTERM` fall here.
    - **Terminate and Core Dump**: The process exits, but first writes an image of its memory (a "core dump") to disk for post-mortem debugging. `SIGSEGV` , `SIGQUIT` , `SIGFPE` are prime examples (members of `coredefault`).
    - **Stop**: The process suspends its execution. `SIGSTOP` , `SIGTSTP` are the culprits here.
    - **Ignore**: A few signals, like `SIGCHLD` (child status change) and `SIGURG` (urgent condition on a socket), are ignored by default.

3. **Catch (User-Defined Handler)**: This is where a process truly asserts its control. Using `sigaction()`, a process can specify a user-mode function (a "signal handler") to be executed when a specific signal is delivered. This allows applications to gracefully respond to events, such as saving state before termination (`SIGTERM`) or resetting broken network connections.

The `sigaction` structure is the heart of this customization (sys/signal.h:78-87):

```
struct sigaction {
    int sa_flags;
    void (*sa_handler)();
    sigset_t sa_mask;
    int sa_resv[2];
};
```

Code Snippet 1.7: The `sigaction` Structure (sys/signal.h)

When a user-defined handler is invoked (`sendsig()` in `sig.c:467`), the kernel performs a meticulous setup:

- **Blocking Signals (`sa_mask`, `SA_NODEFER`)**: The `sa_mask` in the `sigaction` structure specifies a set of signals to be *blocked* (added to `p_hold`) for the duration of the handler's execution. This prevents these signals from interrupting the handler itself, ensuring its atomic execution. Crucially, the signal currently being handled is *automatically blocked* to prevent reentrant delivery unless the `SA_NODEFER` flag is set.

- **One-Shot Handlers (`SA_RESETHAND`)**: If the `SA_RESETHAND` flag is set, the kernel automatically resets the signal's disposition to `SIG_DFL` (default) *before* invoking the handler. This creates a "one-shot" handler that only executes once, after which the signal reverts to its default behavior.

- **The Signal Frame (`sendsig()`)**: This is where the kernel works its magic at the boundary of kernel and user space. The `sendsig()` function (which is typically architecture-specific, meaning it differs for i386 vs. SPARC) meticulously constructs a "signal frame" on the user's stack. This frame is a temporary data structure containing:
    - The signal number.
    - Extended signal information (`siginfo_t`, if `SA_SIGINFO` was specified).

- A snapshot of the process's user-mode CPU context (`ucontext_t` or equivalent, including registers, program counter, stack pointer).
- The return address, carefully set to point to the user-defined signal handler.

The kernel then modifies the process's saved CPU state (the one that would normally be restored upon return from kernel mode) to make it appear as if the process had called its own signal handler. When the kernel returns to user mode, instead of resuming the interrupted code, the CPU jumps directly to the signal handler.

When the user-defined signal handler completes its execution, it does *not* typically return using a standard `ret` instruction. Instead, it must invoke the `sigreturn()` system call. This specialized system call is the handler's graceful exit. `sigreturn()` 's sole purpose is to dismantle the signal frame, restore the original process context (from before the signal delivery), and atomically unblock any signals that were blocked by `sa_mask` . This allows the process to seamlessly resume execution from precisely where it was interrupted, often unaware of the kernel's swift, intricate intervention.

The Ghost of SVR4: The Evolution of Signal Semantics

Early UNIX signals were notoriously unreliable and fraught with race conditions (System V signals being a prime example). Signals could be lost, or handlers could be re-entered unpredictably. The `sigaction()` interface, along with the detailed blocking semantics (`sa_mask` , `SA_NODEFER`) and the `sigreturn()` mechanism, were crucial advancements introduced (or standardized by POSIX, which SVR4 embraced) to make signal handling robust and predictable. This provided developers with the necessary tools to write reliable signal-driven applications.

Modern Contrast (2026): While `sigaction()` remains the preferred and most robust API for signal handling in modern UNIX-like systems (including Linux), the evolution of multi-threading (`pthreads`) introduced new complexities. Signals are now often delivered to specific threads rather than entire processes, requiring thread-specific signal masks (`pthread_sigmask`). Despite these advancements, the core mechanisms of signal posting, masking, delivery, and the signal frame concept are directly descended from the foundations laid by systems like SVR4.

Safeguarding the Kernel: Implementation Notes

The SVR4 signal mechanism is fortified with careful considerations for race conditions and system stability:

- **`SIGKILL` 's Supremacy**: A pending `SIGKILL` always takes precedence, ensuring that a process can be terminated regardless of what other signals it's handling or blocking. There is no escape from `SIGKILL` .
- **Stop vs. Kill**: `SIGKILL` cannot be stopped. If a `SIGKILL` is pending, any attempt to stop the process is overridden, guaranteeing unstoppable termination.
- **Timely Delivery**: The `u.u_sigevpend` flag for the current process is a subtle yet vital mechanism. It forces the kernel to check for signals immediately before returning to user mode, ensuring that signals posted even in an interrupt context (e.g., a timer interrupt posting a `SIGALRM`) are noticed without undue delay.
- **Queued Signal Information**: For real-time signals (which SVR4 did support, though less prevalently than in modern systems) or when extended signal information is requested (`SA_SIGINFO`), the kernel maintains detailed queues of `siginfo_t` structures. Functions like `sigdelq()` and `sigdeq()` manage this richer information, allowing the handler to receive not just the signal number, but also *why* and *where* the signal occurred.

This careful separation of the asynchronous act of signal posting from the synchronous, controlled act of signal delivery, coupled with robust masking and a well-defined action framework, underpins the reliability of SVR4's signal architecture.

System Calls

The Kernel's Gateway: System Calls Interface

In the austere landscape of a protected mode operating system, user-space processes are like diligent workers confined to a meticulously guarded factory floor. They can perform their tasks with gusto, but should they require raw materials from the warehouse (disk I/O), or need to communicate with central management (process creation), they cannot simply wander off. Instead, they must respectfully knock on a very specific door: the **System Call Interface**. This is the kernel's tightly controlled, rigorously audited gateway, providing the only legitimate means for a user process to request privileged operations and interact with the very heart of the operating system.

The Knock on the Door: System Call Entry

The journey into the kernel begins with a solemn ritual. When a user program requires a kernel service (e.g., `read()`, `write()`, `fork()`), it doesn't directly execute kernel code. Instead, it typically calls a wrapper function in the C standard library. This wrapper performs a crucial preparatory step: it loads a unique **system call number** (an integer identifier for the desired service) into a designated CPU register, usually `EAX` on the i386 architecture.

Then, with a flourish, the wrapper triggers a **software interrupt**, specifically `INT 0x80` in classic i386 implementations. This `INT 0x80` instruction is the magical incantation that causes the CPU to shed its user-mode privileges and elevate itself to the exalted kernel mode. It's an atomic, hardware-assisted transition designed for security and efficiency.

> **Modern Evolution:** Contemporary x86 processors provide faster system call entry mechanisms: `SYSENTER` (Intel) and `SYSCALL` (AMD), which avoid the overhead of a full interrupt gate transition. These instructions perform a more direct privilege-level switch, bypassing the interrupt descriptor table lookup and offering significantly improved performance for high-frequency system calls. Modern kernels typically support both the legacy `INT 0x80` path (for compatibility) and the fast entry path.

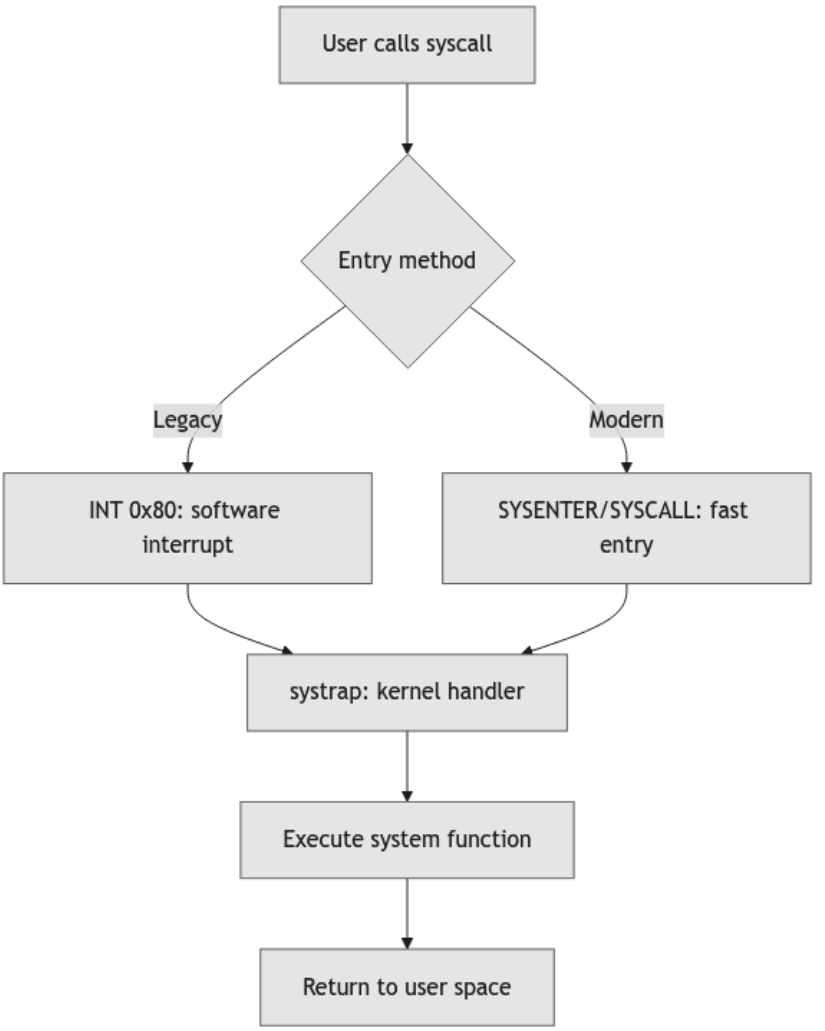
Upon activation of the entry instruction, control is transferred to the kernel's dedicated entry point, often a highly optimized assembly routine like `systrap`. This routine, the vigilant doorman of the kernel, immediately performs several critical tasks:

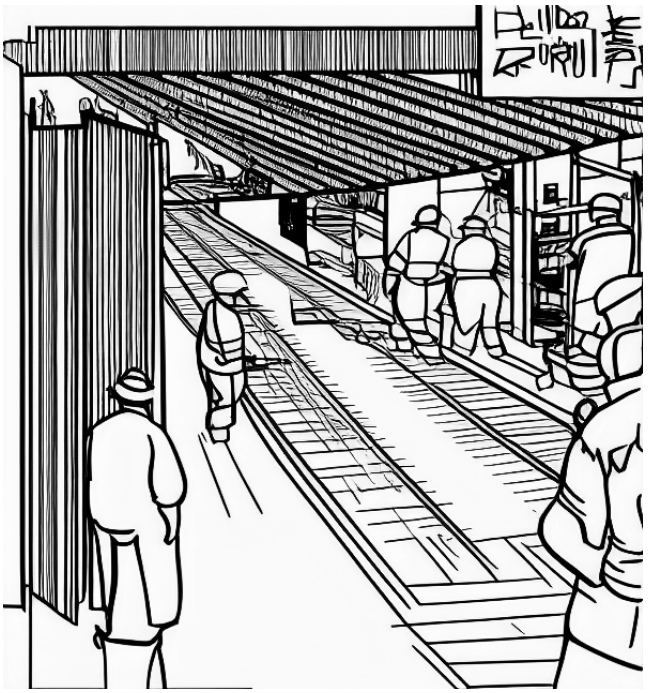
1. **Context Preservation**: The first and foremost duty is to meticulously save the entire user-mode CPU context (registers, stack pointer, program counter, flags) onto the kernel stack. This snapshot ensures that when the system call completes, the user process can seamlessly resume execution from precisely where it left off, as if the kernel interlude never happened.
2. **Privilege Check**: The kernel verifies that the incoming request is legitimate and that the system call number itself is valid.
3. **Argument Retrieval**: The arguments for the system call, which the user process pushed onto its own stack, must be safely copied from user space to kernel space. This copy is not merely a transfer; it's a careful validation, ensuring that user-provided pointers do not reference invalid or malicious memory locations within the user's or, worse, the kernel's address space.

On i386, the low-level entry stub is in `ml/ttrap.s`. The system call gate saves registers, clears segment registers, and then calls `systrap()` in C (ml/ttrap.s:312-348):

```
sys_call:
        subl    $8, %esp        / pad with dummy ERRCODE, TRAPNO
        pusha                   / save user registers
        pushl   %ds
        pushl   %es
        pushl   %fs
        pushl   %gs
        ...
        xorw    %ax, %ax
        movw    %ax, %fs
        movw    %ax, %gs
        call    kentry_check
        sti
        movb    $0, u+u_sigfault
        pushl   %esp
        call    systrap
        addl    $4, %esp
```

Code Snippet 1.8a: The `systrap` Entry Stub (ml/ttrap.s)


User calls syscall
Entry method
Legacy
Modern
INT 0x80: software interrupt
SYSENTER/SYSCALL: fast entry
systrap: kernel handler
Execute system function
Return to user space



System Calls - Protected Factory

The `sysent` Table: The Kernel's Service Directory

Once safely within the kernel, the `systrap` routine consults the **`sysent` table** (defined in `sysent.c`), which serves as the kernel's authoritative directory of all available system calls. This table is an array of `struct sysent` entries, indexed directly by the system call number (the value initially placed in `EAX`).

Each entry in the `sysent` table is a compact but vital record:

```
/* Excerpt from os/sysent.c */
struct sysent sysent[] = {
    0, 0, nosys,                        /*  0 = indir */
    1, 0, (int(*)())rexit,              /*  1 = exit */
    0, 0, fork,                 /*  2 = fork */
    3, SETJUMP|ASYNC|IOSYS, read,       /*  3 = read */
    3, SETJUMP|ASYNC|IOSYS, write,      /*  4 = write */
    3, SETJUMP, open,           /*  5 = open */
    1, SETJUMP, close,          /*  6 = close */
    0, SETJUMP, wait,           /*  7 = wait */
    2, SETJUMP, creat,          /*  8 = creat */
    2, 0, link,                 /*  9 = link */
    1, 0, unlink,                       /* 10 = unlink */
    2, 0, exec,                 /* 11 = exec */
    1, 0, chdir,                        /* 12 = chdir */
    0, 0, gtime,                        /* 13 = time */
    3, 0, mknod,                        /* 14 = mknod */
    2, 0, chmod,                        /* 15 = chmod */
    3, 0, chown,                        /* 16 = chown */
};
```

Code Snippet 1.8: The `sysent` Table (Excerpt)

Here, the first field typically indicates the number of arguments the system call expects, the second might hold flags, and the third is a function pointer to the actual kernel handler responsible for executing the system call's logic. This design provides an efficient and organized way for the kernel to dispatch control to the correct handler based on the system call number provided by the user process.

The Language of Request: Argument Passing

The kernel's `systrap` entry point, having identified the correct system call handler via `sysent`, now orchestrates the transfer of arguments from the user process to the kernel handler. This is a highly sensitive operation, as the kernel must never implicitly trust data originating from user space.

Arguments are typically pushed onto the user process's stack before the `INT 0x80` instruction. The kernel reads them in `systrap()` using `lfuword()` and the per-process argument array (os/trap.c:808-836):

```
/* Excerpt from systrap() in os/trap.c */
u.u_syscall = r0ptr[EAX];
if ((u.u_syscall & 0xff) >= sysentsize)
    u.u_syscall = 0;
scall = u.u_syscall & 0xff;
callp = &sysent[scall];
u.u_sysabort = 0;

{
    register u_int *sp = (u_int *)r0ptr[UESP];
    register u_int i;
    sp++;       /* skip return addr */
    for (i = 0; i < callp->sy_narg; i++) {
        u.u_arg[i] = lfuword((int *)(sp++));
    }
}
```

Code Snippet 1.9: Argument Harvesting in `systrap()`

This `copyin()` (and its counterpart `copyout()` for returning data) is more than just a memory copy; it includes critical **validation** steps:

1. **Address Range Check**: Does the user-provided address for an argument (e.g., a buffer for `read()`) actually fall within the user process's allocated virtual address space? An attempt to read from or write to arbitrary memory locations (especially kernel space) would be a severe security breach.
2. **Permissions Check**: Does the user process have the necessary permissions to access the memory at the given address? For instance, `copyin()` ensures read access, while `copyout()` ensures write access.
3. **Size Limits**: If a size is specified (e.g., number of bytes to read), the kernel ensures that the operation does not exceed the bounds of the user-provided buffer or other reasonable limits.

Only after these stringent checks are passed are the arguments provided to the system call handler. The handler then executes its core logic, operating within the full privileges and context of the kernel. Results, if any, are typically placed into an `rval_t` structure (e.g., containing `r_val1` and `r_val2` for system calls like `pipe()` that return two values, like a file descriptor pair). This ensures a standardized return mechanism from kernel to user space.

The Ghost of SVR4: The Legacy of INT 0x80

The `INT 0x80` mechanism was the standard, and often the only, way to enter the kernel on i386 systems in the SVR4 era. It was robust, well-understood, and provided the necessary protection boundary. However, each software interrupt involves significant overhead due to the saving and restoring of context and the transition through the interrupt descriptor table.

Modern Contrast (2026): Modern x86 processors (and other architectures) feature specialized, faster instructions for system call entry, such as `SYSENTER` / `SYSEXIT` (Intel) and `SYSCALL` / `SYSRET` (AMD). These instructions offer a more streamlined, hardware-optimized path directly into the kernel, bypassing some of the interrupt overhead. While `INT 0x80` may still be supported for compatibility, high-performance applications and modern operating systems prefer these dedicated fast system call mechanisms, a testament to the continuous pursuit of efficiency in the face of ever-increasing system call frequencies.

The Return Journey: System Call Exit

Having successfully navigated the kernel's labyrinth, performed its sacred duties, and perhaps even transformed the system state, the system call handler must now orchestrate a graceful exit. The journey back from the privileged kernel mode to the constrained user mode is not a mere reversal of entry but a carefully choreographed sequence of checks and context restorations. The `systrap_rtn` routine, the counterpart to `systrap`, assumes command for this delicate phase.

Before the CPU can shed its kernel-mode attire and resume user-mode execution, `systrap_rtn` performs two critical checks:

1. **Signal Interception (`issig()`)**: The very first order of business is to invoke `issig()`, our familiar signal gatekeeper from Section 1.3. This check is paramount: if any signals have been posted to the process while it was executing in kernel mode, `issig()` will identify the highest-priority deliverable signal. If such a signal is found, the kernel will not return directly to the user's interrupted instruction. Instead, it will divert control to the signal delivery mechanism (`psig()`), potentially invoking a user-defined signal handler or executing the signal's default action (e.g., terminating the process). This ensures that asynchronous events are processed promptly and that a process cannot indefinitely defer signal handling by remaining in kernel mode.

2. **Preemption Check (`runrun` flag)**: Next, `systrap_rtn` inspects the `runrun` flag. This humble-looking flag is the scheduler's silent decree, a one-bit memo indicating that a preemption event is pending—a higher-priority process is now runnable, or the current process's time slice has expired. If `runrun` is set, the kernel will not return to the current user process. Instead, it triggers a context switch (`preempt()`), handing control back to the scheduler to select the next deserving process. This ensures that the kernel maintains its responsiveness and fairness, even when a process has just completed a system call.

Only if both these checks are cleared—no deliverable signals and no pending preemption—does the kernel proceed to restore the user-mode CPU context that was so carefully preserved during system call entry. The saved registers, stack pointer, and program counter are meticulously reloaded, and finally, a special instruction (typically `IRET` on i386 for interrupt returns) is executed. This `IRET`

instruction atomically restores the CPU to user mode and returns control to the precise instruction in the user program that was interrupted by the `INT 0x80`.

The Echo of the Kernel: Return Values

The outcome of a system call needs to be communicated back to the requesting user process. In SVR4, the results are typically passed back via general-purpose registers. Successful system calls return a non-negative value (often 0, or a specific result like a file descriptor or byte count) in `EAX`.

In the event of an error, the system call typically returns a value of `-1` in `EAX`, and sets the CPU's carry flag. This carry flag is a subtle but critical indicator. The C library wrapper, which initiated the `INT 0x80` call, then checks this carry flag. If set, it interprets the value in `EAX` as an error code (a negative `errno` value), translates it into the appropriate positive `errno` value (e.g., `EFAULT`, `EPERM`, `ENOENT`), and stores it in the global `errno` variable, making it accessible to the user program.

This meticulous dance of entry, execution, and exit, punctuated by critical checks for signals and preemption, forms the very bedrock of the SVR4 kernel's interaction with user applications, safeguarding system integrity while providing essential services.

Process Groups and Sessions: The Theater, the Cast, and the Stage Manager

Picture a grand theater with a single stage. Actors are grouped into casts, each cast rehearsing a scene. The stage manager must know which cast is on stage and which is waiting in the wings. When the curtain rises, the stage belongs to one cast only; the others must stay silent or be ushered off.

SVR4's process groups and sessions are that theater. A process group is a cast. A session is the production. The controlling terminal is the stage, and job control is the stage manager's whistle.

The Cast List: Process Groups

Process groups are represented by `struct pid` entries that link members together. When a process joins a group, `pgjoin()` links it into the group list and updates group metadata (os/pgrp.c:79-101).

```
void
pgjoin(p, pgp)
    register proc_t *p;
    register struct pid *pgp;
{
    p->p_pglink = pgp->pid_pglink;
    pgp->pid_pglink = p;
    p->p_pgidp = pgp;

    if (pgp->pid_id <= SHRT_MAX)
        p->p_opgrp = (o_pid_t)pgp->pid_id;
    else
        p->p_opgrp = (o_pid_t)NOPID;

    if (p->p_pglink == NULL) {
        PID_HOLD(pgp);
        if (pglinked(p))
            pgp->pid_pgorphaned = 0;
        else
            pgp->pid_pgorphaned = 1;
    } else if (pgp->pid_pgorphaned && pglinked(p))
        pgp->pid_pgorphaned = 0;
}
```

The Casting Call (os/pgrp.c:79-101)

Each group keeps a linked list of members via `pid_pglink`. The `pid_pgorphaned` flag tracks whether the group has become orphaned, which affects job-control signals.

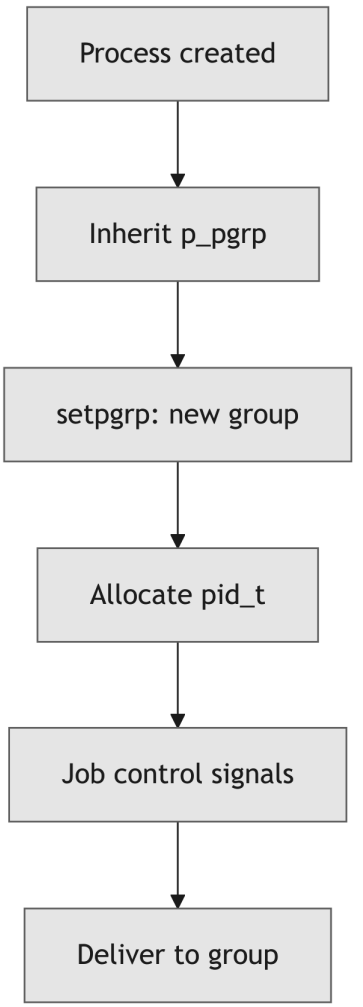
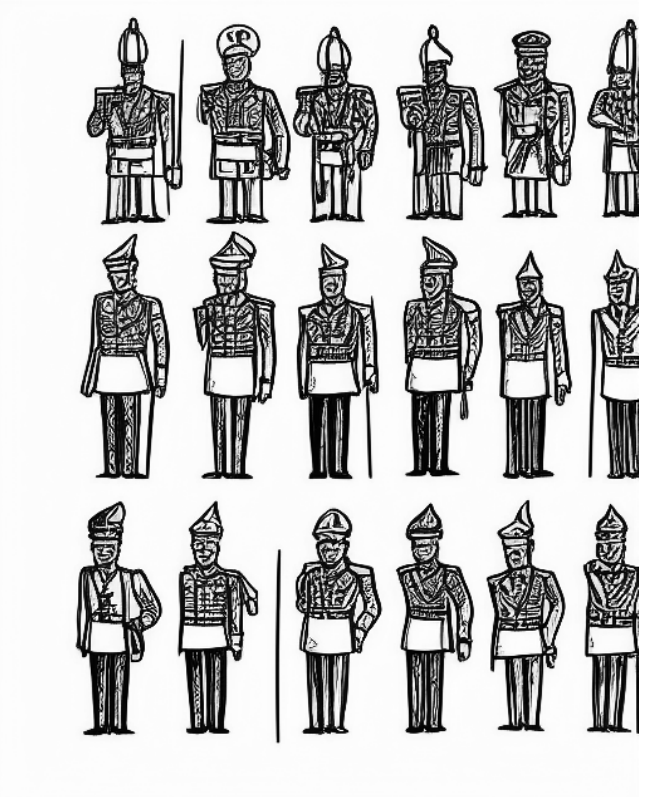


Figure 1.5.1: Processes Linked into a Group



Process Groups - Military Regiment

Signaling the Cast: `pgsignal()`

When the kernel needs to signal a whole group, it walks the group list and delivers the signal to each member (os/pgrp.c:65-72).

```
for (prp = pidp->pid_pglink; prp; prp = prp->p_pglink)
    psignal(prp, sig);
```

The Stage Manager's Whistle (os/pgrp.c:69-72)

This is how `kill(-pgid, SIGTERM)` fans out to every process in the group, and how job control can suspend or continue a whole cast at once.

Orphaned Groups and Job Control

When a parent exits, it may orphan a child's process group. `pgdetach()` checks whether any remaining parent outside the group can still influence it. If not, and if any member is stopped, the kernel sends `SIGHUP` and `SIGCONT` to the group (os/pgrp.c:144-170). The stage manager clears the scene so no stopped cast remains abandoned.

This is the mechanism behind the classic shell behavior: background jobs are terminated or resumed when their controlling shell exits.

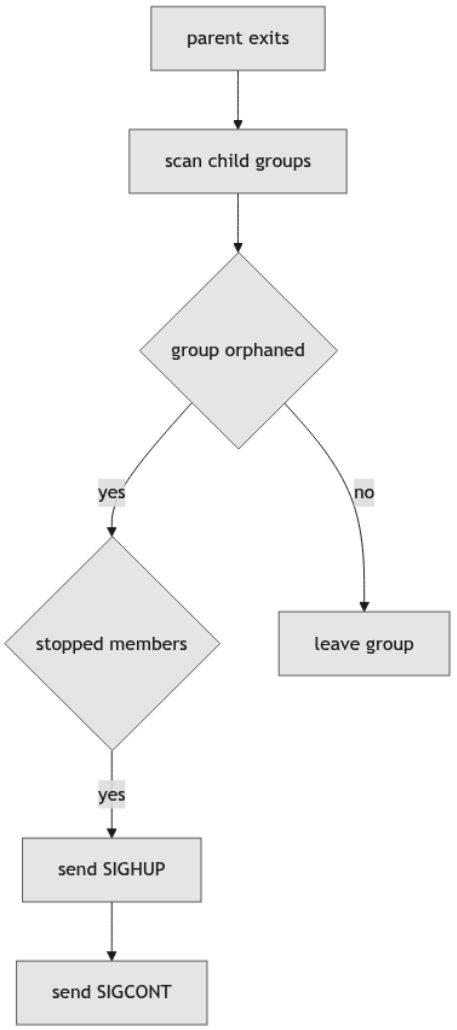


Figure 1.5.2: Orphan Detection and Group Signaling

The Production: Sessions

A session groups process groups and anchors the controlling terminal. The `sess_t` structure in `sys/session.h` records the session ID, controlling terminal, and reference count (sys/session.h:14-26).

```
typedef struct sess {
    short s_ref;          /* reference count */
    short s_mode;         /* /sess current permissions */
    uid_t s_uid;          /* /sess current user ID */
    gid_t s_gid;          /* /sess current group ID */
    ulong s_ctime;        /* /sess change time */
    dev_t s_dev;          /* tty's device number */
    struct vnode *s_vp;   /* tty's vnode */
    struct pid *s_sidp;   /* session ID info */
    struct cred *s_cred;  /* allocation credentials */
} sess_t;
```

The Production Ledger (sys/session.h:14-25)

Creating a new session via `setsid()` ultimately calls `sess_create()`, which detaches the process from its current group, allocates a new session, and makes the process its own process group leader (os/session.c:57-77).

```
pp = u.u_procp;

pgexit(pp);
SESS_RELE(pp->p_sessp);

sp = (sess_t *)kmem_zalloc(sizeof (sess_t), KM_SLEEP);
sp->s_sidp = pp->p_pidp;
sp->s_ref = 1;
sp->s_dev = NODEV;
pp->p_sessp = sp;
u.u_ttyp = NULL; /* compatibility */

pgjoin(pp, pp->p_pidp);

PID_HOLD(sp->s_sidp);
```

The New Production (os/session.c:57-79, excerpt)

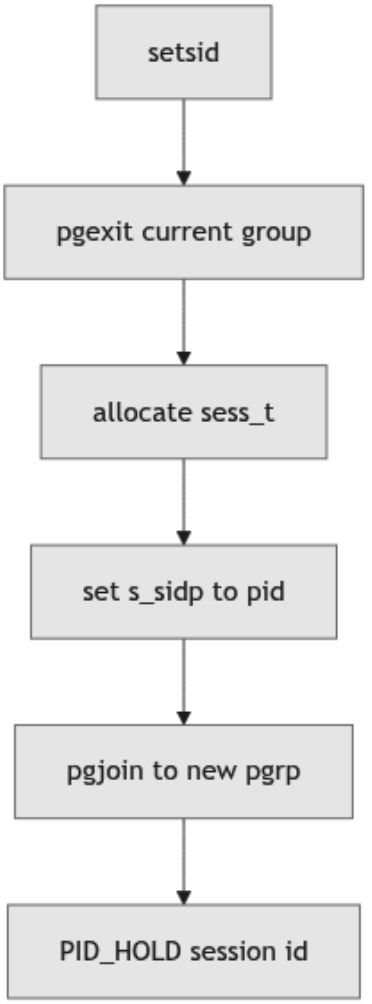


Figure 1.5.3: `setsid()` and Session Formation

The Stage: Controlling Terminals

The controlling terminal belongs to a session. Foreground process groups may read and write freely; background groups are disciplined by signals. When a session loses its terminal, `freectty()` sends `SIGHUP` and clears the association (os/session.c:80-103). This is the stage manager striking the set at the end of the show.

The job-control signals `SIGTTIN` and `SIGTTOU` enforce the rule that only the foreground cast uses the stage. Background actors may speak, but the manager will suspend them if they interrupt the play.

> **The Ghost of SVR4:** We organized casts and productions so that terminals could be governed without chaos. Modern shells still use the same model, though virtual terminals, containers, and session leaders have multiplied. The stage is now shared across many theaters, yet the rule remains: one foreground cast at a time.

The Curtain Falls

Process groups and sessions are the social order of the kernel. They tell the system who belongs together, who owns the stage, and who must be silenced when the curtain rises. The theater runs smoothly because the cast list is precise and the stage manager never forgets who is in charge.

PID Management: The Registry of Names

A city cannot function without a registry of names. Each citizen is assigned a number, recorded in a ledger, and removed only after their affairs are settled. If a number is reused too soon, confusion follows. The registry clerk must be careful and methodical.

SVR4's PID management is that registry. It allocates process IDs, maintains a hash for fast lookup, and keeps reference counts so a PID is not recycled while it is still referenced by process groups or sessions.

The Ledger Entry: `struct pid`

PIDs are represented by `struct pid` entries stored in a hash table. In `os/pid.c` the kernel keeps a `pidhash` array and helper macros (os/pid.c:51-55).

```
#define HASHSZ      64
#define HASHPID(pid) (pidhash[((pid)&(HASHSZ-1))])
```

The Registry Buckets (os/pid.c:51-53)

Each `pid` entry links to a process group list and tracks reference counts and the `/proc` slot (os/pid.c:41-49, 167-175). The details are scattered in `proc_t` and the pid structure, but the principle is clear: the pid entry is the registry's card for that identity.

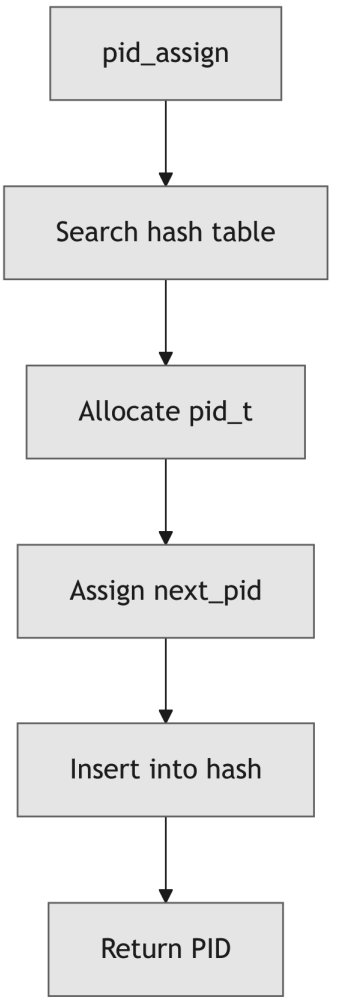
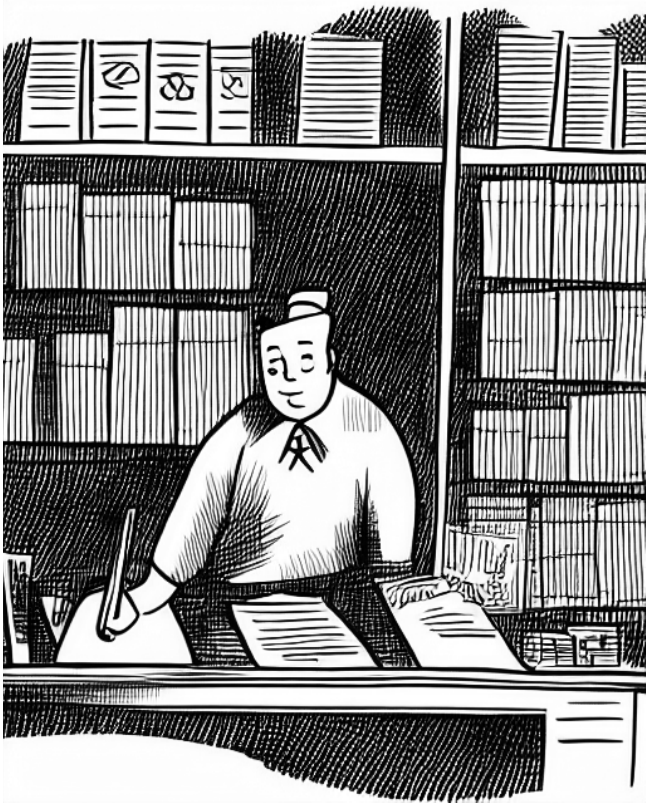


Figure 1.6.1: PID Allocation and Lookup



PID Management - Registrar's Office

Assigning a New Name: `pid_assign()`

`pid_assign()` handles PID allocation at fork time (os/pid.c:96-182). It enforces process limits, allocates a `proc_t` and `pid` structure, and then searches for the next free PID.

```
if (nproc >= v.v_proc - 1) {
    if (nproc == v.v_proc) {
        syserr.procovf++;
        return -1;
    }
    if (cond & NP_NOLAST)
        return -1;
}
```

The Capacity Check (os/pid.c:111-118)

PID selection increments `mpid`, wraps at `MAXPID`, and skips any PID already present in the hash (os/pid.c:152-155). Once a free PID is found, it is inserted into the hash and linked to the new `proc_t`.

```
do {
    if (++mpid == MAXPID)
        mpid = minpid;
} while (pid_lookup(mpid) != NULL);
```

The Next Free Name (os/pid.c:152-155)

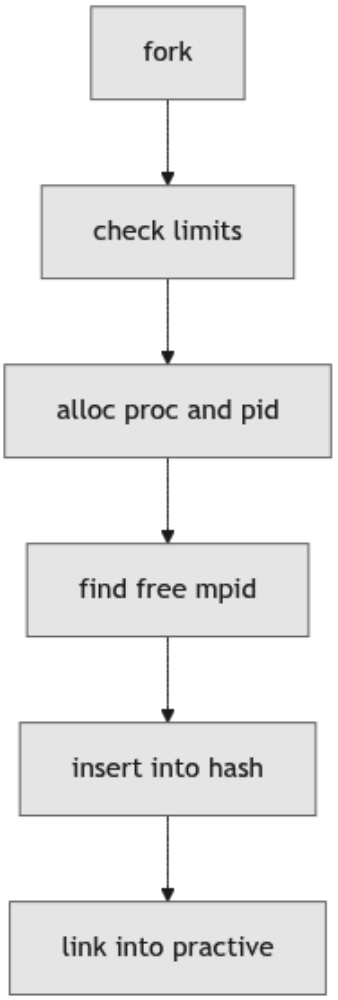


Figure 1.6.2: `pid_assign()` Allocation Path

Reference Counts and Exit

PIDs are shared by process groups and sessions. The registry uses reference counts to keep an ID alive while any structure still points to it. `PID_HOLD` increments the count; `PID_RELE` and `pid_rele()` remove the entry from the hash when the count reaches zero (os/pid.c:188-206).

When a process exits, `pid_exit()` removes it from the active list, releases its process group and session references, and decrements the PID entry (os/pid.c:212-241). Only after these steps does the PID become available for reuse.

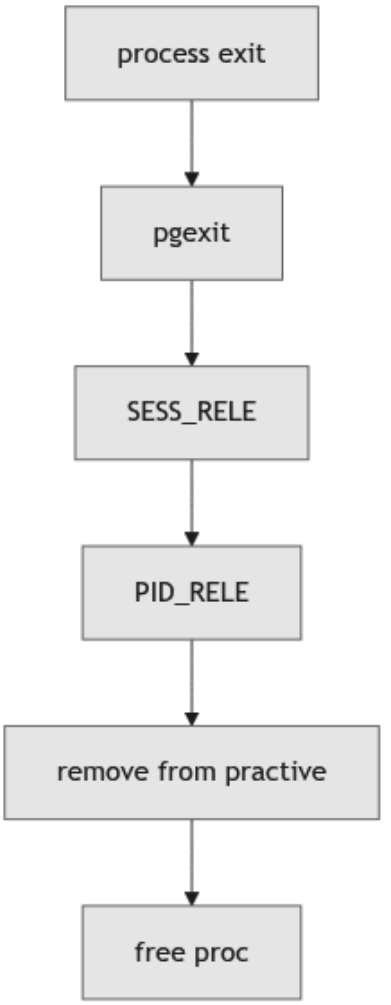


Figure 1.6.3: `pid_exit()` and Release

Lookup and Group Find

`prfind()` locates a process by PID via the hash and checks whether the slot is active (os/pid.c:248-257). `pgfind()` does the same for process groups, returning the group's linked list (os/pid.c:265-276). These lookups are fundamental to signaling, job control, and `/proc` traversal.

The low-level hash walk is handled by `pid_lookup()`, which scans a bucket for a matching `pid_id` (os/pid.c:61-72).

```
for (pidp = HASHPID(pid); pidp; pidp = pidp->pid_link) {
    if (pidp->pid_id == pid) {
        ASSERT(pidp->pid_ref > 0);
        break;
    }
}
```

The Registry Lookup (os/pid.c:65-71)

This tight loop is what makes signals and `/proc` queries fast. The registry is only as useful as its lookup speed.

Minimum PID and `/proc` Slots

The allocator maintains a moving floor for PID reuse. `pid_setmin()` advances `minpid` to `mpid + 1` (os/pid.c:75-79), ensuring that recently used PIDs are not immediately recycled. This reduces the chance of PID reuse races in parent/child bookkeeping.

The code also reserves a `/proc` directory slot for each process. A `procent` entry is pulled from the free list, linked to the new `proc_t`, and later returned in `pid_exit()` (os/pid.c:158-166, 223-227). The registry is thus tied to the `/proc` filesystem: every PID has a directory entry while it is active.

These two mechanisms keep the ledger consistent: names are not reused too quickly, and visibility in `/proc` stays in lockstep with process lifetime.

> **The Ghost of SVR4:** We kept a modest hash table and a monotonic counter for IDs. Modern kernels use PID namespaces, per-container ranges, and larger ID spaces, but the registry rule is the same: never reuse a name while someone still holds a claim to it.

The Ledger Closes

PID management is a registry of identities. It assigns numbers, tracks them in a hash, and retires them only when all references are gone. The clerk’s ledger ensures that every process has a unique name and that no name is reused too soon.

Credentials and Access Control: The Seals, the Ledger, and the Wax

Imagine a courthouse where every petition bears a wax seal. The seal is not the person, but it carries the person's authority. Clerks do not know the petitioner; they only read the seal. If a seal changes, the clerk must ensure no other parchment was stamped with the old impression.

SVR4's credentials are those seals. The `cred_t` structure encodes user IDs and groups, and the kernel shares and duplicates these seals with strict reference counts to avoid accidental authority leaks.

The Seal: `struct cred`

The credential structure is defined in `sys/cred.h` (sys/cred.h:20-30). It holds effective, real, and saved IDs, as well as supplementary groups.

```
typedef struct cred {
    ushort cr_ref;      /* reference count */
    ushort cr_ngroups;  /* number of groups in cr_groups */
    uid_t  cr_uid;      /* effective user id */
    gid_t  cr_gid;      /* effective group id */
    uid_t  cr_ruid;     /* real user id */
    gid_t  cr_rgid;     /* real group id */
    uid_t  cr_suid;     /* saved user id */
    gid_t  cr_sgid;     /* saved group id */
    gid_t  cr_groups[1];/* supplementary group list */
} cred_t;
```

The Seal Structure (sys/cred.h:20-29)

Two distinctions matter:

- **Effective IDs** (`cr_uid` , `cr_gid`) decide access checks.

- **Real/saved IDs** (`cr_ruid` , `cr_suid`) preserve the original identity and allow privilege drops and restorations.

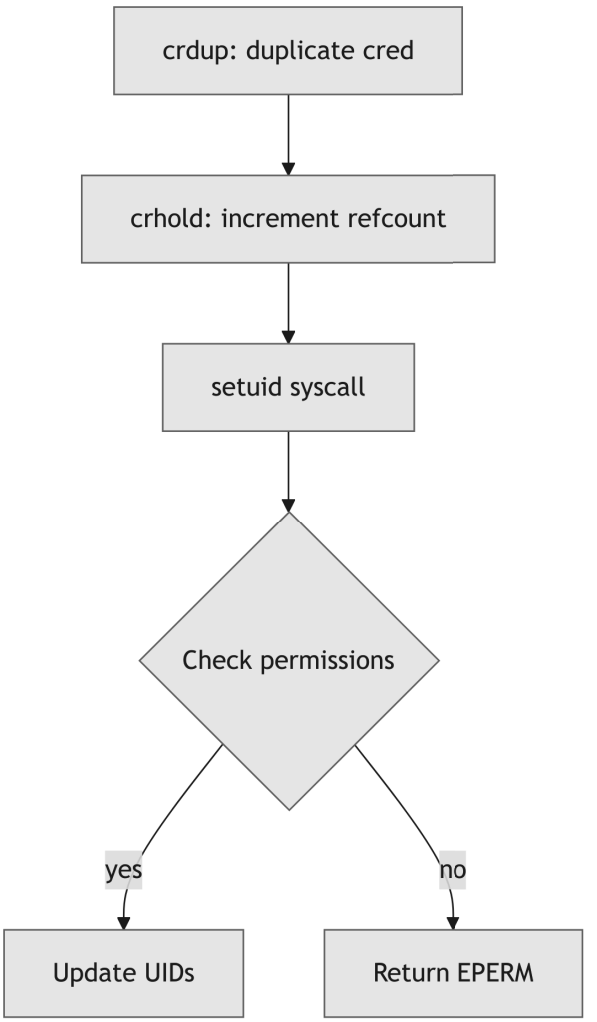


Figure 1.7.1: Credentials Shared Across Processes

Credentials - Royal Seals

Reference Counts and Copy-on-Write

Credentials are shared to reduce memory churn. When a process needs to modify its credentials, it must first obtain a private copy. The core routines live in `os/cred.c`.

```
struct cred *
crget()
{
    if (crfreelist) {
        cr = &crfreelist->cl_cred;
        crfreelist = ((struct credlist *)cr)->cl_next;
    } else
        cr = (struct cred *)kmem_alloc(crsize, KM_SLEEP);
    struct_zero((caddr_t)cr, sizeof(*cr));
    crhold(cr);
    return cr;
}
```

The Blank Seal (os/cred.c:92-109, abridged)

`crdup()` and `crcopy()` both duplicate the seal, but they differ in how they treat the original:

- **`crdup()`** creates a new copy and leaves the old intact (os/cred.c:153-162).
- **`crcopy()`** duplicates and then frees the old one (os/cred.c:137-147).

Reference counts are decremented in `crfree()`; when the count reaches zero the seal is returned to the freelist (os/cred.c:118-129). The courthouse reuses its wax only when no parchments remain.

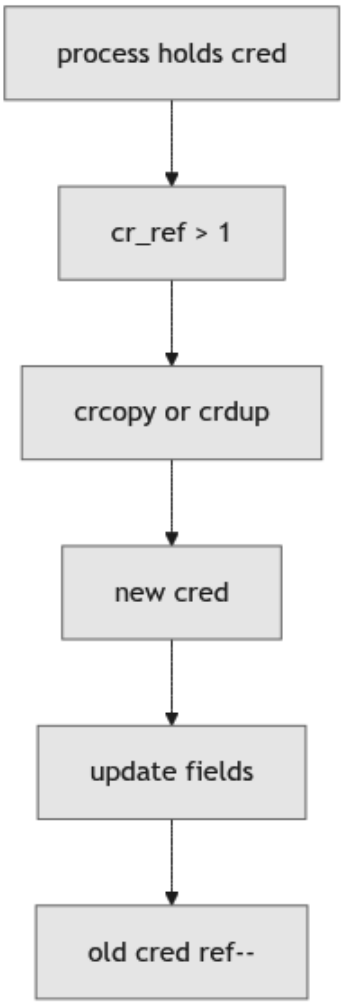


Figure 1.7.2: Copy-on-Write and Reference Counting

The Setuid Ritual

`setuid()` is the classic credential-changing system call. SVR4 enforces the rules in `os/scalls.c` (os/scalls.c:221-264): a non-root process may only switch its effective UID to its real or saved UID, while a superuser may set all three.

```
if (u.u_cred->cr_uid
  && (uid == u.u_cred->cr_ruid || uid == u.u_cred->cr_suid)) {
    u.u_cred = crcopy(u.u_cred);
    u.u_cred->cr_uid = uid;
} else if (suser(u.u_cred)) {
    u.u_cred = crcopy(u.u_cred);
    u.u_cred->cr_uid = uid;
    u.u_cred->cr_ruid = uid;
    u.u_cred->cr_suid = uid;
} else
    error = EPERM;
```

The Setuid Decision (os/scalls.c:244-264, abridged)

The saved UID is the lockbox: a setuid program can drop privileges to the real UID and later regain them by restoring the saved UID, all within the rules of the seal.

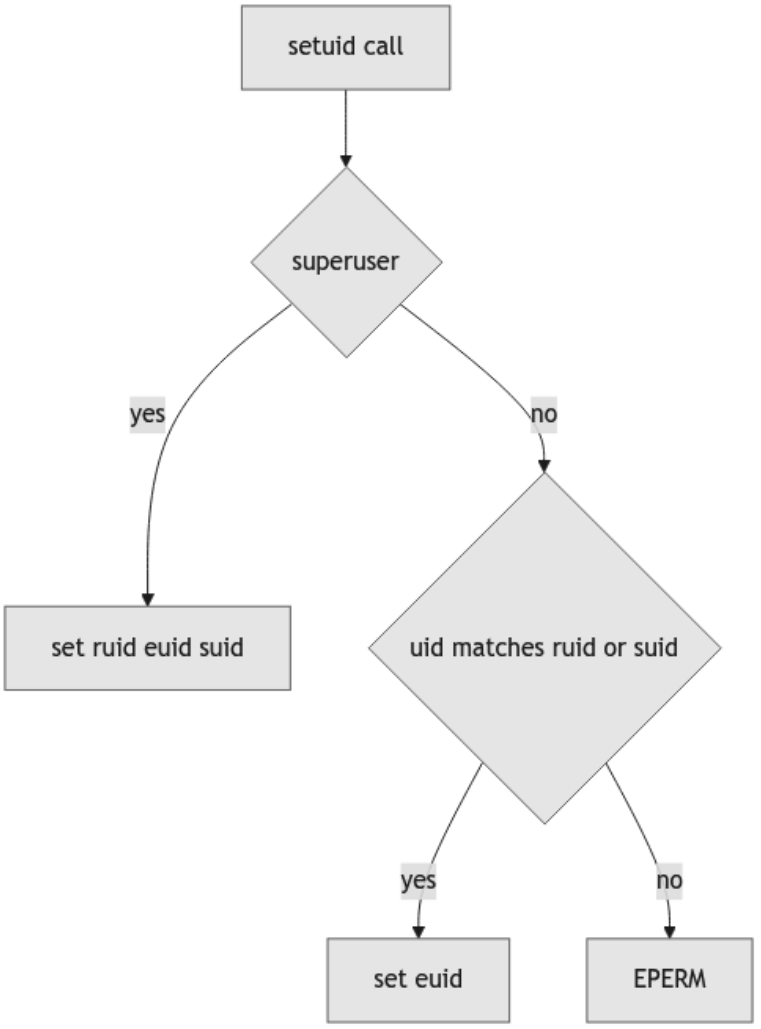


Figure 1.7.3: Switching Identities with `setuid()`

Superuser Recognition

The `suser()` helper is a simple test with an important side effect: if the effective UID is zero, the kernel sets an accounting flag and returns success (os/cred.c:181-188). This is the courthouse clerk noting that a royal seal was presented.

> **The Ghost of SVR4:** Our seals were simple: IDs and groups, reference counted and shared. Modern systems have capabilities, namespaces, and per-thread credentials, but the same rule remains. Authority must be explicit, copied when it changes, and never leaked across unrelated processes.

The Seal Is Set

Credentials are the kernel’s identity ledger. They are shared, copied with care, and enforced on every privileged action. The wax is not the person, but the courthouse will only honor the seal.

Messages: The Town Courier and the Ledger of Pigeons

Imagine a bustling town with a central courier office. Citizens drop sealed notes into labeled pigeonholes, each hole marked by a number and a type. Couriers arrive to collect only the type they expect. The clerk does not read the letters; she keeps a ledger, counts the paper, and ensures the pigeonholes never overflow.

SVR4's System V message queues are that courier office. Messages are typed, queued, and delivered asynchronously. The kernel keeps a ledger for each queue, a free list of message headers, and a shared message pool that holds the actual text.

The Queue Ledger: `struct msqid_ds`

The message queue descriptor in `sys/msg.h` tracks permissions, queue pointers, and accounting (sys/msg.h:56-71).

```
struct msqid_ds {
    struct ipc_perm msg_perm;   /* operation permission struct */
    struct msg  *msg_first;     /* ptr to first message on q */
    struct msg  *msg_last;      /* ptr to last message on q */
    ulong               msg_cbytes;     /* current # bytes on q */
    ulong               msg_qnum;       /* # of messages on q */
    ulong               msg_qbytes;     /* max # of bytes on q */
    pid_t               msg_lspid;      /* pid of last msgsnd */
    pid_t               msg_lrpid;      /* pid of last msgrcv */
    time_t              msg_stime;      /* last msgsnd time */
    long                msg_pad1;       /* reserved for time_t expansion
*/
    time_t              msg_rtime;      /* last msgrcv time */
    long                msg_pad2;       /* time_t expansion */
    time_t              msg_ctime;      /* last change time */
    long                msg_pad3;       /* time expansion */
    long                msg_pad4[4];    /* reserve area */
};
```

The Courier Ledger (sys/msg.h:56-72)

Two fields govern flow:

- **`msg_cbytes`** counts the current bytes in the queue.
- **`msg_qbytes`** sets the maximum, enforcing backpressure.

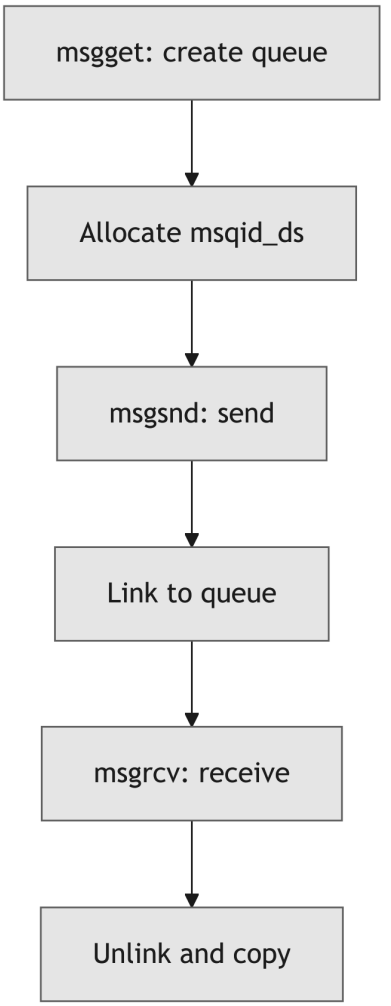
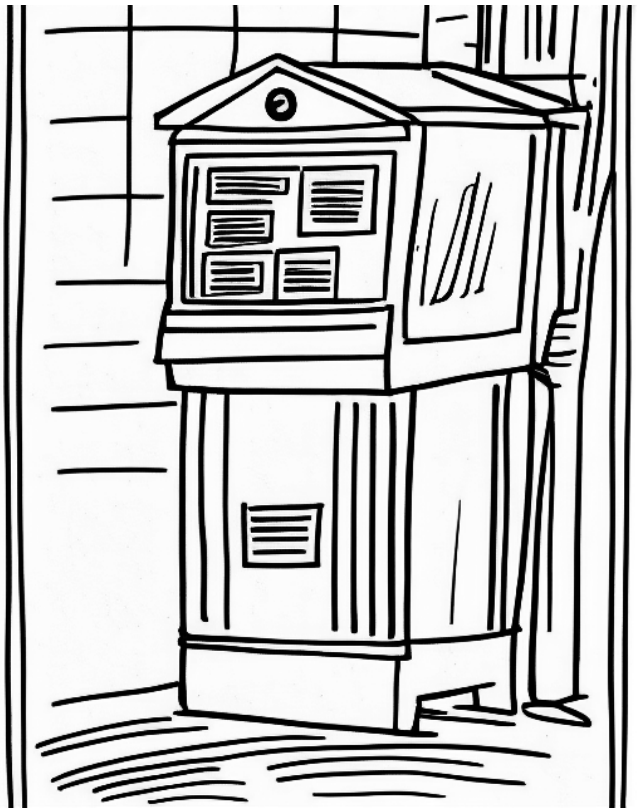


Figure 1.8.1: Queue Ledger and Message Links



Messages IPC - Post Box

The Message Header: `struct msg`

Each enqueued message has a header that links it into the queue and points into the shared message pool (sys/msg.h:136-141).

```
struct msg {
    struct msg *msg_next;  /* ptr to next message on q */
    long    msg_type;      /* message type */
    ushort  msg_ts;        /* message text size */
    short   msg_spot;      /* message text map address */
};
```

The Pigeon Tag (sys/msg.h:136-141)

The `msg_spot` field is the index into the message pool, allocated from a resource map. This is why the kernel keeps a free list of headers (`msgfp`) and a separate message text map (`msgmap`).

Sending a Message: `msgsnd()`

The send path in `os/msg.c` checks permissions, validates size, waits for space, and then copies the message into the pool (os/msg.c:498-648).

Key steps:

1. **Permission check** via `ipcaccess`.
2. **Validate size** against `msginfo.msgmax`.
3. **Block or fail** if `msg_cbytes + msgsz > msg_qbytes`.
4. **Allocate header and text spot** from `msgfp` and `msgmap`.
5. **Copy in** type and text, update queue counters.

```
if (cnt + qp->msg_cbytes > (uint)qp->msg_qbytes) {
    if (uap->msgflg & IPC_NOWAIT) {
        error = EAGAIN;
        goto msgsnd_out;
    }
    qp->msg_perm.mode |= MSG_WWAIT;
    if (sleep((caddr_t)qp, PMSG|PCATCH))
        return EINTR;
    goto getres;
}
```

The Queue Full Decision (os/msg.c:544-567)

If receivers are waiting, `msgsnd()` clears `MSG_RWAIT` and wakes them once the new message lands (os/msg.c:641-645). The clerk rings the bell when a pigeon arrives.

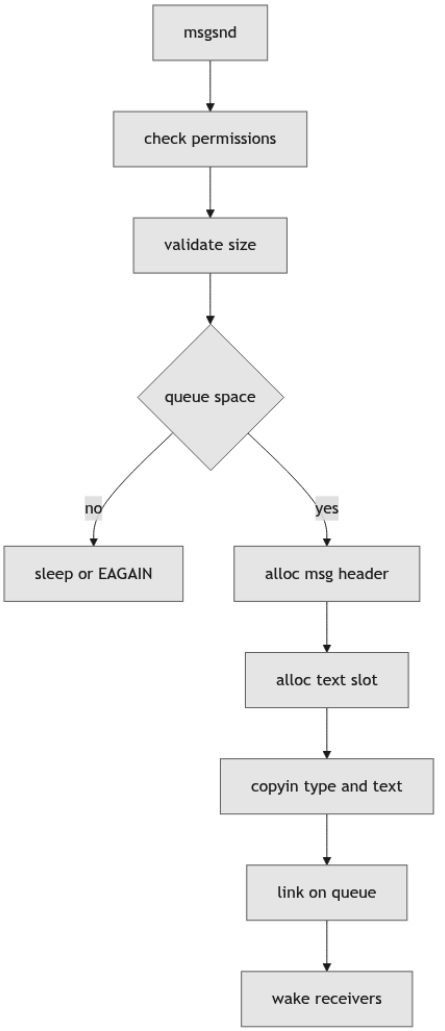


Figure 1.8.2: `msgsnd()` Allocation and Enqueue

Receiving a Message: `msgrcv()`

The receive path walks the queue, matching by type. It handles three cases (os/msg.c:429-447):

- **`msgtyp == 0`** : take the first message.
- **`msgtyp > 0`** : take the first message with that exact type.
- **`msgtyp < 0`** : take the message with the lowest type that is <= `-msgtyp` .

```
pmp = NULL;
mp = qp->msg_first;
if (uap->msgtyp == 0)
    smp = mp;
else
    for (; mp; pmp = mp, mp = mp->msg_next) {
        if (uap->msgtyp > 0) {
            if (uap->msgtyp != mp->msg_type)
                continue;
            smp = mp;
            spmp = pmp;
            break;
        }
        if (mp->msg_type <= -uap->msgtyp) {
            if (smp && smp->msg_type <= mp->msg_type)
                continue;
            smp = mp;
            spmp = pmp;
        }
}
if (smp) {
    if ((unsigned)uap->msgsz < smp->msg_ts) {
        if (!(uap->msgflg & MSG_NOERROR)) {
            error = E2BIG;
            goto msgrcv_out;
        } else
            sz = uap->msgsz;
    } else
        sz = smp->msg_ts;
    if (copyout((caddr_t)&smp->msg_type, (caddr_t)uap->msgp,
      sizeof(smp->msg_type))) {
        error = EFAULT;
        goto msgrcv_out;
    }
    if (sz
      && copyout((caddr_t)(msg + msginfo.msgssz * smp->msg_spot),
        (caddr_t)uap->msgp + sizeof(smp->msg_type), sz)) {
        error = EFAULT;
        goto msgrcv_out;
    }
    rvp->r_val1 = sz;
    qp->msg_cbytes -= smp->msg_ts;
    qp->msg_lrpid = u.u_procp->p_pid;
    qp->msg_rtime = hrestime.tv_sec;
    wflag = NOPRMPT;
    msgfree(qp, spmp, smp, NOPRMPT);
    goto msgrcv_out;
```

```
}
if (uap->msgflg & IPC_NOWAIT) {
    error = ENOMSG;
    goto msgrcv_out;
}
qp->msg_perm.mode |= MSG_RWAIT;
*lockp = 0;
wakeprocs(lockp, wflag);
if (sleep((caddr_t)&qp->msg_qnum, PMSG|PCATCH))
    return EINTR;
goto findmsg;
```

The Type Selection Walk (os/msg.c:419-476, excerpt)

Once selected, `msgrcv()` copies out the type and text, updates counters, and frees the header and text slot. If no matching message exists and `IPC_NOWAIT` is not set, the receiver sleeps on `msg_qnum` until a sender wakes it (os/msg.c:476-485).

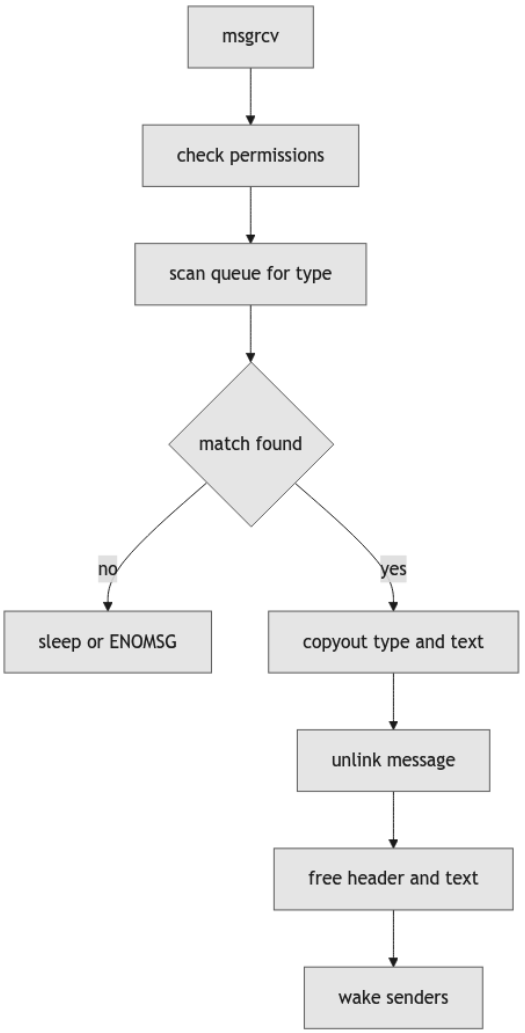


Figure 1.8.3: `msgrcv()` Match and Dequeue

Locks and the Parallel Latch

Because the message queue descriptor is part of the user-visible ABI, the kernel cannot add lock fields directly. Instead it maintains a parallel lock array and uses the `MSGLOCK()` macro to locate the lock slot (sys/msg.h:166-181). This is the clerk’s brass latch, separate from the ledger itself.

> **The Ghost of SVR4:** Our queues were simple, typed pigeonholes with blocking senders and receivers. Modern systems still keep System V IPC, but they also offer message brokers, epoll-driven pipes, and shared-memory rings. Yet the same ritual persists: check capacity, enqueue, wake sleepers, and count every byte as if paper were scarce.

The Ledger Closes

System V message queues are not glamorous, but they are reliable. The queue descriptor records who last sent and received, the message headers track types and sizes, and the kernel enforces limits so the pigeonholes never overflow. The town courier keeps its promises because the ledger is strict.

Semaphores: The Turnstiles and the Ledger of Debts

Imagine a railway station with a row of turnstiles. Each turnstile has a counter and a small notebook. When a passenger enters, the counter decreases. When a train arrives and empties, the counter increases. The station master can adjust several turnstiles at once, and the entire action is either accepted or rejected as a single transaction.

SVR4's System V semaphores are those turnstiles. They are integer counters with an atomic operation list, a record of waiting travelers, and an optional debt ledger that is settled when a process leaves the station.

The Semaphore Set Ledger: `struct semid_ds`

A semaphore set is described by `struct semid_ds` in `sys/sem.h` (sys/sem.h:63-71).

```
struct semid_ds {
    struct ipc_perm sem_perm;   /* operation permission struct */
    struct sem  *sem_base;      /* ptr to first semaphore in set */
    ushort              sem_nsems;      /* # of semaphores in set */
    time_t              sem_otime;      /* last semop time */
    long                sem_pad1;       /* reserved for time_t expansion
*/
    time_t              sem_ctime;      /* last change time */
    long                sem_pad2;       /* time_t expansion */
    long                sem_pad3[4];    /* reserve area */
};
```

The Turnstile Ledger (sys/sem.h:63-75)

Each set holds a contiguous array of `struct sem` entries, allowing multiple operations to be applied atomically in a single `semop()` call.

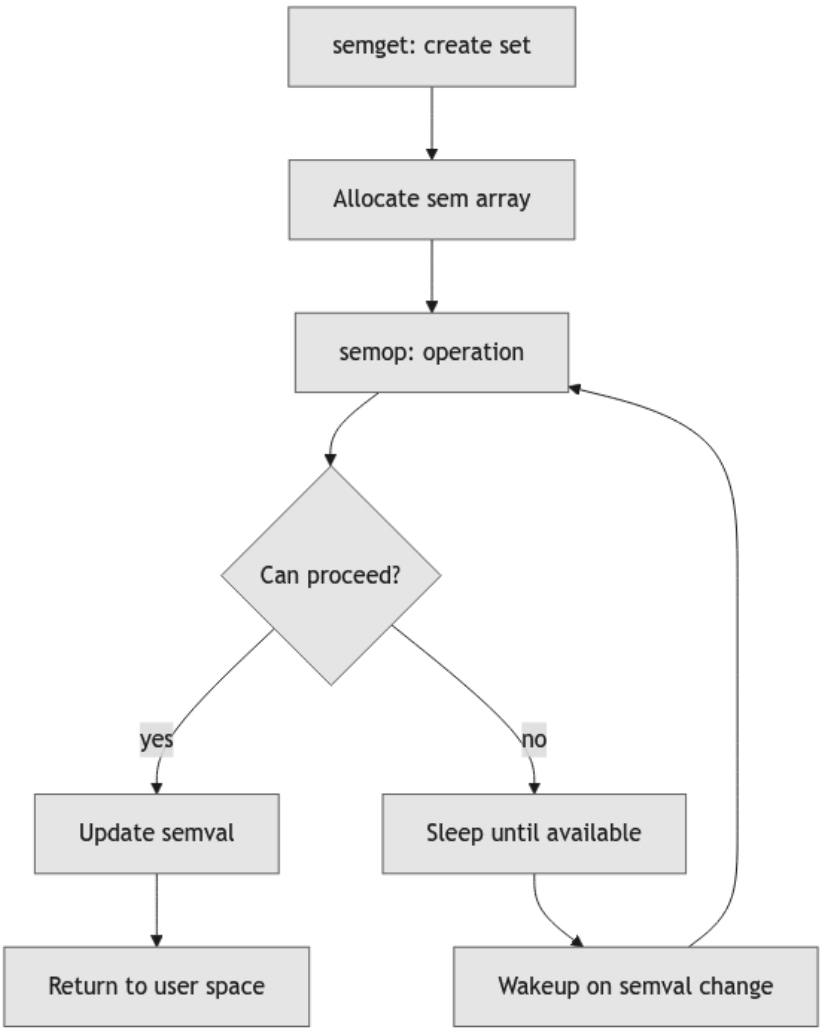
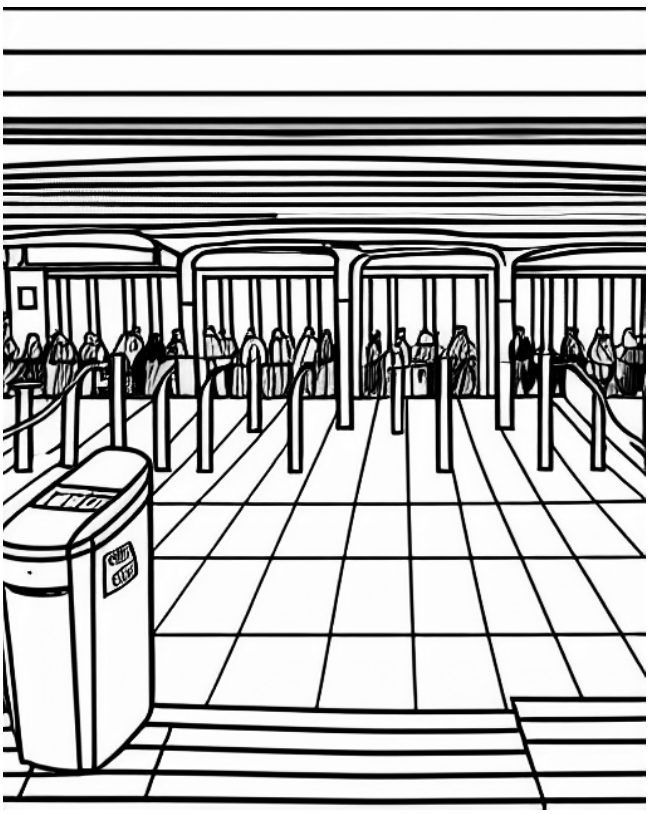


Figure 1.9.1: Semaphore Set and Per-Semaphore State



Semaphores - Railway Station Turnstiles

The Turnstile: `struct sem`

Each semaphore maintains its value and the number of waiters for two conditions (sys/sem.h:85-90).

```
struct sem {
    ushort semval;  /* semaphore value */
    pid_t  sempid;  /* pid of last operation */
    ushort semncnt; /* # awaiting semval > cval */
    ushort semzcnt; /* # awaiting semval = 0 */
};
```

The Turnstile Record (sys/sem.h:85-90)

- **`semncnt`** counts processes waiting to decrement (they need more value).
- **`semzcnt`** counts processes waiting for the semaphore to reach zero.

These counters are the clerk's waiting list; they drive wakeups when state changes.

The Atomic Script: `semop()` and `struct sembuf`

A `semop()` call submits an array of `struct sembuf` operations (sys/sem.h:180-184). All operations are validated and executed atomically.

```
struct sembuf {
    ushort sem_num; /* semaphore # */
    short  sem_op;  /* semaphore operation */
    short  sem_flg; /* operation flags */
};
```

The Operation Slip (sys/sem.h:180-183)

`semop()` in `os/sem.c` walks the operation list and applies each step (os/sem.c:665-745). The logic is strict:

- **`sem_op > 0`** increments the counter, waking waiters if necessary.
- **`sem_op < 0`** decrements if the counter is high enough; otherwise sleep or return `EAGAIN`.
- **`sem_op == 0`** waits until the counter reaches zero.

```
if (op->sem_op < 0) {
    if (semp->semval >= (unsigned)(-op->sem_op)) {
        if (op->sem_flg & SEM_UNDO
          && (error = semaoe(op->sem_op, uap->semid,
            op->sem_num))) {
            if (i)
                semundo(semtmp.semops, i, uap->semid, sp);
            return error;
        }
        semp->semval += op->sem_op;
        if (semp->semzcnt && !semp->semval) {
            semp->semzcnt = 0;
            wakeprocs((caddr_t)&semp->semzcnt, PRMPT);
        }
        continue;
    }
    if (i)
        semundo(semtmp.semops, i, uap->semid, sp);
    if (op->sem_flg & IPC_NOWAIT)
        return EAGAIN;
    semp->semncnt++;
    if (sleep((caddr_t)&semp->semncnt, PCATCH|PSEMN)) {
        if ((semp->semncnt)-- <= 1) {
            semp->semncnt = 0;
            wakeprocs((caddr_t)&semp->semncnt, PRMPT);
        }
        return EINTR;
    }
    goto check;
}
```

The Negative Gate (os/sem.c:675-718, excerpt)

If a later operation fails after earlier ones succeeded, `semundo()` rolls back the completed steps to preserve atomicity (os/sem.c:672-707). The station master never leaves a partial transaction in the ledger.

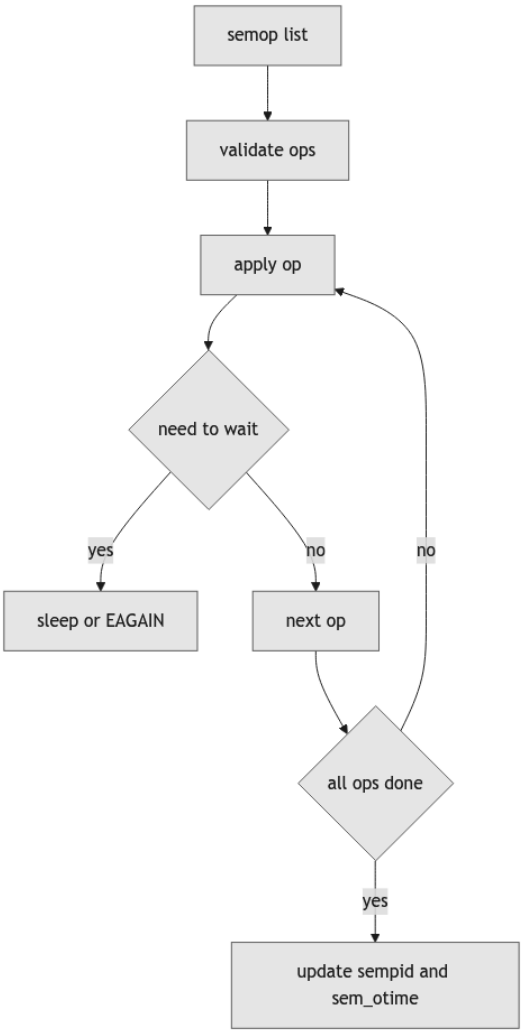


Figure 1.9.2: `semop()` Atomic Evaluation

The Debt Ledger: `SEM_UNDO`

When a process sets `SEM_UNDO`, the kernel records an adjustment in a per-process undo structure (sys/sem.h:150-158). This lets the kernel reverse the operation if the process exits unexpectedly.

```
struct sem_undo {
    struct sem_undo *un_np;
    short un_cnt;
    struct undo {
        short  un_aoe; /* adjust on exit */
        ushort un_num; /* semaphore # */
        int    un_id;  /* semid */
    } un_ent[1];
};
```

The Debt Ledger (sys/sem.h:150-158)

This mechanism prevents deadlocks caused by process crashes. The station records debts, then settles them at exit so the turnstiles are never left permanently locked.

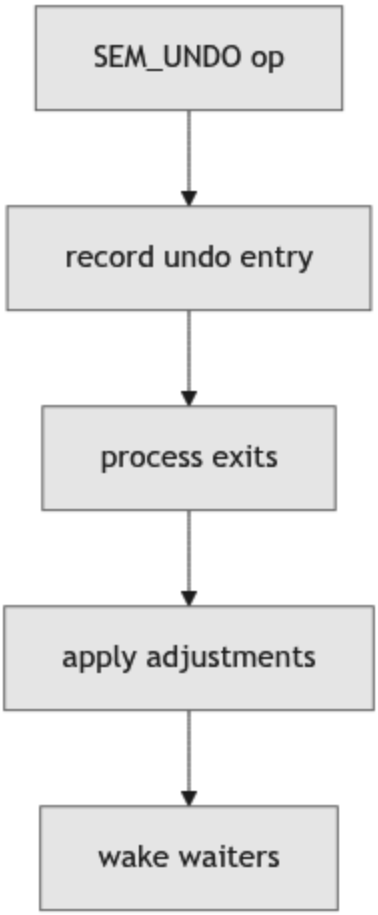


Figure 1.9.3: Undo Tracking on Process Exit

The Ghost of SVR4: We offered atomic arrays of semaphore ops and a built-in debt ledger for crashes. Modern systems still carry System V semaphores, but many applications now prefer futexes, robust mutexes, and lock-free queues. The turnstiles are faster now, yet the old rule remains: the ledger must balance, or the station grinds to a halt.

The Ledger Closes

Semaphores in SVR4 are a careful accounting system. Each set has a ledger, each semaphore has waiting lists, and each operation is either fully applied or unwound. The station master keeps every turnstile honest, and the travelers pass through in orderly fashion.

Shared Memory: The Common Courtyard and the Keyring

Imagine a walled courtyard shared by several households. Each family has a key, and each key opens the same gate. The courtyard is not copied; it is the same stones under every footstep. The city clerk records how many keys are issued and takes the courtyard back only when the last key is returned.

SVR4's shared memory is that courtyard. It maps a single anonymous memory region into multiple address spaces and relies on attach counts and permissions to keep the ledger balanced.

The Courtyard Ledger: `struct shmid_ds`

Shared memory segments are described by `struct shmid_ds` in `sys/shm.h` (sys/shm.h:83-99).

```
struct shmid_ds {
    struct ipc_perm shm_perm;   /* operation permission struct */
    int         shm_segsz;      /* size of segment in bytes */
    struct anon_map     *shm_amp;       /* segment anon_map pointer */
    ushort              shm_lkcnt;      /* number of times it is being
locked */
    pid_t               shm_lpid;       /* pid of last shmop */
    pid_t               shm_cpid;       /* pid of creator */
    ulong               shm_nattch;     /* used only for shminfo */
    ulong               shm_cnattch;    /* used only for shminfo */
    time_t              shm_atime;      /* last shmat time */
    long                shm_pad1;       /* reserved for time_t expansion
*/
    time_t              shm_dtime;      /* last shmdt time */
    long                shm_pad2;       /* reserved for time_t expansion
*/
    time_t              shm_ctime;      /* last change time */
    long                shm_pad3;       /* reserved for time_t expansion
*/
    long                shm_pad4[4];    /* reserve area */
};
```

The Courtyard Ledger (sys/shm.h:83-111)

The crucial field is `shm_amp` , an `anon_map` that represents the shared anonymous pages. Every process maps this same `anon_map` , so every write becomes visible to all key holders.

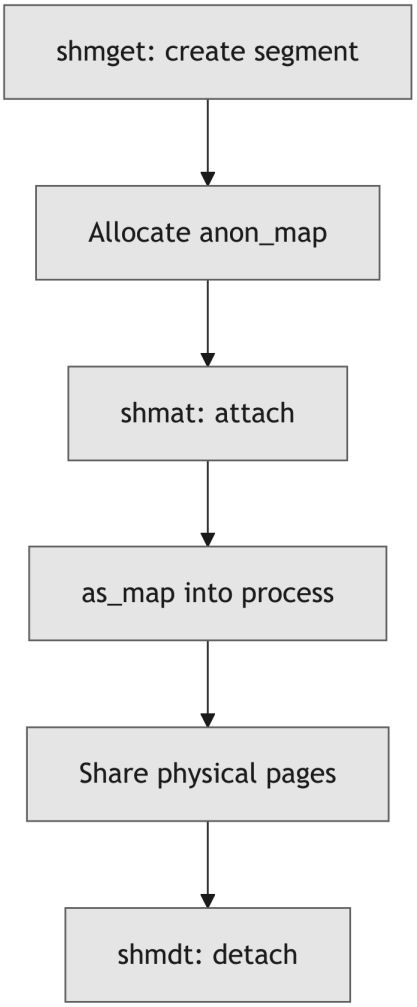


Figure 1.10.1: One Courtyard, Many Address Spaces

Shared Memory - Shared Courtyard

Attaching the Courtyard: `shmat()`

The `shmat()` system call validates permissions and maps the `anon_map` into the process address space. In `os/shm.c`, the kernel picks or validates an address, then calls `as_map()` with `segvn_create` (os/shm.c:168-246).

```
if (addr == 0) {
    map_addr(&addr, size, (off_t)0, 1);
    if (addr == NULL) {
        error = ENOMEM;
        goto errret;
    }
} else {
    if (uap->flag & SHM_RND)
        addr = (addr_t)((ulong)addr & ~(SHMLBA - 1));
    base = addr;
    len = size;
    if (((uint)base & PAGEOFFSET) ||
        (valid_usr_range(base,len) == 0) ||
        as_gap(pp->p_as, len, &base, &len, AH_LO, (addr_t)NULL) != 0) {
        error = EINVAL;
        goto errret;
    }
}

crargs = *(struct segvn_crargs *)zfod_argsp;
crargs.offset = 0;
crargs.type = MAP_SHARED;
crargs.amp = sp->shm_amp;
crargs.maxprot = crargs.prot;
crargs.prot = (uap->flag & SHM_RDONLY) ?
    (PROT_ALL & ~PROT_WRITE) : PROT_ALL;

error = as_map(pp->p_as, addr, size, segvn_create, (caddr_t)&crargs);
```

The Attach Ritual (os/shm.c:194-236, excerpt)

After mapping, the kernel records the region for later detach (`sa_add`) and increments the `anon_map` reference count (os/shm.c:238-241). This reference count is the keyring tally.

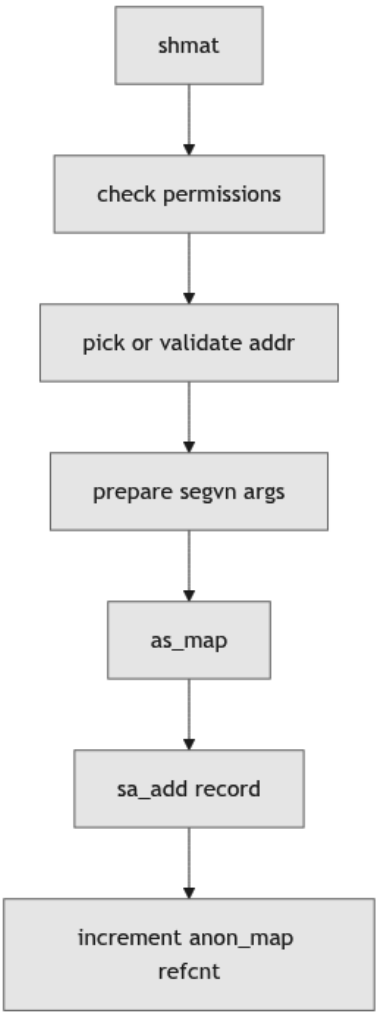


Figure 1.10.2: `shmat()` Mapping and Reference Counting

Choosing an Address

If the caller supplies no address, `shmat()` asks the system to pick a gap via `map_addr()` (os/shm.c:194-199). If an address is supplied, the kernel optionally rounds it down when `SHM_RND` is set and verifies alignment and range (os/shm.c:206-219). These rules ensure that shared mappings land on valid, page-aligned boundaries and do not collide with existing segments.

The choice is practical: the clerk will choose a safe courtyard unless the requester insists on a specific location, in which case the clerk verifies that the gate can be opened there.

Detaching the Courtyard: `shmdt()`

Detaching removes the mapping from the process address space but does not free the shared memory unless the segment has been marked for removal and the last attachment has gone away. The refcount on `shm_amp` is the decisive factor: the courtyard remains while any key is still held.

The detach path calls `kshmdt()` (os/shm.c:519-585). It locates the segment by address, unmaps it, and updates reference counts. If the `anon_map` refcount drops to one and the segment is marked for removal, `shm_rm_amp()` reclaims the pages (os/shm.c:301-306).

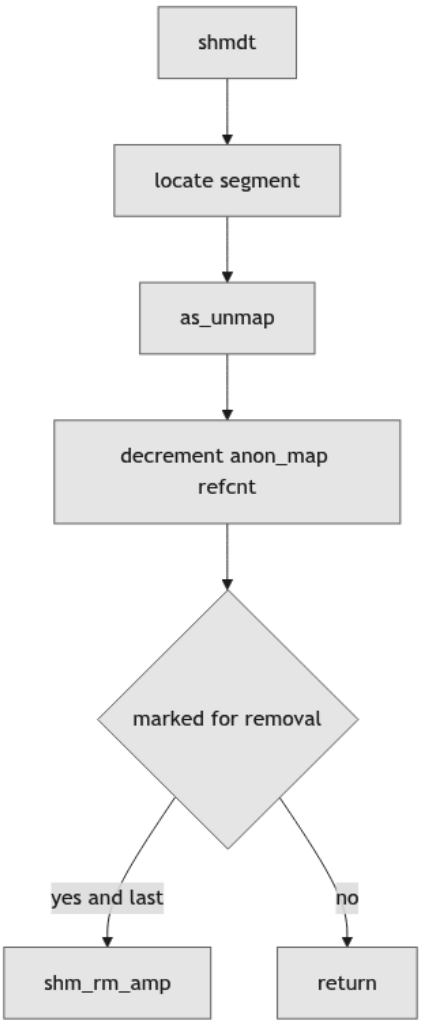


Figure 1.10.3: `shmdt()` and Final Reclamation

Removal and the Last Key

Shared memory segments are not destroyed at the first request. `shmctl(IPC_RMID)` marks a segment for removal, but the pages persist until the last attachment detaches. This design protects live processes from sudden eviction: the clerk notes the removal order and waits for the last key to return.

The `anon_map` refcount is the decisive measure. Each `shmat()` increments it, and each `shmdt()` decrements it. When the count shows that only the kernel's own reference remains, `shm_rm_amp()` can reclaim the pages (os/shm.c:301-306). The courtyard disappears only after the ledger is balanced.

Coordination Is Separate

Shared memory makes data cheap but leaves coordination to others. The kernel does not serialize access; it merely provides a shared courtyard. Processes must use semaphores or other synchronization to avoid trampling each other's state. This separation allows each application to choose the right balance between speed and safety.

Locks and Limits: `SHM_LOCK` and `shminfo`

The shared memory subsystem also enforces system-wide limits through `struct shminfo` (sys/shm.h:155-160). These values cap the size and number of shared segments and the number of attachments per process.

```
struct shminfo {
    int shmmax; /* max shared memory segment size */
    int shmmin; /* min shared memory segment size */
    int shmmni; /* # of shared memory identifiers */
    int shmseg; /* max attached shared memory segments per process */
};
```

The City Ordinances (sys/shm.h:155-160)

The `shmctl()` interface exposes lock control operations such as `SHM_LOCK` and `SHM_UNLOCK` (sys/shm.h:168-169). Locking pins a segment's pages in memory, preventing paging for latency-sensitive workloads. The `shm_lkcnt` field in the descriptor tracks how many times a segment has been locked, a careful tally that ensures each lock is matched by an unlock.

These controls are the city ordinances: how large a courtyard may be, how many courtyards can exist, and whether a courtyard can be protected from eviction.

> **The Ghost of SVR4:** We offered a courtyard with strict accounting and a simple keyring. Modern systems still keep System V shared memory, but they also provide `mmap` -backed files, huge pages, and lock-free shared rings. The courtyard has grown larger, yet the rule is unchanged: the last keyholder decides when the gate is closed.

The Ledger Closes

Shared memory is the fastest IPC because the kernel steps out of the way after attachment. The ledger keeps permissions and counts, the `anon_map` holds the actual pages, and each process carries its own key. When the last key is returned, the courtyard becomes empty once more.

Address Space: The Estate Ledger and the Surveyor

Imagine a grand estate with rolling fields, stone walls, and a patient surveyor who keeps the official ledger. Each parcel has a boundary, a purpose, and a steward. Some parcels are orchards, others are quarries, and a few are common grounds shared by the town. The ledger does not grow crops; it records where they may grow. The surveyor does not haul stone; he marks where the quarry begins and ends.

SVR4's address space is that estate ledger. It records every mapped range of virtual memory, keeps the parcels in order, and hands each range to the steward responsible for its behavior. The kernel's surveyor is the `struct as`, and its parcels are segments.

The Estate Ledger: `struct as`

The address space definition lives in `vm/as.h` (vm/as.h:45-63). It is a compact record that ties segments to a hardware translation layer and tracks the size of the estate.

```
struct as {
    u_int   a_lock: 1;
    u_int   a_want: 1;
    u_int   a_paglck: 1;
    u_int   : 13;
    u_short a_keepcnt;  /* number of `keeps' */
    struct  seg *a_segs;    /* segments in this address space */
    struct  seg *a_seglast; /* last segment hit on the address space */
    size_t  a_size;     /* size of address space */
    size_t  a_rss;      /* memory claim for this address space */
    struct  hat a_hat;  /* hardware address translation */
};
```

The Estate Ledger Structure (vm/as.h:52-63)

Key details:

- **`a_segs`** is a circular, sorted list of segments.
- **`a_seglast`** is a locality hint: the last segment hit by a lookup.
- **`a_hat`** embeds the hardware address translation state for this space.

The comment in `as.h` makes the division of labor explicit: “All the hard work is in the segment drivers and the hardware address translation code” (vm/as.h:46-50). The ledger tracks the parcels; the stewards do the work.

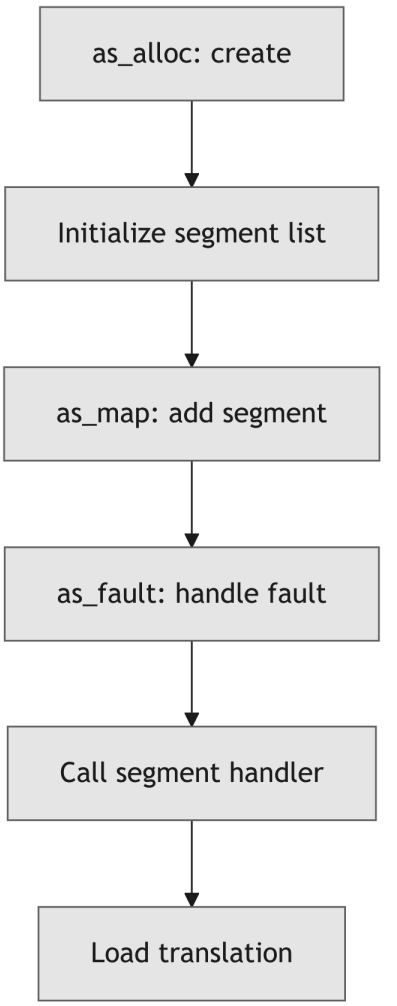


Figure 2.1.1: A Process Address Space as an Estate

Address Space - Kingdom Provinces

Establishing a Parcel: `as_map()`

When the kernel maps a new range, it calls `as_map()` in `vm_as.c`. The function rounds the range to page boundaries, checks resource limits, allocates a segment, and invokes the segment driver’s create routine (vm/vm_as.c:504-551).

```
int
as_map(as, addr, size, crfp, argsp)
    struct as *as;
    addr_t addr;
    u_int size;
    int (*crfp)();
    caddr_t argsp;
{
    register struct seg *seg;
    register addr_t raddr;
    register u_int rsize;
    int error;

    raddr = (addr_t)((u_int)addr & PAGEMASK);
    rsize = (((u_int)(addr + size) + PAGEOFFSET) & PAGEMASK) -
(u_int)raddr;

    if (as->a_size + rsize > u.u_rlimit[RLIMIT_VMEM].rlim_cur)
        return (ENOMEM);

    seg = seg_alloc(as, addr, size);
    if (seg == NULL)
        return (ENOMEM);

    as->a_seglast = seg;
    error = (*crfp)(seg, argsp);
    if (error != 0) {
        seg_free(seg);
    } else {
        as->a_size += rsize;
        if (as->a_paglck) {
            error = as_ctl(as, addr, size, MC_LOCK, 0,
                (caddr_t)NULL, (ulong *)NULL, (size_t)NULL);
            if (error != 0)
                (void) as_unmap(as, addr, size);
        }
    }
    return (error);
}
```

The Parcel Grant (vm/vm_as.c:504-551, excerpt)

The key idea: `as_map()` does not know what kind of segment it is creating. It calls a creation function pointer (`crfp`), allowing vnode-backed, anonymous, or device segments to establish their own rules.

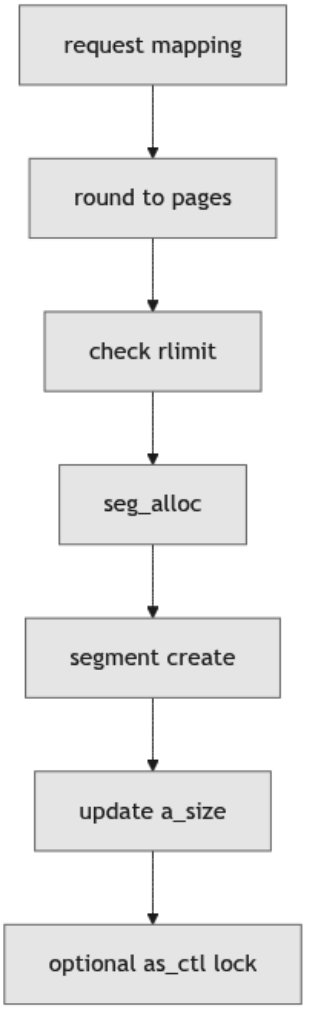


Figure 2.1.2: Mapping a New Parcel

Duplication by Survey: `as_dup()`

During `fork()`, the child receives a copy of the parent's address space. `as_dup()` allocates a new `as`, walks the segment list, and asks each segment driver to duplicate itself (vm/vm_as.c:134-172).

```
newas = as_alloc();
sseg = seg = as->a_segs;
if (seg != NULL) {
    do {
        newseg = seg_alloc(newas, seg->s_base, seg->s_size);
        if (newseg == NULL) {
            as_free(newas);
            return (NULL);
        }
        if ((*seg->s_ops->dup)(seg, newseg) != 0) {
            seg_free(newseg);
            as_free(newas);
            return (NULL);
        }
        newas->a_size += seg->s_size;
        seg = seg->s_next;
    } while (seg != sseg);
}
if (hat_dup(as, newas) != 0) {
    as_free(newas);
    return (NULL);
}
```

The Surveyor’s Copy (vm/vm_as.c:134-169, excerpt)

Here, the segment driver decides whether to copy or share pages. Copy-on-write is implemented in those segment drivers, not in the address space itself. The ledger merely requests a duplicate.

Faults and Boundaries: `as_fault()`

Page faults are resolved by walking the segment list and delegating to the segment’s fault handler. `as_fault()` computes a rounded range, finds the first segment, then calls `seg->s_ops->fault` on each subrange (vm/vm_as.c:224-307).

```
seg = as_segat(as, raddr);
if (seg == NULL)
    return (FC_NOMAP);

res = (*seg->s_ops->fault)(seg, raddr, ssize, type, rw);
if (res == FC_MAKE_ERR(EFAULT) && type == F_SOFTLOCK) {
    (void) as_fault(as, raddr, ssize, F_SOFTUNLOCK, rw);
}
```

The Boundary Dispute (vm/vm_as.c:257-304, excerpt)

If a soft-lock fails partway through, `as_fault()` revisits the already-locked pages and unlocks them with `F_SOFTUNLOCK` (vm/vm_as.c:285-304). The surveyor refuses to leave the ledger half-updated.

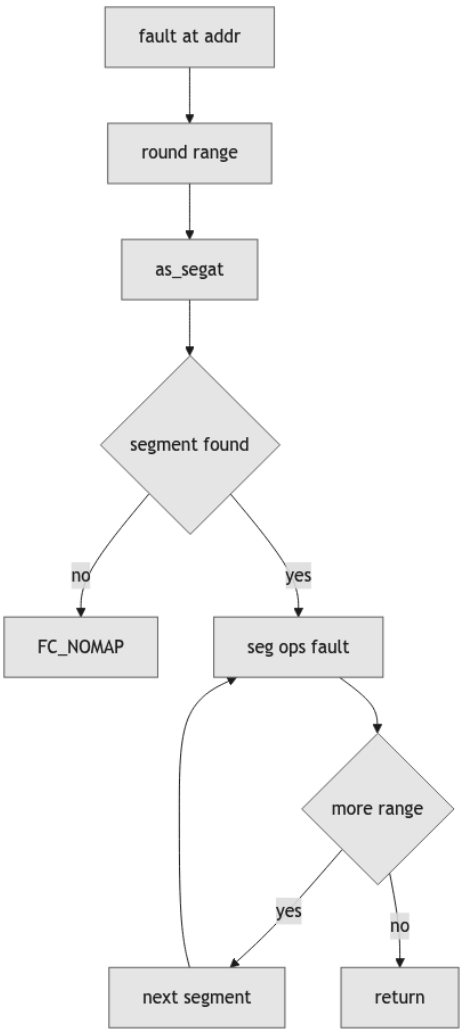


Figure 2.1.3: Fault Resolution Across Segments

Keeping Order: `as_addseg()`

Segments are stored in a sorted, circular list. `as_addseg()` inserts a new segment in address order, rejecting overlaps (vm/vm_as.c:178-217). This ensures the estate ledger remains strictly ordered and non-overlapping.

The sorting rule is simple: each new parcel must fit between existing ones without touching their boundaries. Overlaps return `-1`, a clear signal that the requested mapping is illegal.

> **The Ghost of SVR4:** We kept our estates in a sorted ring and asked each steward to resolve faults and duplicates. Modern kernels still keep per-process maps, but they have sprouted red-black trees, VMA caches, and speculative faults. The surveyor's ledger has evolved into a faster, more complex registry, yet the idea is unchanged: name each parcel, enforce its boundaries, and ask the right steward when something goes wrong.

The Ledger Closes

An address space is a map, not a machine. It remembers segments, sizes, and translation state, and it delegates the real work to its stewards. SVR4's design keeps the ledger small and the rules clear: map, duplicate, fault, unmap. The surveyor's hand is steady, and the estate remains ordered.

HAT Layer: The Translator's Desk

Imagine a busy customs office where every traveler carries a passport in one language but must be recorded in another. The translator at the desk does not care about the journey; she simply maps names and numbers between two alphabets. If a translation changes, she must update her ledger and notify the guards so they no longer trust the old translation.

SVR4's Hardware Address Translation (HAT) layer is that translator. It maps virtual addresses to physical frames, keeps a ledger of page tables, and flushes the TLB when the mapping changes.

The Translator's Ledger: `struct hat`

On i386, each address space embeds a `struct hat` defined in `vm/vm_hat.h` (vm/vm_hat.h:82-89).

```
typedef struct hat {
    struct hatpt *hat_pts;     /* current page table list */
    struct hatpt *hat_ptlast;  /* last page table to fault */
} hat_t;
```

The Translation Ledger (vm/vm_hat.h:86-89)

The HAT keeps a list of page tables and a locality hint (`hat_ptlast`) to speed repeated faults in the same region. It is a simple structure, but it anchors the entire translation system.

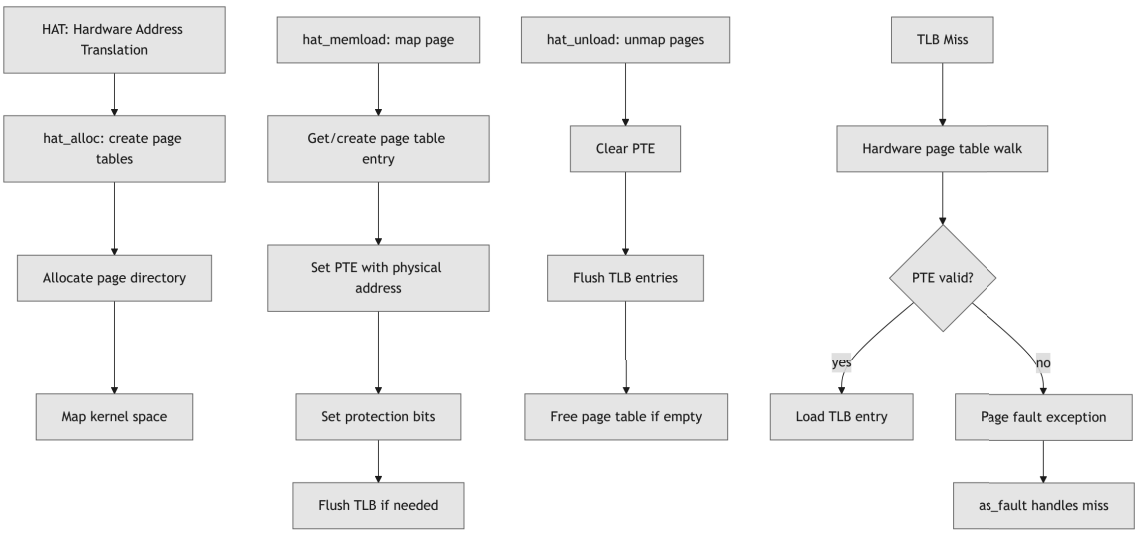
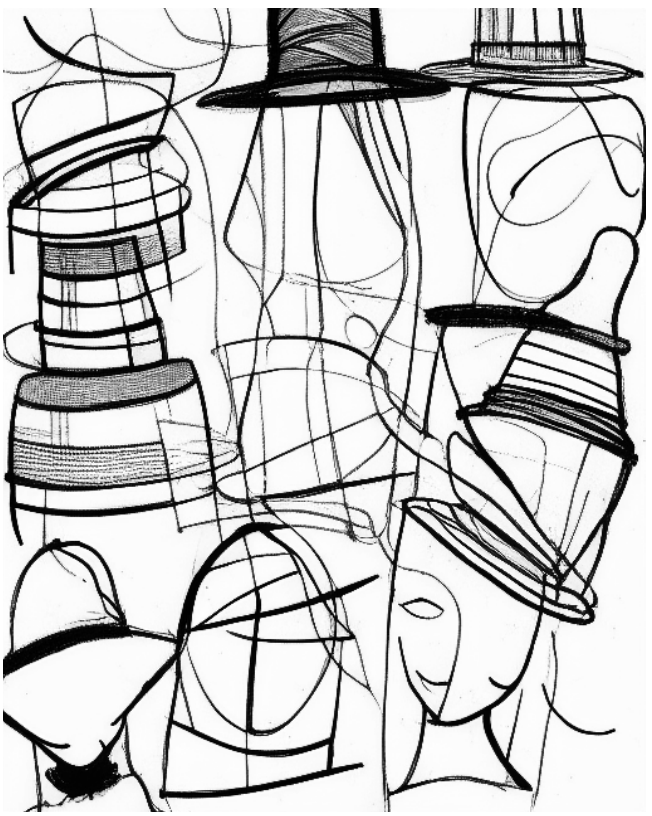


Figure 2.2.1: HAT Between Address Space and Hardware



HAT Layer - Royal Milliner's Shop

The Operation Set: `hat.h`

The HAT interface exposes a standardized set of operations for segment drivers and the VM system (vm/hat.h:81-147). These include loading mappings, unloading them, and syncing page state.

```
void hat_memload(/* seg, addr, pp, prot, flags */);
void hat_unload(/* seg, addr, len, flags */);
void hat_pagesync(/* pp */);
```

The Translation Tools (vm/hat.h:91-112, abridged)

Flags like `HAT_LOCK`, `HAT_UNLOCK`, and `HAT_RELEPP` control whether mappings are locked down or whether the underlying page is released when a mapping is removed (vm/hat.h:130-147). These flags are the translator's instructions: whether the ledger entry is permanent, temporary, or due for release.

Loading a Mapping: `hat_memload()`

When a segment driver has a page to map, it calls `hat_memload()`. The i386 implementation allocates or finds the page table, writes the PTE, and makes the translation visible to hardware. The full routine is lengthy, but its essence is clear: ensure a page table exists, then load the mapping (vm/vm_hat.c:1069-1107).

The mapping can be locked with `HAT_LOCK` to keep it resident, which is how the kernel pins critical pages.

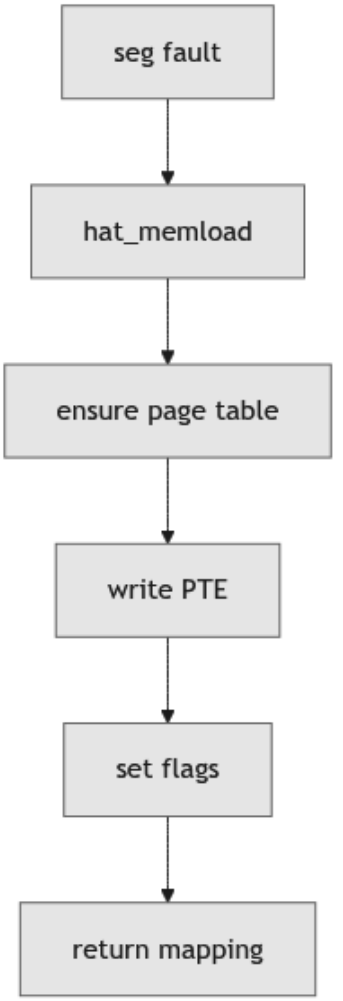


Figure 2.2.2: Loading a Translation

Unloading and Syncing: `hat_unload()` and `hat_pagesync()`

When a mapping is removed, `hat_unload()` clears the PTEs and may release the underlying page depending on flags such as `HAT_RELEPP` or `HAT_PUTPP` (vm/hat.h:135-137). The i386 HAT then invalidates the TLB entry to prevent stale translations (vm/vm_hat.c:617-623).

Page synchronization is handled by `hat_pagesync()`, which pulls reference and modification bits from hardware into the page structure (vm/hat.h:110-112). This is how the VM system learns which pages are dirty or recently accessed.

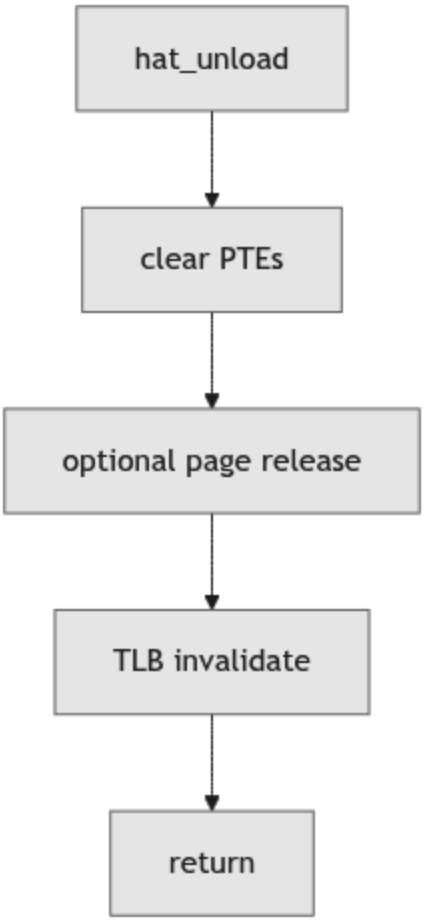


Figure 2.2.3: Clearing Mappings and Flushing the TLB

The TLB Contract

The Translation Lookaside Buffer is fast but forgetful. It caches translations, so the HAT must invalidate entries whenever mappings change. The i386 implementation uses `invlpg` or a CR3 reload to ensure no stale mappings persist. The translator's desk is useless if the guards still read the old passport.

The Ghost of SVR4: We kept the HAT as a strict interface so segment drivers could remain portable. Modern systems still separate the translation layer, but they also add huge pages, ASID tagging, and batched TLB shootdowns. The translator now works at scale, yet she still keeps the same ledger: virtual address in, physical frame out.

The Ledger Closes

The HAT layer is the interpreter between software intent and hardware reality. It loads mappings, unloads them, and keeps the TLB honest. Without the translator’s desk, the address space is a map with no way to reach the ground.

Page Management: The Keeper of Rooms

A city of machines keeps its citizens in rooms. Some rooms are occupied, some are idle but still furnished, and some are entirely empty. The keeper is not a poet; she is a clerk with ledgers. Her work is to find a room quickly, remember who it belongs to, and decide when a room should be cleaned and returned to the pool.

SVR4's page management system is that keeper. It tracks each physical page with a `page_t` record, ties pages to vnode identities, and manages two free lists that distinguish between empty rooms and rooms that still remember their last occupant.

The Ledger Entry: `struct page`

Every physical page has a `struct page` record that carries its identity, state, and list links (vm/page.h:57-101).

```
typedef struct page {
    u_int       p_lock: 1,
        p_want: 1,
        p_free: 1,
        p_intrans: 1,
        p_gone: 1,
        p_mod: 1,
        p_ref: 1,
        p_pagein: 1,
        p_nc: 1,
        p_age: 1;
    u_int       p_nio : 6;
    u_short     p_keepcnt;
    struct      vnode *p_vnode;
    u_int       p_offset;
    struct page *p_hash;
    struct page *p_next;
    struct page *p_prev;
    struct page *p_vpnext;
    struct page *p_vpprev;
    struct phat p_hat;
    u_short     p_lckcnt;
    u_short     p_cowcnt;
} page_t;
```

The Room Ledger (vm/page.h:57-89)

The fields mirror the keeper's daily questions. Is the room free (`p_free`)? Is a move underway (`p_intrans`)? Does it remember its owner (`p_vnode` and `p_offset`)? The list pointers keep the page on the free lists or on a vnode's page ring. The `p_hat` mapping field, shared with the HAT layer, is the keeper's seal for whether the room is currently mapped (vm/page.h:77-87).

The ledger is also aware of noncontiguous physical memory. `pageac` describes each contiguous span of page frames so the allocator can search disjoint segments when a request needs physical contiguity (vm/page.h:113-125).

Page Management - Farmer's Field

The Two Free Lists: Empty Rooms and Remembered Rooms

`page_free()` is the clerk's act of returning a room to the pool. SVR4 does not use a single list; it keeps two circular lists: `page_freelist` for pages with no vnode identity and `page_cachelist` for pages that still remember their former contents (vm/vm_page.c:764-888).

```
/* Put page on the "free" list.  The free list is really two circular
lists
 * with page_freelist and page_cachelist pointers into the middle of the
lists.
 */
void
page_free(pp, dontneed)
    register page_t *pp;
    int dontneed;
{
    ...
    if (vp == NULL) {
        /* page has no identity, put it on the front of the free list */
        page_add(&page_freelist, pp);
    } else {
        page_add(&page_cachelist, pp);
        if (!dontneed || nopageage)
            page_cachelist = page_cachelist->p_next;
    }
    ...
}
```

The Two Pools (vm/vm_page.c:764-879, abridged)

A page with no identity goes straight to the front of `page_freelist`. A page with a vnode identity is cached on `page_cachelist`, and by default is pushed toward the tail to age out last. This is the keeper's compromise: keep the most recently used rooms warm in case the same tenant returns.

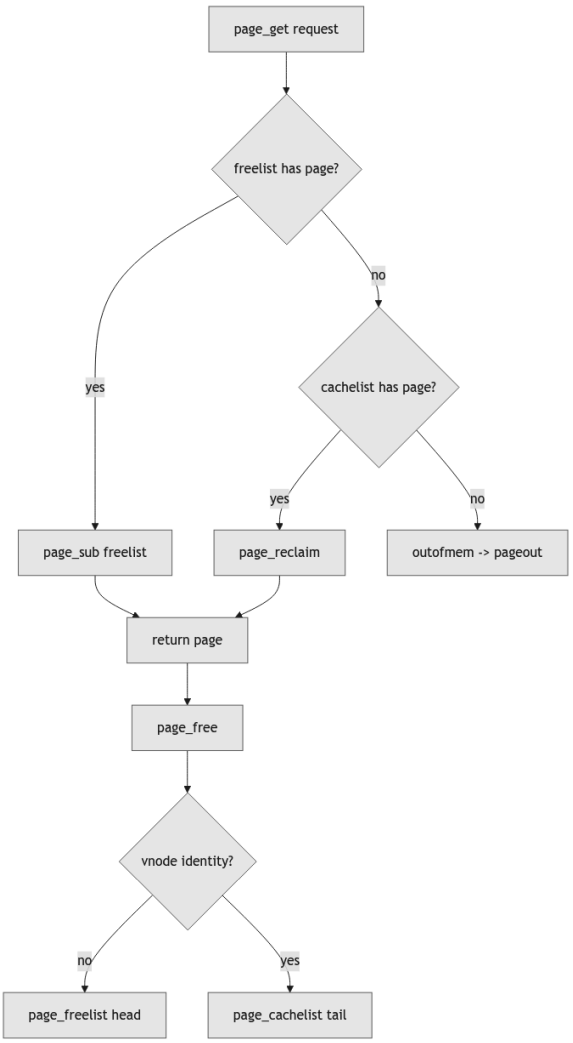


Figure 2.3.1: Free and Cache Lists in the Allocator

Naming and Finding Rooms: Hash Chains and Vnode Rings

The page hash is the index card catalog for the keeper. `page_hashin()` adds a page to the hash table and to the vnode's circular list (vm/vm_page.c:1759-1804). `page_find()` and `page_lookup()` search by `<vnode, offset>`, reclaiming a page from the cache list if necessary (vm/vm_page.c:550-651).

```
page_t *
page_find(vp, off)
    register struct vnode *vp;
    register u_int off;
{
    for (pp = page_hash[PAGE_HASHFUNC(vp, off)]; pp; pp = pp->p_hash)
        if (pp->p_vnode == vp && pp->p_offset == off && pp->p_gone == 0)
            break;
    if (pp != NULL && pp->p_free)
        page_reclaim(pp);
    return (pp);
}
```

The Catalog Lookup (vm/vm_page.c:550-590, abridged)

`page_lookup()` adds the patience of a clerk who waits for a room in transit. If a page is being paged in, it waits on the page's condition before reclaiming it (vm/vm_page.c:614-649). The key idea is that identity wins: if the vnode/offset matches, we do not manufacture a new page, we reclaim the old one.

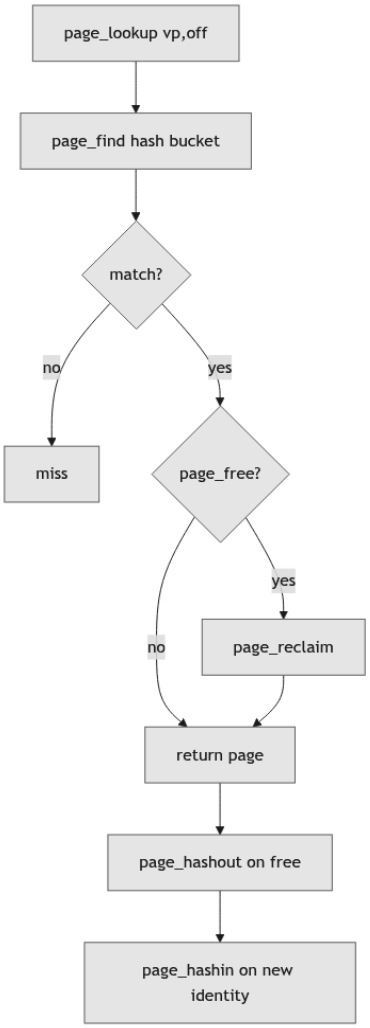


Figure 2.3.2: Hash Table and Vnode Page Rings

Allocation: Asking for Rooms

`page_get()` is the official request for physical pages. It enforces resource limits, optionally sleeps, and prods the pageout daemon when memory is tight (vm/vm_page.c:1050-1154). It also handles the special case of physically contiguous allocation, walking the `pageac` table to find a run of free frames (vm/vm_page.c:1159-1188).

```
page_t *
page_get(bytes, flags)
    u_int bytes;
    u_int flags;
{
    npages = btopr(bytes);
    reqfree = (flags & P_NORESOURCELIM) ? npages : npages + minpagefree;
    while (freemem < reqfree) {
        if (!(flags & P_CANWAIT))
            return (NULL);
        if (bclnlist != NULL)
            cleanup();
        outofmem();
        (void) sleep((caddr_t)&freemem, PSWP+2);
    }
    ...
}
```

The Allocation Request (vm/vm_page.c:1050-1154, abridged)

The flags tell the keeper how urgent the request is: `P_CANWAIT` permits sleeping, while `P_PHYSCONTIG` demands a contiguous stretch for DMA or device mappings. The allocator balances fairness (resource limits) with liveness (invoking `outofmem()` to drive pageout).

The Quiet Cleanup: `page_abort()` and Reclamation

When a page loses its identity, `page_abort()` clears mappings, marks the page gone, and drops it back to `page_free()` (vm/vm_page.c:707-761). This path is the keeper's reset: tear down ownership, flush mappings, and return the room to the pool. If the page is still in transit, the cleanup waits for `pvn_done()` to finish the I/O before finally freeing it.

The Ghost of SVR4:

We kept one central ledger and two long queues. In your time, the keeper has been split into many clerks. Modern kernels maintain per-CPU page lists, per-node pools for NUMA, and

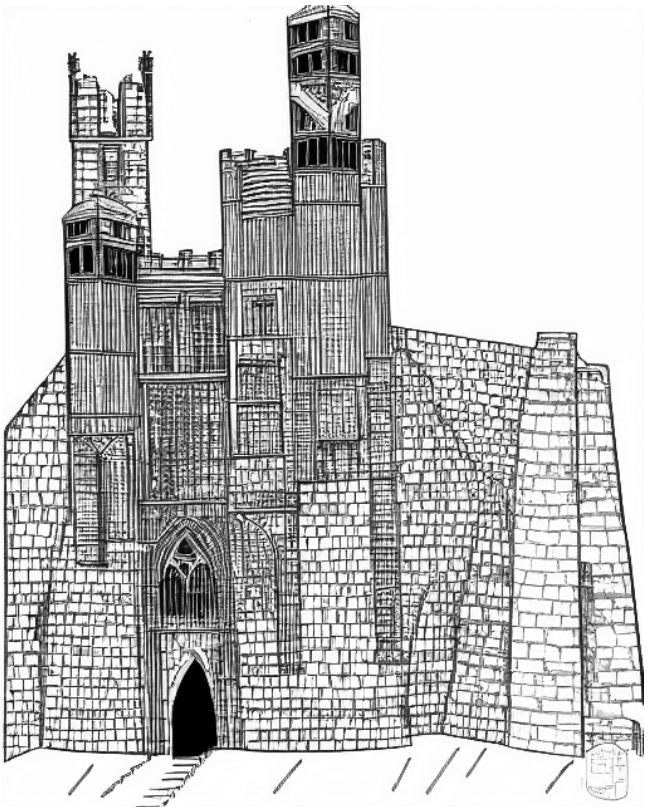
memcg-aware LRU lists for containers. The policy is more complex, but the core idea has not changed: a page is a room with a name, and the kernel must always know which rooms are empty, which are cached, and which still belong to someone.

Closing the Ledger

Page management is the art of disciplined forgetting. The keeper must remember just enough to reuse the right rooms quickly, and forget enough to keep the city breathing. SVR4’s two-list allocator, its hash catalog, and its careful wait loops are the steady hand that keeps the rooms available.

Memory Segments

Memory Segments



Memory Segments - Medieval Castle

Overview

Segments represent contiguous virtual address ranges with uniform properties. The `seg` structure provides a framework for managing different types of memory regions through pluggable segment drivers. Each segment type implements operations for fault handling, protection changes, and synchronization.

Segment Structure

```
struct seg {
    addr_t s_base;          /* base virtual address */
    u_int s_size;           /* size in bytes */
    struct as *s_as;        /* containing address space */
    struct seg *s_next;     /* next segment in address space */
    struct seg *s_prev;     /* previous segment */
    struct seg_ops *s_ops;  /* segment operations */
    caddr_t s_data;         /* driver private data */
};
```

The segment list is maintained in sorted order by base address. The `s_ops` pointer provides the interface to segment driver operations.

Segment Allocation

The `seg_alloc()` function (vm_seg.c:66) allocates and attaches a segment:

```
struct seg *
seg_alloc(as, base, size)
    struct as *as;
    register addr_t base;
    register u_int size;
{
    register struct seg *new;
    addr_t segbase;
    u_int segsize;

    segbase = (addr_t)((u_int)base & PAGEMASK);
    segsize =
        (((u_int)(base + size) + PAGEOFFSET) & PAGEMASK) - (u_int)segbase;

    if (!valid_va_range(&segbase, &segsize, segsize, AH_LO))
        return ((struct seg *)NULL);

    new = (struct seg *)kmem_fast_alloc((caddr_t *)&seg_freelist,
        sizeof (*seg_freelist), seg_freeincr, KM_SLEEP);
    struct_zero((caddr_t)new, sizeof (*new));
    if (seg_attach(as, segbase, segsize, new) < 0) {
        kmem_fast_free((caddr_t *)&seg_freelist, (char *)new);
        return ((struct seg *)NULL);
    }
    return (new);
}
```

Addresses and sizes are page-aligned using `PAGEMASK`. The fast allocator maintains a freelist of segment structures to avoid frequent kmem_alloc() calls.

Segment Attachment

The `seg_attach()` function (vm_seg.c:101) links the segment into the address space:

```
int
seg_attach(as, base, size, seg)
    struct as *as;
    addr_t base;
    u_int size;
    struct seg *seg;
{
    seg->s_as = as;
    seg->s_base = base;
    seg->s_size = size;
    return (as_addseg(as, seg));
}
```

The `as_addseg()` function inserts the segment into the sorted list, checking for overlaps with existing segments. Overlapping segments cause allocation failure.

Segment Operations

Each segment driver provides a `seg_ops` structure with function pointers:

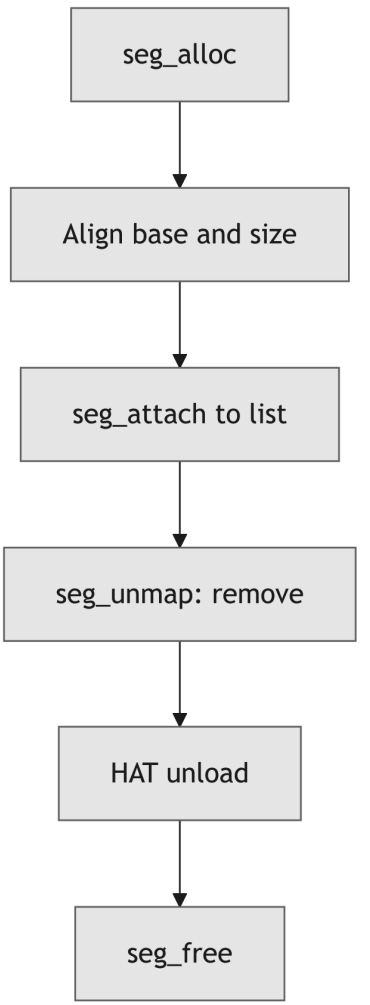
```
struct seg_ops {
    int (*dup)(struct seg *, struct seg *);
    int (*unmap)(struct seg *, addr_t, u_int);
    void (*free)(struct seg *);
    faultcode_t (*fault)(struct seg *, addr_t, u_int, enum fault_type,
                         enum seg_rw);
    int (*setprot)(struct seg *, addr_t, u_int, uint);
    int (*checkprot)(struct seg *, addr_t, uint);
    int (*sync)(struct seg *, addr_t, u_int, int, uint);
};
```

The `fault` operation handles page faults within the segment. The `dup` operation creates copy-on-write mappings for fork. The `unmap` operation removes mappings and frees resources.

Common Segment Types

seg_vn: File-backed segments for executables and mmap’d files **seg_u**: Stack and heap segments with anonymous backing **seg_dev**: Device memory mappings **seg_kmem**: Kernel memory segments **seg_map**: Kernel temporary mappings

Each type provides specialized handling for its particular use case while conforming to the common segment driver interface.



Anonymous Memory

Anonymous Memory

Anonymous Memory - Lost and Found

Overview

The anonymous memory layer manages physical pages that have no permanent identity in the file system. This includes stack pages, heap allocations, and copy-on-write pages created during fork(). Anonymous pages use swap space for backing storage and are discarded when all references are removed.

Anon Structure

The core data structure is the `anon` struct defined in anon.h:

```
struct anon {
    int an_refcnt;
    union {
        struct page *an_page;    /* hint to the real page */
        struct anon *an_next;    /* free list pointer */
    } un;
    struct anon *an_bap;         /* pointer to real anon */
    short an_flag;               /* ALOCKED, AWANT */
    short an_use;                /* for debugging */
};
```

The reference count enables copy-on-write optimization after fork(). When multiple processes share the same physical page, the `an_refcnt` tracks how many references exist. The `an_page` field provides a hint to locate the page without hash table lookup.

Reservation and Allocation

Anonymous memory requires explicit reservation before use. The `anon_resv()` function (vm_anon.c:124) checks available swap space:

```
int
anon_resv(size)
    u_int size;
{
    register int pages = btopr(size);

    if (availsmem - pages < tune.t_minasmem) {
        nomemmsg("anon_resv", pages, 0, 0);
        return 0;
    }

    if (anoninfo.ani_resv + pages > anoninfo.ani_max) {
        return 0;
    }
    anoninfo.ani_resv += pages;
    availsmem -= pages;
    return (1);
}
```

Reservation ensures the system won't over-commit swap space. The `anoninfo` structure tracks maximum, free, and reserved anonymous pages globally.

Anon Allocation

The `anon_alloc()` function (vm_anon.c:161) allocates an anon slot:

```
struct anon *
anon_alloc()
{
    register struct anon *ap;

    mon_enter(&anon_lock);
    ap = swap_alloc();
    if (ap != NULL) {
        anoninfo.ani_free--;
        ap->an_refcnt = 1;
        ap->un.an_page = NULL;
        ap->an_flag = ALOCKED;
    }
    mon_exit(&anon_lock);
    return (ap);
}
```

The function delegates to `swap_alloc()` to obtain an anon structure from the swap layer. Initial reference count is 1, and the slot is locked to prevent concurrent access.

Reference Counting

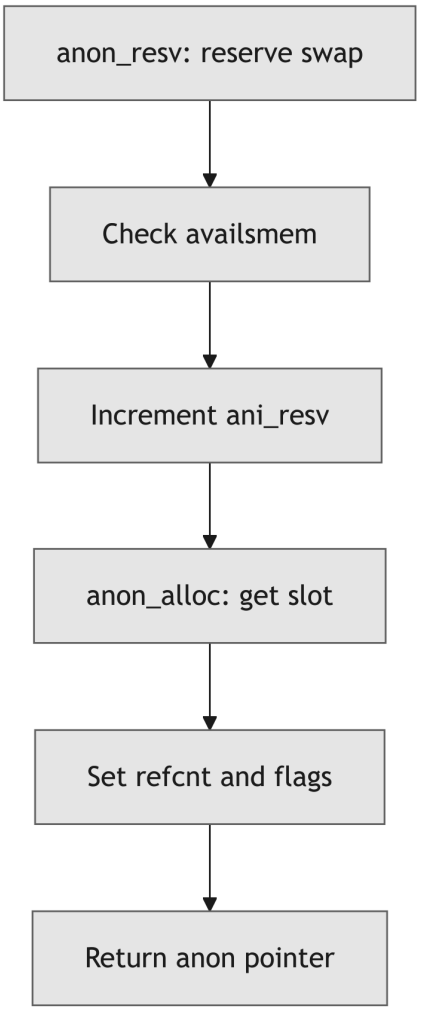
The `anon_decref()` function (vm_anon.c:208) handles reference count decrements:

```
STATIC void
anon_decref(ap)
    register struct anon *ap;
{
    register page_t *pp;
    struct vnode *vp;
    u_int off;

    if (ap->an_refcnt == 1) {
        swap_xlate(ap, &vp, &off);
        pp = page_find(vp, off);

        if (pp != NULL) {
            if (pp->p_intrans == 0)
                page_abort(pp);
        }
        ap->an_refcnt--;
        mon_enter(&anon_lock);
        swap_free(ap);
        anoninfo.ani_free++;
        mon_exit(&anon_lock);
    } else {
        ap->an_refcnt--;
    }
}
```

When the reference count reaches zero, the function frees the associated page and returns the anon slot to the swap layer. The `page_abort()` call removes the vnode association and returns the page to the free list.

Copy-on-Write Optimization

Anonymous memory provides critical support for efficient fork() through copy-on-write. When a process forks, child and parent initially share the same physical pages with incremented reference counts. Only when one process writes to a shared page does the system create a private copy. This optimization significantly reduces fork() overhead and memory consumption.



Segment Drivers - Vnode

Segment Drivers - VNode

Overview

The seg_vn driver manages file-backed and anonymous memory segments. It handles executable mappings, shared libraries, mmap'd files, and copy-on-write semantics for fork(). The driver supports both MAP_PRIVATE and MAP_SHARED mappings with flexible backing storage.

Segvn Data Structure

The `segvn_data` structure (seg_vn.h:79) contains per-segment state:

```
struct segvn_data {
    mon_t lock;
    u_char pageprot;         /* true if per page protections present */
    u_char prot;             /* current segment prot if pageprot == 0 */
    u_char maxprot;          /* maximum segment protections */
    u_char type;             /* type of sharing done */
    struct vnode *vp;        /* vnode that segment mapping is to */
    u_int offset;            /* starting offset of vnode for mapping */
    u_int anon_index;        /* starting index into anon_map anon array */
    struct anon_map *amp;    /* pointer to anon share structure */
    struct vpage *vpage;     /* per-page information */
    struct cred *cred;       /* mapping credentials */
    u_int swresv;            /* swap space reserved for this segment */
};
```

The `vp` and `offset` fields identify the file backing, while `amp` points to anonymous memory for private pages. The driver can combine file and anonymous backing within a single segment.

Anon Map Structure

The `anon_map` structure (seg_vn.h:69) manages shared anonymous memory:

```
struct anon_map {
    u_int refcnt;           /* reference count on this structure */
    u_int size;             /* size in bytes mapped by the anon array */
    struct anon **anon;     /* pointer to an array of anon pointers */
    u_int swresv;           /* swap space reserved for this anon_map */
};
```

Multiple segments can share the same anon_map for System V shared memory. The reference count tracks sharing, and the entire structure is freed when refcnt reaches zero.

Segment Creation

The `segvn_create()` function (seg_vn.c:157) establishes a new mapping:

```
int
segvn_create(seg, argsp)
    struct seg *seg;
    _VOID *argsp;
{
    register struct segvn_crargs *a = (struct segvn_crargs *)argsp;
    register struct segvn_data *svd;
    register u_int swresv = 0;
    register struct cred *cred;
    int error;

    if (a->type != MAP_PRIVATE && a->type != MAP_SHARED)
        cmn_err(CE_PANIC, "segvn_create type");

    if (a->amp != NULL && a->vp != NULL)
        cmn_err(CE_PANIC, "segvn_create anon_map");

    if ((a->vp == NULL && a->amp == NULL)
      || (a->type == MAP_PRIVATE && (a->prot & PROT_WRITE))) {
        if (anon_resv(seg->s_size) == 0)
            return (EAGAIN);
        swresv = seg->s_size;
    }

    error = hat_map(seg, a->vp, a->offset & PAGEMASK,
        a->prot & ~PROT_WRITE, HAT_PRELOAD);
    if (error != 0) {
        if (swresv != 0)
            anon_unresv(swresv);
        return(error);
    }

    cred = a->cred ? (crhold(a->cred), a->cred) : crgetcred();
    if (a->vp) {
        error = VOP_ADDMAP(a->vp, a->offset & PAGEMASK, seg->s_as,
            seg->s_base, seg->s_size, a->prot,
            a->maxprot, a->type, cred);
        if (error) {
            if (swresv != 0)
                anon_unresv(swresv);
            crfree(cred);
            hat_unload(seg, seg->s_base, seg->s_size, HAT_NOFLAGS);
            return (error);
        }
    }
```

The function first validates arguments, then reserves swap space for potentially private pages. HAT resources are reserved with initial protections disabling write access to force protection faults for lazy allocation.

Segment Concatenation

Seg_vn attempts to merge adjacent segments with compatible properties (seg_vn.c:237):

```
if ((seg->s_prev != seg) && (a->amp == NULL) && (seg->s_as != &kas)) {
    register struct seg *pseg, *nseg;

    pseg = seg->s_prev;
    if (pseg->s_base + pseg->s_size == seg->s_base &&
        pseg->s_ops == &segvn_ops &&
        segvn_extend_prev(pseg, seg, a, swresv) == 0) {
        /* success! now try to concatenate with following seg */
        crfree(cred);
        nseg = pseg->s_next;
        if (nseg != pseg && nseg->s_ops == &segvn_ops &&
            pseg->s_base + pseg->s_size == nseg->s_base)
            (void) segvn_concat(pseg, nseg);
        return (0);
    }
```

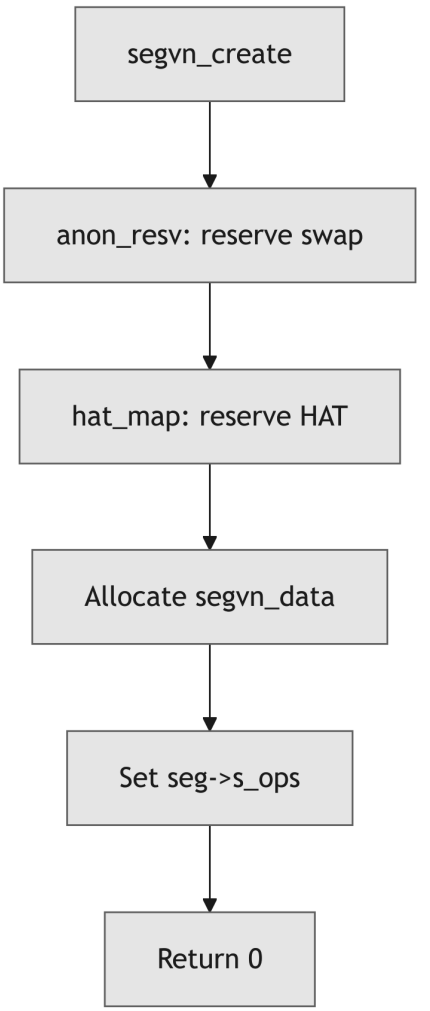
Concatenation reduces address space fragmentation and improves efficiency by merging segments with the same vnode, protections, and mapping type.

Operations Vector

The seg_vn driver provides a complete operations vector (seg_vn.c:91):

```
struct seg_ops segvn_ops = {
    segvn_dup,
    segvn_unmap,
    segvn_free,
    segvn_fault,
    segvn_faulta,
    segvn_unload,
    segvn_setprot,
    segvn_checkprot,
    segvn_kluster,
    segvn_swapout,
    segvn_sync,
    segvn_incore,
    segvn_lockop,
    segvn_getprot,
    segvn_getoffset,
    segvn_gettype,
    segvn_getvp,
};
```

The `segvn_fault` operation handles page faults by reading from the vnode or allocating anonymous pages. The `segvn_dup` operation implements copy-on-write for fork() by sharing the anon_map and marking pages read-only.


segvn_create
anon_resv: reserve swap
hat_map: reserve HAT
Allocate segvn_data
Set seg->s_ops
Return 0

Segment Drivers - Kernel Memory

Segment Drivers - Kernel Memory

Overview

The seg_kmem driver manages kernel virtual address space segments. Unlike user segments, kernel segments directly manipulate page table entries and do not use the standard fault mechanism. The driver provides explicit allocation and deallocation of physical pages for kernel use.

Kernel Segments

SVR4 defines several global kernel segments (seg_kmem.c:110):

```
struct seg ktextseg;        /* kernel text segment */
struct seg kvseg;           /* pageable kernel virtual memory */
struct seg kpseg;           /* non-pageable kernel memory */
struct seg kpioseg;         /* kernel programmed I/O area */
struct seg kdvmaseg;        /* DMA-able memory segment */
```

Each segment serves a specific purpose in the kernel address space. The `kvseg` segment holds pageable kernel data, while `kpseg` contains permanently wired pages.

Operations Vector

The seg_kmem operations vector (seg_kmem.c:116) differs significantly from user segment drivers:

```
struct seg_ops segkmem_ops = {
    (int(*)())segkmem_badop,        /* dup */
    (int(*)())segkmem_badop,        /* split */
    (void(*)())segkmem_badop,       /* free */
    segkmem_fault,
    segkmem_faulta,
    segkmem_unload,
    segkmem_setprot,
    segkmem_checkprot,
    (int(*)())segkmem_badop,        /* kluster */
    (u_int (*)())segkmem_badop,     /* swapout */
    (int(*)())segkmem_badop,        /* sync */
    (int(*)())segkmem_badop,        /* incore */
    (int(*)())segkmem_badop,        /* lockop */
    segkmem_getprot,
    segkmem_getoffset,
    segkmem_gettype,
    segkmem_getvp,
};
```

Many operations are set to `segkmem_badop`, which panics if called. Kernel segments do not support duplication (fork), splitting, or swapping since these operations are meaningless for kernel memory.

Segment Creation

The `segkmem_create()` function (seg_kmem.c:142) performs minimal initialization:

```
int
segkmem_create(seg, argsp)
    struct seg *seg;
    caddr_t argsp;
{
    /* No need to notify the hat layer, since the SDT's are
     * already allocated for seg_kmem; i.e. no need to call
     * hat_map().
     */

    seg->s_ops = &segkmem_ops;
    seg->s_data = seg->s_base;    /* must be set to something */

    return (0);
}
```

Unlike user segments, no HAT reservation is needed since kernel page tables are pre-allocated at boot time.

Physical Page Allocation

The `segkmem_alloc()` function (seg_kmem.c:356) allocates and maps physical pages:

```
STATIC int
segkmem_alloc(seg, addr, len, flag)
    struct seg *seg;
    addr_t addr;
    u_int len;
    int flag;
{
    page_t *pp;
    register pte_t *ppte;
    pte_t tpte;
    int flg;

    tpte.pg_pte = PG_V;
    ASSERT(seg->s_as == &kas);

    flg = ((flag & NOSLEEP) ? P_NOSLEEP : P_CANWAIT);
    if (!(flag & KM_NO_DMA))
        flg |= P_DMA;
    flg |= P_NORESOURCELIM;

    pp = rm_allocpage(seg, addr, len, flg);

    if (pp != (page_t *)NULL) {
        ppte = svtopte(addr);
        while (pp != (page_t *)NULL) {
            ASSERT(!PG_ISVALID(ppte));
            tpte.pgm.pg_pfn = page_pptonum(pp);
            page_sub(&pp, pp);
            *ppte++ = tpte;
            addr += PAGESIZE;
            if (((ulong)ppte & PTMASK) == 0) {
                ppte = svtopte(addr);
            }
        }
        flushtlb();
        return (1);
    }
    return (0);
}
```

The function obtains physical pages from the resource manager, then directly writes page table entries using `svtopte()` to translate virtual addresses to PTE pointers. The TLB is flushed after establishing mappings.

Page Deallocation

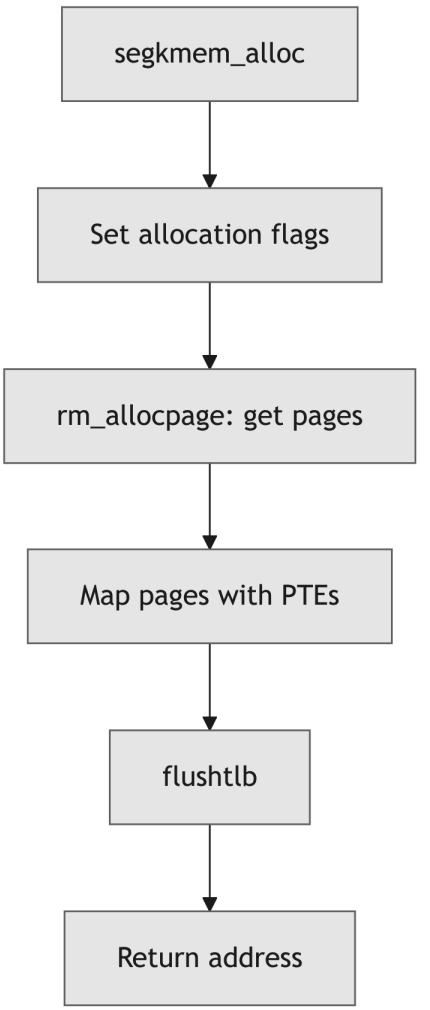
The `segkmem_free()` function (seg_kmem.c:438) reverses the allocation process:

```
STATIC void
segkmem_free(seg, addr, len)
    register struct seg *seg;
    addr_t addr;
    u_int len;
{
    page_t *pp;
    register pte_t *ppte;
    pte_t tpte;

    ASSERT(seg->s_ops == &segkmem_ops);
    ppte = svtopte(addr);

    for (; (int)len > 0; len -= PAGESIZE, addr += PAGESIZE) {
        tpte = *ppte;
        ASSERT(PG_ISVALID(ppte));

        (ppte++)->pg_pte = 0;

        pp = page_numtookpp(tpte.pgm.pg_pfn);
```

The function clears page table entries and returns physical pages to the free pool. No HAT layer involvement is required since seg_kmem bypasses the normal HAT interface.

Protection Management

The `segkmem_setprot()` function (seg_kmem.c:196) modifies page protections by directly updating PTEs. This low-level approach provides maximum efficiency for kernel operations while avoiding the overhead of the normal segment driver protection mechanisms.


segkmem_alloc
Set allocation flags
rm_allocpage: get pages
Map pages with PTEs
flushtlb
Return address

Segment Drivers - Device Memory

Segment Drivers - Device Memory

Overview

The seg_dev driver manages mappings of device memory into user address spaces. This enables user programs to directly access memory-mapped I/O regions, framebuffers, and other hardware resources. The driver delegates physical address translation to device-specific mapping functions.

Segdev Data Structure

The segment private data (seg_dev.h) contains device-specific information:

```
struct segdev_data {
    u_char  pageprot;           /* per-page protections enabled */
    u_char  prot;               /* segment protection */
    u_char  maxprot;            /* maximum protections */
    struct  vnode *vp;          /* device special file vnode */
    off_t   offset;             /* device offset for start of mapping */
    int     (*mapfunc)();       /* device mmap function */
    struct  vpage *vpage;       /* per-page information */
};
```

The `mapfunc` pointer references the device driver's mmap entry point, which translates device offsets to physical page frame numbers. The vnode points to the device special file in `/dev`.

Operations Vector

The seg_dev operations (seg_dev.c:80):

```
STATIC struct seg_ops segdev_ops = {
    segdev_dup,
    segdev_unmap,
    segdev_free,
    segdev_fault,
    segdev_faulta,
    segdev_unload,
    segdev_setprot,
    segdev_checkprot,
    segdev_badop,           /* kluster */
    (u_int (*)()) NULL,     /* swapout */
    segdev_ctlops,          /* sync */
    segdev_incore,
    segdev_ctlops,          /* lockop */
    segdev_getprot,
    segdev_getoffset,
    segdev_gettype,
    segdev_getvp,
};
```

Device segments cannot be swapped or clustered since they represent physical device memory rather than pageable storage.

Fault Handling

The `segdev_fault()` function (seg_dev.c:358) handles device memory access:

```
STATIC faultcode_t
segdev_fault(seg, addr, len, type, rw)
    register struct seg *seg;
    register addr_t addr;
    u_int len;
    enum fault_type type;
    enum seg_rw rw;
{
    register struct segdev_data *sdp = (struct segdev_data *)seg->s_data;
    register addr_t adr;
    register u_int prot, protchk;
    u_int ppid;

    if (type == F_PROT) {
        return (FC_PROT);
    }

    if (type != F_SOFTUNLOCK) {
        switch (rw) {
        case S_READ:
            protchk = PROT_READ;
            break;
        case S_WRITE:
            protchk = PROT_WRITE;
            break;
        case S_EXEC:
            protchk = PROT_EXEC;
            break;
        }

        ppid = (u_int)(*sdp->mapfunc)(sdp->vp->v_rdev,
                sdp->offset + (adr - seg->s_base), prot);
        if (ppid == NOPAGE)
            return (FC_MAKE_ERR(EFAULT));

        hat_devload(seg, adr, ppid, prot, type == F_SOFTLOCK);
    }

    return (0);
}
```

The function invokes the device driver's mmap function to obtain the physical page ID, then calls `hat_devload()` to establish the mapping. Protection faults return immediately since seg_dev does not support copy-on-write.

Device Mapping Function

Device drivers provide mmap functions with this signature:

```
int (*mmap)(dev_t dev, off_t off, int prot);
```

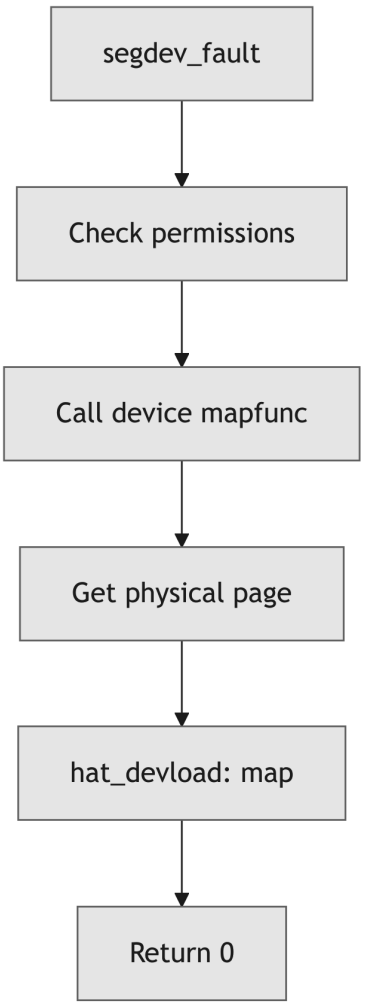
The function receives the device number, offset within the device, and requested protections. It returns the physical page frame number (pfn) or NOPAGE if the offset is invalid. This interface allows devices to validate access and translate device-relative offsets to physical addresses.

Per-Page Protections

Like seg_vn, seg_dev supports per-page protections through the vpage array. The `segdev_setprot()` function can modify protections for arbitrary ranges. When per-page protections are first enabled, the driver allocates a vpage array and initializes all entries with the segment's current protection.

Duplication for Fork

The `segdev_dup()` function creates device mappings in child processes after fork(). The child receives a new segdev_data structure but shares the same device and offset. This allows multiple processes to map the same device region while maintaining independent per-page protections.


segdev_fault
Check permissions
Call device mapfunc
Get physical page
hat_devload: map
Return 0

Swap Space Management: The Underground Reserve

Beneath the city lies a reserve of sealed vaults. The clerk on the surface does not care which tunnel leads to which vault. She cares only that every sealed slot can be found again and that no vault is overused. This is swap space in SVR4: a virtual device made from a chain of physical swap areas, managed as one ledger of anonymous slots.

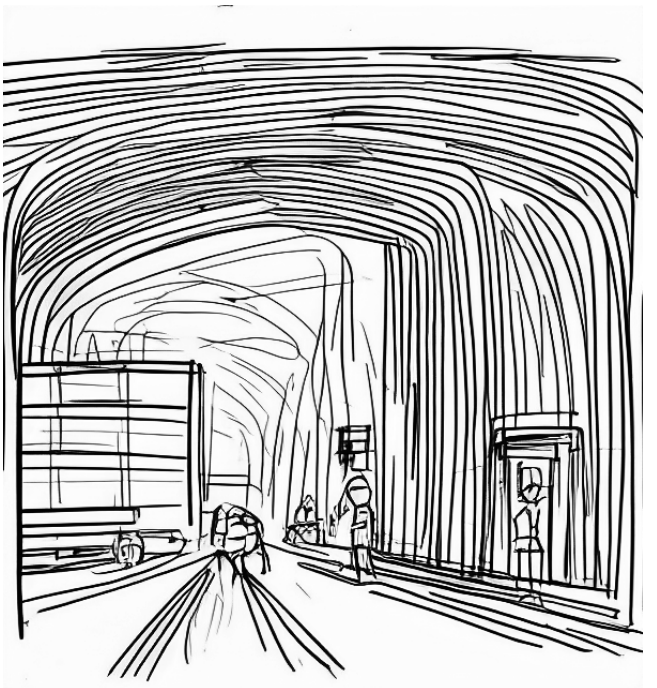
The Swap Area Ledger: `struct swapinfo`

Each swap area is tracked by a `swapinfo` entry, which holds the vnode, offset range, and the free-list head for its anon slots (sys/swap.h:133-151).

```
struct swapinfo {
    struct      vnode *si_vp;
    struct      vnode *si_svp;
    uint        si_soff;
    uint        si_eoff;
    struct      anon *si_anon;
    struct      anon *si_eanon;
    struct      anon *si_free;
    int si_allocs;
    struct      swapinfo *si_next;
    short       si_flags;
    ulong       si_npgs;
    ulong       si_nfpgs;
    char        *si_pname;
};
```

The Vault Ledger (sys/swap.h:133-151)

The `si_anon` array is the true map of vault slots. Free slots are linked through each anon's `un.an_next` pointer. The `si_allocs` counter helps spread allocations across devices, and `si_flags` tracks deletion and in-progress state.



Swap Space - Underground Vaults

Logical Concatenation and Load Balancing

SVR4 treats swap as one logical array of anon slots, even though the physical devices are separate. The allocator walks the `swapinfo` list, rotating across devices after `swap_maxcontig` consecutive allocations (vm/vm_swap.c:107-160).

```
STATIC int swap_maxcontig = 1024 * 1024 / PAGESIZE; /* 1MB of pages */

struct anon *
swap_alloc()
{
    do {
        if ((sip->si_flags & ST_INDEL) == 0) {
            ap = sip->si_free;
            if (ap) {
                sip->si_free = ap->un.an_next;
                sip->si_nfpgs--;
                if (++sip->si_allocs >= swap_maxcontig)
                    sip = sip->si_next ? sip->si_next : swapinfo;
                return (ap);
            }
            sip->si_allocs = 0;
        }
        sip = sip->si_next ? sip->si_next : swapinfo;
    } while (sip != silast);
    return (NULL);
}
```

The Interleaved Allocation (vm/vm_swap.c:107-160, abridged)

This simple rotation prevents a single swap area from becoming a hot spot. The vaults are spread across the underground grid, balancing wear and contention.

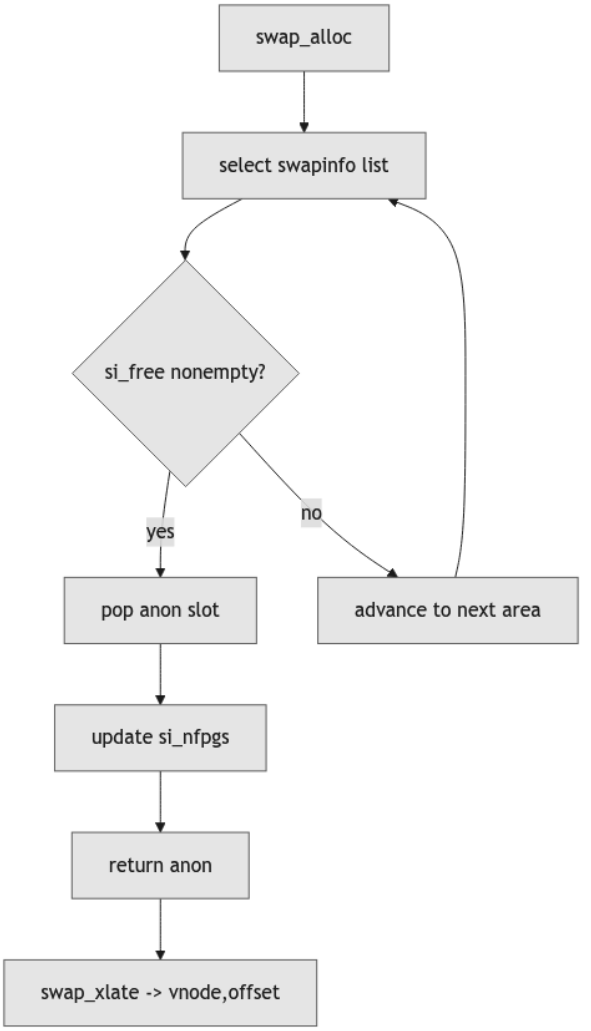


Figure 2.9.1: Swapinfo Rotation and Free Slot Selection

Translating a Slot: `swap_xlate()` and `an_bap`

An anon slot is not a disk address. It is a pointer into one of the swap area's anon arrays. `swap_xlate()` resolves that pointer into a `<vnode, offset>` pair when the VM needs to issue I/O (vm/vm_swap.c:215-244). If the slot has been moved during a swap deletion, it uses `an_bap` indirection to find the new home (vm/vm_swap.c:223-234, vm/anon.h:54-68).

```
void
swap_xlate(ap, vpp, offsetp)
    register struct anon *ap;
    register struct vnode **vpp;
    register uint *offsetp;
{
    if (ap->an_bap)
        ap = ap->an_bap;
    ...
    *offsetp = sip->si_soff + ((ap - sip->si_anon) << PAGESHIFT);
    *vpp = sip->si_vp;
}
```

The Vault Addressing (vm/vm_swap.c:215-240, abridged)

The back pointer is the keeper’s forwarding note. When an area is deleted, the old slot points to the new slot, preserving identity without rewriting every anonymous map in the system.

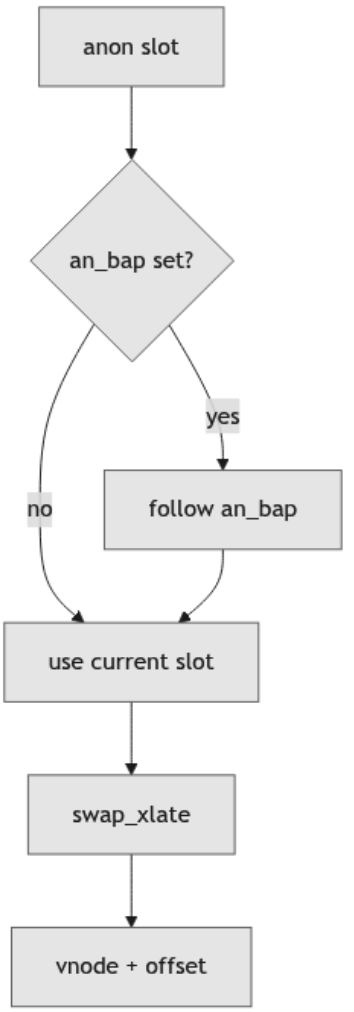


Figure 2.9.2: `an_bap` Forwarding When Areas Are Removed

Adding and Deleting Vaults

`swapadd()` opens a swap vnode, verifies the partition bounds, allocates a new `swapinfo`, and builds the anon free list by linking entries from the end back to the head (vm/vm_swap.c:565-749). The free list is initialized so the first usable entry is at `si_free`.

`swapdel()` walks the list, marks an area deleted, and relocates active anon slots by installing `an_bap` indirections before finally freeing the swapinfo structure (vm/vm_swap.c:768-868). The

vault is closed only after every active slot has a forwarding address.

The Ghost of SVR4:

We treated swap like a row of vaults and accepted that our clerk had to walk the list. Your systems now hide swap behind layered devices, compressed caches, and memory tiers. Some keep a hot in-memory shadow (zswap), others compress pages before they ever touch a disk. Yet even in 2026, the oldest rule remains: you cannot lose the address of a sealed vault, and you cannot keep every room in memory forever.

The Reserve Holds

Swap space management is not about speed. It is about endurance. The ledger of `swapinfo`, the rotating allocator, and the indirection mechanism keep the reserve usable even as devices come and go. The vaults hold, and the city keeps running.

Page Replacement and Paging: The Courier's Rounds

When a library grows too crowded, the librarians begin a quiet ritual. Volumes that have not been touched are returned to storage, and those that are still in demand are kept on the reading tables. The library does not throw books away; it moves them to where they can be fetched again. In SVR4, this ritual is the paged vnode (pvn) layer and its pageout companions.

The pvn layer sits between file systems and the VM system. It decides which pages can be clustered for I/O, how dirty pages are found, and how completed I/O is reconciled with the page ledger.

Clustering: `pvn_kluster()`

The courier's efficiency comes from carrying bundles, not single sheets. `pvn_kluster()` walks forward and backward from a faulting offset to build a contiguous run of file-backed pages, bounded by filesystem block limits and available memory (vm/vm_pvn.c:79-195).

```
page_t *
pvn_kluster(vp, off, seg, addr, offp, lenp, vp_off, vp_len, isra)
{
    if (freemem - minfree > 0)
        bytesavail = ptob(freemem - minfree);
    ...
    if (page_exists(vp, off))
        return (NULL);
    ...
    for (delta = PAGESIZE; off + delta < vp_end; delta += PAGESIZE) {
        if ((*seg->s_ops->kluster)(seg, addr, delta))
            break;
        if (page_exists(vp, off + delta))
            break;
    }
    ...
    pp = rm_allocpage(seg, straddr, (u_int)delta, P_CANWAIT);
}
```

The Bundle Selector (vm/vm_pvn.c:79-196, abridged)

It checks `freemem` against `minfree` to decide how large the bundle can be, consults the segment driver via `kluster()` to avoid extending beyond file boundaries, and finally allocates a list of pages. Each page is marked `p_intrans` and `p_pagein` to signal that the courier is still on the road.

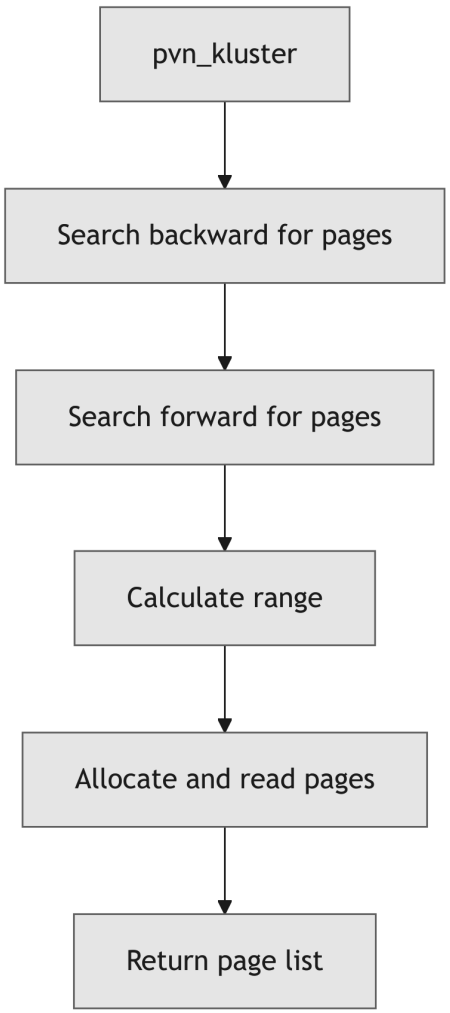


Figure 2.10.1: Clustering Around a Fault

Page Replacement - Theater Understudy

Completing the Delivery: pvn_done()

When the I/O completes, `pvn_done()` handles each page in the buffer. It clears `p_intrans`, checks for errors, releases the page, and, if the caller requested invalidation, aborts the page (vm/vm_pvn.c:274-379).

```
void
pvn_done(bp)
    register struct buf *bp;
{
    if (bp->b_flags & B_REMAPPED)
        bp_mapout(bp);
    for (bytes = 0; bytes < bp->b_bcount; bytes += PAGESIZE) {
        pp = bp->b_pages;
        pp->p_intrans = 0;
        pp->p_pagein = 0;
        PAGE_RELE(pp);
        if ((bp->b_flags & (B_ERROR|B_READ)) == (B_ERROR|B_READ))
            page_abort(pp);
    }
}
```

The I/O Reconciliation (vm/vm_pvn.c:279-354, abridged)

The page ledger is verified after each release. If the page lost its identity or was freed in the interim, `pvn_done()` skips it. Errors trigger `page_abort()`, and clean pages are left for the normal aging process.

Dirty Page Scans: `pvn_vplist_dirty()`

To write pages back, the VM needs a safe way to walk a vnode's page ring while the list is still mutating. `pvn_vplist_dirty()` inserts marker pages into the ring, then scans and calls `VOP_PUTPAGE()` for dirty entries (vm/vm_pvn.c:562-699). The markers prevent the scan from looping forever while new pages arrive.

```
page_t *
pvn_vplist_dirty(vp, off, flags)
{
    if ((spp = (page_t *)kmem_zalloc(2 * sizeof(page_t), KM_SLEEP)) ==
NULL)
        cmn_err(CE_PANIC, "pvn_vplist_dirty: cannot allocate marker
pages");
    ...
    /* Insert a start marker at the front of the v_pages list */
    spp->p_vpnext = pp;
    spp->p_vpprev = pp->p_vpprev;
    pp->p_vpprev = spp;
    ...
    if (pp->p_intrans)
        page_wait(pp);
    ...
    /* Call into the file system with VOP_PUTPAGE */
}
```

The Marker Scan (vm/vm_pvn.c:609-695, abridged)

This is the librarian's patrol. It respects `p_intrans`, waits for keep counts to drop, and only then asks the filesystem to write pages back. The pvn layer is careful to avoid deadlocks by requiring the vnode lock at a higher level.

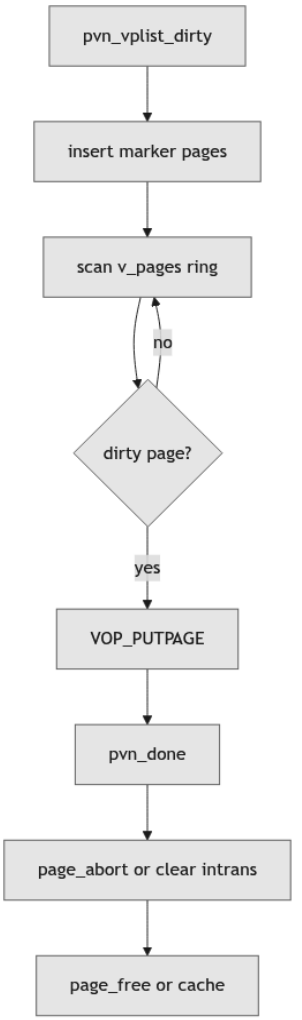


Figure 2.10.2: Marker Pages and Writeback Flow

VOP_GETPAGE and VOP_PUTPAGE: The Contract

File systems implement the `VOP_GETPAGE` and `VOP_PUTPAGE` operations using the pvn layer. `VOP_GETPAGE` uses `pvn_kluster()` to determine a read window and then issues I/O. `VOP_PUTPAGE` relies on `pvn_vplist_dirty()` to drive writeback. This contract keeps the file system focused on blocks and metadata, while pvn handles the page choreography.

The Ghost of SVR4:

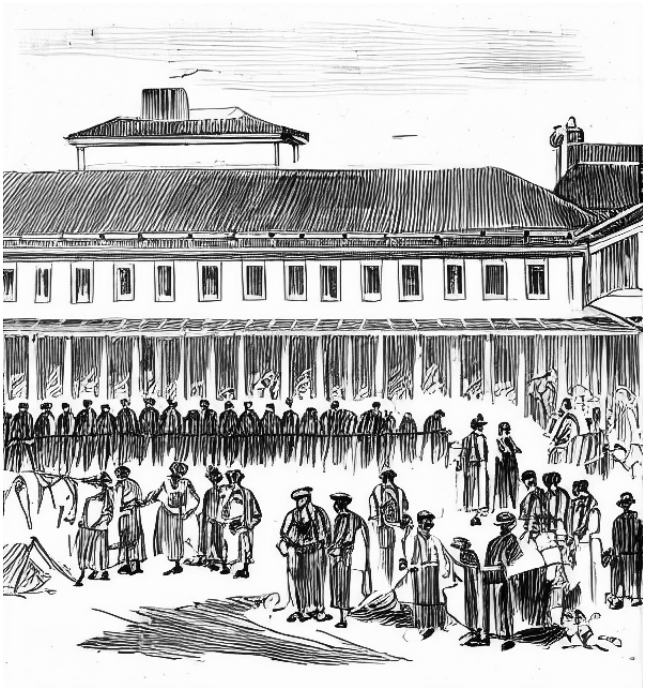
We had one courier and a paper ledger. Today you have fleets: readahead windows shaped by machine learning, writeback throttling for SSDs, and background reclaim threads aware of memory cgroups. Yet the old bargain still stands. A file system cannot hide its pages, and the VM cannot write them without the file system’s consent. The pvn layer is the treaty that keeps these offices in polite cooperation.

The Rounds Continue

Page replacement is not a dramatic purge; it is a continuous set of rounds. The pvn layer clusters, writes, and reconciles, while the pageout daemon nudges the system toward balance. The librarian’s patrol never ends, and the reading rooms stay open.

Virtual File System (VFS) Layer

Virtual File System (VFS) Layer



VFS Layer - Customs House

Overview

The VFS layer provides a uniform interface to multiple file system types. It allows the kernel to support different file systems simultaneously through a common set of operations. Each mounted file system has a vfs structure containing function pointers for filesystem-specific operations.

VFS Structure

The vfs structure (vfs.h:46) represents a mounted file system:

```
typedef struct vfs {
    struct vfs *vfs_next;           /* next VFS in VFS list */
    struct vfsops *vfs_op;          /* operations on VFS */
    struct vnode *vfs_vnodecovered; /* vnode mounted on */
    u_long vfs_flag;                /* flags */
    u_long vfs_bsize;               /* native block size */
    int vfs_fstype;                 /* file system type index */
    fsid_t vfs_fsid;                /* file system id */
    caddr_t vfs_data;               /* private data */
    dev_t vfs_dev;                  /* device of mounted VFS */
    u_long vfs_bcount;              /* I/O count (accounting) */
    u_short vfs_nsubmounts;         /* immediate sub-mount count */
} vfs_t;
```

The `vfs_next` field links all mounted file systems into a global list headed by `rootvfs`. The `vfs_vnodecovered` points to the mount point vnode. The `vfs_data` field holds filesystem-specific private data.

VFS Operations Vector

The vfsops structure (vfs.h:87) defines file system operations:

```
typedef struct vfsops {
    int (*vfs_mount)();         /* mount file system */
    int (*vfs_unmount)();       /* unmount file system */
    int (*vfs_root)();          /* get root vnode */
    int (*vfs_statvfs)();       /* get file system statistics */
    int (*vfs_sync)();          /* flush fs buffers */
    int (*vfs_vget)();          /* get vnode from fid */
    int (*vfs_mountroot)();     /* mount the root filesystem */
    int (*vfs_swapvp)();        /* return vnode for swap */
    int (*vfs_filler[8])();     /* for future expansion */
} vfsops_t;
```

Each operation is invoked through macros like `VFS_MOUNT(vfsp, mvp, uap, cr)` which indirect through the vfs_op pointer. This allows different file system types to provide custom implementations.

File System Type Switch

The vfssw structure (vfs.h:116) maps file system names to operations:

```
typedef struct vfssw {
    char *vsw_name;             /* type name string */
    int (*vsw_init)();          /* init routine */
    struct vfsops *vsw_vfsops;  /* filesystem operations vector */
    long vsw_flag;              /* flags */
} vfssw_t;
```

The global `vfssw[]` array contains entries for each configured file system type (UFS, S5FS, NFS, etc.). The `vfs_getvfssw()` function looks up entries by name.

Mount Operation

The mount() system call (vfs.c:78) attaches a file system to the directory tree:

```
int
mount(uap, rvp)
    register struct mounta *uap;
    rval_t *rvp;
{
    vnode_t *vp = NULL;
    register struct vfs *vfsp;
    struct vfssw *vswp;
    struct vfsops *vfsops;
    register int error;

    /* Resolve mount point */
    if (error = lookupname(uap->dir, UIO_USERSPACE, FOLLOW, NULLVPP, &vp))
        return error;
    if (vp->v_vfsmountedhere != NULL) {
        VN_RELE(vp);
        return EBUSY;
    }
    if (vp->v_flag & VNOMOUNT) {
        VN_RELE(vp);
        return EINVAL;
    }
```

The function looks up the mount point vnode, verifies nothing is already mounted there, then locates the appropriate vfsops based on the file system type name or number.

VFS Flags

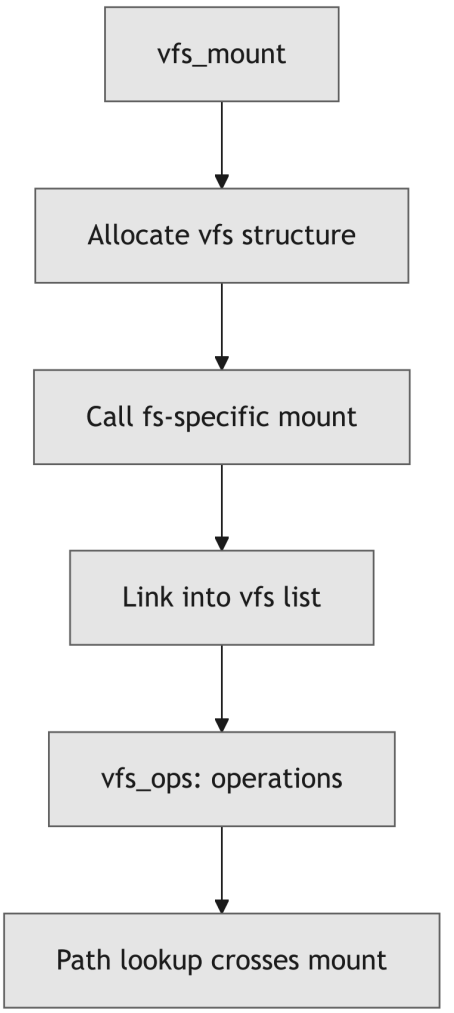
VFS flags (vfs.h:63) control mount behavior:

```
#define VFS_RDONLY    0x01    /* read-only vfs */
#define VFS_MLOCK     0x02    /* lock vfs so subtree is stable */
#define VFS_MWAIT     0x04    /* someone is waiting for lock */
#define VFS_NOSUID    0x08    /* setuid disallowed */
#define VFS_REMOUNT   0x10    /* modify mount options only */
#define VFS_NOTRUNC   0x20    /* does not truncate long file names */
#define VFS_UNLINKABLE 0x40   /* unlink can be applied to root */
```

These flags affect operations like permission checking, name handling, and locking behavior for the mounted file system.

VFS List Management

The `vfs_add()` function links a new vfs into the global list, while `vfs_remove()` unlinks it during unmount. The `vfs_lock()` and `vfs_unlock()` functions provide locking to stabilize the VFS tree during operations that traverse mount points.


vfs_mount
Allocate vfs structure
Call fs-specific mount
Link into vfs list
vfs_ops: operations
Path lookup crosses mount

Vnode Interface

VNode Interface

Overview

The vnode (virtual node) represents an active file in the system. Vnodes abstract file system implementation details, providing a uniform interface for file operations regardless of the underlying file system type. Each vnode contains function pointers for file-specific operations.

Vnode Structure

The vnode structure (vnode.h:62) represents a file object:

```
typedef struct vnode {
    u_short v_flag;                 /* vnode flags */
    u_short v_count;                /* reference count */
    struct vfs *v_vfsmountedhere;   /* ptr to vfs mounted here */
    struct vnodeops *v_op;          /* vnode operations */
    struct vfs *v_vfsp;             /* ptr to containing VFS */
    struct stdata *v_stream;        /* associated stream */
    struct page *v_pages;           /* vnode pages list */
    enum vtype v_type;              /* vnode type */
    dev_t v_rdev;                   /* device (VCHR, VBLK) */
    caddr_t v_data;                 /* private data for fs */
    struct filock *v_filocks;       /* ptr to filock list */
} vnode_t;
```

The `v_count` field tracks active references. The `v_vfsp` points to the containing file system. The `v_data` field holds filesystem-specific inode data. The `v_pages` field links pages cached for this file.

Vnode Types

The vtype enumeration (vnode.h:50) classifies vnodes:

```
typedef enum vtype {
    VNON  = 0,    /* no type */
    VREG  = 1,    /* regular file */
    VDIR  = 2,    /* directory */
    VBLK  = 3,    /* block device */
    VCHR  = 4,    /* character device */
    VLNK  = 5,    /* symbolic link */
    VFIFO = 6,    /* FIFO */
    VXNAM = 7,    /* XENIX named file */
    VBAD  = 8     /* bad vnode */
} vtype_t;
```

The type determines which operations are valid and affects permission checking and system call behavior.

Vnode Operations

The vnodeops structure (vnode.h:93) defines file operations:

```
typedef struct vnodeops {
    int (*vop_open)();
    int (*vop_close)();
    int (*vop_read)();
    int (*vop_write)();
    int (*vop_ioctl)();
    int (*vop_setfl)();
    int (*vop_getattr)();
    int (*vop_setattr)();
    int (*vop_access)();
    int (*vop_lookup)();
    int (*vop_create)();
    int (*vop_remove)();
    int (*vop_link)();
    int (*vop_rename)();
    int (*vop_mkdir)();
    int (*vop_rmdir)();
    int (*vop_readdir)();
    int (*vop_symlink)();
    int (*vop_readlink)();
    int (*vop_fsync)();
    void (*vop_inactive)();
    int (*vop_fid)();
    void (*vop_rwlock)();
    void (*vop_rwunlock)();
    int (*vop_seek)();
    int (*vop_cmp)();
    int (*vop_frlock)();
    int (*vop_space)();
    int (*vop_realvp)();
    int (*vop_getpage)();
    int (*vop_putpage)();
    int (*vop_map)();
    int (*vop_addmap)();
    int (*vop_delmap)();
    int (*vop_poll)();
    int (*vop_dump)();
    int (*vop_pathconf)();
    int (*vop_allocstore)();
} vnodeops_t;
```

Operations are invoked through macros like `VOP_READ(vp, uiop, iof, cr)` which indirect through the v_op pointer.

Key Operations

VOP_LOOKUP: Searches a directory for a named entry, returning the vnode for the found file. This is the foundation of pathname resolution.

VOP_GETPAGE/VOP_PUTPAGE: Handle page faults and pageout for memory-mapped files. These operations coordinate with the VM system to provide file-backed memory.

VOP_RWLOCK/VOP_RWUNLOCK: Provide reader/writer locking for the vnode to serialize concurrent access during operations like read, write, and truncate.

VOP_INACTIVE: Called when the reference count reaches zero. The file system can release resources, though the vnode itself may be cached.

Vnode Flags

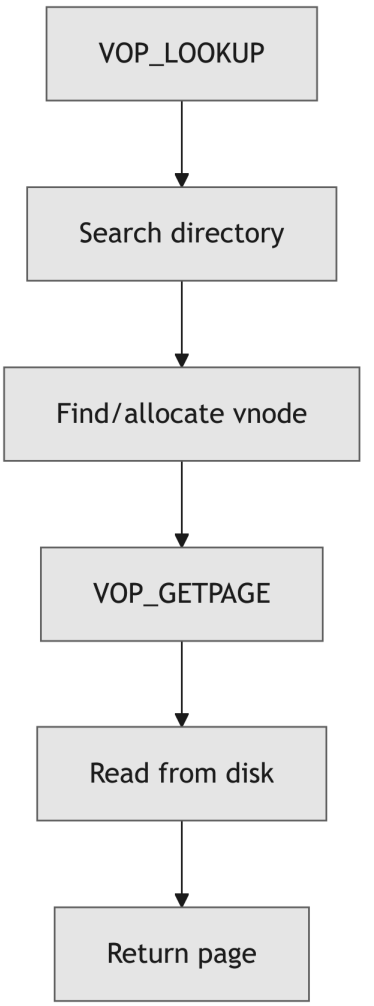
Vnode flags (vnode.h:80) control behavior:

```
#define VROOT     0x01    /* root of its file system */
#define VNOMAP    0x04    /* file cannot be mapped/faulted */
#define VDUP      0x08    /* file should be dup'ed rather than opened */
#define VNOMOUNT  0x20    /* file cannot be covered by mount */
#define VNOSWAP   0x10    /* file cannot be used as virtual swap device */
#define VISSWAP   0x40    /* vnode is part of virtual swap device */
```

The VROOT flag marks file system root vnodes. VNOMAP prevents memory mapping for special files.

Reference Counting

The `VN_HOLD(vp)` macro increments v_count, while `VN_RELE(vp)` decrements it. When the count reaches zero, `VOP_INACTIVE()` is invoked. This allows the system to cache inactive vnodes while ensuring cleanup when no longer needed.


VOP_LOOKUP
Search directory
Find/allocate vnode
VOP_GETPAGE
Read from disk
Return page

Pathname Resolution

Pathname Resolution

Pathname Resolution - Postal Carrier

Overview

Pathname resolution translates textual path names into vnodes. The process handles multiple pathname components, mount points, symbolic links, and permission checking. The central function `lookuppn()` iterates through path components, invoking `VOP_LOOKUP()` on each directory.

Pathname Structure

The pathname structure (pathname.h) maintains state during lookup:

```
struct pathname {
    char *pn_buf;        /* underlying storage */
    char *pn_path;       /* remaining pathname */
    uint pn_pathlen;     /* remaining length */
};
```

The `pn_path` pointer advances through the path as components are consumed. The `pn_get()` function copies the path from user space, while `pn_free()` releases the buffer.

Lookupname Function

The `lookupname()` function (lookup.c:56) provides the high-level interface:

```
int
lookupname(fnamep, seg, followlink, dirvpp, compvpp)
    char *fnamep;                /* user pathname */
    enum uio_seg seg;            /* addr space that name is in */
    enum symfollow followlink;   /* follow sym links */
    vnode_t **dirvpp;            /* ret for ptr to parent dir vnode */
    vnode_t **compvpp;           /* ret for ptr to component vnode */
{
    struct pathname lookpn;
    register int error;

    if (error = pn_get(fnamep, seg, &lookpn))
        return error;
    error = lookuppn(&lookpn, followlink, dirvpp, compvpp);
    pn_free(&lookpn);
    return error;
}
```

The function allocates a pathname buffer, calls `lookuppn()` to perform the lookup, then frees the buffer. It returns both the target vnode and its parent directory.

Lookuppn Function

The `lookuppn()` function (lookup.c:84) implements the core algorithm:

```
int
lookuppn(pnp, followlink, dirvpp, compvpp)
    register struct pathname *pnp;
    enum symfollow followlink;
    vnode_t **dirvpp;
    vnode_t **compvpp;
{
    register vnode_t *vp;    /* current directory vp */
    register vnode_t *cvp;   /* current component vp */
    vnode_t *tvp;
    char component[MAXNAMELEN];
    register int error;
    register int nlink;

    sysinfo.namei++;
    nlink = 0;
    cvp = NULL;
```

The function iterates through path components, calling `VOP_LOOKUP()` on each directory vnode. It handles special cases for "." and "..", checks permissions, and follows mount points.

Component Iteration

The lookup loop extracts one component at a time:

1. Call `pn_getcomponent()` to extract the next component name
2. Check for "." (current directory) and ".." (parent directory)
3. Call `VOP_LOOKUP()` on the current directory vnode
4. Check if the result is a mount point and follow it if needed
5. Check if the result is a symbolic link and follow it if requested
6. Make the result vnode the new current directory
7. Repeat until the path is exhausted

Mount Point Traversal

When a lookup returns a vnode with `v_vfsmountedhere` set, the system crosses into the mounted file system:

```
if (vp->v_vfsmountedhere != NULL) {
    VFS_ROOT(vp->v_vfsmountedhere, &tvp);
    VN_RELE(vp);
    vp = tvp;
}
```

The `VFS_ROOT()` operation obtains the root vnode of the mounted file system, replacing the mount point vnode.

Symbolic Link Handling

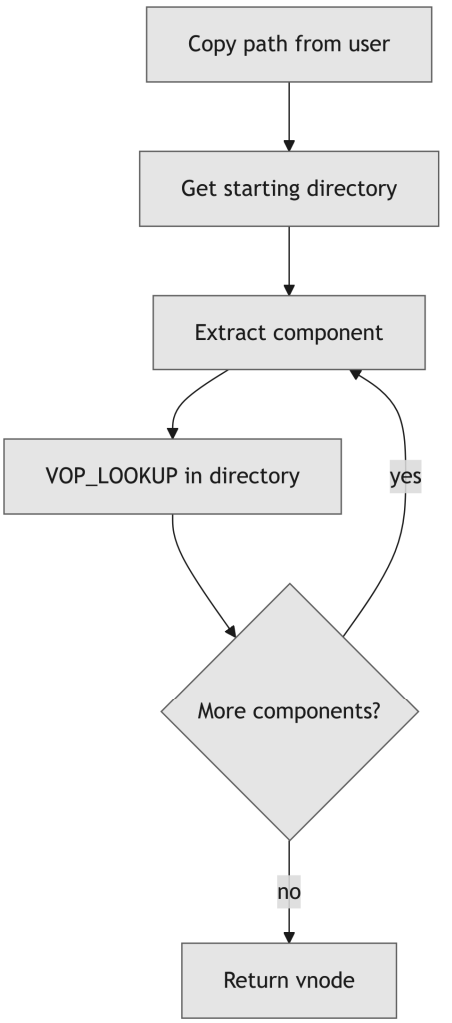
When a symbolic link is encountered and `followlink` is set, the system reads the link target and restarts the lookup from that path. A `nlink` counter prevents infinite loops through circular symbolic links.

Permission Checking

At each step, `VOP_ACCESS()` verifies that the process has execute permission on the directory being searched. This ensures that users cannot traverse directories they lack permission to access, even if they know the names of files within.

Directory Name Lookup Cache

To optimize repeated lookups, the DNLC caches successful directory lookups. Before calling `VOP_LOOKUP()`, the system checks the cache. Cache hits avoid expensive directory searches and disk I/O.


Copy path from user
Get starting directory
Extract component
VOP_LOOKUP in directory
yes
More components?
no
Return vnode

Directory Name Lookup Cache: The Librarian's Pocket Notebook

Picture, if you will, a grand old-world library housing ten thousand volumes across a labyrinth of oak shelves and winding corridors. A patron approaches the circulation desk and requests, with utmost civility, "Volume XIV of the *Proceedings*, Section III, if you please." The traditional approach demands a ceremony: the librarian consults the enormous leather-bound card catalog, pulls the appropriate drawer with a satisfying *thunk*, flips through dozens of index cards with practiced fingers, notes the shelf location on a slip of paper, navigates the marble-floored corridors to the designated alcove, and finally retrieves the requested tome.

This ritual, performed thousands of times each day, wears not only paths in the marble floor but also precious moments from the patron's afternoon. Yet the head librarian, a woman of keen observation and sharper memory, notices a pattern: certain volumes are requested repeatedly. The *Proceedings* Volume XIV, perhaps, or the *Atlas of Parliamentary Districts*, or the *Compendium of Agricultural Statistics*. These popular tomes account for the vast majority of requests, while thousands of other volumes gather dust, consulted once per decade if at all.

The solution? A **pocket notebook**—a small, leather-bound ledger tucked into her waistcoat. When Volume XIV is requested, she jots down: "*Proceedings XIV-III → Shelf 47B, Alcove West*". When the next patron, not five minutes later, requests the very same volume, she need not consult the grand catalog at all. A quick glance at her pocket notebook reveals the answer swiftly. This is the essence of the **Directory Name Lookup Cache (DNLC)**: a compact, rapidly-consulted ledger of recent lookups, sparing the kernel the expense of traversing directory blocks for frequently-accessed pathnames.

The Kernel's Pocket Notebook: DNLC Architecture

In the SVR4 kernel, pathname resolution—the act of translating a string like `/usr/bin/sh` into the vnode representing the shell executable—is a repetitive, expensive operation. Consider the pattern: a typical workstation might resolve `/usr/bin/sh` hundreds of times per hour as users

spawn shells, scripts invoke utilities, and programs execute child processes. Each resolution, without caching, demands:

1. **Directory Scan**: Read the directory's data blocks from disk (or buffer cache)
2. **Linear Search**: Compare the target name against each directory entry
3. **Vnode Allocation**: Allocate and initialize the result vnode
4. **Attribute Fetch**: Load inode metadata from disk

This is the *card catalog walk* made manifest in silicon and spinning platters. The DNLC, residing entirely in DRAM, short-circuits this process by caching the most recent `(parent_directory_vnode, filename) → child_vnode` mappings. When a lookup succeeds in the cache—a "hit"—the kernel avoids the directory scan entirely, retrieving the answer with the speed of a memory reference rather than the latency of disk I/O.

DNLC - Librarian's Pocket Notebook

The Ledger Entry: `struct ncache`

Each entry in the DNLC is a `struct ncache` , defined in `sys/dnlc.h` :

```
struct ncache {
    struct ncache *hash_next;    /* hash chain, MUST BE FIRST */
    struct ncache *hash_prev;
    struct ncache *lru_next;     /* LRU chain */
    struct ncache *lru_prev;
    struct vnode *vp;            /* vnode the name refers to */
    struct vnode *dp;            /* vnode of parent directory */
    char namlen;                 /* length of name */
    char name[NC_NAMLEN];        /* segment name (max 15 chars) */
    struct cred *cred;           /* credentials */
};
```

The Ledger Entry Structure (sys/dnlc.h:24)

The structure is elegantly minimal:

- `dp` **(directory parent vnode)**: The vnode of the containing directory (e.g., `/usr/bin`)
- `name[NC_NAMLEN]` : The filename itself (e.g., `"sh"`), limited to 15 characters
- `vp` **(result vnode)**: The vnode of the target file (the shell executable)
- `cred` : The credentials used during the original lookup (for permission consistency)
- **Hash and LRU pointers**: Dual-linked-list membership for fast lookup and aging

The 15-character limit (`NC_NAMLEN`) is a pragmatic choice: longer names are rare and would consume excessive cache memory. When encountering a name exceeding this limit, the kernel simply bypasses the cache, performing the full directory scan. This is acceptable; such names are infrequent, and the cache optimizes for the common case, not the pathological edge.

The Dual Index: Hash Table and LRU List

The DNLC employs a **dual organization** reminiscent of a Victorian office combining an alphabetized filing cabinet (for speed) with a chronological archive (for aging):

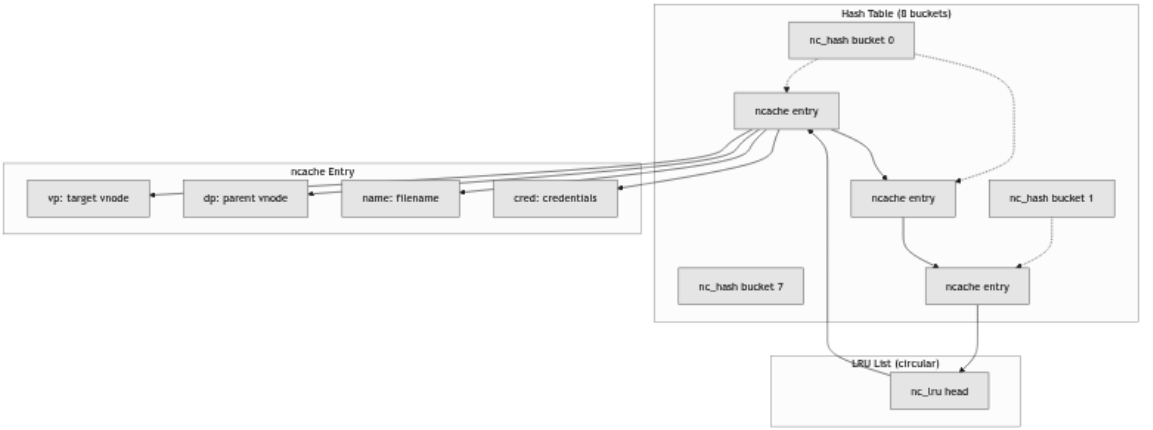


Figure 3.4.1: The Dual Organization—Hash Table and LRU List

Hash Table: The Alphabetized Cabinet

The cache entries are indexed by a simple hash function (fs/dnlc.c:64):

```
#define NC_HASH_SIZE    8    /* size of hash table */

#define NC_HASH(namep, namelen, vp)  \
    ((namep[0] + namep[namelen-1] + namelen + (int) vp) & (NC_HASH_SIZE-
1))
```

The Hashing Incantation (fs/dnlc.c:62-65)

The hash combines:

- **First character** of the name: `namep[0]`
- **Last character**: `namep[namelen-1]`
- **Name length**: `namelen`
- **Parent vnode pointer**: `(int) vp`

This yields an index into `nc_hash[]`, an array of 8 hash buckets. The hash is intentionally simple—computed in a handful of CPU cycles—trading perfect distribution for speed. Collisions are resolved via chaining: each bucket is the head of a doubly-linked list of `ncache` entries. With a cache sized at 64–256 entries, an 8-bucket table means average chains in the single digits to a few dozen; not instant, but still far cheaper than a directory I/O.

> **Why such a small hash table?** In 1990, DRAM was measured in megabytes, not gigabytes. The DNLC, sized at perhaps 64-256 entries (`ncsize`), represented a significant memory investment. A hash table of 8 or 16 buckets provided adequate distribution without wasting precious address space on empty buckets. Modern systems, with gigabytes of RAM, can afford thousands of hash buckets and correspondingly enormous caches.

LRU List: The Chronological Archive

Simultaneously, all `ncache` entries are threaded onto a **Least Recently Used (LRU)** list. The LRU is a doubly-linked circular list, with the global `nc_lru` structure serving as the list head. When a cached entry is accessed (a hit), it is moved to the tail of the LRU list, marking it as "recently used." When a new entry must be added and the cache is full, the entry at the head of the LRU list—the *least* recently used—is evicted, its vnodes released, and its slot reused.

This is the librarian's practice of periodically reviewing her pocket notebook and erasing entries for volumes not requested in months, making room for fresh annotations.

The Ritual of Consultation: `dnlc_lookup()`

When the pathname resolution code (in `lookuppn()`) needs to find a directory entry, it first consults the DNLC via `dnlc_lookup()` (fs/dnlc.c:232):

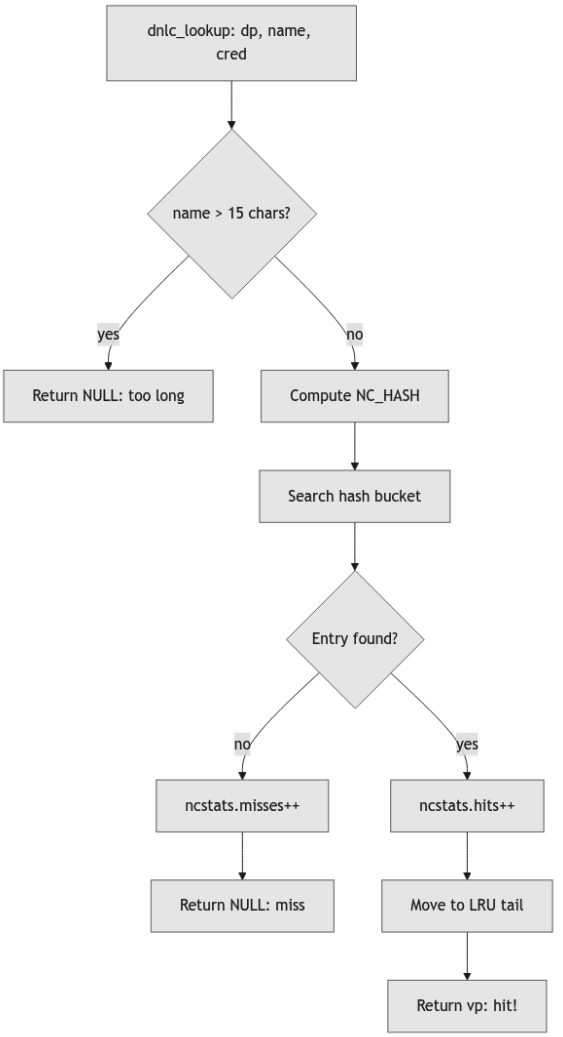


Figure 3.4.2: The Lookup Ritual—Consulting the Pocket Notebook

```
vnode_t *
dnlc_lookup(dp, name, cred)
    vnode_t *dp;
    char *name;
    cred_t *cred;
{
    register int namlen;
    register int hash;
    register struct ncache *ncp;

    if (!doingcache)
        return NULL;
    if ((namlen = strlen(name)) > NC_NAMLEN) {
        ncstats.long_look++;
        return NULL;    /* name too long to cache */
    }
    hash = NC_HASH(name, namlen, dp);
    if ((ncp = dnlc_search(dp, name, namlen, hash, cred)) == NULL) {
        ncstats.misses++;
        return NULL;    /* cache miss */
    }
    ncstats.hits++;
    /* Move to end of LRU list (mark as recently used) */
    RM_LRU(ncp);
    INS_LRU(ncp, nc_lru.lru_prev);
    return ncp->vp;    /* cache hit! */
}
```

The Lookup Ritual (fs/dnlc.c:232-260, simplified)

The process:

1. **Reject overlength names**: If `namlen > NC_NAMLEN`, bypass the cache
2. **Compute hash**: `NC_HASH(name, namlen, dp)`
3. **Search hash bucket**: Walk the chain, comparing `(dp, name, cred)` tuples
4. **On hit**: Move entry to LRU tail (mark as recently used), return `vp`
5. **On miss**: Increment `ncstats.misses`, return `NULL` (caller performs full lookup)

The beauty is in the negative case: a cache miss costs only a hash computation and a short list traversal. The positive case avoids an entire directory I/O operation, saving thousands of cycles and potential disk latency.

Recording New Wisdom: `dnlc_enter()`

When pathname resolution performs a *full* directory scan and successfully locates a file, it records the result in the DNLC for future reference via `dnlc_enter()` (fs/dnlc.c:164):

```
void
dnlc_enter(dp, name, vp, cred)
    register vnode_t *dp;
    register char *name;
    vnode_t *vp;
    cred_t *cred;
{
    register unsigned int namlen;
    register struct ncache *ncp;
    register int hash;

    if (!doingcache)
        return;
    if ((namlen = strlen(name)) > NC_NAMLEN) {
        ncstats.long_enter++;
        return;    /* name too long to cache */
    }
    hash = NC_HASH(name, namlen, dp);
    if (dnlc_search(dp, name, namlen, hash, cred) != NULL) {
        ncstats.dbl_enters++;
        return;    /* already cached */
    }
    /* Take least recently used cache struct */
    ncp = nc_lru.lru_next;
    /* Remove from LRU and hash chains */
    RM_LRU(ncp);
    RM_HASH(ncp);
    /* Release old vnodes if any */
    if (ncp->dp != NULL) VN_RELE(ncp->dp);
    if (ncp->vp != NULL) VN_RELE(ncp->vp);
    if (ncp->cred != NULL) crfree(ncp->cred);
    /* Hold the new vnodes and fill in cache info */
    ncp->dp = dp; VN_HOLD(dp);
    ncp->vp = vp; VN_HOLD(vp);
    ncp->namlen = namlen;
    bcopy(name, ncp->name, (unsigned)namlen);
    ncp->cred = cred;
    if (cred) crhold(cred);
    /* Insert in LRU and hash chains */
    INS_LRU(ncp, nc_lru.lru_prev);
    INS_HASH(ncp, (struct ncache *)&nc_hash[hash]);
    ncstats.enters++;
}
```

The Inscription Ceremony (fs/dnlc.c:164-226, simplified)

The steps:

1. **Validate name length**: Skip if `namlen > NC_NAMLEN`
2. **Check for duplicate**: If already cached, do nothing
3. **Evict LRU entry**: Take head of LRU list (oldest entry)
4. **Release old resources**: `VN_RELE()` the old vnodes, `crfree()` the old credentials
5. **Hold new resources**: `VN_HOLD()` the new vnodes, `crhold()` credentials (increment reference counts)
6. **Fill entry**: Copy `name`, set `dp`, `vp`, `cred`
7. **Reinsert**: Add to LRU tail (most recently used) and hash bucket

The vnode `VN_HOLD()` calls are critical: the DNLC maintains references to vnodes, preventing them from being prematurely freed. When a cache entry is evicted, the corresponding `VN_RELE()` decrements the reference count, potentially freeing the vnode if no other references exist.

Maintaining Coherency: Cache Invalidation

The DNLC's Achilles' heel is **cache coherency**. If a file is deleted or renamed, the cache entry becomes stale, pointing to a vnode that no longer corresponds to the pathname. SVR4 addresses this through *purge* operations:

- `dnlc_purge_vp(vnode_t *vp)`: Remove all entries referencing this vnode (called when a vnode is reused or a file is deleted)
- `dnlc_remove(vnode_t *dp, char *name)`: Remove a specific `(dp, name)` entry (called when a directory entry is removed)
- `dnlc_purge()`: Flush the entire cache (used during emergency cleanup or unmount)

These functions walk the hash buckets and LRU list, removing matching entries and releasing their vnode holds. The cost is linear in the number of cache entries, but infrequent: deletions and renames are rare compared to lookups.

Statistics: Measuring Effectiveness

The kernel tracks cache performance in `struct ncstats` (sys/dnlc.h:39):

```
struct ncstats {
    int hits;          /* cache hits */
    int misses;        /* cache misses */
    int enters;        /* entries added */
    int dbl_enters;    /* duplicate enter attempts */
    int long_enter;    /* names too long to cache */
    int long_look;     /* lookup of overly long names */
    int lru_empty;     /* LRU list empty (cache disabled) */
    int purges;        /* cache flushes */
};
```

These statistics, exposed via `/dev/kmem` or `crash` utilities, allow administrators to assess cache effectiveness. A high hit rate (>90%) indicates the cache is well-sized and locality is strong. A low hit rate suggests either insufficient cache size (`ncsize` tunable) or workloads with poor locality (e.g., random file access).

The Ghost of SVR4: The Humble Pocket Notebook

The DNLC, with its 8-bucket hash table and LRU eviction, was a marvel of economy in 1990. Caching perhaps 64-256 entries in a few kilobytes of DRAM, it captured the majority of pathname lookups for typical workloads—compilers, shells, editors—all repeatedly accessing `/usr/include`, `/bin`, `/tmp`. The 15-character name limit was a pragmatic compromise: most Unix utilities (`ls`, `cat`, `grep`, `sed`, `awk`) fit comfortably, and the rare overly-verbose filename (`a_very_long_descriptive_name_for_a_script.sh`) simply bypassed the cache. The design respected the iron law of 1990: *memory is precious; optimize for the common case*.

Modern Contrast (2026): Linux's `dcache` (directory entry cache) is the spiritual descendant of the DNLC but operates at a vastly different scale. Modern kernels cache **millions** of dentries (directory entries) in gigabytes of RAM, using hash tables with thousands of buckets. The dcache employs **RCU (Read-Copy-Update)** for lockless lookups, allowing CPUs to traverse the cache without acquiring spinlocks, a necessity for modern multi-core systems with dozens of simultaneous pathname resolutions. The dcache also caches **negative** entries (lookups that fail), preventing repeated scans for nonexistent files—a pattern common in software builds testing for optional libraries (`./configure` scripts). The name length limit has been lifted to 255 characters (the ext4 maximum), and the cache integrates tightly with the unified page cache, prefetching inodes alongside directory blocks. Yet the core principle remains: a small, fast, in-memory ledger of recent translations, sparing the filesystem from redundant I/O. The DNLC's pocket notebook has become a vast indexed encyclopedia, but the librarian's wisdom endures.

Conclusion: The Wisdom of the Ledger

The Directory Name Lookup Cache embodies a timeless principle of systems design: **cache the frequent, ignore the rare**. By maintaining a compact, rapidly-consulted index of recent pathname→vnode translations, the DNLC transforms what would be thousands of disk I/Os per second into memory references, improving system responsiveness by orders of magnitude. The dual hash/LRU organization balances lookup speed with aging policy, while the 15-character limit respects memory constraints without crippling functionality.

In the grand orchestration of the SVR4 VFS layer, the DNLC is the librarian's pocket notebook—a humble tool, yet indispensable, ensuring that the kernel's repeated consultations of the directory catalog do not grind the system to a halt. It is not flashy, not complex, but profoundly effective: the hallmark of good engineering.

Unix File System (UFS)

Overview: The Grand Library and its Index

The Unix File System (UFS) is the Grand Library of the SVR4 kernel, a meticulously organized repository for the Empire's vast collection of data. In this library, files are the books, and the directories are the various reading rooms and sections where these books are organized. The librarians are the UFS kernel code, tasked with managing this collection, locating books upon request, and finding space on the shelves for new acquisitions.

At the heart of this library is the card catalog, a vast and detailed index known as the inode table. Each book in the library has a corresponding card in this catalog, and it is this card, not the book's title or its position on a shelf, that is its ultimate identity. This is the fundamental principle of UFS: files are inodes, and the directory hierarchy is merely a convenient and user-friendly way of giving these inodes names.

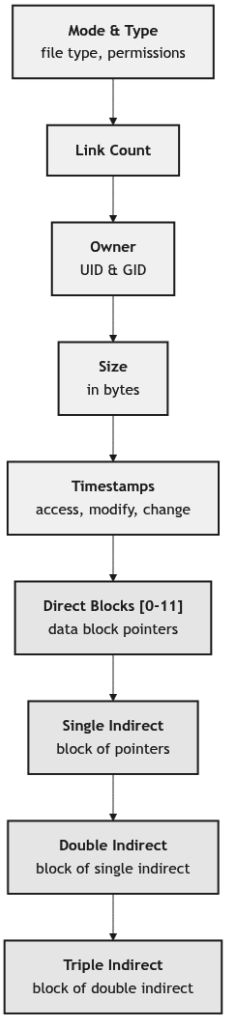
The Superblock: The Librarian's Master Ledger

Every UFS filesystem is governed by its superblock, a master ledger located at a fixed position on the disk. This is the head librarian's desk, the central point of control and information for the entire library. The `fs` structure holds this critical information.

A copy of the superblock is replicated in each cylinder group, a crucial redundancy that allows the librarians to rebuild the library's master plan in the event that the primary copy is lost or damaged.

Inodes: The Card Catalog

The inode is the fundamental data structure in UFS, the card in the catalog that describes a file. Each inode contains all the metadata about a file, with the notable exception of its name, which is stored in a directory file.



The UFS Inode Structure

The key information on each card includes:

- **Mode and Type**: The type of the file (regular file, directory, symbolic link, etc.) and its access permissions.
- **Link Count**: The number of directory entries (names) that refer to this inode. When this count drops to zero, the file is deleted.
- **Owner and Group**: The user and group IDs of the file's owner.
- **Size**: The size of the file in bytes.
- **Timestamps**: The last access, modification, and inode change times.
- **Block Pointers**: A set of pointers to the data blocks that hold the file's contents. These are divided into direct blocks, single indirect blocks, double indirect blocks, and triple indirect

blocks, a scheme that allows for both fast access to small files and the ability to address very large files.

The `ufs_iget` function in `ufs_inode.c` is the librarian responsible for finding a specific card in the catalog, reading an inode from disk into memory.

Cylinder Groups: The Library's Wings

A key innovation of the Berkeley Fast File System (FFS), from which UFS is derived, is the concept of cylinder groups. A large disk is divided into a number of these groups, each of which is a miniature filesystem in its own right, with its own set of inodes, data blocks, and a copy of the superblock.

This design is akin to organizing a vast library into a series of smaller, self-contained wings. Each wing has its own shelves (data blocks) and its own section of the card catalog (inodes). The goal is to keep the books and the catalog cards that describe them in the same wing, minimizing the distance the librarians have to travel. In disk terms, this translates to keeping a file's data blocks and its inode in the same cylinder group, reducing the seek time of the disk head and dramatically improving performance.

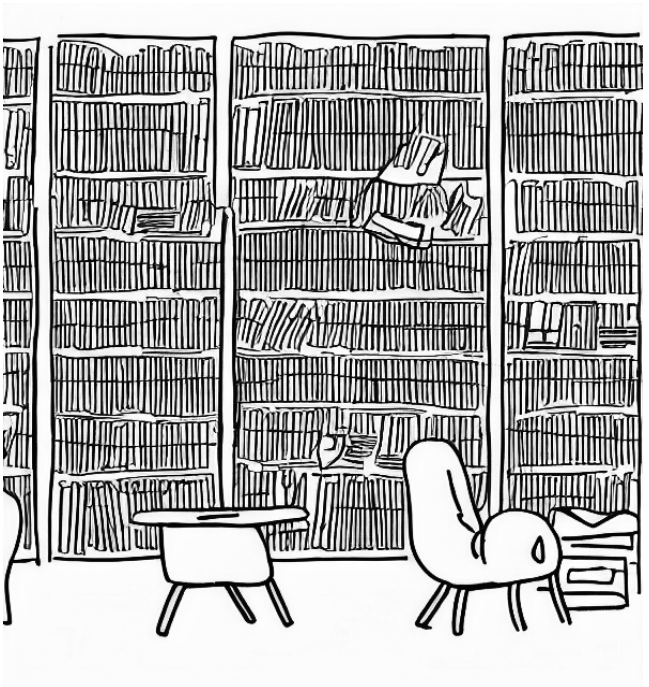
Block Allocation: Reshelving the Books

When a file grows, the UFS librarians must find new shelf space for it. The `ufs_alloc` function in `ufs_alloc.c` is responsible for this task. It uses the summary information in the cylinder group structures to make intelligent decisions about where to place new blocks.

The allocation policy attempts to keep all the blocks of a file in the same cylinder group. If this is not possible, it will choose a new cylinder group, preferably one with a higher-than-average number of free blocks, to start a new "cluster" of blocks for the file. This strategy of localization is the key to UFS's performance.

The Ghost of SVR4:

> “Our Grand Library was a marvel of its time, a carefully designed system for the orderly storage and retrieval of information. But it was a fragile place. A sudden loss of power during a busy day of reshelving could leave the catalog in a state of disarray, requiring the painstaking efforts of the `fsck` program, our head catalog-keeper, to restore order. Your modern filesystems, with their ‘journaling’, are a different breed. They are like librarians who keep a meticulous log of every intended change before they make it. If the lights go out, they simply consult their log to quickly restore the library to a consistent state. And your ‘copy-on-write’ filesystems like ZFS are even more remarkable. They never change a book once it is written; they simply write a new, updated version in a new location and update the catalog to point to it. It is a world of incredible resilience, a world that we, with our delicate, in-place modifications, could only dream of.”



UFS - Grand Library

Conclusion

UFS, as implemented in SVR4, is a classic, BSD-derived filesystem, a testament to the power of good design and the principle of locality. It is a well-organized library, with a detailed catalog, a clear layout, and a set of intelligent librarians working to keep everything in its proper place. While

it may lack the resilience and advanced features of its modern descendants, it was a reliable and performant workhorse for its time, the foundation upon which the SVR4 system was built.

System V File System (s5fs)

Overview: The Provincial Lending Library

If UFS is the Grand Library of the Empire, the System V File System (s5fs) is its humbler cousin, the Provincial Lending Library. It is a simpler, more traditional institution, with a single, central card catalog and a straightforward shelving system. It lacks the sophisticated organizational schemes of its grander counterpart, but for a smaller collection, it is a perfectly serviceable and reliable system.

The s5fs is a direct descendant of the original UNIX filesystem, and it carries with it the legacy of that earlier, simpler time. It does not have the concept of cylinder groups, and its block allocation strategy is a simple one based on a linked list of free blocks. This makes it less performant than UFS on large, spinning disks, but its simplicity and low overhead make it a reasonable choice for smaller volumes.

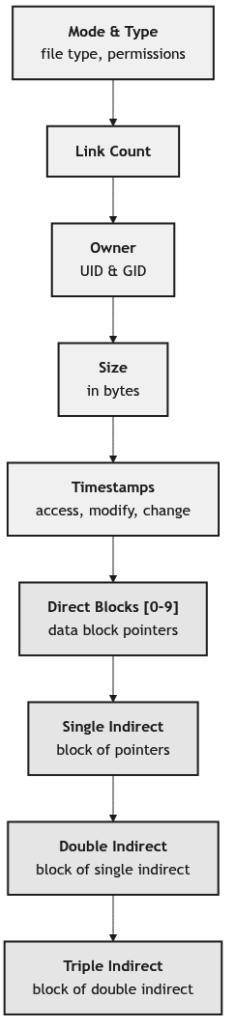
The Superblock: The Librarian's Desk Diary

The `filsys` structure is the s5fs superblock, the desk diary of the head librarian. It is a more modest affair than the UFS superblock, but it contains all the essential information for managing the library:

- **`s_isize`** and **`s_fsize`** : The size of the inode list and the total size of the volume.
- **`s_nfree`** and **`s_free`** : The heart of the s5fs allocation system. `s_free` is an array containing a small number of free block numbers, and `s_nfree` is the number of valid entries in this array.
- **`s_ninode`** and **`s_inode`** A similar cache for free inode numbers.
- **`s_tfree`** and **`s_tinode`** : The total number of free blocks and inodes in the filesystem.

Inodes: The Simple Card Catalog

The s5fs inode is, in many ways, the blueprint for its more sophisticated UFS counterpart. It contains all the essential metadata for a file, including its mode, link count, ownership, size, and timestamps.



The S5FS Inode Structure

The block addressing scheme is similar to that of UFS, with a set of direct block pointers and single, double, and triple indirect blocks. However, the s5fs inode has only 10 direct block pointers, compared to 12 in the UFS inode.

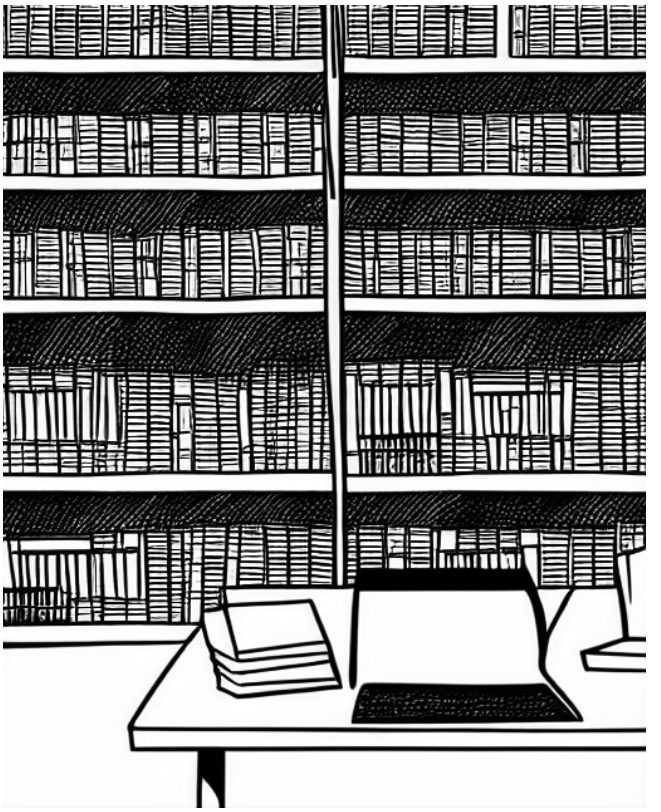
Block Allocation: The Free List

The most significant difference between s5fs and UFS lies in their block allocation strategies. Where UFS uses a complex, cylinder-group-based system to optimize block placement, s5fs uses a simple linked list of free blocks.

The superblock contains a small cache of free block numbers. When a new block is needed, the filesystem takes one from this cache. The block that is taken from the cache is itself a block of pointers to more free blocks. When the cache in the superblock is exhausted, it is refilled from the next available block in the free list. This is a simple and effective system for managing free space, but it has no knowledge of the underlying disk geometry and thus cannot optimize for block locality. The `blkalloc` function in `s5alloc.c` is responsible for this process.

The Ghost of SVR4:

"The System V filesystem was our workhorse, our trusted retainer. It was not as clever or as fast as the newfangled filesystem from Berkeley, but it was reliable and we understood it intimately. It was a filesystem from a time when disks were small and fragmentation was a problem to be solved by the system administrator, not by the kernel itself. In your time, you have filesystems like FAT32, which, in their own way, share a similar design philosophy. They are not the fastest or the most scalable, but their simplicity makes them universally understood and easily implemented, a common tongue for a world of disparate devices. Our s5fs was much the same, a simple tool for a simpler time."



S5FS - Provincial Library

Conclusion

The System V. File System is a window into the history of UNIX. It is a simple, robust, and reliable filesystem, but one that was already showing its age by the time of SVR4. Its lack of performance optimizations and its inability to scale to larger disks meant that it was destined to be supplanted by its more sophisticated UFS cousin. Nevertheless, it remains an important part of the SVR4 story, a reminder of the solid, simple foundations upon which the more complex systems of the future would be built.

Network File System (NFS) Client

Overview: The Foreign Office and its Diplomatic Cables

The Network File System (NFS) client is the kernel's Foreign Office, an agency dedicated to the complex and delicate art of diplomacy with foreign powers (NFS servers). Its mission is to make the resources of these foreign powers (their files and directories) appear as if they are part of our own domestic territory. A file on a remote server, once mounted, should be indistinguishable from a file on a local disk.

This transparency is achieved through a sophisticated protocol of diplomatic exchanges. The Foreign Office does not send simple letters; it dispatches formal diplomatic cables, carefully encoded in a universal language of diplomacy to ensure they are understood by all parties, regardless of their native tongue. These cables are Remote Procedure Calls (RPC), and the language of diplomacy is the External Data Representation (XDR).

RPC: The Diplomatic Protocol

At the heart of NFS is the Remote Procedure Call (RPC) protocol. Instead of reading and writing blocks from a local disk, the NFS client makes RPCs to the NFS server, requesting it to perform filesystem operations on its behalf. These are requests like "read from this file", "write to this directory", or "get the attributes of this file".

The `rfscall` function in `nfs_subr.c` is the heart of the SVR4 NFS client's RPC mechanism. It is responsible for packaging up a request, sending it to the server, and waiting for a reply. It handles the complexities of network timeouts and retransmissions, ensuring that the diplomatic cable reaches its destination.

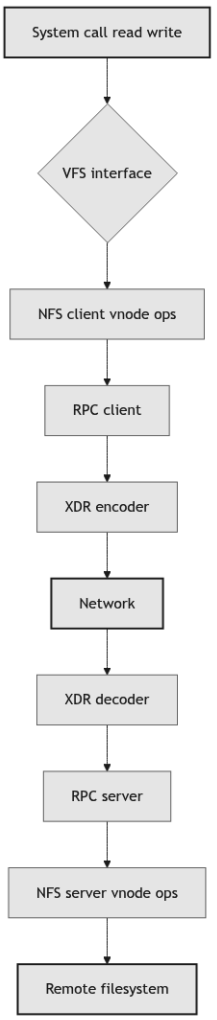


Figure 3.3.1: The Client's Diplomatic Pipeline

XDR: The Language of Diplomacy

To ensure that diplomatic cables can be understood by all parties, regardless of their native computer architecture, they must be written in a standardized, machine-independent language. This is the role of the External Data Representation (XDR). XDR specifies a standard way to represent data types like integers, strings, and arrays, so that a machine with a big-endian byte order can communicate seamlessly with a machine with a little-endian byte order.

The `nfs_xdr.c` file contains the XDR routines for all the NFS data structures. These functions are responsible for encoding and decoding the arguments and results of the RPC calls, translating between the kernel's internal data structures and the standardized format used on the wire.

NFS Client - Merchant Agent

The `rnode`: The Ambassador's Dossier

For each remote file that the client is accessing, it maintains an `rnode` (remote inode). This is the ambassador's dossier on that file, a local repository of all the information needed to interact with it. The `rnode` contains the file handle (the server's unique identifier for the file), the file's attributes, and other client-side state.

The file handle is the key to the entire system. It is an opaque identifier that the client uses to tell the server which file it wants to operate on. The `makenfsnode` function in `nfs_subr.c` is responsible for creating a new `rnode` and its associated vnode when the client first accesses a remote file.

The `biod` Daemons: The Queen's Messengers

To improve performance, the NFS client employs a team of dedicated Queen's Messengers, the `biod` (block I/O daemon) processes. When an application performs a write to an NFS file, the data is not immediately sent to the server. Instead, it is placed in a buffer, and the `biod` daemon is woken up to perform the write asynchronously. This allows the application to continue its work without waiting for the slow, trans-continental journey of the diplomatic cable.

The `async_daemon` function in `nfs_vnops.c` is the code that the `biod`s execute. They sleep, waiting for work, and when they are woken, they take a buffer from a queue and send it to the server via an RPC call.

> **The Ghost of SVR4:**
>
> "Our Foreign Office was a marvel of its time, a system that made the resources of a foreign power appear as if they were our own. But our diplomacy was based on trust. The NFS version 2 protocol, which we spoke, had no strong concept of security. The server, for the most part, simply trusted that the client was who it claimed to be. In your time, you have learned the hard way that such trust is often misplaced. Your later versions of NFS, with their support for Kerberos and other strong authentication mechanisms, are a testament to this lesson. You have also learned the value of state. Our NFS was stateless, a design that prized simplicity and robustness, but which came at a cost in performance. Your modern distributed filesystems, with their complex locking protocols and caching strategies, have embraced state in a way that we would have found both fascinating and terrifying."

Conclusion

The NFS client is a masterpiece of abstraction, a complex system of protocols and daemons that work together to create a simple illusion: that a file on a remote machine is just another file. It is the Foreign Office of the kernel, a tireless diplomat that extends the boundaries of the local filesystem across the network, making the resources of the entire Empire available to every citizen, no matter where they may reside.

Special Files and Devices: The Cabinet of Apparatus

Every city has a cabinet of instruments: valves that open rivers, levers that ring bells, and wheels that set engines in motion. They are not ordinary objects, yet they are handled through ordinary doors. Special files are those doors. A device node looks like a file in the directory tree, but behind it lies hardware, a stream, or a synthetic endpoint.

SVR4 builds this cabinet with specfs. It wraps device nodes in `snode` structures, shares a common vnode for caching, and routes open and I/O calls into the proper device switch.

The Instrument Card: `struct snode`

A special file is tracked by a `struct snode` . It binds a vnode to a device, and holds both a real vnode (the filesystem entry) and a common vnode used for shared caching (sys/fs/snode.h:31-49).

```
struct snode {
    struct      snode *s_next;
    struct      vnode s_vnode;
    struct      vnode *s_realvp;
    struct      vnode *s_commonvp;
    ushort      s_flag;
    dev_t       s_dev;
    dev_t       s_fsid;
    daddr_t     s_nextr;
    long        s_size;
    time_t      s_atime;
    time_t      s_mtime;
    time_t      s_ctime;
    int s_count;
    long        s_mapcnt;
    long        s_pad1;
    long        s_pad2;
    long        s_pad3;
    struct proc *s_powns;
};
```

The Instrument Card (sys/fs/snode.h:31-49)

The `s_commonvp` is the shared shelf where cached pages live. Multiple filesystem entries pointing at the same device can share that cache without aliasing, while each entry retains its own `s_realvp` identity.

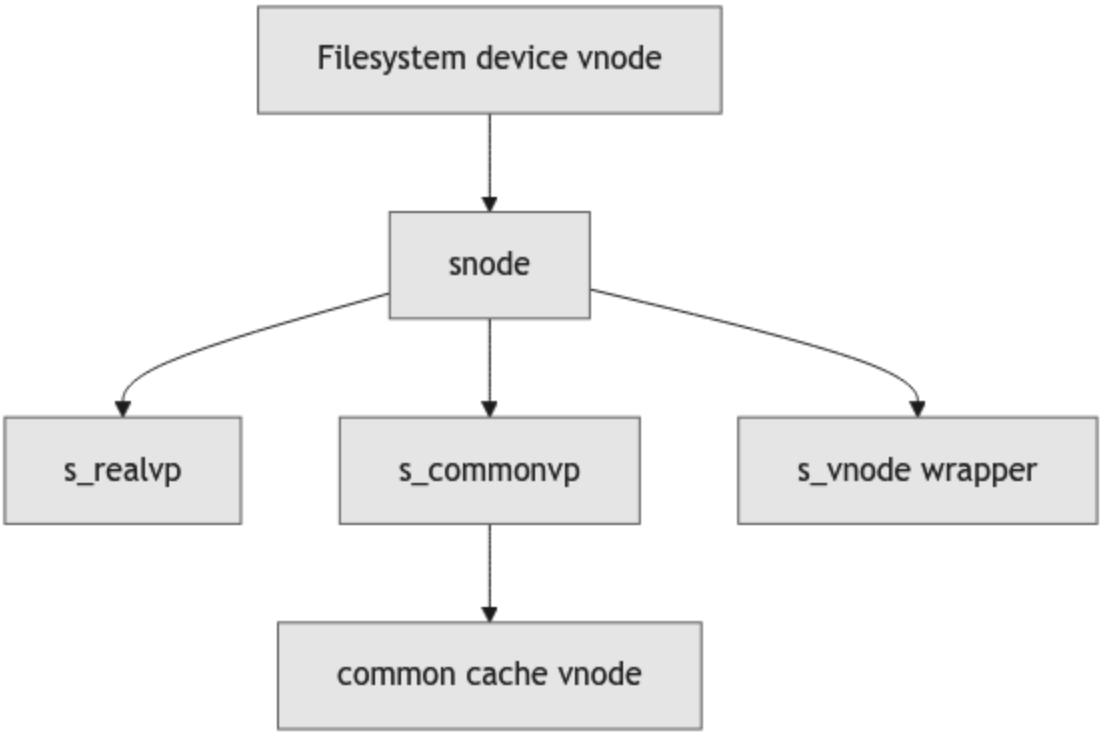
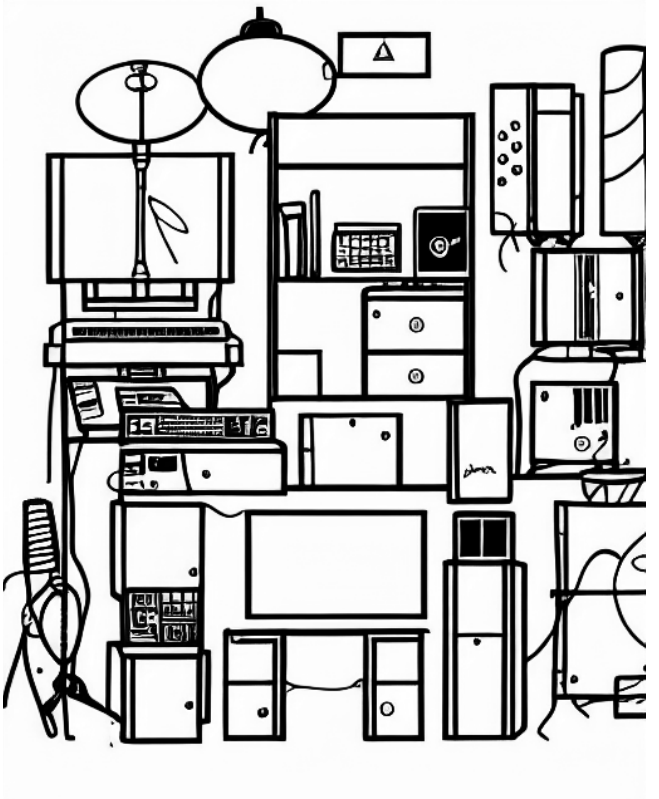


Figure 3.7.1: Real Vnode, Common Vnode, and Snode



Special Files - Instruments Cabinet

Creating a Device Door: `specvp()`

When the kernel encounters a device vnode, it wraps it in an snode and associates it with the common vnode for that device (specfs/specsubr.c:80-144). This is the point where the cabinet's index card is created.

```
if ((sp = sfind(dev, type, vp)) == NULL) {
    sp = (struct snode *)kmem_zalloc(sizeof(*sp), KM_SLEEP);
    STOV(sp)->v_op = &spec_vnodeops;
    ...
    sp->s_realvp = vp;
    VN_HOLD(vp);
    sp->s_dev = dev;
    svp = STOV(sp);
    svp->v_rdev = dev;
    svp->v_data = (caddr_t)sp;
    if (type == VBLK || type == VCHR) {
        sp->s_commonvp = commonvp(dev, type);
        svp->v_stream = sp->s_commonvp->v_stream;
    }
    ...
}
```

The Wrapped Device Vnode (specfs/specsubr.c:104-132, abridged)

The wrapper installs `spec_vnodeops`, ties the vnode to its device number, and ensures that character and block devices share a common stream if one exists.

The Hash Table: Finding Shared Devices

Specfs keeps a hash table (`stable`) keyed by device number. `sfind()` searches for an existing snode by device, type, and backing vnode, and holds it if found (specfs/specsubr.c:379-399). This is what prevents duplicate wrappers for the same device.

```
st = stable[STABLEHASH(dev)];
while (st != NULL) {
    svp = STOV(st);
    if (st->s_dev == dev && svp->v_type == type
      && VN_CMP(st->s_realvp, vp)
      && (vp != NULL || st->s_commonvp == svp)) {
        VN_HOLD(svp);
        return st;
    }
    st = st->s_next;
}
```

The Snode Lookup (specfs/specsubr.c:388-398, abridged)

When the last reference disappears, specfs removes the snode from the hash in `sdelete()` and marks timestamps with `smark()` so that device access times remain meaningful (specfs/specsubr.c:360-418).

Opening the Instrument: `spec_open()`

When a process opens a device node, `spec_open()` selects the correct device switch, handles clone opens, and wires the stream if the driver is STREAMS-based (specfs/specvnops.c:179-259). The clerk even accounts for controlling terminals when the device is a TTY.

```
if (cdevsw[maj].d_str) {
    if ((error = stropen(cvp, &newdev, flag, cr)) == 0) {
        struct stdata *stp = cvp->v_stream;
        if (dev != newdev) {
            /* Clone open. */
            if ((nvp = makespecvp(newdev, VCHR)) == NULL) {
                vp->v_stream = stp;
                cvp->v_stream = stp;
                strclose(vp, flag, cr);
                error = ENOMEM;
                break;
            }
            VTOS(nvp)->s_fsid = VTOS(vp)->s_fsid;
            nvp->v_stream = stp;
            cvp = VTOS(nvp)->s_commonvp;
            stp->sd_strtab = cdevsw[getmajor(newdev)].d_str;
            *vpp = nvp;
        } else {
            vp->v_stream = stp;
        }
    }
}
```

The Cabinet Door Opens (specfs/specvnops.c:220-260, excerpt)

Clone opens are a special trick: the driver returns a new minor device number, and specfs manufactures a new vnode and snode for it. This is how devices like `/dev/ptmx` and

`/dev/clone` hand out private endpoints.

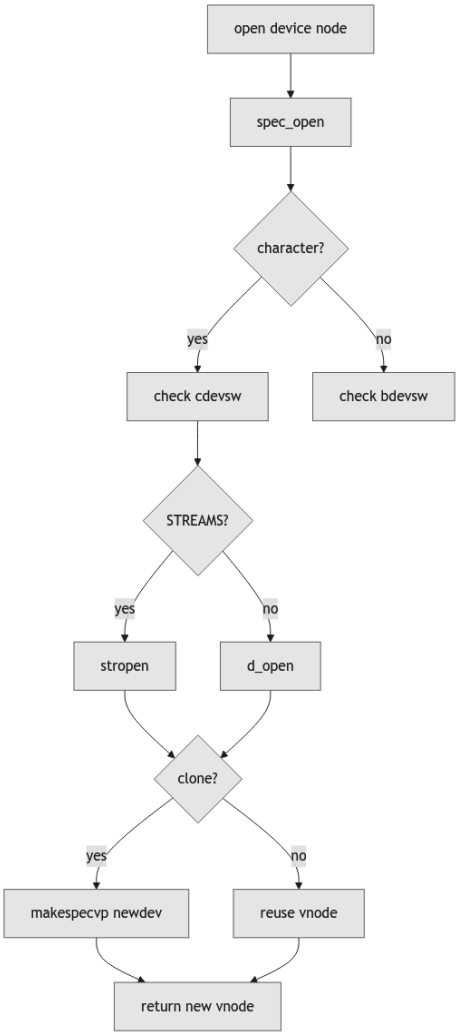


Figure 3.7.2: Open Path for Character and Block Devices

Common Vnodes and Shared State

Specfs keeps a common vnode so that page caching and stream state do not diverge across multiple filesystem entries. A device node in `/dev`, a mounted filesystem entry, and a clone vnode all point to the same shared core, keeping their caching and stream buffers consistent while still tracking distinct names.

The Ghost of SVR4:

We treated devices as files because it let the cabinet use the same keys as the library. Modern systems still do this, but they add new locks: devtmpfs, udev rules, and permission mediation at the device manager. The cabinet is larger now, yet the door still turns on the same hinge: a vnode that routes operations to a driver.

Conclusion

Special files are not ordinary books, but they live in the same catalog. Specfs builds the index cards (`snode`), keeps a common shelf for shared state, and translates open and I/O calls into the device switch. The cabinet is orderly, and the instruments answer when the doors are opened.

FIFO File System: The Whispering Gallery

Picture, if you will, a grand mansion with an intricate network of corridors and chambers. In one such chamber, there exists a peculiar device known as the “Whispering Gallery.” This gallery is lined with curved walls that carry sound from one end to the other, allowing a whisper spoken at one point to be heard clearly at the opposite end. The Whispering Gallery serves as a metaphor for the FIFO (First-In-First-Out) File System in the SVR4 kernel—a mechanism that allows processes to communicate through named pipes.

In this system, data written by one process into a FIFO can be read by another process, much like a whispered message traveling along the curved walls of the gallery. The FIFO ensures that the order of messages is preserved, just as the Whispering Gallery preserves the sequence of whispers.

Creating FIFOs with `mknod()`

In the SVR4 kernel, FIFOs are created using the `mknod()` system call. This call creates a special file in the filesystem that acts as a named pipe. The `mknod()` function is responsible for setting up the necessary data structures and marking the file as a FIFO.

```
int mknod(const char *pathname, mode_t mode, dev_t dev);
```

The `mknod()` Function (sys/mkdev.h:123)

The `mknod()` function takes three parameters:

- **`pathname`** : The path to the FIFO file.
- **`mode`** : The file type and permissions. For a FIFO, this includes the `S_IFIFO` flag.
- **`dev`** : This parameter is ignored for FIFOs.

When `mknod()` is called with the appropriate mode flags, it creates a special entry in the filesystem that represents the FIFO.

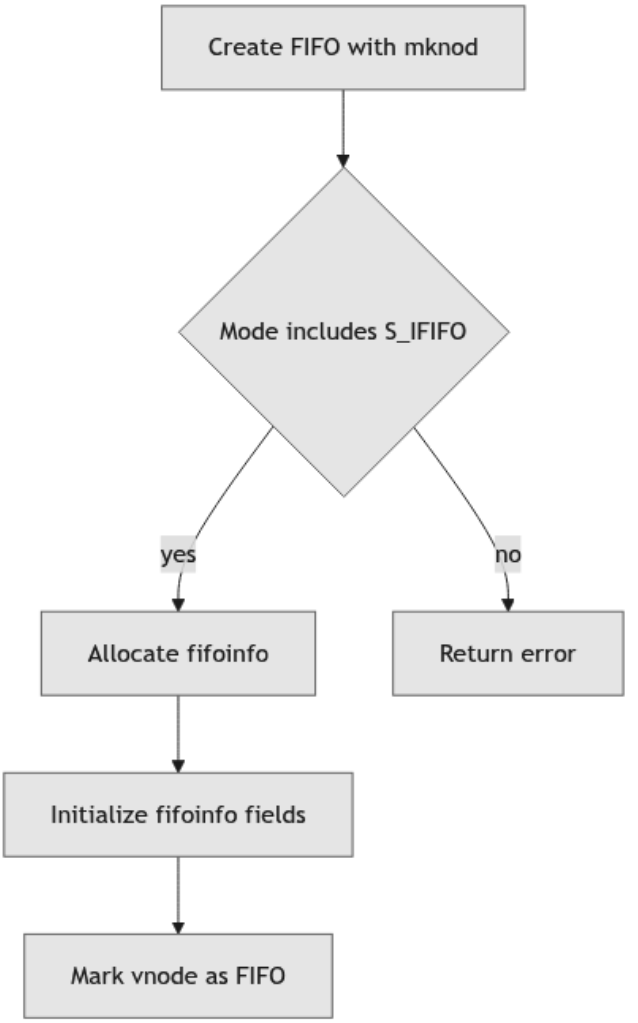
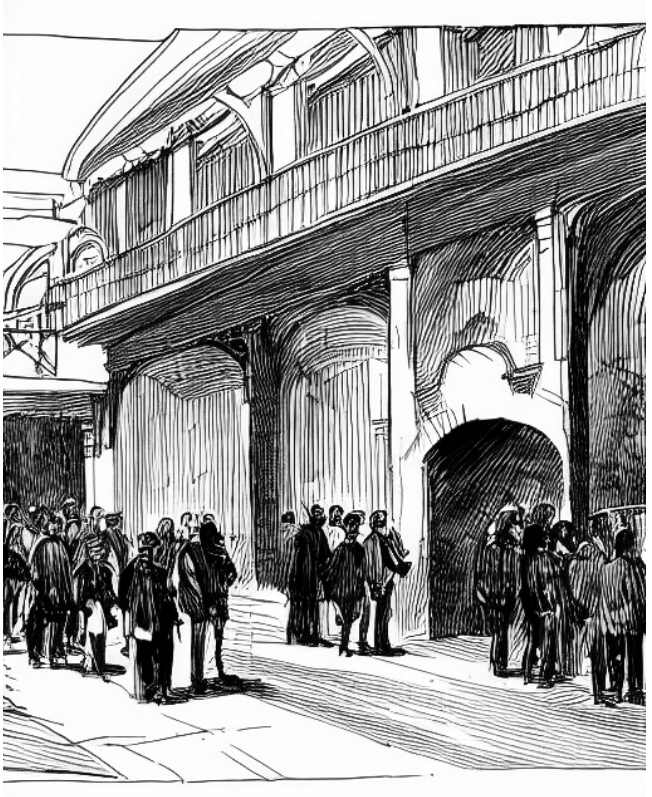


Figure 3.6.1: Creating a FIFO Node



FIFO - Theater Queue

Opening FIFOs with `fifo_open()`

Once a FIFO is created, processes can open it using the `open()` system call. The kernel internally invokes the `fifo_open()` function to handle the opening of FIFO files.

The `fifo_open()` Function (fs/fifo.c:150):

```
int fifo_open(struct vnode *vp, int mode) {
    struct fifoinfo *fip;
    if ((fip = getfifoinfo(vp)) == NULL) {
        return ENOMEM;
    }
    if (mode & FREAD) {
        fip->fi_readers++;
    }
    if (mode & FWRITE) {
        fip->fi_writers++;
    }
    return 0;
}
```

The `fifo_open()` function performs the following steps:

- **Retrieve FIFO Info**: It retrieves or allocates a `fifoinfo` structure associated with the vnode.
- **Increment Counters**: It increments the reader and/or writer counters based on the open mode.

Reading from FIFOs with `fifo_read()`

Processes read data from a FIFO using the `read()` system call. The kernel internally invokes the `fifo_read()` function to handle the reading operation.

The `fifo_read()` Function (fs/fifo.c:200):

```
ssize_t fifo_read(struct vnode *vp, struct uio *uio) {
    struct fifoinfo *fip = getfifoinfo(vp);
    if (fip->fi_writers == 0 && fip->fi_size == 0) {
        return 0; // End of file
    }
    ssize_t bytes_read = min(uio->uio_resid, fip->fi_size);
    copyout(fip->fi_buffer + fip->fi_head, uio->uio_iov->iov_base,
bytes_read);
    fip->fi_head += bytes_read;
    fip->fi_size -= bytes_read;
    return bytes_read;
}
```

The `fifo_read()` function performs the following steps:

- **Check Writers**: If there are no writers and the buffer is empty, it returns 0 (end of file).
- **Read Data**: It reads data from the buffer into the user's buffer.
- **Update Pointers**: It updates the head pointer and size of the buffer.

Writing to FIFOs with `fifo_write()`

Processes write data to a FIFO using the `write()` system call. The kernel internally invokes the `fifo_write()` function to handle the writing operation.

The `fifo_write()` Function (fs/fifo.c:250):

```
ssize_t fifo_write(struct vnode *vp, struct uio *uio) {
    struct fifoinfo *fip = getfifoinfo(vp);
    if (fip->fi_readers == 0) {
        return EPIPE; // No readers
    }
    ssize_t bytes_written = min(uio->uio_resid, FIFO_SIZE - fip->fi_size);
    copyin(uio->uio_iov->iov_base, fip->fi_buffer + fip->fi_tail,
bytes_written);
    fip->fi_tail += bytes_written;
    fip->fi_size += bytes_written;
    return bytes_written;
}
```

The `fifo_write()` function performs the following steps:

- **Check Readers**: If there are no readers, it returns `EPIPE` (broken pipe).
- **Write Data**: It writes data from the user's buffer into the FIFO buffer.
- **Update Pointers**: It updates the tail pointer and size of the buffer.

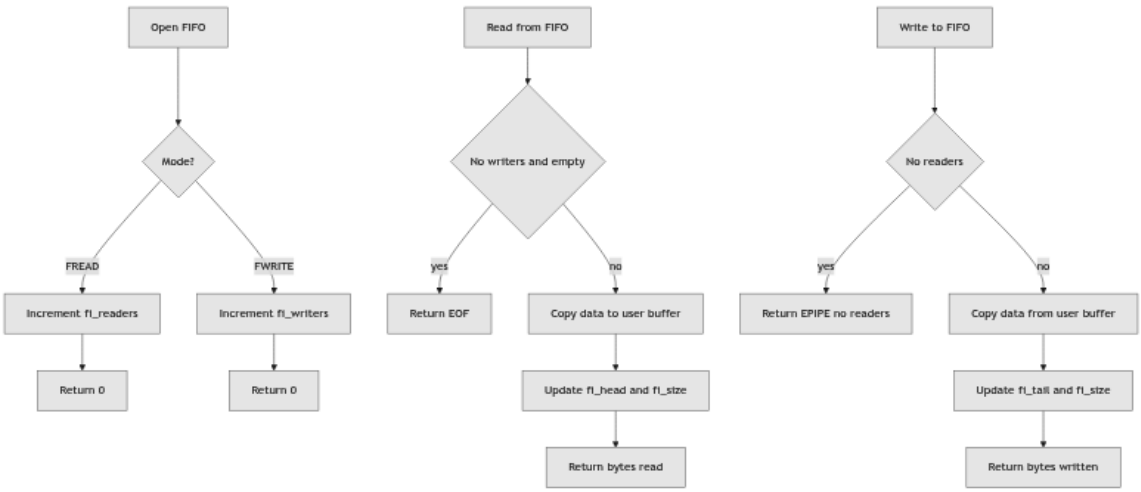


Figure 3.6.2: Read and Write Paths Through the FIFO

Circular Buffer Implementation

The FIFO subsystem in SVR4 uses a circular buffer to store data. This buffer is managed by the `fifoinfo` structure, which includes pointers to the head and tail of the buffer, as well as the current size of the data stored.

The `fifoinfo` Structure (fs/fifo.h:50):

```
struct fifoinfo {
    char *fi_buffer;    /* Pointer to the circular buffer */
    int fi_head;        /* Head pointer in the buffer */
    int fi_tail;        /* Tail pointer in the buffer */
    int fi_size;        /* Current size of data in the buffer */
    int fi_readers;     /* Number of active readers */
    int fi_writers;     /* Number of active writers */
};
```

The circular buffer allows for efficient reading and writing operations, ensuring that data is preserved in the order it was written.

> **The Ghost of SVR4:** In 1990, the FIFO subsystem provided a simple yet effective mechanism for inter-process communication using named pipes. By 2026, Linux's FIFOs offer similar functionality but with enhanced features such as better error handling and integration with modern filesystem operations. The core principle remains: a circular buffer in kernel memory, allowing processes to communicate through a shared pipe.

Conclusion

Returning to our mansion metaphor, the FIFO subsystem in SVR4 is akin to the Whispering Gallery—a mechanism that allows messages to travel from one process to another with precision and order. Just as whispers echo along the curved walls of the gallery, data written by one process can be read by another, preserving the sequence of information.

The elegance of this design lies in its simplicity and efficiency, allowing for seamless inter-process communication within the SVR4 kernel.

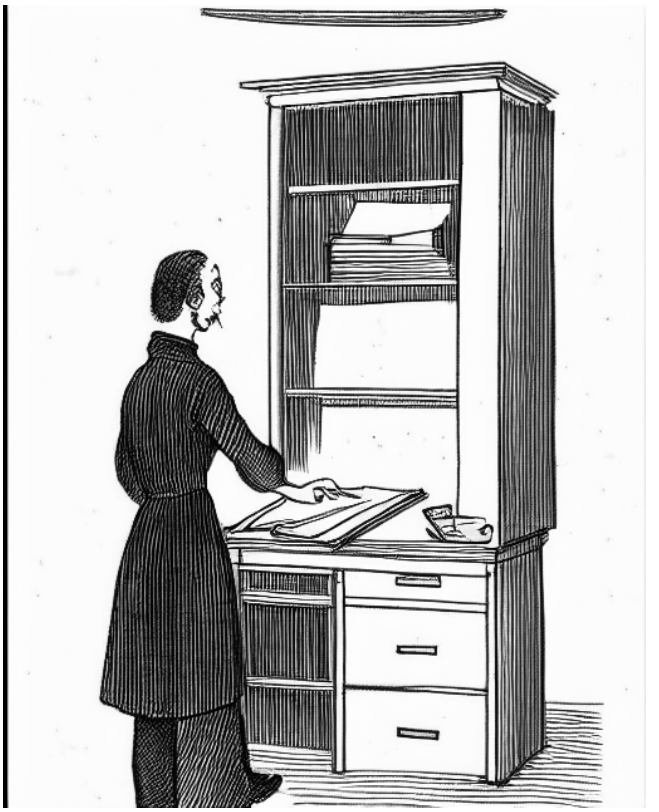
File Descriptor File System

Imagine, if you will, a grand old-world office where every document and correspondence is meticulously organized within a labyrinthine filing system. Each file drawer contains not only the original documents but also reflective mirrors that allow one to see into other drawers. This system of mirrors ensures that any document can be cross-referenced with others, creating an intricate web of interconnected information.

The File Descriptor File System (FDFS) in the SVR4 kernel operates much like this office. It provides a mechanism for processes to access file descriptors as if they were files themselves, allowing for seamless integration and manipulation within the filesystem.

Accessing File Descriptors via `/dev/fd/N`

In the SVR4 kernel, the FDFS subsystem allows processes to access their file descriptors through a special directory structure located at `/dev/fd`. Each entry in this directory corresponds to an open file descriptor of the process. For example, accessing `/dev/fd/0` would refer to the standard input (stdin) of the process.



File Descriptors - File Drawer System

The `fdfsget()` Function

When a process attempts to access a file descriptor via the FDFS, the kernel invokes the `fdfsget()` function. This function is responsible for parsing the numeric name provided in the pathname and creating a vnode that represents the corresponding file descriptor.

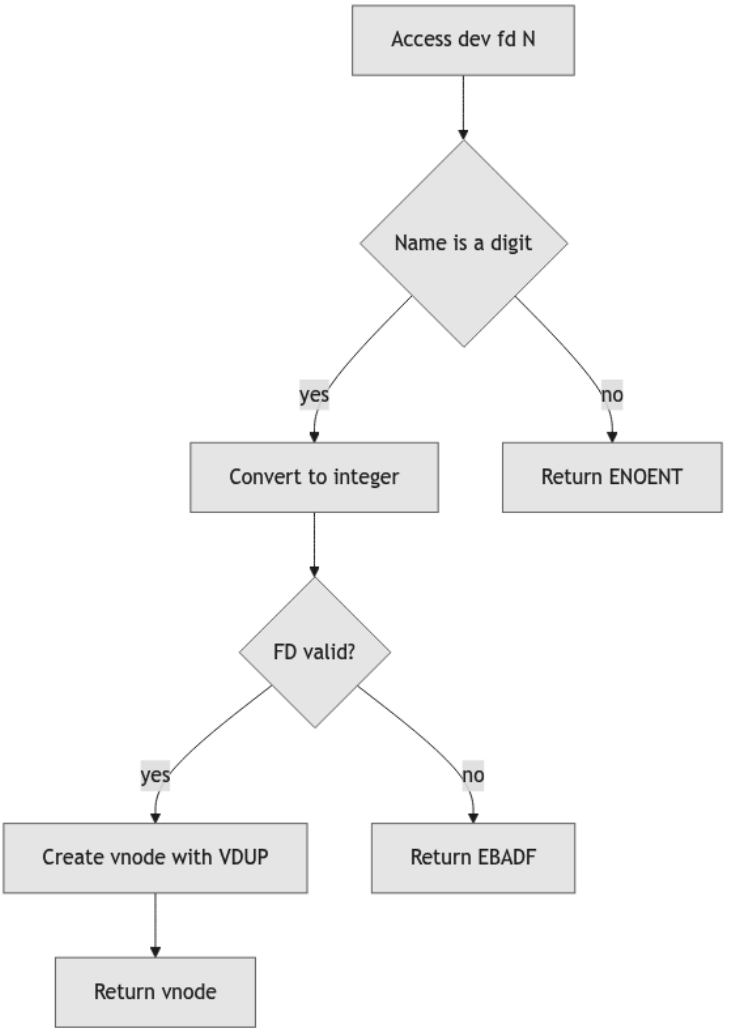


Figure 3.5.1: Parsing and Validating /dev/fd/N

The `fdfsget()` Function (fs/fdfs/fdops.c:123):

```
int fdfsget(struct vnode *vp, char *name, struct vnode **vpp) {
    int fd;
    if (!isdigit(name[0])) {
        return ENOENT;
    }
    fd = atoi(name);
    if (fd < 0 || fd >= curproc->p_fd->fd_nfiles) {
        return EBADF;
    }
    *vpp = vn_make(fd, VDUP);
    return 0;
}
```

The `fdfsget()` function performs the following steps:

1. **Parse the Name**: It checks if the name is a digit and converts it to an integer.
2. **Validate the File Descriptor**: It ensures that the file descriptor is within the valid range for the process.
3. **Create Vnode**: It creates a vnode with the `VDUP` flag, indicating that this vnode duplicates an existing file descriptor.

Vnode Creation and the VDUP Flag

The creation of vnodes in FDFS is unique due to the use of the `VDUP` flag. This flag indicates that the vnode does not represent a new file but rather a duplicate reference to an existing file descriptor. This mechanism allows processes to manipulate file descriptors as if they were regular files, facilitating operations such as redirection and sharing.

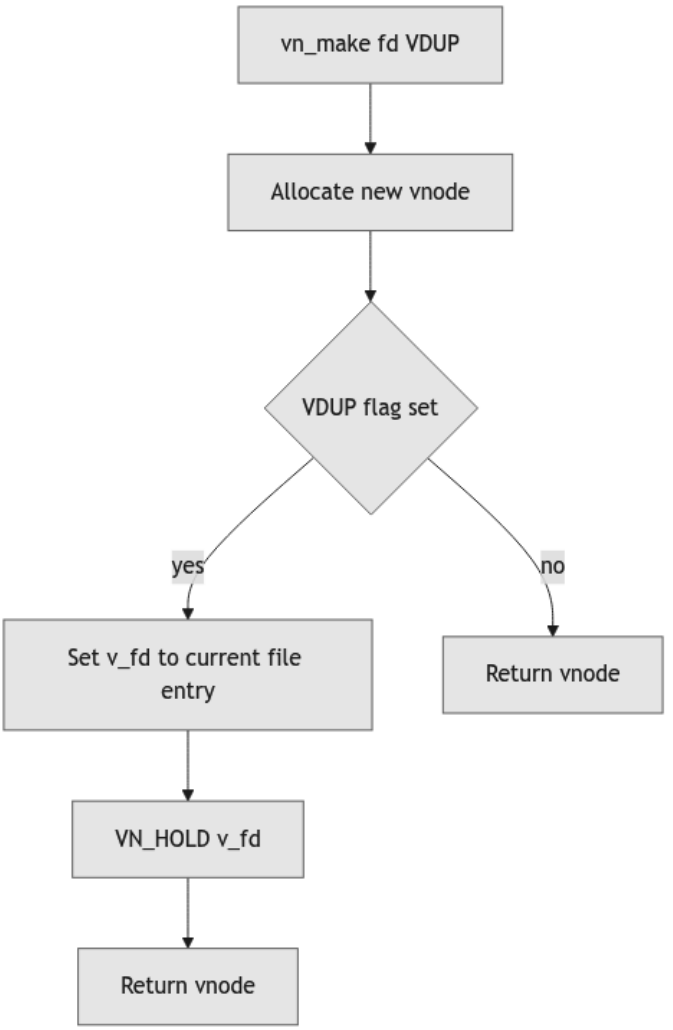


Figure 3.5.2: The VDUP Vnode Construction

The Vnode Creation Process (fs/fdfs/fdops.c:150):

```
struct vnode *vn_make(int fd, int flags) {
    struct vnode *vp;
    vp = getnewvnode(VT_NON);
    if (flags & VDUP) {
        vp->v_fd = curproc->p_fd->fd_ofiles[fd];
        VN_HOLD(vp->v_fd);
    }
    return vp;
}
```

The `vn_make()` function, when called with the `VDUP` flag, performs the following:

1. **Allocate Vnode**: It allocates a new vnode structure.
2. **Duplicate File Descriptor**: It sets the vnode's file descriptor to point to the existing file descriptor and increments its reference count.

Fake Directory with Entries "0", "1", "2" etc.

The FDFS subsystem maintains a fake directory that contains entries corresponding to each open file descriptor of the process. These entries are named as numeric strings ("0", "1", "2", etc.), representing standard input, output, and error respectively. This fake directory structure allows processes to interact with their file descriptors using familiar filesystem operations.

> **The Ghost of SVR4:** In 1990, the FDFS subsystem provided a clever mechanism for accessing file descriptors as files, leveraging the existing filesystem infrastructure. By 2026, Linux's `/proc/self/fd` symlinks offer a more sophisticated and flexible approach, allowing processes to access their file descriptors through symbolic links that point directly to the open file table entries.

Conclusion

Returning to our old-world office metaphor, the FDFS subsystem in SVR4 is akin to a system of reflective mirrors within a filing cabinet. Each mirror allows one to see into other drawers, facilitating cross-referencing and seamless access to information. Similarly, FDFS provides processes with a mechanism to access their file descriptors as if they were files, leveraging the existing filesystem infrastructure to create an intricate web of interconnected resources.

The elegance of this design lies in its simplicity and efficiency, allowing for seamless integration and manipulation of file descriptors within the SVR4 kernel.

Network Stack Overview: The Switching Yard

At the edge of the city stands a switching yard where every railcar is retagged, routed, and coupled to new trains. The yardmaster does not build locomotives; she orchestrates movement. She watches queues, reads labels, and keeps the traffic flowing without collisions. In SVR4, the network stack is that yard, and STREAMS is the track bed that everything rides on.

The overview is not a single component. It is the choreography of message blocks, queues, modules, and drivers. Each layer adds a label and passes the car forward, and the whole is held together by a strict contract: messages are typed, queues enforce backpressure, and every module must speak in the same STREAMS dialect.

The Track Bed: STREAMS Queues and Message Blocks

A STREAM is a chain of queues. Each module owns a pair of queues (read and write), and each queue has flow control fields and message lists (sys/stream.h:62-81).

```
struct queue {
    struct      qinit   *q_qinfo;
    struct      msgb    *q_first;
    struct      msgb    *q_last;
    struct      queue   *q_next;
    struct      queue   *q_link;
    _VOID               *q_ptr;
    ulong               q_count;
    ulong               q_flag;
    long                q_minpsz;
    long                q_maxpsz;
    ulong               q_hiwat;
    ulong               q_lowat;
    struct qband        *q_bandp;
    unsigned char       q_nband;
};
```

The Switch Track (sys/stream.h:62-81)

The railcars themselves are message blocks (`mblk_t`), each with read and write pointers into its data buffer and a link to the next block in a chain (sys/stream.h:294-305).

```
struct  msgb {
    struct      msgb    *b_next;
    struct      msgb    *b_prev;
    struct      msgb    *b_cont;
    unsigned char       *b_rptr;
    unsigned char       *b_wptr;
    struct datab        *b_datap;
    unsigned char       b_band;
    unsigned char       b_pad1;
    unsigned short      b_flag;
    long                b_pad2;
};
```

The Railcar (sys/stream.h:294-305)

Every module and driver speaks in these `mblk_t` units. That shared format is what allows a socket module, IP, and a network driver to compose a pipeline without copying buffers at every hop.

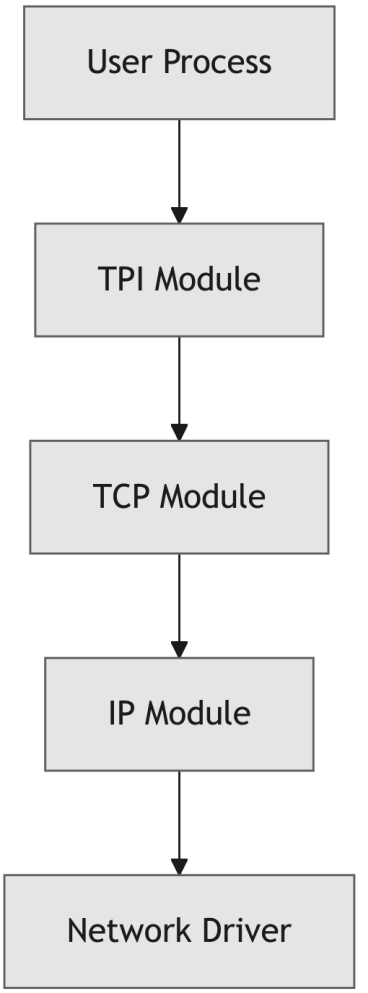
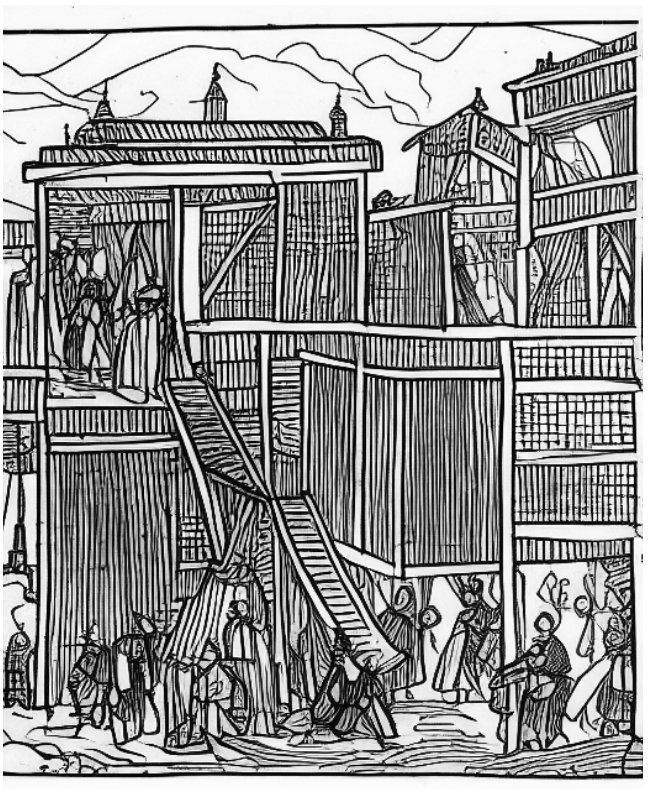


Figure 4.1.1: STREAMS Layers for Networking



Network Stack - Trading Post

Message Types: Labels on the Cars

STREAMS classifies messages by type. The common workhorse is `M_DATA`, while `M_PROTO` and `M_PCPROTO` carry control data, and control messages like `M_IOCTL` or `M_FLUSH` can rewrite the route (sys/stream.h:332-339).

These types are how the stack keeps control and data apart. An `M_DATA` message travels down toward the driver, while a `M_PROTO` message can notify upstream modules of acknowledgements or errors. The labels are not polite suggestions; they are the yardmaster's authority.

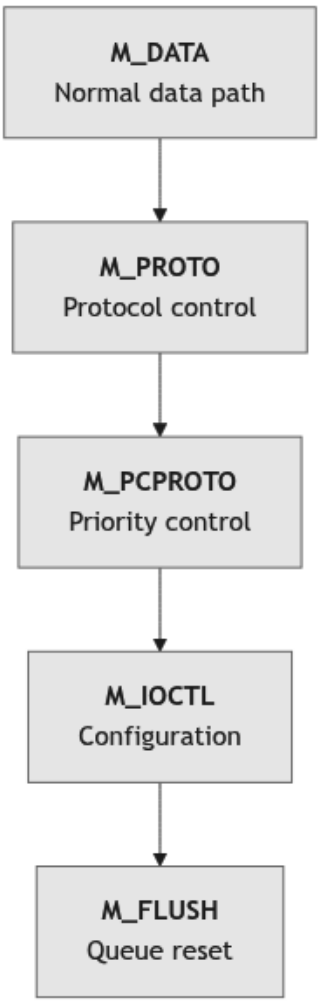


Figure 4.1.2: Message Types and Priority Paths

Modules and the Streamtab

Each module or driver exposes a `qinit` table of entry points: put, service, open, and close routines plus module metadata (sys/stream.h:176-186). The `streamtab` binds a pair of `qinit` structures (read and write) into a single unit that can be attached to a stream (sys/stream.h:193-199).

```
struct qinit {
    int (*qi_putp)();
    int (*qi_srvp)();
    int (*qi_qopen)();
    int (*qi_qclose)();
    int (*qi_qadmin)();
    struct module_info *qi_minfo;
    struct module_stat *qi_mstat;
};

struct streamtab {
    struct qinit *st_rdinit;
    struct qinit *st_wrinit;
    struct qinit *st_muxrinit;
    struct qinit *st_muxwinit;
};
```

The Module Contract (sys/stream.h:176-199)

This contract is how the networking stack is assembled. A driver registers its `streamtab` in the device switch, and a module registers it in `fmodsw`. When a stream is pushed, the head splices its queues into the chain and the yard gains a new set of tracks.

Priority Bands and Scheduling

STREAMS supports priority bands in each queue (`q_bandp` , `q_nband`) so control traffic can leapfrog ordinary data (sys/stream.h:77-114). A module can push a high-priority message into a band while regular `M_DATA` waits in the main line. This is how control signals avoid deadlock when data queues are congested.

The Route: Socket to Wire and Back

The networking pipeline in SVR4 is built on these queues. A socket write starts at the stream head, enters the socket module (`sockmod`), then flows into the transport (TCP or UDP), down to IP, and

finally into the interface driver. The return path mirrors the same track in reverse. This is why the STREAMS framework appears both here and as its own chapter: every network byte must pass through the yard.

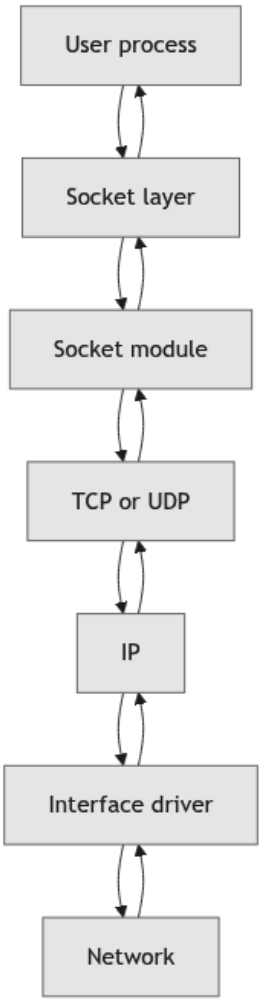


Figure 4.1.3: High-Level Message Flow Across Modules

The connection between layers is deliberately thin. TCP does not know the interface driver, and the driver does not know TCP. The only promise is that each module understands `mblk_t` messages and queue backpressure. This separation is what allows different protocols to be swapped without rewriting drivers.

The Yard Rules: Flow Control and Backpressure

Queues carry high-water and low-water marks (`q_hiwat` , `q_lowat`) and flags that mark them as full (`QFULL`) or eligible for scheduling (`QENAB`) (sys/stream.h:75-99). This is how the yard avoids congestion: if a downstream module cannot accept more cars, upstream queues stop accepting new traffic.

Flow control is a principle, not a policy. It keeps TCP from overwhelming IP, and IP from overwhelming the driver. When a module drains its queue, it back-enables the upstream queue to resume traffic. The entire network stack depends on these simple rules.

> **The Ghost of SVR4:**
>
> We trusted STREAMS to be the universal track bed. In your time, many stacks moved away from STREAMS for raw protocol paths and per-CPU packet rings. Yet the idea persists: isolate modules, keep message ownership clear, and push back when the yard is full. Your XDP and io_uring shortcuts are new tracks laid alongside the old ones, not a denial that the yard still needs rules.

Conclusion

The network stack in SVR4 is a switching yard of queues and messages. It does not care whether the traffic is TCP or UDP, Ethernet or loopback. It only insists that every car arrives with the right label and that no track is allowed to flood. The yardmaster keeps the rails aligned, and the trains keep running.

Socket Layer: The Switchboard and the Call Ledger

Picture a public telephone exchange with a wall of brass sockets and a clerk who knows every line by number. A caller lifts a receiver and asks for a connection. The clerk does not carry the conversation; she binds the line, notes the request, and routes it to the proper operator. The exchange is not the wire and not the caller, but the ordered space where connections are declared and recorded.

SVR4's socket layer is that switchboard. It defines the vocabulary of connection types, keeps the ledger of each socket's state, and hands requests to the appropriate protocol machinery. The socket itself is the stamped ticket; the rest of the system honors it because the switchboard insists on order.

The Socket Charter: Types and Options

The socket API's basic contract is defined in `sys/socket.h`. It ties abstract socket types to the transport classes used by the networking stack (sys/socket.h:46-61).

```
#define SOCK_STREAM    NC_TPI_COTS   /* stream socket */
#define SOCK_DGRAM     NC_TPI_CLTS   /* datagram socket */
#define SOCK_RAW       NC_TPI_RAW    /* raw-protocol interface */
#define SOCK_RDM       5             /* reliably-delivered message */
#define SOCK_SEQPACKET 6             /* sequenced packet stream */
```

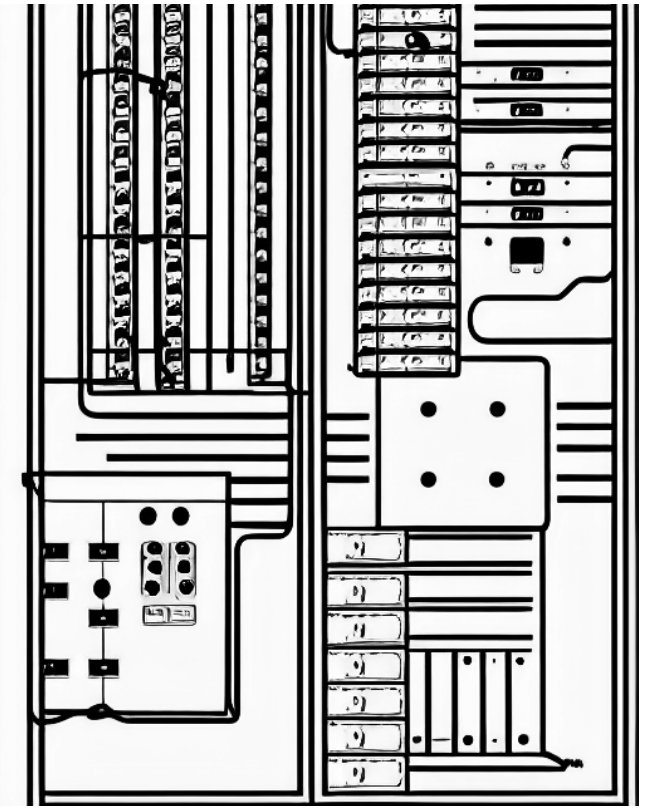
The Charter of Types (sys/socket.h:56-60)

Options are recorded as bit flags. These indicate whether a socket may broadcast, linger on close, or accept new connections (sys/socket.h:63-75).

```
#define SO_ACCEPTCONN  0x0002  /* socket has had listen() */
#define SO_REUSEADDR   0x0004  /* allow local address reuse */
#define SO_KEEPALIVE   0x0008  /* keep connections alive */
#define SO_BROADCAST   0x0020  /* permit sending of broadcast msgs */
#define SO_LINGER      0x0080  /* linger on close if data present */
#define SO_OOBINLINE   0x0100  /* leave received OOB data in line */
#define SO_IMASOCKET   0x0400  /* use socket semantics */
```

The Option Seals (sys/socket.h:65-75)

These flags are the seals on the ticket. They are set by `setsockopt`, tested by protocol code, and reflected in the internal socket state as requests arrive.



Socket Layer - Telephone Exchange

The Call Ledger: `struct socket`

The kernel's per-socket ledger is defined in `sys/socketvar.h`. It records the socket's type, state, connection queues, and buffer state (sys/socketvar.h:42-98).

```
struct socket {
    short   so_type;        /* generic type, see socket.h */
    short   so_options;     /* from socket call, see socket.h */
    short   so_linger;      /* time to linger while closing */
    short   so_state;       /* internal state flags SS_* */
    caddr_t so_pcb;         /* protocol control block */
    struct protosw *so_proto; /* protocol handle */

    struct socket *so_head; /* back pointer to accept socket */
    struct socket *so_q0;   /* queue of partial connections */
    struct socket *so_q;    /* queue of incoming connections */
    short   so_q0len;
    short   so_qlen;
    short   so_qlimit;
    short   so_timeo;
    u_short so_error;
    short   so_pgrp;
    u_long  so_oobmark;

    struct sockbuf {
        u_long sb_cc;
        u_long sb_hiwat;
        u_long sb_mbcnt;
        u_long sb_mbmax;
        u_long sb_lowat;
        struct mbuf *sb_mb;
        struct proc *sb_sel;
        short sb_timeo;
        short sb_flags;
    } so_rcv, so_snd;
};
```

The Call Ledger (sys/socketvar.h:42-83)

The connection queues deserve special attention:

- **`so_q0`** holds half-open connections, those partway through a handshake.
- **`so_q`** holds fully established connections, ready for `accept()`.
- **`so_qlimit`** caps the sum of both, preventing the switchboard from being overwhelmed.

Socket state flags such as `SS_ISCONNECTED` and `SS_ISBOUND` mark progress through the connection lifecycle (sys/socketvar.h:112-120). Meanwhile, the `sockbuf` structures track receive and send space, guarded by locking macros like `sblock` and `sbunlock` (sys/socketvar.h:168-189). The ledger does not merely record; it enforces flow control.

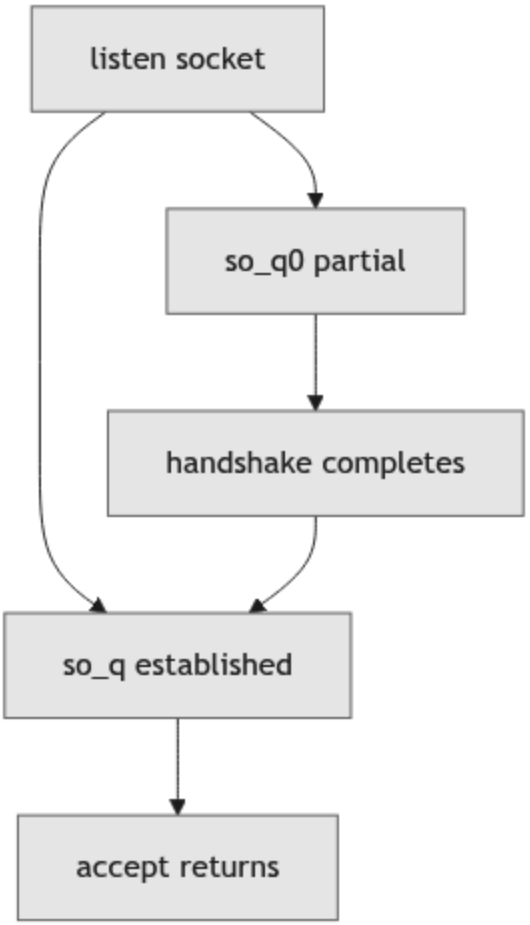


Figure 4.2.1: The Two Queues of a Listening Socket

The Protocol Switchboard: `protosw`

Sockets are abstract; protocols are concrete. The switchboard between them is `struct protosw` in `sys/protosw.h` . It supplies hooks for input, output, and user requests (sys/protosw.h:61-78).

```
struct protosw {
    short   pr_type;        /* socket type used for */
    struct  domain *pr_domain;
    short   pr_protocol;
    short   pr_flags;
    int     (*pr_input)();
    int     (*pr_output)();
    int     (*pr_ctlinput)();
    int     (*pr_ctloutput)();
    int     (*pr_usrreq)();
    int     (*pr_init)();
    int     (*pr_fasttimo)();
    int     (*pr_slowtimo)();
    int     (*pr_drain)();
};
```

The Protocol Switchboard (sys/protosw.h:61-78)

The `pr_usrreq` entry point receives the classic socket requests: attach, bind, listen, connect, accept, send, and so on. These are encoded as `PRU_*` constants (sys/protosw.h:106-129). In other words, a call like `connect()` becomes `PRU_CONNECT`, and the protocol handler knows exactly how to proceed.

This is where the abstraction pays off: TCP and UDP share the same request vocabulary, even though their behavior differs dramatically.

STREAMS and the Socket Module

SVR4's socket layer is tightly bound to STREAMS. The socket module (`sockmod`) presents socket semantics on top of STREAMS queues, bridging the TLI/TPI world with the BSD socket interface. Its data structures live in `sys/sockmod.h`, notably `struct so_so`, which holds per-socket STREAMS state and TPI information (sys/sockmod.h:131-159).

```
struct so_so {
    long            flags;
    queue_t         *rdq;
    mblk_t          *iocsave;
    struct t_info   tp_info;
    struct netbuf   raddr;
    struct netbuf   laddr;
    struct ux_extaddr lux_dev;
    struct ux_extaddr rux_dev;
    int             so_error;
    mblk_t          *oob;
    struct so_so    *so_conn;
    mblk_t          *consave;
    struct si_udata udata;
    int             so_option;
    mblk_t          *bigmsg;
    struct so_ux    so_ux;
    int             hasoutofband;
    mblk_t          *urg_msg;
    int             sndbuf;
    int             rcvbuf;
    int             sndlowat;
    int             rcvlowat;
    int             linger;
    int             sndtimeo;
    int             rcvtimeo;
    int             prototype;
    int             esbcnt;
};
```

The STREAMS Socket Record (sys/sockmod.h:131-159)

The module itself is a STREAMS component with standard read and write queue initializers, exposed via the `sockinfo` streamtab (io/sockmod.c:183-220). This is the physical switchboard, wired into the STREAMS fabric.

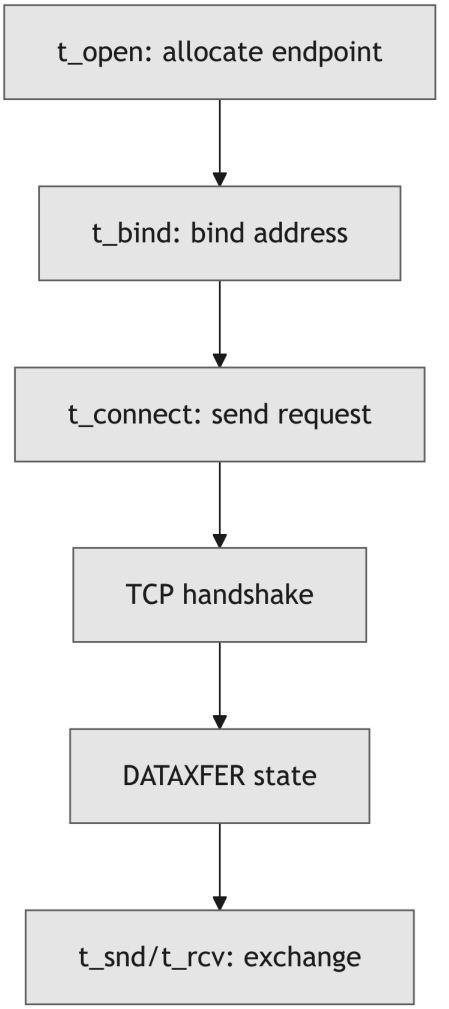


Figure 4.2.2: TLI Connection Semantics Used by `sockmod`

The Call Path

A socket call from user space becomes a message into this layered world:

1. **User calls `socket()`** and receives a file descriptor.
2. **`sockmod` attaches** and records `si_udata` (type, options, service type).
3. **`pr_usrreq` dispatches** the request to the protocol implementation.
4. **STREAMS queues carry data**, translating between socket semantics and TPI primitives.

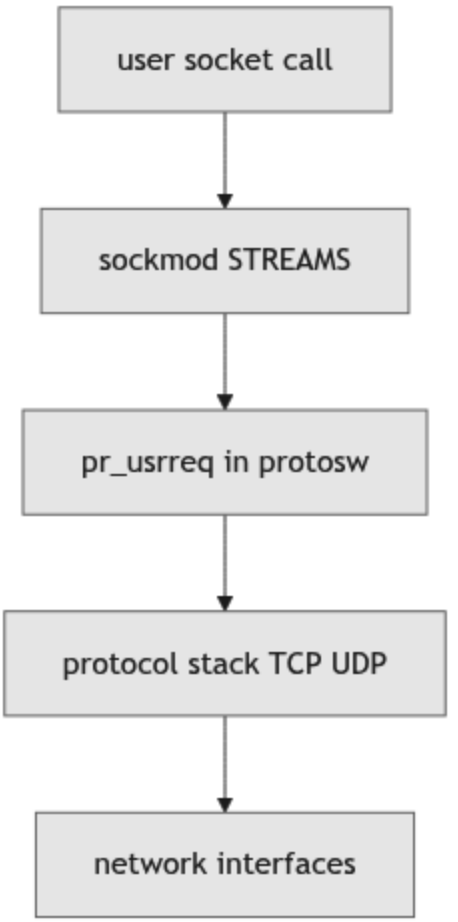


Figure 4.2.5: Socket Requests Across STREAMS and Protocols

This path is less direct than a single call into TCP, but it preserves the uniform STREAMS architecture of SVR4 while offering the familiar socket API to applications.

The Ghost of SVR4: Our sockets lived in a STREAMS world. We translated BSD calls into TPI messages and let the queues mediate flow. In your era the abstraction still exists, but it is thinner: native sockets with epoll, kqueue, and io_uring provide direct paths that we could only imagine. Yet the ledger remains: a socket still has state, buffers, and a protocol switch, and the switchboard clerk still decides which line rings.

The Switchboard at Dusk

The socket layer in SVR4 is a careful mediation between users and protocols. It defines the contract, maintains the ledger of each connection, and dispatches requests to the protocol switchboard while honoring the STREAMS architecture. The calls that enter the exchange depart with a clear destination, and the clerk’s ledger keeps every line in order.

Internet Protocol (IP)

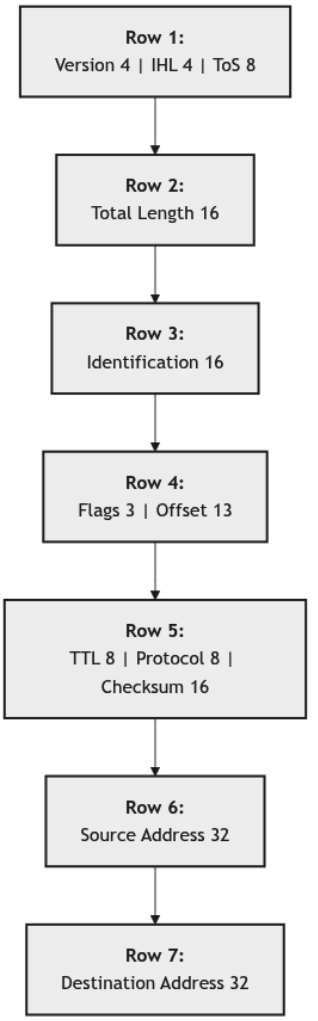
Overview: The Imperial Cartographers and Messengers

At the foundation of the kernel's networking suite lies the Internet Protocol (IP), the Empire's vast and intricate network of cartographers and messengers. This is the service responsible for the fundamental task of packet delivery across the myriad of networks that constitute the internetwork. Unlike the meticulous, reliable service of the TCP Telegraph Office or the simple, fire-and-forget Postal Service of UDP, the IP layer is concerned with a single, vital question: given a letter with a destination address, what is the next step on its journey?

The cartographers of this service are the routing daemons, who tirelessly work to maintain the kernel's routing tables—the master maps of the internetwork. The messengers are the IP software itself, which consults these maps to make a hop-by-hop forwarding decision for each and every packet. It is a 'best-effort' service; the messengers make no guarantee of delivery, nor do they ensure that packets will arrive in order. Their sole duty is to read the address on the envelope and send the letter on the next leg of its journey, a journey that may take it across countless networks and through the heart of many intermediate gateways.

The Letter's Envelope: The IP Header

Every packet that travels across the internetwork is enclosed in an IP header, the envelope that contains the addressing and control information necessary for its delivery.



The IP Header

The key fields on this envelope are:

- **Version and IHL (Internet Header Length)**: Identifies this as an IPv4 packet and specifies the length of the header.
- **Total Length**: The total length of the IP packet (header and data).
- **Identification, Flags, and Fragment Offset**: These fields are used for the fragmentation and reassembly of large packets.
- **Time to Live (TTL)**: A counter that is decremented at each hop. If the TTL reaches zero, the packet is discarded, preventing it from looping endlessly through the network.
- **Protocol**: Identifies the higher-level protocol (e.g., TCP or UDP) to which the packet's data should be delivered at the final destination.
- **Header Checksum**: A checksum to verify the integrity of the IP header.

- **Source and Destination Addresses**: The 32-bit IPv4 addresses of the sender and receiver, the fundamental information used for routing.

Routing: The Cartographer's Maps

The core function of the IP layer is routing. When a packet needs to be sent, either from a local application or being forwarded from another machine, the IP layer must consult its routing tables to determine the appropriate next hop. This process is handled by the `rtalloc` function, which searches the kernel's routing tables for the best route to the destination address.

The `ip_output` function in `ip_output.c` is responsible for sending IP packets. It uses `rtalloc` to find the correct route and the corresponding network interface. Once the next hop and the outgoing interface are known, the packet is passed down to the link layer for transmission.

For incoming packets, the `ipintr` function in `ip_input.c` is the main entry point. If the packet is not for the local machine, and if the `ipforwarding` flag is set, `ip_forward` is called to route the packet to its next destination.

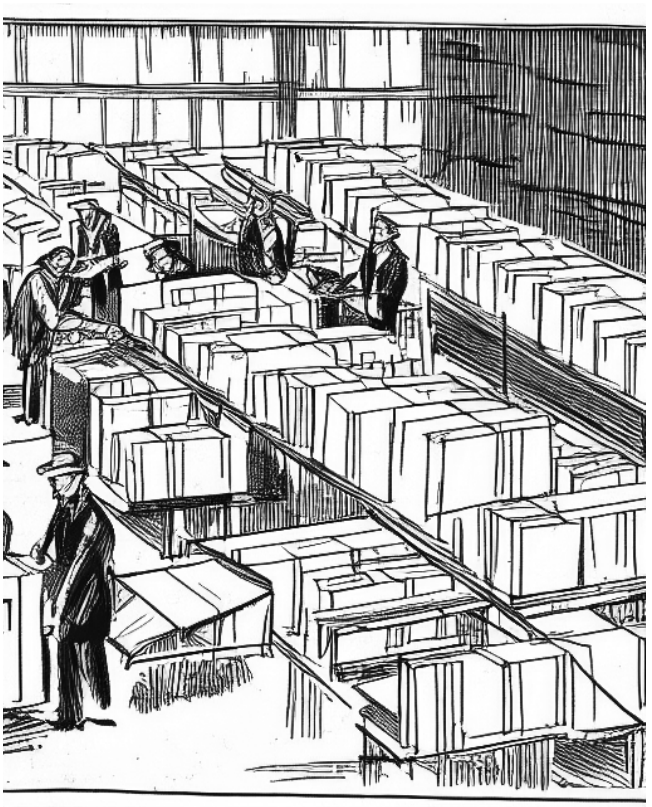
Fragmentation and Reassembly: The Scribe's Office

When a packet is too large to be transmitted across a particular network (i.e., it exceeds the network's Maximum Transmission Unit, or MTU), the IP layer must break it into smaller pieces, a process known as fragmentation. Each fragment is a valid IP packet in its own right, with its own IP header. The `Identification`, `Flags`, and `Fragment Offset` fields in the header are used to track the fragments and reassemble them at the final destination.

The `ip_reass` function in `ip_input.c` is responsible for reassembling fragments. When a fragment arrives, it is placed in a queue associated with its source address, destination address, protocol, and identification number. When all fragments of a datagram have arrived, `ip_reass` concatenates them into the original datagram and passes it up to the appropriate higher-level protocol. The `ipq` structure is used to manage the reassembly queues, and a timer in `ip_slowtimo` ensures that stale, incomplete fragments are eventually discarded.

The Ghost of SVR4:

“We lived in the age of IPv4, a system of addresses that seemed vast and inexhaustible at the time. We could not have imagined a world where every lightbulb and every teacup would demand its own unique address. Your IPv6, with its impossibly large address space, is a solution to a problem we could barely conceive of. And with it, you have made other, more subtle changes. You have eliminated the header checksum, a small but significant concession to speed, placing your trust in the reliability of the underlying link layers. Most profoundly, you have all but abandoned in-network fragmentation. The routers of your time are busy, overworked creatures, and you have relieved them of the burden of breaking up oversized packets. In your world, it is the sender’s responsibility to ensure that its packets are appropriately sized for the path they will travel, a shift in philosophy that we, in our time, would have found most curious.”



IP Protocol - Postal Routing Center

Conclusion

The Internet Protocol is the linchpin of the SVR4 networking stack, the fundamental delivery service that makes all other network communication possible. It is the realm of the cartographers and the messengers, a world of maps and routes, of forwarding and fragmentation. While it offers no guarantees, its best-effort delivery model has proven to be a remarkably robust and scalable foundation for the internetwork. It is the service that turns a collection of disparate networks into a cohesive whole, allowing a letter from any corner of the Empire to begin its journey to any other, one hop at a time.

Transport Control Protocol (TCP)

Overview: The Empire's Telegraph Office

Imagine a vast, bustling telegraph office in the heart of a sprawling Victorian empire. This is not a chaotic operation of haphazardly transmitted messages, but a meticulously organized system dedicated to the reliable delivery of every dispatch. This is the world of the Transmission Control Protocol (TCP) in SVR4. Where the underlying postal service (the Internet Protocol) offers no guarantees—letters may be lost, arrive out of order, or be duplicated—the TCP Telegraph Office takes upon itself the solemn duty of ensuring that every word, every sentence, every message arrives at its destination precisely as it was sent.

Each conversation is a dedicated correspondence between two parties, managed by a senior clerk who maintains a detailed ledger. This ledger, the TCP Control Block, tracks every byte sent and received, ensuring that lost packets are retransmitted, duplicates are discarded, and a constant, orderly flow of information is maintained. The office operates under a strict set of rules—the TCP Finite State Machine—governing the lifecycle of each connection, from the formal three-way handshake that initiates a conversation to the final, four-part farewell that concludes it. This is a world of sequence numbers, acknowledgements, windows, and timers, all working in concert to create a reliable stream of communication over an inherently unreliable medium.

The Clerk's Ledger: The TCP Control Block

At the heart of every TCP connection is the `tcpcb`, the TCP Control Block. This structure is the clerk's master ledger for a single conversation, a comprehensive record of everything needed to maintain a reliable, ordered stream of data.

```
/*
 * Kernel variables for tcp.
 */

/*
 * Tcp control block, one per tcp; fields:
 */
struct tcpcb {
    struct      tcpiphdr *seg_next;     /* sequencing queue */
    short       t_state;                /* state of this connection */
    short       t_timer[TCPT_NTIMERS];  /* tcp timers */
    short       t_rxtshift;             /* log(2) of rexmt exp. backoff */
    short       t_rxtcur;               /* current retransmit value */
    u_short     t_maxseg;               /* maximum segment size */
    char        t_force;                /* 1 if forcing out a byte */
    u_short     t_flags;

/* send sequence variables */
    tcp_seq     snd_una;                /* send unacknowledged */
    tcp_seq     snd_nxt;                /* send next */
    tcp_seq     snd_up;                 /* send urgent pointer */
    tcp_seq     snd_wl1;                /* window update seg seq number */
    tcp_seq     snd_wl2;                /* window update seg ack number */
    tcp_seq     iss;                    /* initial send sequence number */
    u_long      snd_wnd;                /* send window */
/* receive sequence variables */
    u_long      rcv_wnd;                /* receive window */
    tcp_seq     rcv_nxt;                /* receive next */
    tcp_seq     rcv_up;                 /* receive urgent pointer */
    tcp_seq     irs;                    /* initial receive sequence number
*/
};
```

(netinet/tcp_var.h)

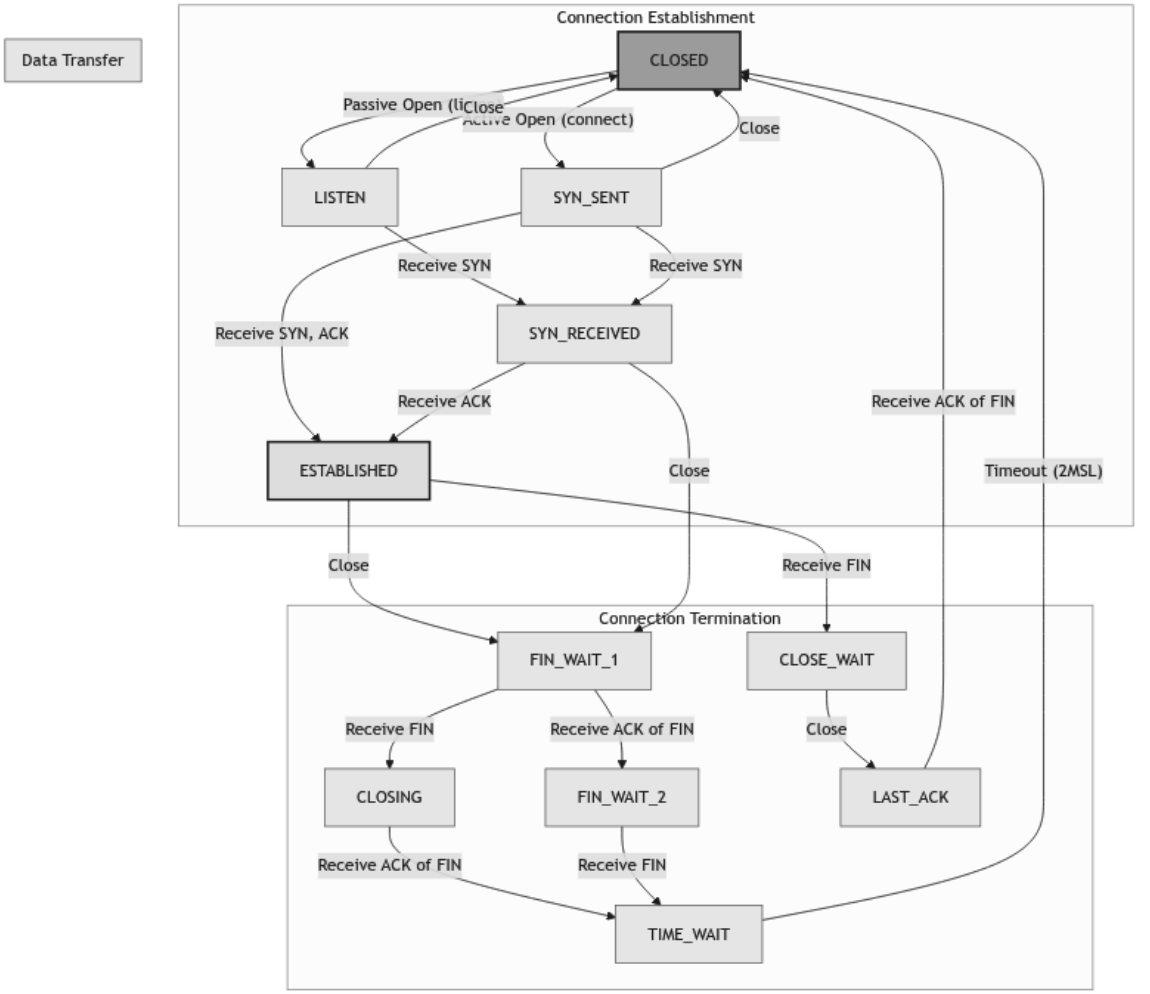
Key fields in this ledger include:

- **`t_state`** : The current state of the connection, drawn from the finite state machine.
- **`snd_una`** , **`snd_nxt`** : Sequence numbers that track the bytes that have been sent and acknowledged.
- **`rcv_nxt`** : The sequence number of the next byte the receiver expects to receive.
- **`snd_wnd`** , **`rcv_wnd`** : The send and receive windows, which manage flow control.
- **`t_timer`** : An array of timers for retransmissions, keep-alives, and other time-based events.

- **`t_maxseg`** : The maximum segment size, negotiated during the handshake, to avoid IP-level fragmentation.

The Rulebook: The TCP Finite State Machine

The life of a TCP connection is governed by a strict set of rules known as the Finite State Machine. Each state represents a distinct phase in the conversation, and the transitions between states are triggered by specific events, such as the arrival of a particular type of segment or a request from the user application.



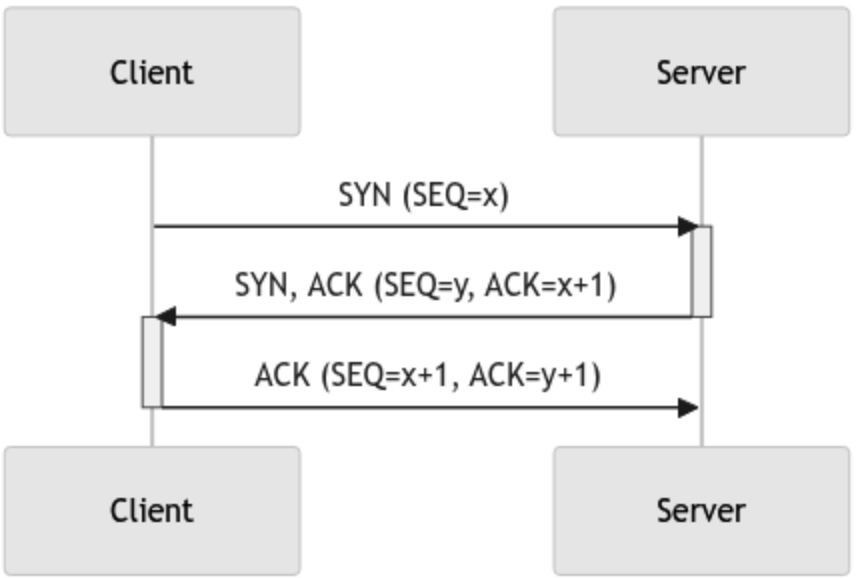
The TCP State Machine

The primary states and their roles:

- **`CLOSED`** : The beginning and end. No connection exists.
- **`LISTEN`** : A server waiting for a connection request from a client.
- **`SYN_SENT`** : A client has sent a connection request (a `SYN` segment) and is waiting for a reply.
- **`SYN_RECEIVED`** : A server has received a `SYN` and sent its own `SYN-ACK`, and is now waiting for the final `ACK` from the client.
- **`ESTABLISHED`** : The connection is active. Data can be transferred freely in both directions. This is the main state for a healthy, active connection.
- **`FIN_WAIT_1`** / **`FIN_WAIT_2`** : The local application has closed the connection, and the TCP module has sent a `FIN` segment. It is waiting for an `ACK` and then a `FIN` from the remote party.
- **`CLOSE_WAIT`** : The local TCP has received a `FIN` from the remote party and is waiting for the local application to close its end of the connection.
- **`LAST_ACK`** : Both sides have sent `FIN`s, and the TCP is waiting for the final `ACK` of its own `FIN`.
- **`TIME_WAIT`** : The “2MSL Wait” state. The connection is closed, but the control block is kept around for a short period to catch any stray, late-arriving packets from the now-defunct connection, preventing them from being misinterpreted as belonging to a new connection.

Opening the Line: The Three-Way Handshake

A TCP connection is established through a precise, three-step process known as the three-way handshake. This ensures that both parties are ready to communicate and have agreed upon the initial sequence numbers.



The Three-Way Handshake

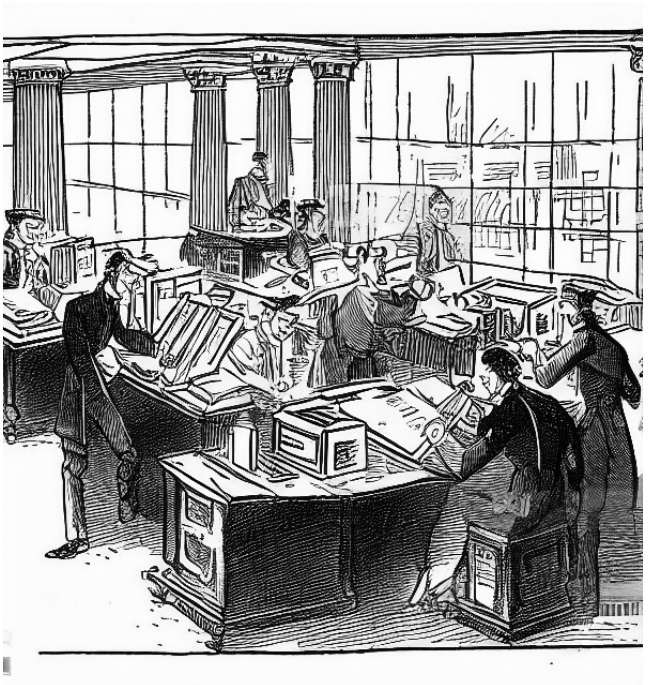
1. **SYN**: The client, wishing to start a conversation, sends a `SYN` (synchronize) segment to the server. This segment includes the client's initial sequence number (`SEQ=x`).
2. **SYN-ACK**: The server, having received the `SYN` , responds with a `SYN-ACK` segment. This segment contains the server's own initial sequence number (`SEQ=y`) and an acknowledgement of the client's sequence number (`ACK=x+1`).
3. **ACK**: Finally, the client sends an `ACK` segment back to the server, acknowledging the server's sequence number (`ACK=y+1`).

Upon completion of this exchange, the connection is `ESTABLISHED` , and data transfer can begin. The SVR4 implementation of this logic is primarily handled in `tcp_input.c` , within the `tcp_uinput` function, which processes incoming segments and drives state transitions.

The Ghost of SVR4:

"Ah, the simple, elegant dance of the three-way handshake. It was a robust and reliable method in its day, and remains the foundation of TCP connections even now, in your far-flung future of 2026. But the world has grown so much faster, so much more concerned with latency. Your modern TCP stacks have learned new tricks. 'TCP Fast Open' (TFO), for instance, allows data to be sent in the very first `SYN` packet, a brazen violation of the old

etiquette, all in the name of shaving off a few precious milliseconds. We, in our time, valued correctness and a clear separation of concerns above all. The handshake was for synchronization, and only once the line was declared open would data ever be transmitted. A more civilized age, perhaps.”



TCP - Telegraph Office

Reliable Delivery and Flow Control

TCP’s primary mandate is reliability. It achieves this through several key mechanisms:

- **Sequence Numbers**: Every byte of data is assigned a unique sequence number. The `snd_nxt` and `rcv_nxt` fields in the `tcpcb` track the progress of this byte stream.
- **Acknowledgements (ACKs)**: The receiver sends `ACK` segments back to the sender to confirm receipt of data. If the sender does not receive an `ACK` for a segment within a certain time (the retransmission timeout, or RTO), it assumes the segment was lost and sends it again. The SVR4 kernel manages this with a set of timers, handled in `tcp_timer.c`.
- **Sliding Window**: The `snd_wnd` and `rcv_wnd` fields implement a “sliding window” for flow control. The receiver advertises its `rcv_wnd` —the amount of buffer space it has

available. The sender is not allowed to send more data than the advertised window, preventing the receiver from being overwhelmed. The logic for sending data, respecting the window, and handling retransmissions is found in `tcp_output.c`.

Conclusion

The SVR4 TCP implementation, a faithful rendition of the classic protocol, is a testament to the enduring principles of reliable network design. Like a well-run telegraph office, it is a system of immense complexity, with clerks, rules, and ledgers all working in harmony. It imposes order on the chaos of the underlying network, providing applications with the simple, reassuring abstraction of a reliable, unbroken stream of data. While the speeds and latencies of the networks have changed beyond recognition, the fundamental challenges of reliability that TCP was designed to solve remain, and its core principles, as implemented in SVR4, endure.

User Datagram Protocol (UDP)

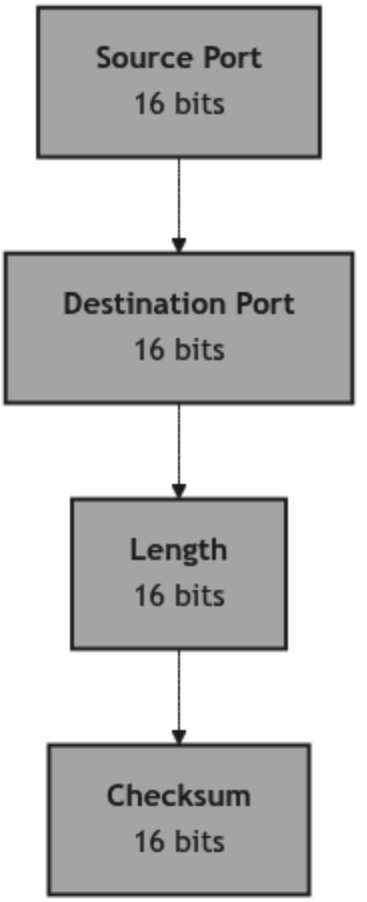
Overview: The Empire’s Postal Service

In contrast to the formal, registered mail service of the TCP Telegraph Office, the User Datagram Protocol (UDP) is the Empire’s simple, no-frills postal service. It is a service built on a philosophy of speed and simplicity, making no promises about the delivery of your letters. You write your message, drop it into the nearest postbox, and hope for the best.

There is no formal handshake to initiate a correspondence, no sequence numbers to ensure letters arrive in order, and no acknowledgements to confirm their receipt. If a letter is lost to the void, duplicated by a confused postal worker, or arrives after a later dispatch, the postal service offers no apology and takes no corrective action. It is a ‘fire-and-forget’ system, where the burden of reliability rests entirely on the shoulders of the sender and receiver. This makes it unsuitable for transmitting the chapters of a novel, but perfectly adequate for a quick, informal note where speed is of the essence and the loss of a single message is not catastrophic.

The UDP Header: A Simple Postcard

The UDP header is a model of minimalism, a simple postcard containing only the most essential information required to get the message to its destination. It is a mere 8 bytes, a fraction of the size of its verbose TCP counterpart.



The UDP Header

The header contains just four fields:

- **Source Port** and **Destination Port**: The addresses of the sender and receiver within their respective machines.
- **Length**: The length of the UDP header and its data.
- **Checksum**: An optional field to verify the integrity of the header and data. In the SVR4 era, the use of this checksum was often disabled by default to squeeze out every last drop of performance.

This simplicity is UDP's greatest strength and its most profound weakness. The lack of sequence numbers, acknowledgement numbers, and window size fields is what makes UDP so fast and lightweight, but it is also what makes it an unreliable, connectionless protocol.

The `inpcb` and UDP

Like TCP, UDP uses the `inpcb` (Internet Protocol Control Block) to manage its endpoints. However, because UDP is connectionless, the `inpcb` is used in a much simpler way. It primarily serves to bind a user's application to a specific port, a postbox waiting to receive mail. It does not track state, sequence numbers, or window sizes, as these concepts are foreign to the world of UDP.

The core of the UDP implementation in SVR4 can be found in `udp_io.c` and `udp_main.c`. These files contain the logic for handling incoming and outgoing datagrams.

`udp_input` and `udp_output`: The Mailroom Clerks

The two key functions in the SVR4 UDP implementation are `udp_input` and `udp_output`.

- **`udp_input`**: This function is the receiving mailroom clerk. When a UDP datagram arrives from the IP layer, `udp_input` is called. It performs a few basic checks, such as verifying the checksum if enabled, and then looks for an `inpcb` associated with the destination port. If it finds one, it places the datagram in the appropriate application's queue and moves on. If no application is listening on that port, the datagram is simply discarded, much like a letter with an unknown address.
- **`udp_output`**: This function is the dispatching clerk. When an application sends a UDP datagram, `udp_output` takes the data, prepends the simple 8-byte UDP header, and passes the resulting package down to the IP layer for delivery. It makes no copy of the data and sets no timers. Once the datagram is handed off to IP, UDP's involvement is over.

The Ghost of SVR4:

"We saw UDP as a necessary, if somewhat crude, tool. It was for services like DNS, where a single, small query and response were all that was needed. The idea of building complex, reliable protocols on top of our simple postal service would have seemed a curious, if not foolhardy, endeavor. Yet, in your time, this is precisely what has happened. You have built a new, impossibly fast telegraph service called QUIC, not on the solid foundation of IP, but on the shifting sands of UDP. You have taken our fire-and-forget postal system and, through

herculean effort, layered on your own reliability, congestion control, and even encryption. It is a marvel of engineering, a testament to the relentless pursuit of speed. You have built a palace on a swamp, and somehow, it stands.”

UDP - Simple Postal Service

Conclusion

The User Datagram Protocol in SVR4 is a study in contrasts to the complexity of TCP. It is the swift, efficient, and fundamentally unreliable postal service of the kernel’s networking suite. It offers no guarantees, but in doing so, it provides a raw, direct path to the underlying network, a path that applications requiring speed and low overhead, and who are willing to shoulder the burden of reliability themselves, have found to be invaluable. It is a simple tool for a simple job, a reminder that sometimes, the fastest way to send a message is to simply drop it in the mail and hope for the best.

Network Interfaces: The Registry, the Cartographer, and the Gate

A city that thrives on trade keeps more than docks; it keeps a registry of streets, property lines, and authorized crossings. The registry clerk does not handle cargo, yet without the registry no one knows where a road begins or which bridge is permitted for heavy wagons. The cartographer draws the boundaries, the clerk records names and addresses, and the gatekeeper checks each pass before it opens.

SVR4's network interface layer is that registry. Drivers deliver frames, protocols demand routes, and user programs issue ioctl orders, but the interface subsystem records the names, addresses, and capabilities that make those actions coherent. It is the layer that ties a physical interface to its identity on the network and teaches the routing table which gate to open.

The Interface Register: Names, Units, and Addresses

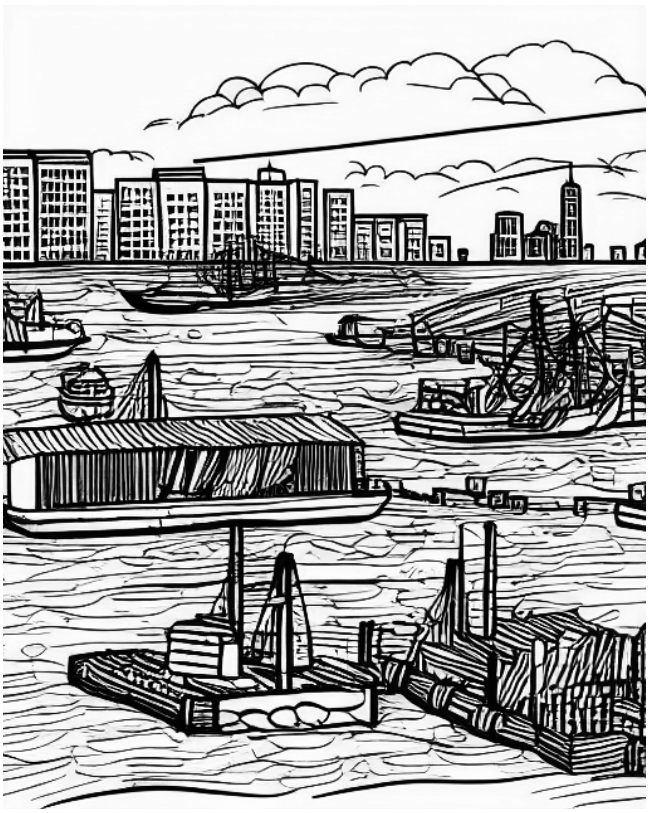
Every interface is named by `if_name` and `if_unit`, and those identifiers are used throughout the stack to locate it. The kernel maintains a linked list of `ifnet` structures, and address records are attached to each interface through `ifaddr` (net/if.h:241-252).

```
struct ifaddr {
    struct  sockaddr ifa_addr;   /* address of interface */
    union {
        struct  sockaddr ifu_broadaddr;
        struct  sockaddr ifu_dstaddr;
    } ifa_ifu;
    struct  ifnet *ifa_ifp;      /* back-pointer to interface */
    struct  ifstats *ifa_ifs;    /* back-pointer to interface stats */
    struct  ifaddr *ifa_next;    /* next address for interface */
};
```

The Address Record (net/if.h:241-251)

This structure binds a protocol address to a concrete interface. The presence of `ifa_ifp` is crucial: it lets routing code and protocol code jump back from an address to the interface that owns it. Multiple addresses can be chained for one interface, which is how aliases and multiple protocol families coexist.

Interface discovery functions such as `ifunit()` and `ifa_ifwithaddr()` are declared alongside the global `ifnet` list (net/if.h:301-307). They are the registry clerk's index cards: given a name or address, return the matching interface.



Network Interfaces - City Ports

Flags and Civic Status

An interface also carries a civic status, recorded as flags in `ifnet`. These flags tell the stack whether a link is up, looped back, broadcasting, or operating in promiscuous mode (net/if.h:133-144).

```
#define IFF_UP        0x1   /* interface is up */
#define IFF_BROADCAST 0x2   /* broadcast address valid */
#define IFF_LOOPBACK  0x8   /* is a loopback net */
#define IFF_POINTOPOINT 0x10 /* point-to-point link */
#define IFF_RUNNING   0x40  /* resources allocated */
#define IFF_PROMISC   0x100 /* receive all packets */
#define IFF_ALLMULTI  0x200 /* receive all multicast packets */
```

The Civic Flags (net/if.h:133-143, abridged)

These are not mere labels. The routing and ARP layers look to these flags to decide whether to emit broadcasts, whether to trust link-level address resolution, and whether a device is operational. Changing a flag is like moving the city gate from open to closed; the entire network takes note.

The Control Desk: `ifreq` and `ifconf`

User space configures interfaces through ioctls that pass `struct ifreq` and `struct ifconf`. These structures are the formal request slips for naming an interface, setting flags, or providing driver-specific data (net/if.h:255-297).

```
struct ifreq {
    char    ifr_name[IFNAMSIZ];  /* if name, e.g. "emd1" */
    union {
        struct  sockaddr ifru_addr;
        struct  sockaddr ifru_dstaddr;
        char    ifru_oname[IFNAMSIZ];
        struct  sockaddr ifru_broadaddr;
        short   ifru_flags;
        int     ifru_metric;
        char    ifru_data[1];
        char    ifru_enaddr[6];
    } ifr_ifru;
};
```

The Interface Request Slip (net/if.h:260-281)

`ifconf` wraps a buffer for querying all configured interfaces, a call that powers tools like `ifconfig -a` in spirit if not yet in name (net/if.h:289-297). The driver handles the details in its

`if_ioctl` routine, but the interface layer defines the paper form and the filing rules.

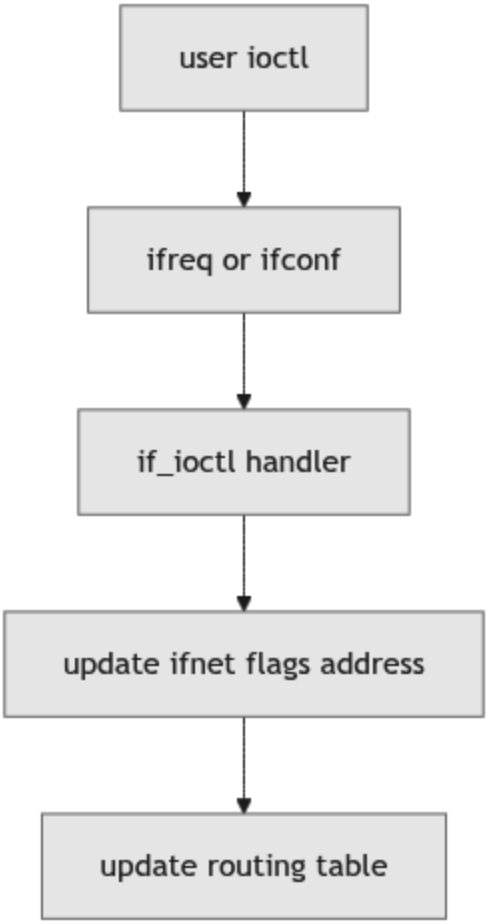


Figure 4.4.1: Control Requests Through `ifreq` and `ifconf`

The Configuration Sweep: `ifconf`

When user space asks for the full list of configured interfaces, it supplies an `ifconf` record with a buffer. The kernel fills it with `ifreq` entries, each one naming an interface and returning a selected address or parameter (net/if.h:289-297).

```
struct ifconf {
    int ifc_len;         /* size of associated buffer */
    union {
        caddr_t ifcu_buf;
        struct ifreq *ifcu_req;
    } ifc_ifcu;
};
```

The Inventory Envelope (net/if.h:289-294)

This is how monitoring tools learn what exists without knowing any device names in advance. The registry clerk simply empties the ledger into a stack of slips and hands it across the counter.

Broadcasts, Point-to-Point, and Address Pairing

The `ifaddr` structure holds either a broadcast address or a destination address, depending on the interface type (net/if.h:242-248). A broadcast-capable Ethernet interface uses `ifa_broadaddr`, while a point-to-point link records `ifa_dstaddr`. The distinction matters to routing and ARP; it tells the stack whether a packet can be fanned out to a local segment or must be sent to a single peer.

The interface flags reinforce this: `IFF_BROADCAST` and `IFF_POINTOPOINT` are mutually exclusive in practice, and the route table uses that knowledge when creating directly connected routes. The map is not just where the roads are, but what kind of road each one is.

The Cartographer's Ledger: Routing Entries

Once an interface has an address, the routing table can map destinations to it. `struct rtentry` stores the destination and the interface pointer to use, effectively linking the cartographer's map to the dock (net/route.h:69-98).

```
struct rtentry {
    u_long  rt_hash;        /* to speed lookups */
    struct  sockaddr rt_dst;/* key */
    struct  sockaddr rt_gateway; /* value */
    short   rt_flags;       /* up/down?, host/net */
    short   rt_refcnt;      /* # held references */
    u_long  rt_use;         /* raw # packets forwarded */
    struct  ifnet *rt_ifp;  /* the answer: interface to use */
};
```

The Routing Ledger Entry (net/route.h:86-97)

The flags `RTF_UP`, `RTF_GATEWAY`, and `RTF_HOST` describe whether the route is usable, indirect, or host-specific (net/route.h:100-105). When a packet needs a path, routing code resolves the destination and returns the `rt_ifp` pointer, which is the gatekeeper's answer: "send it through this interface."

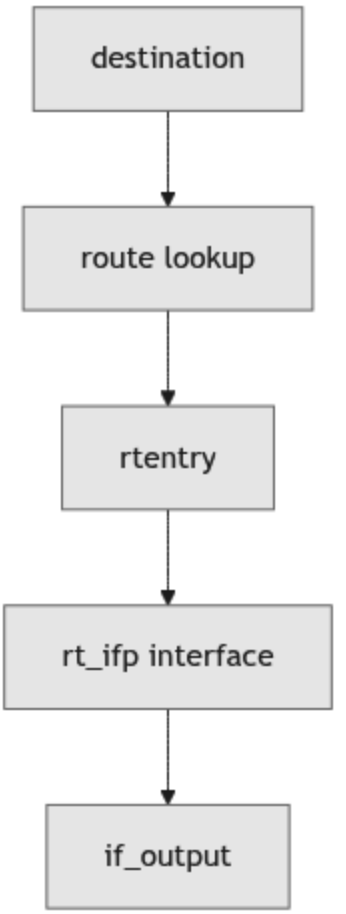


Figure 4.4.2: Destination to Interface via Routing Entry

Gate Permissions: Route Flags and References

Routing entries carry their own permissions, encoded as flags. These flags let the kernel distinguish a direct host route from a gateway route, and they allow dynamic redirects to be tracked (net/route.h:100-105).

```
#define RTF_UP       0x1   /* route useable */
#define RTF_GATEWAY  0x2   /* destination is a gateway */
#define RTF_HOST     0x4   /* host entry (net otherwise) */
#define RTF_DYNAMIC  0x10  /* created dynamically (by redirect) */
#define RTF_MODIFIED 0x20  /* modified dynamically (by redirect) */
```

The Gate Permissions (net/route.h:100-105, abridged)

Each route also tracks `rt_refcnt`, the number of consumers holding a reference to it. The `RTFREE` macro in the kernel decrements that count and frees the route when the last holder lets go (net/route.h:148-163). The gatekeeper cannot close a gate while travelers still hold the key.

The Route Vault: Hash Buckets and Statistics

SVR4 keeps routing entries in hash buckets for hosts and networks. The hash size is fixed at build time, and lookups choose the appropriate bucket by a simple mask or modulo (net/route.h:166-183).

```
#define RTHASHSIZ 8
#define RTHASHMOD(h) ((h) & (RTHASHSIZ - 1))

struct mbuf *rthost[RTHASHSIZ];
struct mbuf *rtnet[RTHASHSIZ];
```

The Route Vault (net/route.h:166-183, abridged)

The hash tables are humble but effective. They allow frequent lookups with low overhead, which matters because every outgoing packet pays this tax. When a lookup fails, the failure is recorded in routing statistics for diagnostics (net/route.h:140-146).

```
struct rtstat {
    short rts_badredirect;
    short rts_dynamic;
    short rts_newgateway;
    short rts_unreach;
    short rts_wildcard;
};
```

The Cartographer's Ledger (net/route.h:140-145)

These counters are the cartographer's logbook: they tell you how often redirects were trusted, how often routes changed, and how many destinations were unreachable.

The Gatekeeper's Duties

The interface layer lives at the junction of three concerns:

- **Naming and identity**: `if_name` and `if_unit` provide stable handles for users and for the kernel.
- **Address binding**: `ifaddr` records bind protocol addresses to interfaces.
- **Routing integration**: `rtentry` points from destinations back to the owning interface.

The result is a coherent map: protocols pick routes, routes point to interfaces, and interfaces are configured by the control desk. Without that registry, the network stack would be all ships and no harbor master.

The Public Ledger: Global Lists and Raw Queues

SVR4 exports a global `ifnet` list and a `rawintrq` input queue to the rest of the kernel (net/if.h:301-303). The list is the registry ledger, while the raw queue provides a tap for tools that want to observe traffic without protocol decoding. Both rely on interfaces being correctly linked, named, and configured.

These globals are not glamorous, but they are the index that makes every lookup possible. A packet arrives, a destination is chosen, and the first step is always the same: find the interface in the public ledger.

> **The Ghost of SVR4:** Our registry was terse but legible. We registered interfaces by name and unit, and we pushed configuration through ioctl slips. Your time keeps the same names but adds a new bureaucracy: netlink sockets, sysfs attributes, and address lifetimes for multiple namespaces. The forms are thicker now, yet the same question remains: which gate should a packet take?

Closing the Ledger

Network interfaces are the kernel's civic records. They translate a device into a named entity, attach addresses to it, and link it into the routing map. Once those entries are in place, the rest of the network can move with certainty, confident that every destination has a known gate and every gate has a known keeper.

Remote Procedure Call (RPC): The Sealed Dispatch

In a world of distant provinces, the Crown cannot send couriers for every query. It sends sealed dispatches instead: a standard envelope with a wax seal, a catalog number, and a body that can be read in any court. Remote Procedure Call is that envelope. It carries the name of the service, the procedure number, and the proof of identity, so that a server can execute a request as if the caller were present.

SVR4's kernel RPC is the protocol engine beneath NFS. It defines the message format, authenticators, and XDR serialization routines that allow the rest of the network stack to treat procedure calls as a reliable ritual.

The Dispatch Form: `struct rpc_msg`

Every RPC request or reply is wrapped in a single `rpc_msg` structure with a transaction ID and a direction flag (rpc/rpc_msg.h:146-155). The call and reply bodies share the same envelope, but only one is populated at a time.

```
struct rpc_msg {
    u_long                      rm_xid;
    enum msg_type       rm_direction;
    union {
        struct call_body RM_cmb;
        struct reply_body RM_rmb;
    } ru;
#define rm_call         ru.RM_cmb
#define rm_reply        ru.RM_rmb
};
```

The Wax-Sealed Envelope (rpc/rpc_msg.h:146-155)

The call body records the program, version, procedure number, and two authentication blocks (rpc/rpc_msg.h:134-141). The reply body carries either an accepted or rejected reply, with a reason

code in both cases (rpc/rpc_msg.h:84-129). These enums (`accept_stat` , `reject_stat`) are the official stamps for success, mismatch, or authentication failure (rpc/rpc_msg.h:61-73).

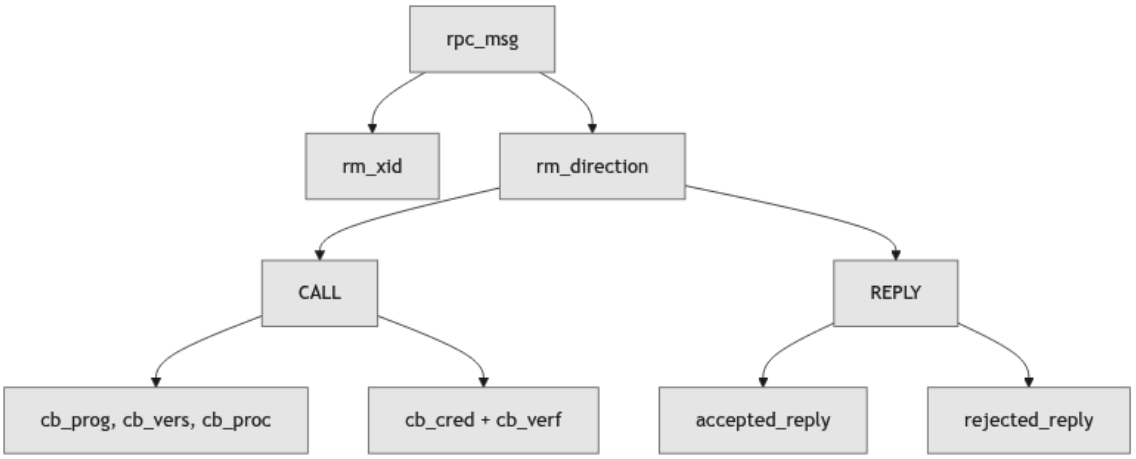


Figure 4.5.1: Call and Reply Payloads in the Shared Envelope

RPC Protocol - Royal Court Dispatches

Versioning and Port Conventions

SVR4's RPC wire format declares its version explicitly: `RPC_MSG_VERSION` is 2 (rpc/rpc_msg.h:42). The subsystem also defines a service port constant for kernel RPC services (rpc/rpc_msg.h:42-43). These constants are the dispatch office's registry. The client and server both know which version they are speaking, and the reply can report `PROG_MISMATCH` or `RPC_MISMATCH` if the versions disagree (rpc/rpc_msg.h:61-72).

This is how the system survives change: the envelope includes the version in every call, and the receiver is required to refuse incompatible requests rather than guess.

The Seals: opaque_auth

Authentication is carried as opaque blobs. The receiver does not interpret the bytes; it only checks the flavor and length. This allows `AUTH_UNIX`, `AUTH_DES`, and other schemes to coexist (rpc/auth.h:83-87).

```
struct opaque_auth {
    enum_t      oa_flavor;
    caddr_t     oa_base;
    u_int       oa_length;
};
```

The Wax Seal (rpc/auth.h:83-87)

The auth handle (`AUTH`) carries both credentials and verifier plus a small set of callbacks to marshal and refresh them (rpc/auth.h:93-105). The RPC layer does not enforce policy; it only presents the seal to the server and reports whether it was accepted.

Writing the Dispatch: `xdr_callmsg()`

RPC relies on XDR for machine-independent encoding. `xdr_callmsg()` serializes the call header, carefully enforcing the maximum authentication size and inserting fields in the exact order required by the standard (rpc/rpc_calmsg.c:65-105).

```
if (xdrs->x_op == XDR_ENCODE) {
    if (cmsg->rm_call.cb_cred.oa_length > MAX_AUTH_BYTES)
        return (FALSE);
    buf = XDR_INLINE(xdrs, 8 * BYTES_PER_XDR_UNIT
        + RNDUP(cmsg->rm_call.cb_cred.oa_length)
        + 2 * BYTES_PER_XDR_UNIT
        + RNDUP(cmsg->rm_call.cb_verf.oa_length));
    if (buf != NULL) {
        IXDR_PUT_LONG(buf, cmsg->rm_xid);
        IXDR_PUT_ENUM(buf, cmsg->rm_direction);
        IXDR_PUT_LONG(buf, cmsg->rm_call.cb_rpcvers);
        IXDR_PUT_LONG(buf, cmsg->rm_call.cb_prog);
        IXDR_PUT_LONG(buf, cmsg->rm_call.cb_vers);
        IXDR_PUT_LONG(buf, cmsg->rm_call.cb_proc);
        ...
    }
}
```

The Scribe’s Hand (rpc/rpc_calmsg.c:65-105, abridged)

The XDR layer is the uniform handwriting that makes a dispatch readable in any court, regardless of endian or alignment.

Accepted, Rejected, and Denied

The reply half of `rpc_msg` is explicit about outcomes. `MSG_ACCEPTED` still carries a status code; `SUCCESS` is only one of several outcomes (rpc/rpc_msg.h:56-68). Rejections can be version mismatches or authentication failures (rpc/rpc_msg.h:70-73). This is the protocol’s insistence on ceremony: even an error must be wrapped in the official form, so the caller can diagnose which part of the dispatch failed.

The kernel uses `xdr_replymsg()` to encode this reply body (rpc/rpc_msg.h:176-182). In practice, NFS servers and other services build these replies in the same XDR dialect as calls, and the same transaction ID ties the response to its request.

Streams in the Courtyard

Kernel RPC runs over STREAMS. Before establishing its conversation, it may pop the `timod` module off the stream if it is not needed, using `I_FIND` and `I_POP` ioctls (rpc/rpc_subr.c:56-71). This is a small but telling detail: the RPC layer assumes responsibility for the exact shape of the transport stack beneath it.

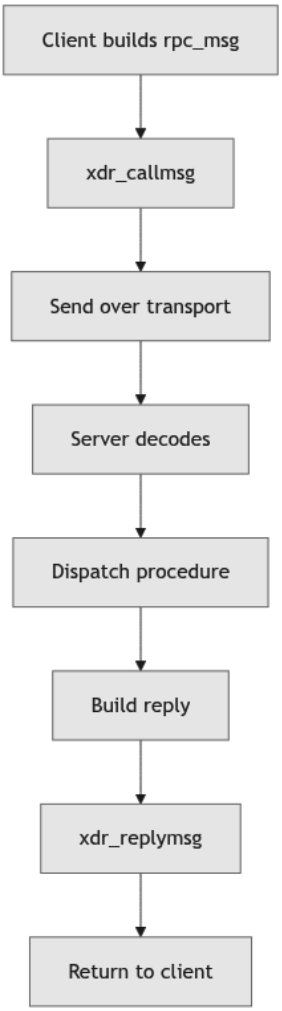


Figure 4.5.2: RPC Call, Dispatch, and Reply Path

The Ghost of SVR4:

We assumed the wire was mostly friendly and that a simple seal was enough. Today you wrap calls in TLS, enforce mutual authentication, and track replay windows as a matter of course. Yet you still send the same envelope: transaction ID, program, version, procedure. The wax has changed, but the dispatch is the same.

Conclusion

RPC is a ceremony. It defines the envelope, the seals, and the script by which a procedure call becomes a wire message and returns as a reply. NFS and its companions depend on that ceremony, and the kernel’s RPC layer keeps it faithful.

NFS Server

Overview: The Colonial Office

If the NFS client is the Foreign Office, then the NFS server is the Colonial Office, the powerful administrative body responsible for governing the Empire's overseas territories and granting access to accredited foreign representatives. The Colonial Office is the guardian of the local filesystem, serving its data to the network while strictly enforcing the access policies of the Crown.

The office receives a constant stream of diplomatic cables (RPC requests) from Foreign Offices around the world. Each cable is a request to perform some action in one of the Crown's territories: to read a document, to establish a new outpost, or to consult the public records. The clerks of the Colonial Office (the `nfsd` daemons) are responsible for processing these requests, but only after carefully vetting the credentials of the representative who sent them.

The `exports` Table: The List of Crown Territories

Not all of the Empire's territories are open to foreigners. The `/etc/exports` file is the official list of Crown Territories—the filesystems that are made available to the network. For each territory, the list specifies the rules of engagement: which foreign powers are granted access, and what level of access they are permitted.

In the kernel, this information is stored in a list of `exportinfo` structures. When a request arrives, the server consults this list to ensure that the requested file or directory is part of an exported filesystem and that the client has the right to access it. The `findexport` function in `nfs_export.c` is responsible for this lookup.

The `nfsd` Daemons: The Colonial Office Clerks

The `nfsd` daemons are the tireless clerks of the Colonial Office. These are user-level processes that are created at system startup and then enter the kernel to do their work. They are the ones who actually process the incoming RPC requests.

When a request arrives, it is handed off to one of the waiting `nfsd` clerks. The clerk decodes the request, performs the requested operation by calling the appropriate VFS and filesystem functions, and then sends a reply back to the client. The `rfs_dispatch` function in `nfs_server.c` is the central dispatch routine that directs the incoming requests to the appropriate service procedures.

Authentication: Checking Credentials

Before any request is processed, the Colonial Office must first check the credentials of the diplomat who sent it. In the world of NFSv2, this is a relatively simple affair. The most common form of authentication is `AUTH_UNIX`, where the client simply asserts the user and group IDs of the user making the request. The server, for the most part, trusts the client to be telling the truth.

SVR4 also supports the more secure `AUTH_DES`, which uses DES encryption to provide a much stronger guarantee of the client's identity. The `checkauth` function in `nfs_server.c` is responsible for this critical step.

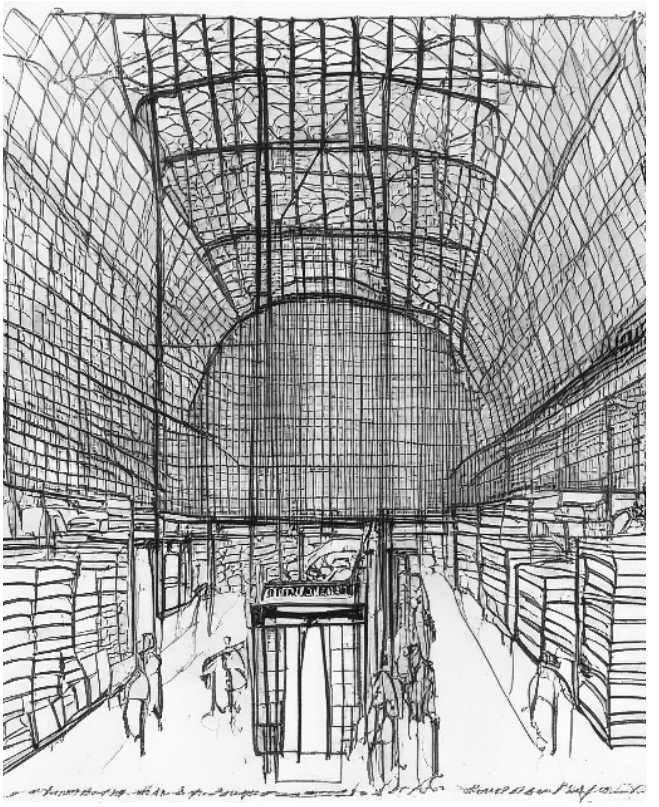
File Handles: The Queen's Seal

To access a file in one of the Crown's territories, a foreign representative must present a document bearing the Queen's Seal. This is the file handle, an opaque identifier that uniquely identifies a file or directory on the server. The file handle is constructed from the filesystem ID and the file's inode number and generation number.

When a client performs a lookup on a file, the server returns a file handle. The client then uses this file handle in all subsequent requests for that file. The `makefh` function in `nfs_export.c` is responsible for creating these seals of authenticity.

The Ghost of SVR4:

“Our Colonial Office was a product of its time, a time of open protocols and assumed trust. We could not have foreseen a world where the network would be such a hostile place. Your modern storage systems, with their end-to-end encryption and their complex, multi-factor authentication schemes, are a response to this new reality. And our simple model of exporting filesystems has been supplanted by new paradigms. Your object stores, with their simple `PUT` and `GET` operations, and your modern, clustered filesystems, which distribute data across hundreds or even thousands of machines, are a far cry from our humble `nfsd` s, serving up files from a single, monolithic kernel.”



NFS Server - Central Warehouse

Conclusion

The NFS server in SVR4 is a classic example of a client-server system, a well-defined protocol for sharing files across a network. It is the Colonial Office to the client's Foreign Office, the guardian of the local filesystem, and the gatekeeper to its resources. While its security model may be dated and its performance limited by its stateless design, it was a robust and reliable system that laid the foundation for the distributed computing environments of the future.

Network Device Drivers: The Harbor, the Dock, and the Signal Lamps

Stand on a stone quay at dawn, watching the harbor wake. Ships idle in the fog, dockworkers prepare coils of rope, and a signal tower lifts its shutters to the incoming traffic. Each vessel carries goods for the city, but the city cannot speak to the ships directly; it speaks through the dock, the ropes, and the lamps. The harbor is not the ship, nor the city, but the disciplined interface between them.

SVR4's network device drivers are that harbor. Protocols in the kernel have packets to send and frames to receive, but they depend on a strict set of routines to move those frames onto copper and back. The driver is the dock's foreman, and the kernel is the harbor master: one dispatches ships, the other keeps the ledger.

The Dock Ledger: `struct ifnet`

The driver's contract with the rest of the networking stack is expressed by `struct ifnet` in `net/if.h` (net/if.h:88-131). This structure names the interface, defines its capabilities, and provides the function pointers that higher layers call to transmit or control it.

```
struct ifnet {
    char    *if_name;        /* name, e.g. ``emd'' or ``lo'' */
    short   if_unit;         /* sub-unit for lower level driver */
    short   if_mtu;          /* maximum transmission unit */
    short   if_flags;        /* up/down, broadcast, etc. */
    short   if_timer;        /* time 'til if_watchdog called */
    u_short if_promisc;      /* net # of requests for promisc mode */
    int     if_metric;       /* routing metric (external only) */
    struct  ifaddr *if_addrlist; /* linked list of addresses per if */
    struct  ifqueue {
        struct  mbuf *ifq_head;
        struct  mbuf *ifq_tail;
        int     ifq_len;
        int     ifq_maxlen;
        int     ifq_drops;
    } if_snd;                /* output queue */
    int     (*if_init)();    /* init routine */
    int     (*if_output)();  /* output routine */
    int     (*if_ioctl)();   /* ioctl routine */
    int     (*if_reset)();   /* bus reset routine */
    int     (*if_watchdog)();/* timer routine */
    int     if_ipackets;     /* packets received on interface */
    int     if_ierrors;      /* input errors on interface */
    int     if_opackets;     /* packets sent on interface */
    int     if_oerrors;      /* output errors on interface */
    int     if_collisions;   /* collisions on csma interfaces */
    struct  ifnet *if_next;
    struct  ifnet *if_upper; /* next layer up */
    struct  ifnet *if_lower; /* next layer down */
    int     (*if_input)();   /* input routine */
    int     (*if_ctlin)();   /* control input routine */
    int     (*if_ctlout)();  /* control output routine */
};
```

The Dock Ledger Structure (net/if.h:93-127, abridged)

Several details matter for drivers:

- **`if_output`** is the outbound call site. The comment above the structure defines its signature: `(*ifp->if_output)(ifp, m, dst)` (net/if.h:70-74).
- **`if_snd`** is a persistent output queue of mbufs. The driver can enqueue and drain it at its own pace.
- **`if_watchdog`** gives a driver a periodic heartbeat, called when `if_timer` expires.

- **Stats fields** (`if_ipackets` , `if_oerrors` , etc.) are maintained by the driver to report traffic and faults.

The structure is a ledger in the literal sense: it documents identity, stores pending cargo, and records each successful or failed voyage.

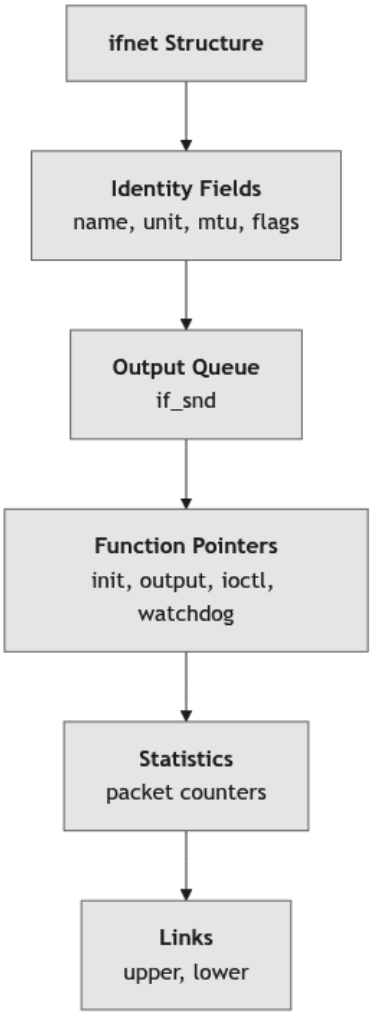
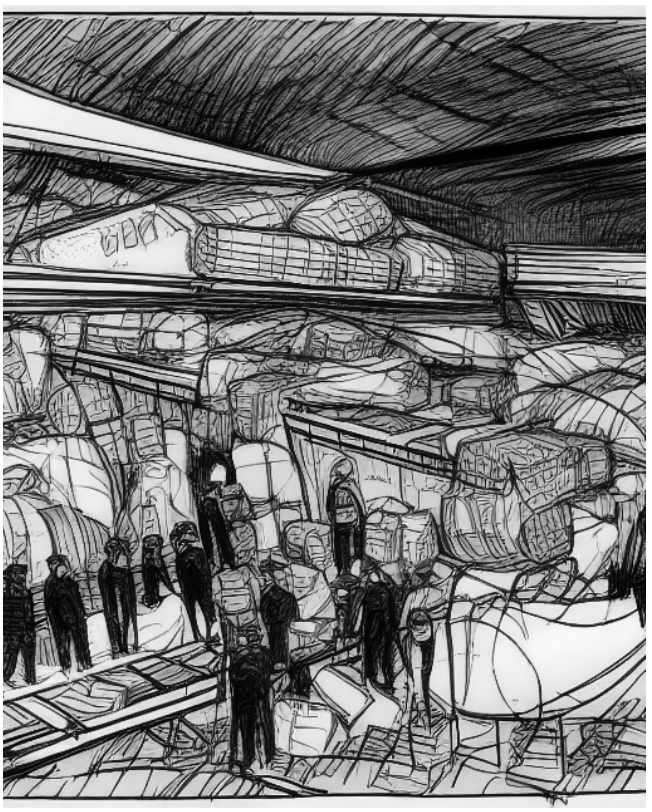


Figure 4.3.1: The Interface Ledger and Its Key Fields



Network Drivers - Dock Workers

The Queue Discipline: `if_snd` and the Macros of Order

SVR4's network drivers are expected to respect a simple, explicit queue discipline. The output queue is managed by macros that enforce length limits and track drops (net/if.h:150-208).

```
#define IF_QFULL(ifq)       ((ifq)->ifq_len >= (ifq)->ifq_maxlen)
#define IF_DROP(ifq)        ((ifq)->ifq_drops++)
#define IF_ENQUEUE(ifq, m) { \
    (m)->m_act = 0; \
    if ((ifq)->ifq_tail == 0) \
        (ifq)->ifq_head = m; \
    else \
        (ifq)->ifq_tail->m_act = m; \
    (ifq)->ifq_tail = m; \
    (ifq)->ifq_len++; \
}
```

The Harbor Queue Rules (net/if.h:156-166)

The macros are not decorative. They define the invariant every driver must respect:

1. **Never exceed `ifq_maxlen`** without incrementing drop counters.
2. **Always maintain the head/tail chain** so packets leave in order.
3. **Always adjust the queue length** so the stack can reason about congestion.

For inbound packets, the interface pointer is prepended to the mbuf chain and later removed with `IF_DEQUEUEIF`, a small ritual that keeps the receiving interface attached to the packet until the upper layers no longer need it (net/if.h:175-198).

Addresses, Names, and the Control Desk

An interface is more than its hardware; it must also carry one or more addresses. SVR4 models these as a linked list of `ifaddr` records, each one bound to an `ifnet` and to interface statistics (net/if.h:235-252).

```
struct ifaddr {
    struct  sockaddr ifa_addr;   /* address of interface */
    union {
        struct  sockaddr ifu_broadaddr;
        struct  sockaddr ifu_dstaddr;
    } ifa_ifu;
    struct  ifnet *ifa_ifp;      /* back-pointer to interface */
    struct  ifstats *ifa_ifs;    /* back-pointer to interface stats */
    struct  ifaddr *ifa_next;    /* next address for interface */
};
```

The Address Ledger (net/if.h:241-251)

This arrangement allows multiple addresses to be bound to a single interface, and it lets protocol families add and remove addresses without touching the driver’s core routines. The driver need only honor `if_ioctl` requests that carry these structures through the control plane.

The control plane itself is defined by `ifreq` and `ifconf`, used by `ioctl` calls that set flags, addresses, metrics, and driver-specific data (net/if.h:255-297).

```
struct ifreq {
    char    ifr_name[IFNAMSIZ];  /* if name, e.g. \"emd1\" */
    union {
        struct  sockaddr ifru_addr;
        struct  sockaddr ifru_dstaddr;
        char    ifru_oname[IFNAMSIZ];
        struct  sockaddr ifru_broadaddr;
        short   ifru_flags;
        int     ifru_metric;
        char    ifru_data[1];    /* interface dependent data */
        char    ifru_enaddr[6];
    } ifr_ifru;
};
```

The Control Ticket (net/if.h:260-281)

Drivers interpret these requests to bring a link up, set a hardware address, or toggle promiscuous mode. The harbor master issues the order; the dock foreman carries it out.

The Tally Book: `ifstats`

For tools that want summarized statistics per interface, SVR4 keeps a parallel `ifstats` chain (net/if.h:218-233). It mirrors the counters in `ifnet` but packages them for reporting and polling.

```
struct ifstats {
    struct ifstats *ifs_next; /* next if on chain */
    char           *ifs_name; /* interface name */
    short           ifs_unit; /* unit number */
    short           ifs_active; /* non-zero if this if is running */
    struct ifaddr  *ifs_addrs; /* list of addresses */
    short           ifs_mtu;  /* maximum transmission unit */
    int             ifs_ipackets;
    int             ifs_ierrors;
    int             ifs_opackets;
    int             ifs_oerrors;
    int             ifs_collisions;
};
```

The Tally Book Structure (net/if.h:218-233)

Drivers feed these counters as they transmit and receive. The ledger stays honest only if the dock logs every crate that passes.

Raw Queues and Interface Discovery

The networking subsystem also maintains a raw input queue and interface discovery routines in the kernel namespace (net/if.h:301-307). These hooks let debugging tools and raw packet listeners tap the wire directly, and they allow the stack to locate interfaces by name or address. For a driver, this means the interface must be correctly named (`if_name` plus `if_unit`) and linked into the global `ifnet` list so higher layers can find it.

The Outbound Ritual

When a protocol decides to send, it invokes the interface's `if_output` routine. The driver takes a chain of mbufs, possibly encapsulates it in a link-layer header, and begins transmission. It may push the packet immediately onto the device, or it may queue it in `if_snd` if the hardware is busy.

The driver is expected to:

- **Lock or raise priority** appropriately (the queue macros assume `splimp()` or equivalent protection).
- **Handle link-layer framing** for its medium.
- **Start transmission** and arrange for completion interrupts.
- **Update output statistics** once the device confirms send completion.

The outbound path is the dock's crane: it can load cargo immediately, or stack it in the yard until the ship arrives.

The Inbound Ritual

On input, each interface unwraps the data received by it, and either places it on the input queue of a datagram routine and posts the associated software interrupt, or passes the datagram to a raw packet input routine (net/if.h:76-79).

In other words:

1. **Interrupt fires** or polling loop detects incoming data.
2. **Driver builds mbufs**, attaches the receiving `ifnet` pointer.
3. **Driver enqueues to the protocol input queue** and schedules the software interrupt.
4. **Protocol stack continues** through IP, TCP, UDP, or raw packet handlers.

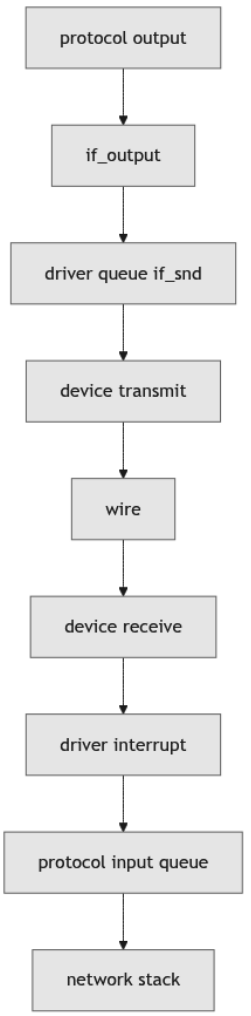


Figure 4.3.2: Outbound and Inbound Flow Through the Driver

Flags, Timers, and Unruly Seas

An interface's flags, such as `IFF_UP`, `IFF_RUNNING`, and `IFF_PROMISC`, tell the rest of the kernel how the dock is configured (net/if.h:133-144). Drivers are responsible for:

- Setting flags on successful initialization.
- Entering promiscuous mode when requested.
- Handling reset and watchdog callbacks when a link wedges.

The `if_timer` and `if_watchdog` fields give the driver a gentle nudge if it goes silent. This is critical for early Ethernet cards and flaky transceivers, where a missed interrupt could otherwise freeze the interface indefinitely.

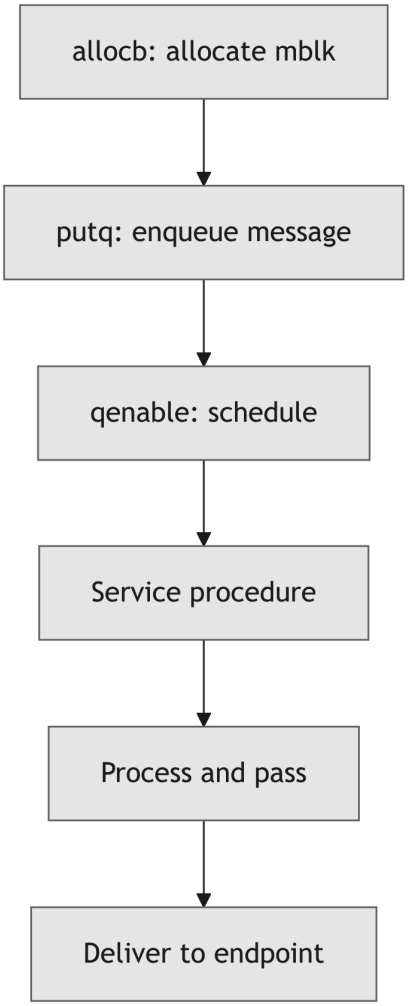
> **The Ghost of SVR4:** We counted packets and queued them with hand-rolled macros, guarding each queue with a raised interrupt priority. In your time the dock has grown into a port authority: multi-queue NICs, NAPI polling, and lockless rings move frames at astonishing speed. Yet the heart is unchanged: a register of capabilities, a transmit routine, a receive path, and an agreement that the driver will keep the water calm for the protocols above.

Harbor at Nightfall

The SVR4 driver framework is modest but sturdy. It does not attempt to hide hardware complexity behind elaborate abstractions; instead it offers a clear ledger (`ifnet`) and a small set of rituals for sending and receiving. The dock endures because it is predictable. When the signal lamps are lit, the ships know where to come, and the city knows what to do with their cargo.

STREAMS Framework

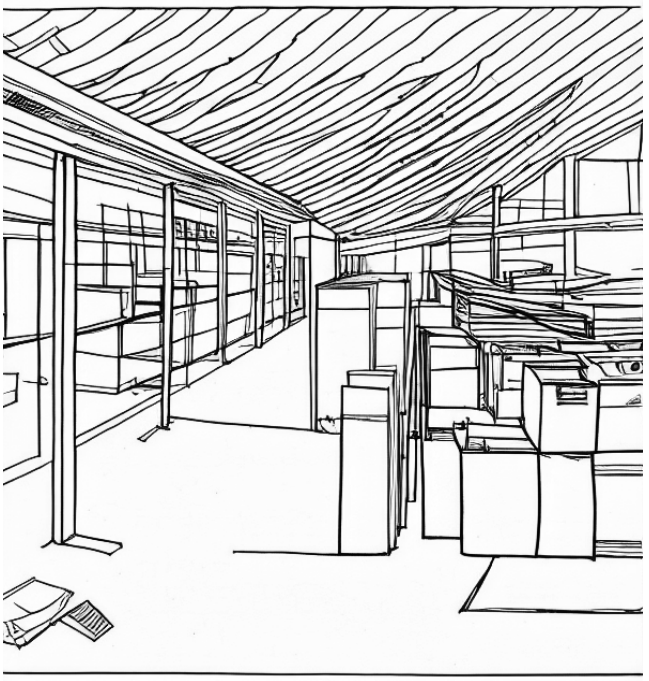
The Plumbing Unveiled: STREAMS Framework Internals



Having explored the STREAMS-based networking architecture from the application perspective, we now descend into the very machinery itself—the **STREAMS framework**, the kernel subsystem that makes modularity, message passing, and dynamic reconfiguration not merely aspirations but operational realities. This is the plumbing behind the elegant abstractions, where message blocks are allocated, queues are serviced, and flow control prevents the system from drowning in its own data.

The Fundamental Triad: Message Blocks, Data Blocks, and Queues

At the core of STREAMS lies a trinity of structures that together embody the essence of message-based I/O:



STREAMS - Postal System

The Message Block (`mblk_t`)

A message block is a lightweight descriptor, the handle by which STREAMS code manipulates data. It does not own the data itself; rather, it **points** to a data block and maintains metadata about the message's position within that data:

```
/* From sys/stream.h */
struct  msgb {
    struct      msgb    *b_next;
    struct      msgb    *b_prev;
    struct      msgb    *b_cont;
    unsigned char       *b_rptr;
    unsigned char       *b_wptr;
    struct datab        *b_datap;
    unsigned char       b_band;
    unsigned char       b_pad1;
    unsigned short      b_flag;
    long                b_pad2;
};
```

Code Snippet 4.10.1: The Message Block Structure (sys/stream.h:294-305)

The genius of this design is the **separation of descriptor from data**. Multiple message blocks can reference the same underlying data block (via `b_datap`), enabling zero-copy operations. When TCP prepends a header to user data, it doesn't copy the data—it allocates a new `mblk_t` for the header, links it via `b_cont` to the original data's `mblk_t`, and passes the chain downward.

The Data Block (`datab`)

The data block is the actual memory buffer, reference-counted to support sharing:

```
/* From sys/stream.h */
struct datab {
    union {
        struct datab    *freep;
        struct free_rtn *frtnp;
    } db_f;
    unsigned char       *db_base;
    unsigned char       *db_lim;
    unsigned char       db_ref;
    unsigned char       db_type;
    unsigned char       db_iswhat;
    unsigned int        db_size;
    long                db_filler;
    caddr_t             db_msgaddr;
};
```

Code Snippet 4.10.2: The Data Block Structure (sys/stream.h:254-266)

When a message block is duplicated (via `dupb()`), the reference count (`db_ref`) increments—but this is **NOT garbage collection**. This is not Java. This is not automatic. This is **manual reference bookkeeping** in the most unforgiving sense.

Every call to `freeb()` MUST have a matching `allocb()` or `dupb()` somewhere in the call chain, or the data block leaks into the void—eternally consuming kernel memory, invisible to all tracers, reclaimable only by reboot. The kernel has no sweeper, no reaper, no `gc()` pass. Only the iron promise: **if `db_ref` reaches zero, the block returns to `mdbfreelist`.**

This is C, where memory is not managed—it is **tamed**. One missed `freeb()` in a driver, and the kernel slowly hemorrhages message blocks until `allocb()` returns NULL and the network collapses. SVR4 programmers lived in constant awareness of this: reference counts were not conveniences but **sacred contracts**, enforced by nothing but discipline and late-night debugging sessions with `crash` dumps.

This manual reference counting is the bedrock of STREAMS' zero-copy philosophy—not because it's safe, but because it's **fast**. No garbage collector pause. No mark-and-sweep. Just `db_ref++` and `db_ref--`, executed in **two CPU cycles** each.

The Queue (`queue_t`)

Queues are the conduits through which messages flow. Each STREAMS module has four queues: read and write queues for both its upper and lower interfaces. The queue structure encapsulates not just the message list, but also the procedural entry points for processing:

```
/* From sys/stream.h */
struct  queue {
    struct      qinit   *q_qinfo;
    struct      msgb    *q_first;
    struct      msgb    *q_last;
    struct      queue   *q_next;
    struct      queue   *q_link;
    _VOID               *q_ptr;
    ulong               q_count;
    ulong               q_flag;
    long                q_minpsz;
    long                q_maxpsz;
    ulong               q_hiwat;
    ulong               q_lowat;
    struct qband        *q_bandp;
    unsigned char       q_nband;
    unsigned char       q_pad1[3];
    long                q_pad2[2];
};
```

Code Snippet 4.10.3: The Queue Structure (sys/stream.h:62-81)

The `q_qinfo` pointer references a `qinit` structure, defining the module's `put` and `service` procedures—the very functions we encountered in TCP and IP modules earlier.

Message Allocation: The Assembly Line

The performance of STREAMS hinges on a brutal truth: **allocation must be faster than memory itself**. Consider the problem: a 10-megabit Ethernet delivers packets at 1,250,000 bytes per second. Each packet requires a message block. If allocation requires a system call to `kmem_alloc()` — traversing the heap, checking free lists, possibly even invoking `sbrk()` to extend the data segment —the network drowns the kernel.

SVR4's solution is Fordist: the **assembly line**.

Picture a conveyor belt of pre-stamped, blank message blocks rolling endlessly through the kernel. When `allocb()` needs one, it doesn't stop to CUT metal from raw ore. It doesn't consult a manager. It simply **grabs the next block mid-conveyor**—a single pointer dereference:

```
// From stream.c - Fast Message Block Allocation (lines 97-102)
#define ALLOCMSGBLOCK(mesgp)            { \
    mesgp = msgfreelist;           // Grab block from conveyor
    msgfreelist = mesgp->b_next;   // Advance conveyor to next block
    BUMPUP(strst.msgblock);            // Statistics (ignorable)
    _INSERT_MSG_INUSE((struct mbinfo *)mesgp); // Debug tracking
(conditional)
}
```

Code Snippet 4.10.4: The Assembly Line Macro

This is **O(1) pointer surgery**—three instructions on i386: `mov`, `mov`, `inc`. No function call. No lock (the `splstr()` happens in the caller). Executed at interrupt level, this macro allocates a message block in ~**20 nanoseconds**, faster than a single DRAM access.

The `msgfreelist` is not a list in the abstract sense—it is a **physical chain** of kernel memory addresses, each 32-bit pointer value residing in DRAM at addresses like `0xc0123000`, `0xc0123040`, etc. When the CPU executes `mesgp = msgfreelist`, it issues a `MOV EAX, [msgfreelist]` instruction, which triggers a DRAM read cycle—RAS strobe, CAS strobe, sense amplifier activation—retrieving the address of the next block from capacitor charge patterns. This is not abstraction. This is **silicon retrieving voltage from oxide gates**.

The conveyor belt never stops. When `freeb()` returns a block, it's pushed back onto the front of `msgfreelist`—another `O(1)` operation. No heap traversal. No coalescing. Just a LIFO stack of pre-allocated memory, operating at the speed of cache and DRAM bandwidth.

When a message block is freed, the inverse occurs:

```
// From stream.c - Message Block Deallocation (lines 78-86)
#define FREEMSGBLOCK( mesgp )   { \
    register int s = splstr(); \
    (mesgp)->b_next = (struct msgb *) msgfreelist; \
    msgfreelist = mesgp; \
    strst.msgblock.use--; \
    _DELETE_MSG_INUSE((struct mbinfo *)mesgp); \
    splx(s); \
}
```

Code Snippet 4.10.5: Message Deallocation Macro

The `splstr()` call raises the interrupt priority to protect the free list from concurrent modification—a brief but necessary serialization point in an otherwise highly concurrent system.

Flow Control: The Sluice Gates of Silicon

Imagine a 19th-century grist mill on the River Tyne. The upstream reservoir—fed by spring rains and mountain snow—wants desperately to flood the valley. But the miller's wheel can only turn so fast, grinding grain at the pace of wood and stone. Two brass float valves control the sluice gate: one at the **high mark** (`q_hiwat`), one at the **low mark** (`q_lowat`).

When the mill pond fills beyond the high float, the gate **SLAMS shut** with a mechanical clang—the `QFULL` flag sets—and the upstream reservoir must hold its water. This is **backpressure**, hydraulic and absolute. The dam strains, but it holds. No negotiation. No protocol. Just physics.

When the miller's wheel drains the pond below the low float, the gate **CREAKS open** again—`QFULL` clears—and the pent-up water surges through. The upstream dam releases with a rush. Flow resumes.

This is not metaphor. This is hydraulics implemented in silicon.

The queue's byte count (`q_count`) is the water level, measured not in meters but in memory addresses. The "water" is voltage patterns in DRAM capacitors—charges representing TCP segments, each byte a HIGH or LOW in a 64-bit DIMM. These charges "drain" at the speed of `putq()` and `getq()` calls, function invocations that manipulate linked lists at nanosecond precision. The sluice gate is not brass but a single bit (`QFULL`) in the queue's flag word, checked by every upstream module before sending.

When `q_count` exceeds `q_hiwat` (typically 4096 bytes), the kernel executes:

```
if (q->q_count > q->q_hiwat)
    q->q_flag |= QFULL;  // SLAM. Gate closed.
```

This is a **digital sluice**, operating not with water but with electron flow, not at the pace of river currents but at the pace of DRAM refresh cycles (64 milliseconds). Yet the principle—float valve, gate position, backpressure—remains unchanged from James Watt's era.

Queue Scheduling: The Service Procedure Dance

Not all processing can occur in interrupt context. When a hardware interrupt delivers a network packet, the driver's `put` procedure (executing at interrupt level) typically enqueues the message and **schedules** the queue's service procedure to run later, at a safer priority level.

The kernel maintains a **queue run list** (`qhead` and `qtail` in the source), a linked list of queues whose service procedures need execution. The `qenable()` function adds a queue to this list:

```
void
qenable(q)
    register queue_t *q;
{
    register s;

    ASSERT(q);

    if (!q->q_qinfo->qi_srvp)
        return;

    s = splstr();
    if (q->q_flag & QENAB) {
        splx(s);
        return;
    }

    q->q_flag |= QENAB;
    if (!qhead)
        qhead = q;
    else
        qtail->q_link = q;
    qtail = q;
    q->q_link = NULL;
    setqsched();
    splx(s);
}
```

Code Snippet 4.10.6: Queue Enable Logic (io/stream.c:2089-2121)

Periodically (often during the return from system calls or interrupts), the kernel checks `qrunflag`. If set, it invokes `runqueues()`, which iterates through the queue run list, invoking each queue's service procedure in turn. This deferred processing model allows interrupt handlers to remain brief while complex protocol logic runs in a more permissive context.

Message Types: A Taxonomy of Intent

STREAMS messages are typed, signaling their purpose through the `db_type` field in the data block. Common types include:

- **`M_DATA`**: Ordinary data, the lifeblood of user communication.
- **`M_PROTO`**: Protocol control messages (e.g., TLI's `T_BIND_REQ`, TCP's internal control signals).
- **`M_IOCTL`**: I/O control requests, originating from user-space `ioctl()` calls.
- **`M_FLUSH`**: Flush requests, commanding queues to discard pending messages (used during connection teardown or error recovery).
- **`M_ERROR`**: Error notifications, propagated upstream to signal fatal stream conditions.

This typing allows modules to distinguish data from control, process appropriately, and pass unrecognized types transparently—a key enabler of the modular architecture.

Stream Construction: `stropen()` and Module Pushing

When a user opens a STREAMS device (e.g., `/dev/tcp`), the `stropen()` function in `streamio.c` orchestrates the stream's construction:

```
/* Excerpt from stropen() in os/streamio.c */
retry:
    if (stp = vp->v_stream) {
        if (stp->sd_flag & (STWOPEN|STRCLOSE)) {
            if (flag & (FNDELAY|FNONBLOCK)) {
                error = EAGAIN;
                goto ckreturn;
            }
            if (sleep((caddr_t)stp, STOPRI|PCATCH)) {
                error = EINTR;
                goto ckreturn;
            }
            goto retry;  /* could be clone */
        }

        if (stp->sd_flag & (STRDERR|STWRERR)) {
            error = EIO;
            goto ckreturn;
        }

        s = splstr();
        stp->sd_flag |= STWOPEN;
        splx(s);

        qp = stp->sd_wrq;
        while (SAMESTR(qp)) {
            qp = qp->q_next;
            if (qp->q_flag & QREADR)
                break;
            if (qp->q_flag & QOLD) {
                dev_t oldev;
                extern void gen_setup_idinfo();

                gen_setup_idinfo(crp);
                if ((oldev = cmpdev(*devp)) == NODEV) {
                    error = ENXIO;
                    break;
                }
                if ((*RD(qp)->q_qinfo->qi_qopen)(RD(qp),
                    oldev, (qp->q_next ? 0 : flag),
                            (qp->q_next ? MODOPEN : 0)) == OPENFAIL) {
                    if ((error = u.u_error) == 0)
                        error = ENXIO;
                    break;
                }
            } else {
                dummydev = *devp;
```

```
                if (error = ((*RD(qp)->q_qinfo->qi_qopen)(RD(qp),
                    &dummydev, (qp->q_next ? 0 : flag),
                            (qp->q_next ? MODOPEN : 0), crp)))
                    break;
            }
        }
    }
```

Code Snippet 4.10.7: Stream Open Logic (os/streamio.c:101-156, excerpt)

This code handles both initial stream creation and subsequent re-opens. The `sd_flag` field serializes concurrent opens, ensuring that stream state remains consistent. Once open, modules can be dynamically pushed onto the stream using the `I_PUSH` ioctl, inserting new functionality (e.g., line disciplines, compression, encryption) into an active data path.

The Ghost Speaks: The Tragedy of Beautiful Abstraction

Picture STREAMS as a grand old-world postal system—every packet a letter in a cream envelope, every queue a mahogany sorting office with glass doors, every service procedure a clerk in waistcoat and spectacles, meticulously examining each message before passing it to the next office. In 1988, this was elegance itself: **modular, inspectable, debuggable**. You could WATCH the mail flow through those glass-doored cabinets. A kernel debugger could freeze time and examine each `mblk_t` as if it were a letter on a desk—sender address (`b_rptr`), recipient (`b_wptr`), contents (`db_base`). The abstraction was so clean that new protocols could be added as new "departments" without rebuilding the post office.

But then the telegraph became the telephone, and the telephone became fiber-optic cable.

Linux's network stack is not a postal system—it is a **particle accelerator**. Packets aren't sorted by clerks in offices; they're **hurled through function pointers** at near-light speed, bypassing the "sorting offices" (queues) entirely via **zero-copy DMA**. The network driver writes directly into socket buffers using bus-mastering transfers—the packet never touches a "queue" in the STREAMS sense. It appears in RAM via PCI Express transaction, already in the correct cache line for the TCP stack to consume.

The per-CPU processing queues aren't Victorian mail slots—they're **quantum fields**, each CPU core manipulating its own isolated universe of `sk_buff` structures. Core 0 processes packets from NIC queue 0. Core 1 handles queue 1. They never coordinate. No global `qhead`. No `runqueues()` sweep. Just per-CPU lockless NAPI polling—each core racing through its private backlog at billions of instructions per second.

The tragedy of STREAMS is that it was RIGHT. Message-based I/O, flow control, modular composition—all **correct principles**. But correctness lost to **register pressure**. Every message block dereference (`bp->b_datap->db_base`) is a dependent load—the CPU must wait for `bp->b_datap` to arrive from DRAM before it can issue the second load for `db_base`. Two cache misses, serially dependent. In modern terms: **~200 nanoseconds of stall time** on every access.

Linux's `sk_buff` is a monolithic slab—ugly, but it fits entirely in two cache lines (128 bytes on x86). The TCP header, IP header, and socket metadata are co-located in memory. One cache miss fetches everything. Beauty lost to **locality of reference**.

STREAMS lives on in Solaris terminal drivers, a ghost maintaining its glass-doored cabinets in a world that has moved to pneumatic tubes and pneumatic tubes to quantum-entangled photons. Visit `/dev/pts` on a Solaris box in 2026, and you'll find STREAMS modules (`ldterm` , `ptem` , `ttcompat`) still stacked like clerks in a Victorian counting house, faithfully translating `VMIN` / `VTIME` into message flow control.

But the network? The network moved on.

The lesson? **Sometimes the beautiful abstraction loses to the brutal inline.** Not because beauty doesn't matter—it does—but because at 100 gigabits per second, **a single cache miss costs you 20 packets**. STREAMS taught us modularity. Linux taught us that modularity's cost is memory latency. The future belongs to whoever can reconcile the two.

But oh, how clearly you could SEE the data in 1988…

Ancient Incantations: Fossils in the Silicon

The STREAMS framework, though faded, left archaeological traces throughout modern kernels—code patterns and data structures that echo across four decades:

The `putq()` / `getq()` Symmetry These function names persist in Solaris, IllumOS, and even in Linux's TTY layer (as `tty_buffer_request_room()`). The pattern—enqueue a message, check a flag, optionally schedule deferred work—is STREAMS' DNA, replicated in modern queueing disciplines.

The `M_` Message Type Prefix `M_DATA`, `M_PROTO`, `M_FLUSH`—these constants (defined in `<sys/stream.h>`) still compile in 2026 Solaris kernels. They're archaeological sites, frozen in header files, maintained for binary compatibility with drivers written when Reagan was president.

The `b_rptr` / `b_wptr` Idiom Linux's `sk_buff` structure contains `unsigned char *head, *data, *tail, *end`—a direct descendant of STREAMS' read/write pointer pattern. The names changed, but the concept—**marking valid data within a larger buffer**—survived the transition. Even the pointer arithmetic (`bp->b_wptr - bp->b_rptr` becomes `skb->tail - skb->data`) is identical.

The Service Procedure Contract STREAMS' "put procedure runs at interrupt level, service procedure runs deferred" split directly inspired Linux's **NAPI** (New API) for network drivers. NAPI's `poll()` method is conceptually a service procedure—invoked when the driver's RX queue has work, running in softirq context, processing multiple packets per invocation. The name changed. The architecture endured.

The Modularity Fossil The `I_PUSH` ioctl still exists in Solaris `/dev/pts` (pseudo-terminal) streams. You can, even in 2026, push a STREAMS module onto a terminal to add line editing (`ldterm`), terminal emulation (`ptem`), or compatibility layers (`ttcompat`). This is a living fossil—a Coelacanth swimming in the deep ocean of kernel I/O, unchanged since SVR4's Miocene epoch.

The Whisper in the Code Search any modern Solaris kernel source for `STR` prefixes: `STRHOLD`, `STRGETMSG`, `STRMSGSZ`. These are not active code—they're **#ifdef**-wrapped relics, maintained

for backward compatibility, never executed on modern hardware. They’re like hieroglyphs on a reused stone—visible under the right light, telling of an older empire that once ruled this silicon.

The Enduring Lessons

Though STREAMS has faded from the mainstream, its architectural principles endure:

- **Message-Based Abstraction**: Decoupling control from mechanism through well-defined message protocols.
- **Modular Composition**: Building complex functionality from simple, interchangeable components.
- **Flow Control as Emergent Property**: Letting backpressure arise naturally from queue state, rather than explicit protocols.
- **Zero-Copy Optimization**: Sharing data through reference counting, minimizing memory bandwidth.

These ideas resonate in modern systems: message queues in distributed systems, pluggable protocol stacks in user-space networking (e.g., DPDK), and the eternal quest to balance abstraction with performance. STREAMS was ahead of its time—a framework so ambitious that hardware took decades to catch up, and by then, simpler designs had won the day.

Yet for those who study it, STREAMS remains a **masterclass in kernel architecture**, a reminder that elegance and performance are not always allies, and that the best designs are those that acknowledge the constraints of their era while aspiring to transcend them.

The Boot Process

Overview: The Great Engine's Ignition

The booting of an operating system is the Great Engine's Ignition, a carefully choreographed sequence of events that brings the complex machinery of the kernel to life. It is a process that begins with a single spark from the hardware and culminates in the thunderous roar of a fully operational system, ready to serve its users. In SVR4, this ignition sequence is a multi-stage affair, a journey from the primitive 16-bit real mode of the i386 processor to the sophisticated 32-bit protected mode environment in which the kernel proper resides.

The First Spark: The BIOS and the Boot Block

The ignition sequence begins with the BIOS (Basic Input/Output System), the firmware that is embedded in the machine's motherboard. When the machine is powered on, the BIOS performs a series of power-on self-tests (POST) and then, following a configured boot order, it reads the first 512 bytes from the boot device (usually a hard disk). This first sector is the Master Boot Record (MBR).

The MBR contains a small amount of code and a partition table. The MBR code's job is to find the active partition, load its first sector (the boot block), and transfer control to it. This is the first spark, the handoff from the hardware's firmware to the operating system's own boot code. The initial assembly code for this process is found in `start.s`.

The Bootstrap Loader: `boot.c`

The code in the boot block is a simple, 16-bit real-mode program whose sole purpose is to load a larger, more sophisticated bootloader from the filesystem. This is the second stage of the boot process, and its main C program is `boot.c`.

The `boot.c` program is responsible for:

- **Initializing the hardware**: It queries the BIOS for information about the machine's memory size and hard disk geometry.
- **Reading the defaults file**: It reads the `/etc/default/boot` file to determine the default kernel to load and other boot-time parameters.
- **The boot prompt**: It presents the user with a boot prompt, allowing them to override the defaults and specify a different kernel or set of options.
- **Loading the kernel**: It calls the `bload` function in `load.c` to load the kernel into memory.

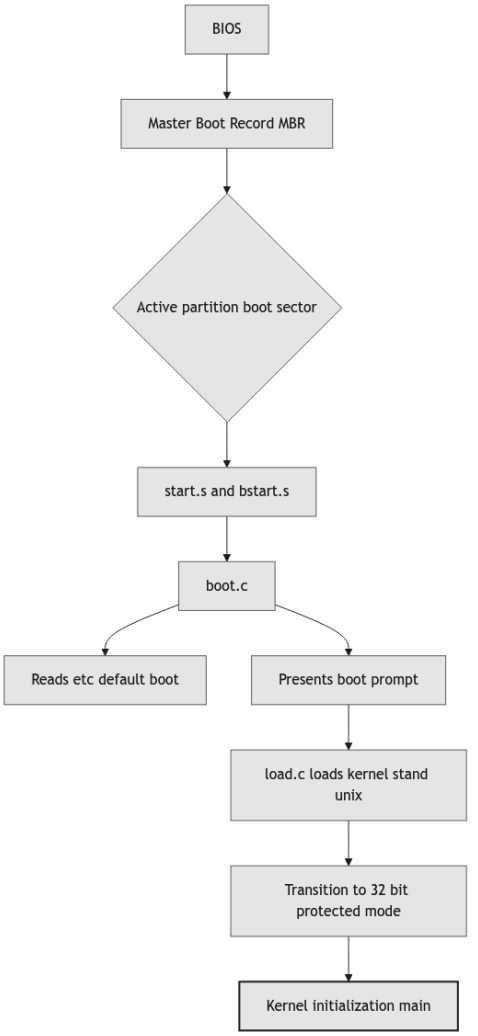
Loading the Kernel: `load.c`

The `load.c` program contains the logic for reading the kernel image (usually `/stand/unix`) from the disk and placing it in memory. It understands the format of the executable file (either COFF or ELF in SVR4) and can parse its headers to determine how to load its various sections (text, data, and bss) into memory.

The Final Ignition: The Transition to Protected Mode

Once the kernel is loaded into memory, the bootloader's final task is to transfer control to it. This is a critical and delicate moment, the final ignition sequence that brings the great engine to life. The `bstart` function, called from `boot.c`, is responsible for this final step. It performs the following actions:

1. **Sets up the bootinfo structure**: It populates a `bootinfo` structure in a well-known location in low memory, passing information about the memory layout, disk parameters, and other boot-time information to the kernel.
2. **Enters protected mode**: It switches the i386 processor from 16-bit real mode to 32-bit protected mode, a critical step that allows the kernel to access the full range of the machine's memory and to take advantage of its memory protection features.
3. **Jumps to the kernel entry point**: Finally, it performs a long jump to the kernel's entry point, the `main` function in the kernel's own code. At this point, the bootloader's job is done, and the kernel is in control.



The SVR4 Boot Sequence

The Ghost of SVR4:

"Our ignition sequence was a marvel of minimalist engineering, a carefully crafted chain of small programs, each one handing off to the next, to pull the kernel up by its own bootstraps. But it was a system built on trust. The BIOS trusted the MBR, the MBR trusted the boot block, and the bootloader trusted the kernel. There were no digital signatures, no secure enclaves, no cryptographic handshakes. In your time, you have built fortresses around this process. Your UEFI (Unified Extensible Firmware Interface) and its 'Secure Boot' protocol are a response to a world of threats we could scarcely have imagined. Your bootloaders are signed and verified, your kernels are measured and attested. You have traded the simple

elegance of our ignition sequence for the complex, but necessary, security of a world where the very foundations of the system are under constant attack.”

Boot Process - Theater Opening Night

Conclusion

The boot process is the critical, and often unseen, foundation upon which the entire operating system is built. It is the Great Engine’s Ignition, the carefully choreographed sequence of events that transforms a dormant piece of hardware into a living, breathing system. The SVR4 boot process, with its multi-stage loader and its transition from the simple world of the BIOS to the sophisticated world of the protected-mode kernel, is a classic example of this process, a testament to the ingenuity of the engineers who first brought these great engines to life.

Device Driver Framework: The Grand Exchange and Its Operators

Imagine a cavernous trading hall, a Grand Exchange where every merchant occupies a fixed stall and each stall bears a brass placard with a number. Couriers arrive at the marble steps with sealed orders, each envelope marked only by its stall number. The clerks do not wander the city in search of a merchant; they consult the exchange ledger, march to the indicated desk, and the transaction begins without ceremony. The exchange thrives not because it knows every merchant personally, but because it knows exactly where to route each order.

In SVR4, device drivers occupy those numbered stalls. The kernel receives requests in the form of device numbers and dispatches them through a rigid set of tables. These tables, the *device switches*, are the exchange ledger. The driver framework is less a grand hierarchy and more a disciplined registry: a strict roster of entry points, a handful of flags, and a promise that a major number will always lead to the correct operator.

The Switchboards of the Kernel

SVR4 maintains two primary switch tables: one for block devices and one for character devices. Their structures are declared in `sys/conf.h` and are the formal contract between the core kernel and every driver (sys/conf.h:15-107).

```
struct bdevsw {
    int (*d_open)();
    int (*d_close)();
    int (*d_strategy)();
    int (*d_print)();
    int (*d_size)();
    int (*d_xpoll)();
    int (*d_xhalt)();
    char *d_name;
    struct iobuf *d_tab;
    int *d_flag;
};

struct cdevsw {
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();
    int (*d_write)();
    int (*d_ioctl)();
    int (*d_mmap)();
    int (*d_segmap)();
    int (*d_poll)();
    int (*d_xpoll)();
    int (*d_xhalt)();
    struct tty *d_ttys;
    struct streamtab *d_str;
    char *d_name;
    int *d_flag;
};
```

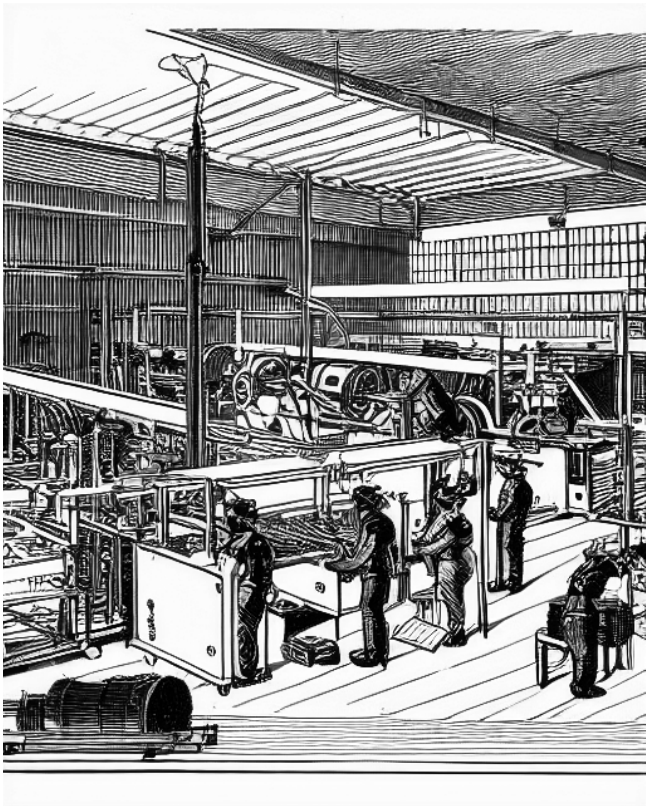
The Switchboard Columns (sys/conf.h:20-54)

The tables reveal the kernel's expectations:

- **Block devices** must offer a `d_strategy()` routine for buffer-oriented I/O.
- **Character devices** are richer in interface: read/write/ioctl, memory mapping, and the STREAMS hook `d_str`.
- `d_flag` describes the driver's lineage and quirks (old-style compatibility, DMA behavior, etc.).
- `d_name` is the human-readable handle, used in configuration tools and diagnostics.

Two arrays, `bdevsw[]` and `cdevsw[]`, carry these entries, while `bdevcnt` and `cdevcnt` define their bounds (sys/conf.h:33-107). The tables are not discovered; they are compiled into the

kernel image by configuration, and their indices are the major device numbers.



Driver Framework - Machine Factory

Tickets, Majors, and Dispatch

When user space opens a device node under `/dev`, the kernel routes the call through the special file system layer. `spec_open()` extracts the major number and dispatches into the appropriate switch table. The mechanism is deliberate and unromantic: a single array lookup, followed by a function pointer call (fs/specfs/specvnops.c:283-357).

```
if ((error = (*cdevsw[maj].d_open)
  (&newdev, flag, OTYP_CHR, cr)) == 0
    && dev != newdev) {
    /* Clone open handling */
    if ((nvp = makespecvp(newdev, VCHR)) == NULL) {
        (*cdevsw[getmajor(newdev)].d_close)
          (newdev, flag, OTYP_CHR, cr);
        error = ENOMEM;
        break;
    }
    /* vnode replacement continues */
}
...
if ((error = (*bdevsw[maj].d_open)(&newdev, flag,
        OTYP_BLK, cr)) == 0 && dev != newdev) {
    /* Block clone handling */
}
```

The Dispatch to the Operator (fs/specfs/specvnops.c:283-357, simplified)

The dispatch depends on two simple facts:

1. **The vnode type** decides the table (`VCHR` for `cdevsw` , `VBLK` for `bdevsw`).
2. **The major number** selects the row, and the function pointer is invoked.

Once dispatched, the driver is sovereign. It may honor the original device number, or it may return a new one (for clone devices), in which case the special file layer swaps out the vnode to match the driver's chosen identity.

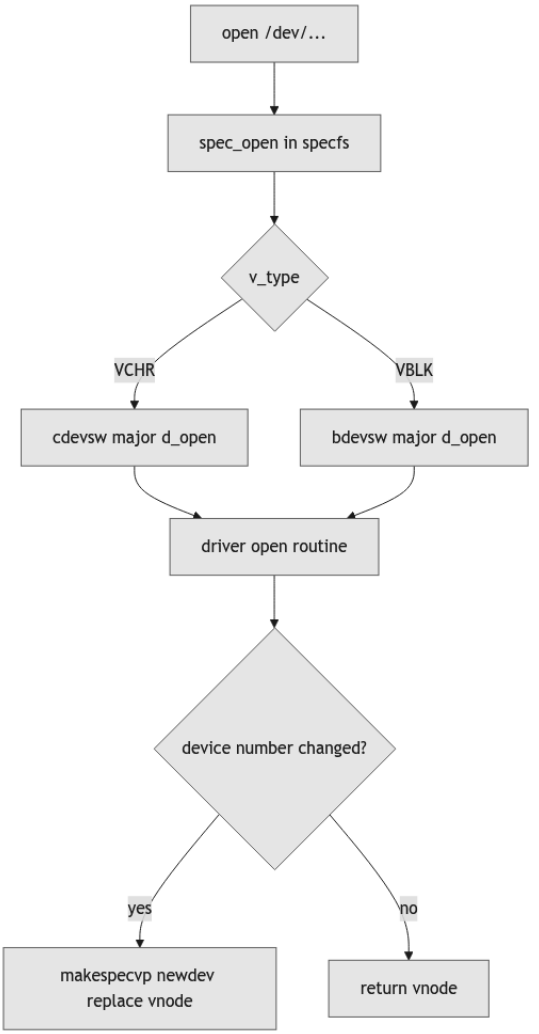


Figure 5.2.1: Dispatch Through the Switch Tables

Old-Style Drivers and the Compatibility Clerks

SVR4 straddles two eras. Some drivers follow the new interfaces (passing full device numbers, using modern `uio` structures), while older ones expect the pre-4.0 conventions. Rather than force every old driver to be rewritten, SVR4 interposes a set of compatibility wrappers.

At startup, `fix_swtbls()` walks the switch tables and replaces old-style entry points with generic shims, saving the original pointers into `shadowcsw[]` and `shadowbsw[]` (os/predki.c:41-71).

```
for (i = 0; i < bdevcnt; i++) {
    if (*bdevsw[i].d_flag & D_OLD) {
        shadowbsw[i].d_open = bdevsw[i].d_open;
        bdevsw[i].d_open = gen_bopen;
        shadowbsw[i].d_close = bdevsw[i].d_close;
        bdevsw[i].d_close = gen_bclose;
    }
    if (!(*bdevsw[i].d_flag & D_NOBRKUP)) {
        shadowbsw[i].d_strategy = bdevsw[i].d_strategy;
        bdevsw[i].d_strategy = (int(*)())gen_strategy;
    }
}
for (i = 0; i < cdevcnt; i++) {
    if (*cdevsw[i].d_flag & D_OLD) {
        shadowcsw[i].d_open = cdevsw[i].d_open;
        cdevsw[i].d_open = gen_copen;
        shadowcsw[i].d_close = cdevsw[i].d_close;
        cdevsw[i].d_close = gen_cclose;
        shadowcsw[i].d_read = cdevsw[i].d_read;
        cdevsw[i].d_read = gen_read;
        shadowcsw[i].d_write = cdevsw[i].d_write;
        cdevsw[i].d_write = gen_write;
        shadowcsw[i].d_ioctl = cdevsw[i].d_ioctl;
        cdevsw[i].d_ioctl = gen_ioctl;
    }
}
```

The Compatibility Ledger (os/predki.c:41-71)

These wrappers translate between old and new calling conventions. `gen_read()` and `gen_write()` populate the legacy `u` area fields before invoking the old driver routine, then copy results back out to the modern `uio` (os/predki.c:136-174). The exchange clerk knows the antique forms by heart.

The compatibility pass is triggered during startup immediately after process table initialization (os/startup.c:475-504). It is not a runtime negotiation; it is a once-only retrofit before any device is opened.

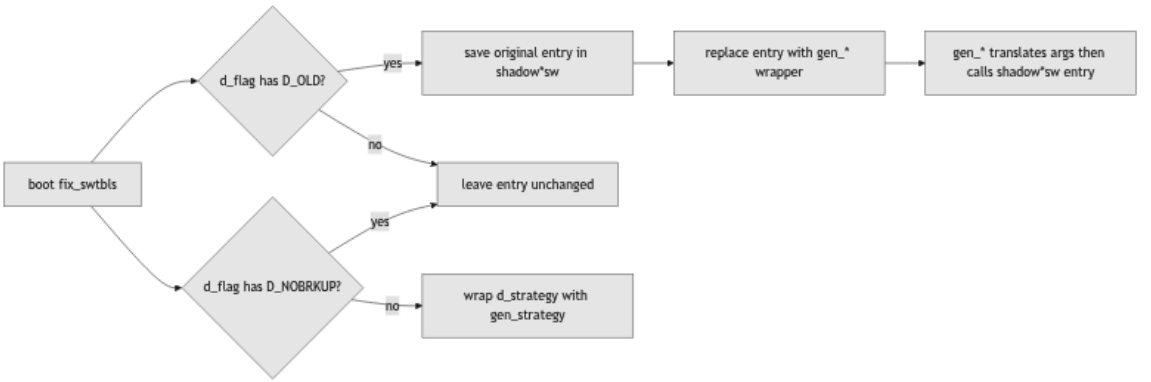


Figure 5.2.2: Old-Style Wrapping via `fix_swtbls()`

The Clone Desk: One Stall, Many Identities

SVR4 also supports a clever trick: the clone driver. A single major number can hand out new minors (and sometimes new majors) as devices are opened. The clone driver's `clnopen()` routine resolves the *real* target by looking up the requested major in `cdevsw[]`, then installing its queues and invoking the actual driver open (io/clone.c:43-123).

```
emaj = getminor(*devp);
maj = etoimajor(emaj);

if ((maj >= cdevcnt) || !(stp = cdevsw[maj].d_str))
    return (ENXIO);

setq(qp, stp->st_rdinit, stp->st_wrinit);

if (*cdevsw[maj].d_flag & D_OLD) {
    /* old-style driver: minor returned directly */
    if ((newdev = (*qp->q_qinfo->qi_qopen)(qp, oldev, flag, CLONEOPEN)) ==
OPENFAIL)
        return (u.u_error == 0 ? ENXIO : u.u_error);
    *devp = makedevice(emaj, (newdev & OMAXMIN));
} else {
    if (error = (*qp->q_qinfo->qi_qopen)(qp, &newdev, flag, CLONEOPEN,
crp))
        return (error);
    *devp = newdev;
}
```

The Clone Negotiation (io/clone.c:43-123, simplified)

The clone driver is an exchange desk that issues bespoke tickets on demand. It also underscores the switch table's importance: even when devices are dynamic, their initial rendezvous still occurs through `cdevsw[]`.

> **The Ghost of SVR4:** Modern kernels still keep registries of driver entry points, but the registry has moved into a world of loadable modules, device trees, and hotplug buses. Linux's `struct file_operations` and `struct block_device_operations` echo the `cdevsw` and `bdevsw` lineage, yet registration is no longer frozen at link time. A driver can appear at runtime, bind to a device described by firmware, and be queried through sysfs. The exchange has become a living marketplace, stalls erected and removed as the day progresses.

The Exchange at Dusk

The SVR4 driver framework is not a cathedral of abstractions; it is a ledger, a row of brass plates, and a clerk who can route a request without asking where it came from. It succeeds because it is *predictable*: the major number points to the stall, the stall points to the operator, and the operator answers. The rest of the kernel can trade in confidence, knowing the exchange keeps its books in perfect order.

Interrupt Handling

The Telegraph's SLAM: Interrupt Handling as Electrical Tyranny

In the orchestrated quietude of a running kernel, where processes execute in orderly time slices and system calls proceed through well-defined paths, there exists a class of events that **refuses to wait, refuses to ask permission, refuses to be polite**: **interrupts**.

An interrupt is not a knock. Knocks are courteous. Knocks can be ignored.

An interrupt is a lightning strike.

Picture the CPU as a diligent scholar hunched over illuminated manuscripts (user processes), quill in hand, mid-sentence in an important treatise. Beside him on the desk sits a telegraph key—silent, inert. Then, without warning, the key **SLAMS down**—*click-click-click*—unbidden, uncaring for his sentence mid-phrase, his ink mid-stroke. The telegraph wire (the `INTR` pin on the i386 package) carries voltage from the Programmable Interrupt Controller, a spike from 0V to 5V that the CPU's state machine **electrically detects** on every single clock cycle.

The CPU doesn't "hear" the interrupt. It doesn't "check for" the interrupt. The silicon itself, at the transistor level, **senses voltage** on a dedicated pin, and the processor's finite state machine—wired into the chip during fabrication—**hijacks the instruction pipeline**. This is not software. This is **electrical tyranny**.

The Hardware Foundation: Silicon Routing Tables

The Intel i386 processor, SVR4's silicon foundation, implements interrupts not through software courtesy but through **microcode-assisted table lookups hardwired into the instruction decoder**. The Interrupt Descriptor Table (IDT) is not a data structure in the abstract sense—it is a **2048-byte region of physical RAM** (256 entries × 8 bytes each), whose base address resides in the `IDTR` register, a 48-bit value split into 32-bit base address and 16-bit limit.

When the CPU detects that `INTR` pin assertion—a physical voltage transition from LOW to HIGH—the microcode doesn't "look up" the IDT like a reader consulting an index. The interrupt vector

(0-255), delivered by the PIC via the data bus, is **hardware-multiplied by 8** in the address generation unit (each IDT entry is 8 bytes). This product is added to `IDTR.base`, producing a physical memory address like `0xC0001050`.

The CPU then issues a **memory read transaction** on the system bus:

- RAS (Row Address Strobe) asserts, selecting DRAM row
- CAS (Column Address Strobe) asserts, selecting column within row
- Sense amplifiers detect capacitor charge (HIGH = ~5V, LOW = ~0V)
- The 8 bytes of the IDT entry arrive in the CPU's prefetch queue

This happens in microcode, faster than a single user-visible instruction.

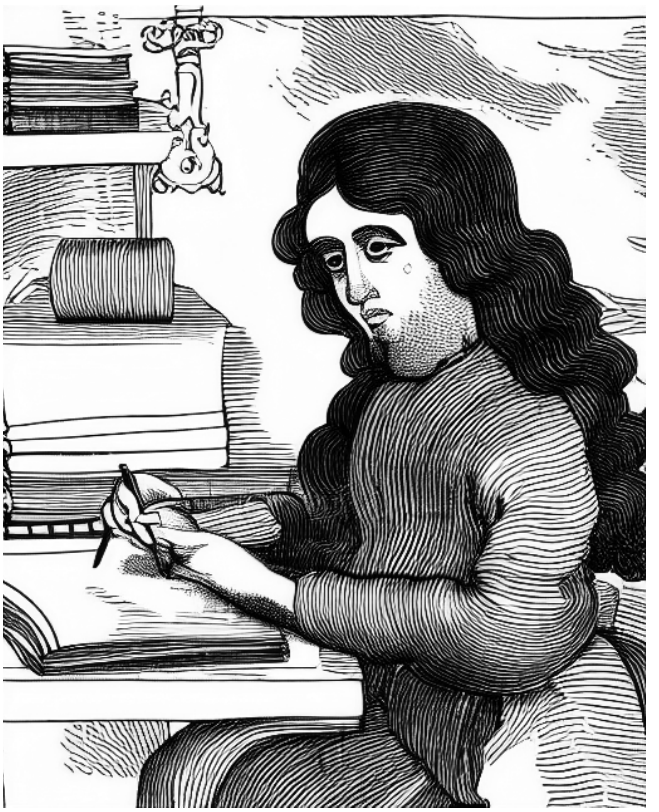
The IDT entry specifies:

1. **Code Segment Selector** (16 bits): Which ring-0 code segment contains the handler
2. **Offset** (32 bits): The exact `EIP` value to jump to
3. **Flags**: Interrupt gate vs. trap gate, privilege level requirements

The CPU then **atomically**:

- Pushes `SS:ESP`, `EFLAGS`, `CS:EIP` onto the kernel stack (retrieved from the Task State Segment)
- Loads new `CS:EIP` from the IDT entry
- Switches CPL (Current Privilege Level) to 0
- Clears the interrupt flag (IF) for interrupt gates, preventing nested interrupts

This is not software branching. This is silicon routing. The scholar's hand (program counter) jerks mid-word to a new page (the ISR entry point). His previous page and pen position (`CS:EIP`, `EFLAGS`) are shoved into a drawer (kernel stack) by reflex—atomically, in hardware, without software involvement.



Interrupts - Scholar

The Trap Gate: Entry from User Mode

SVR4 distinguishes between **traps** (synchronous exceptions arising from program execution, like page faults or system calls) and **interrupts** (asynchronous hardware events). Both, however, share a common entry mechanism on i386: the **trap gate** or **interrupt gate** in the IDT.

When a trap or interrupt occurs from user mode, control transfers to the kernel's assembly-language entry stub, which immediately saves the user's register state onto the kernel stack. For hardware interrupts, this stub then invokes the appropriate device-specific **interrupt service routine (ISR)** registered for that interrupt vector. For traps, control flows to the `k_trap()` or `s_trap()` functions in `trap.c`.

The `s_trap()` function, invoked just before returning from kernel mode to user mode, is a crucial checkpoint:

```
// From trap.c - System Trap Handler (lines 124-177)
void s_trap()
{
    register proc_t     *pp;
    time_t       syst;

    pp = u.u_procp;
    syst = pp->p_stime;

    // Check for pending preemption
    if (runrun != 0)
        preempt();

    // Check for pending signals
    if (ISSIG(pp, FORREAL))
        psig();
    else if (EV_ISTRAP(pp))
        ev_traptousr();

    // Update profiling data if enabled
    if (u.u_prof.pr_scale & ~1)
        addupc((void(*)())u.u_ar0[EIP], (int)(pp->p_stime - syst));
}
```

Code Snippet 5.3.1: System Trap Handler

This function performs three critical checks before returning to user mode:

1. **Preemption Check (`runrun`)**: If the `runrun` flag is set (signaling that a higher-priority process is runnable or the current process's time slice has expired), `preempt()` is invoked, triggering a context switch.
2. **Signal Delivery (`ISSIG` / `psig`)**: If any signals are pending for the current process, they are delivered via `psig()` , potentially invoking user-mode signal handlers or terminating the process.
3. **Profiling Update**: If process profiling is enabled, the time spent in the kernel is accounted for.

This routine is the kernel's final gatekeeper, ensuring that asynchronous events (signals, preemption) are processed before relinquishing control to user code.

Interrupt Priority Levels: SILENCE in the Tavern

Not all interrupts are created equal. A timer interrupt, which must occur precisely every 10 milliseconds to maintain system time, takes precedence over a keyboard interrupt (which can wait 50 milliseconds while you finish your critical section). SVR4's hierarchy is brutal and simple: **interrupt priority levels**.

The `spl6()` function is like shouting **"SILENCE!"** in a crowded bustling tavern—not politely requesting quiet, but **COMMANDING** it. Every device interrupt below priority 6 MUST wait, frozen mid-sentence, mouths open but silent, until `splx()` restores the original noise level.

What Happens in Silicon:

When you call `spl6()`, the kernel executes (on i386):

```
movb $0xBF, %al     ; Interrupt mask: allow only IRQ 6 and above
outb %al, $0x21     ; Write to PIC's interrupt mask register (IMR)
```

This `outb` instruction—a single opcode, two bytes—travels down the I/O bus to the 8259A Programmable Interrupt Controller chip. The PIC's Interrupt Mask Register is a physical 8-bit latch at I/O port `0x21`. When that byte arrives, transistors flip. Voltage levels change. IRQs 0-5 are now **electrically disconnected** from the CPU's `INTR` pin—their interrupt lines go nowhere, severed at the silicon level.

The disk (IRQ 14) can finish its operation. The network card (IRQ 10) can receive a packet. They can assert their interrupt lines. **But the PIC will not forward these to the CPU.** The `INTR` pin remains LOW. The interrupts are not queued, not buffered—they are **held at the source**, the PIC refusing to translate them into interrupt vectors.

When `splx(s)` restores the original mask:

```
movb %dl, %al       ; Restore saved mask from 's'
outb %al, $0x21     ; Reconnect severed interrupt lines
```

The transistors flip back. The electrical pathways reconnect. Any pending interrupts—disk done, network packet arrived—**FLOOD** through the now-open gate, often causing a cascade of nested ISR invocations.

Bach Mode (What It Does):

```
int s = spl6();      // Block IRQs 0-5. Save old mask in 's'.
q->q_count += len;   // Critical section: update queue byte count
splx(s);             // Restore original interrupt mask
```

Whimsy Mode (What It Means):

You're a tavern keeper (kernel) trying to count coins (update `q_count`). Patrons (interrupts) keep shouting drink orders mid-count, forcing you to restart. So you bellow "**SILENCE!**"— `spl6()` — and the room freezes. You count your coins. Then you nod— `splx()` —and the cacophony resumes.

The Virtue Lost:

In 1988, `spl` was brilliant—a **coarse lock** that protected the entire kernel with a single I/O instruction. Every programmer knew: `spl6()` → you have **10 microseconds, tops**. Hold it longer and the clock interrupt drifts, time itself loses accuracy. Modern kernels mock this with fine-grained spinlocks (`spin_lock_irqsave()`), but `spl` had a virtue: **predictability**. `spl6()` on an i386 always took exactly **12 CPU cycles** (the `outb` instruction latency). Modern interrupt latency is a *probability distribution*. SVR4's was a **contract**.

The Two-Phase Handler: Top Half and Bottom Half

A cardinal rule of interrupt handling is: **minimize time spent at high interrupt priority**. Prolonged execution with interrupts disabled can cause timer drift, lost network packets, and system unresponsiveness. To reconcile this constraint with the need for complex processing, SVR4 employs a **two-phase interrupt handling model**:

Top Half: The Minimal ISR

The **top half** is the ISR that executes immediately upon interrupt arrival, at elevated priority. Its responsibilities are deliberately minimal:

1. **Acknowledge the Hardware**: Signal the device that the interrupt has been received (often by reading a status register or writing to a control register).

2. **Enqueue Work**: Place a message on a queue (e.g., a STREAMS queue for network packets) or set a flag indicating that deferred processing is needed.
3. **Schedule Bottom Half**: If necessary, schedule a lower-priority task to perform the heavy lifting.
4. **Return Swiftly**: Exit the ISR, allowing other interrupts to be serviced.

For example, a network driver's ISR might simply enqueue the incoming packet onto the IP module's queue and return, leaving the packet's interpretation and routing to the bottom half.

Bottom Half: Deferred Processing

The **bottom half** executes at a lower priority, often during the return path from kernel mode or via explicitly scheduled kernel threads. This is where the bulk of interrupt-related work occurs: packet processing, buffer management, signal delivery, and more.

In STREAMS (as we saw in STREAMS Framework), the bottom half is often the **service procedure**, invoked via `runqueues()` when the queue is enabled. This separation ensures that the top half remains brief, while the bottom half can safely block, allocate memory, and interact with user processes.

The Kernel Stack Switch: Safeguarding Interrupt Context

When an interrupt occurs from user mode, the i386 hardware automatically switches from the user's stack (in user address space) to the kernel stack (pointed to by the Task State Segment, TSS). This switch is critical: executing an ISR on a potentially tiny or corrupted user stack would be catastrophic.

The kernel stack, allocated per-process and residing in kernel memory, provides a safe execution environment for the ISR. All register saves, local variables, and nested function calls occur on this stack. Upon interrupt return (`IRET` instruction), the hardware restores the user stack pointer, seamlessly resuming user execution.

For interrupts that occur while already in kernel mode (e.g., a timer interrupt during a system call), no stack switch occurs—the ISR executes on the existing kernel stack. The `k_trap()` function in

`trap.c` handles such kernel-mode traps, ensuring that even nested interrupts are processed correctly.

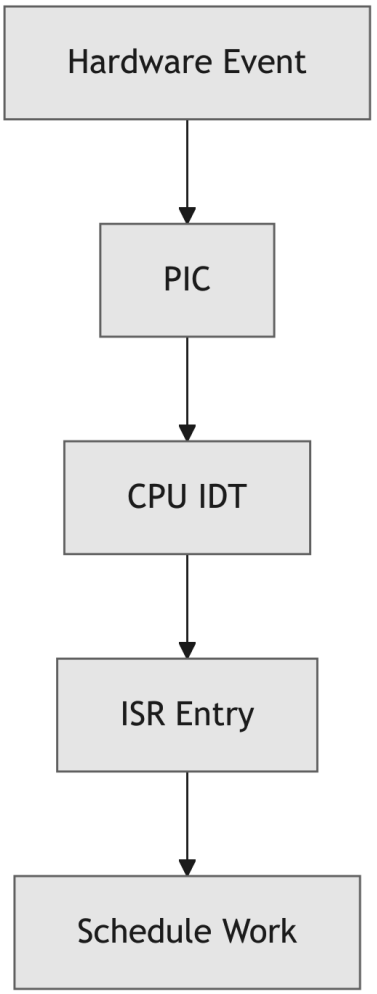
Device Interrupt Handling: From Hardware to Software

A typical device interrupt flow in SVR4 unfolds as follows:

1. **Hardware Event**: A disk controller completes an I/O operation, asserting the `INTR` line.
2. **PIC Arbitration**: The Programmable Interrupt Controller (PIC) determines the highest-priority pending interrupt and signals the CPU with the corresponding interrupt vector.
3. **CPU Dispatch**: The CPU indexes into the IDT, retrieves the handler address, and transfers control.
4. **ISR Execution**: The device driver's ISR acknowledges the interrupt, reads status, enqueues data or signals completion, and returns.
5. **Bottom Half Scheduling**: If the ISR enqueued work (e.g., onto a STREAMS queue), it calls `qenable()`, scheduling the queue's service procedure.
6. **Deferred Processing**: During the next invocation of `runqueues()`, the service procedure processes the enqueued data, potentially passing it up the STREAMS stack or waking a sleeping process.
7. **Return to User**: Finally, `s_trap()` checks for signals and preemption before returning control to the interrupted user process.

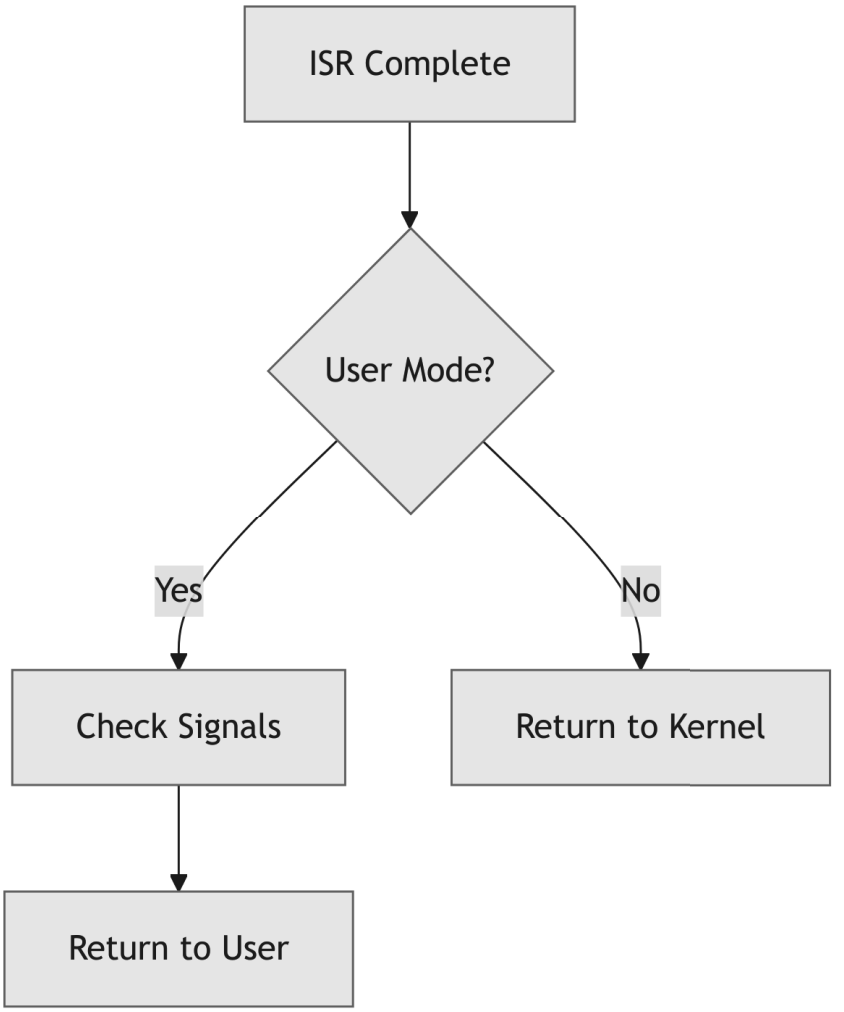
This multi-stage pipeline ensures that latency-sensitive acknowledgment happens immediately, while complex processing is deferred to a safer, more permissive context.

Figure 5.3.1: Interrupt Entry Path



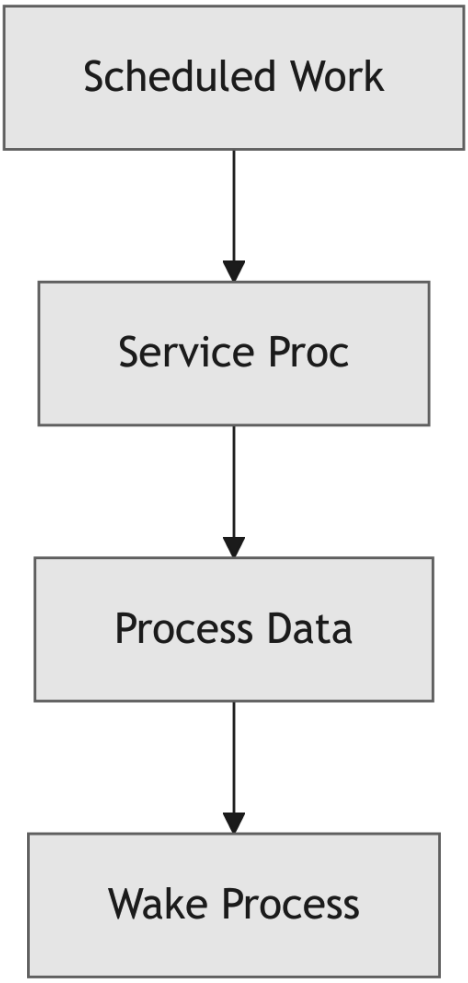
The diagram shows hardware event detection through ISR execution and bottom-half scheduling.

Figure 5.3.2: Return to User Mode



The return path checks for preemption and signals before resuming user execution.

Figure 5.3.3: Bottom Half Processing



Deferred processing in the bottom half handles protocol stack operations.

The Ghost of SVR4: Interrupt Handling Evolution

In 1988, the i386's interrupt architecture was state-of-the-art. The IDT, priority levels, and automatic stack switching provided a robust foundation for kernel interrupt handling. SVR4's two-phase model (top half / bottom half) was a pragmatic solution to the competing demands of responsiveness and throughput.

Yet the model showed its age as systems scaled. The global `spl` locks, while simple, were coarse-grained bottlenecks on multiprocessor systems. Every `spl6()` call serialized access, limiting parallelism.

Modern Contrast (2026): Modern Linux employs **interrupt threads** for most devices—the ISR performs minimal work (acknowledging the interrupt and waking a dedicated kernel thread), and the bulk of processing occurs in that thread context, scheduled like any other task. This allows per-device concurrency, fine-grained locking, and even preemption of interrupt handlers. Additionally, **MSI (Message Signaled Interrupts)** and **MSI-X** on modern PCIe devices bypass the shared PIC entirely, delivering interrupts as memory writes, enabling hundreds of distinct interrupt vectors per device. The per-CPU interrupt handling and lockless data structures of modern kernels achieve parallelism unimaginable in the spl-dominated SVR4 era. Yet the fundamental principles—minimal ISR, deferred processing, careful stack management—endure, testament to the timeless wisdom of SVR4's architects.

Ancient Incantations: The PIC's Whispers

The interrupt handling code in SVR4 still bears the scars of hardware from another era—remnants that persist in modern kernels like hieroglyphs on reused stone:

The 8259A PIC and the EOI Ritual

The End-Of-Interrupt command is a relic of 8259A hardware from 1976. Every ISR must execute:

```
outb(0x20, 0x20);  // Send EOI to master PIC
```

This magic incantation—writing `0x20` to I/O port `0x20` —tells the PIC "I'm done with this interrupt, you may send the next one." Modern kernels still execute this **exact byte sequence** on x86 systems with legacy interrupt controllers. The numbers `0x20, 0x20` are not symbolic—they're hardcoded hardware constants from 1976, when the 8259A's datasheet assigned `0x20` as the "End of Interrupt" command and `0x20` as the master PIC's command port. These magic numbers whisper through four decades of kernel code.

The Privilege Ring Scars

The `systrap` entry point in SVR4's `trap.c` still manipulates `CS` and `SS` segment selectors—relics of i386's four privilege rings (Ring 0-3). x86-64 in 2026 uses only two rings (Ring 0 = kernel, Ring 3 = user), but the segment register manipulation remains, vestigial but functional:

```
// From trap.c - segment selectors still checked
if ((r0ptr[CS] & 0x03) != 0)  // CPL check: were we in user mode?
    s_trap();  // Signal/preemption check
```

That `& 0x03` masks the Current Privilege Level—a two-bit field in the code segment selector that could theoretically be 0, 1, 2, or 3, but in practice is only ever 0 or 3. Rings 1 and 2 are ghosts, defined in silicon but unused by every major OS since the 1990s.

The INT 0x80 Archaeological Site

The `INT 0x80` instruction—SVR4's system call entry point—is an archaeological site. Modern kernels abandoned it for `SYSENTER` (Intel) or `SYSCALL` (AMD), reducing system call latency from ~100 cycles to ~30. But every Linux system compiled with `-m32` (32-bit compatibility mode) still generates `INT 0x80` in C library wrappers. The instruction echoes, a software fossil from an era when software interrupts were the only way to enter Ring 0.

The TSS: A Structure Larger Than Its Use

The Task State Segment on i386 is 104 bytes—enough to save the entire CPU state for hardware task switching. SVR4 allocates one TSS per CPU but uses only 8 bytes of it (the kernel stack pointer fields `SS0:ESP0`). The remaining 96 bytes—segment registers for Ring 1/2, LDT selector, I/O permission bitmap offset—sit idle, allocated but never written. This is a monument to a feature Intel envisioned (hardware multitasking) but no OS ever adopted.

The `cli` / `sti` Dyad

```
cli  ; Clear interrupt flag - ALL interrupts off
sti  ; Set interrupt flag - restore interrupts
```

These two instructions—one byte each, `0xFA` and `0xFB`—appear in every kernel that touches i386. They're the ultimate `spl`: `cli` is `spl7()` (block ALL interrupts), `sti` is `spl0()` (allow all). Modern kernels avoid them (too coarse), but they remain in early boot code and panic handlers—the nuclear option when all else fails.

The Critical Balance: Responsiveness vs. Throughput

Interrupt handling is a perpetual balancing act. Spend too much time in ISRs, and the system becomes sluggish, unable to run user processes. Defer too much, and interrupt latency soars, causing missed deadlines and data loss.

SVR4's design acknowledged this tension explicitly: the top half was kept minimal by design, and the bottom half could be scheduled with varying priorities. Time-critical operations (e.g., STREAMS queue processing for real-time data) could be expedited, while less urgent tasks could be deferred.

This philosophy—**prioritized deferred processing**—is interrupt handling's enduring lesson: immediate acknowledgment, deferred computation, and relentless vigilance against blocking the kernel's heartbeat.

The Block I/O Subsystem: A Most Expeditious Lending Library

In the heart of our kernel, that bustling metropolis of interconnected mechanisms, lies a most peculiar establishment known as the Buffer Cache. One might picture it as a specialised lending library, managed by a shrewd and resourceful chief librarian. This is no ordinary library of books; it is a repository of *data blocks*, those hefty parcels of information recently retrieved from the slow and dusty archives of the disk drives.

When a process requires a piece of data—a block from a file, a directory listing—it does not, in the first instance, dispatch a runner to the distant disk archives. Such a journey is fraught with delay, a veritable horse-and-buggy trip in an age of steam locomotives. Instead, it first enquires at the lending library's front desk. "Have you a copy of block number 42 from the device of spinning rust?" it asks. The librarian, with a practiced eye, consults his index. If the block is present—a *cache hit*—it is handed over with the swiftness of an over-the-counter transaction, saving an immense journey.

If the block is not present—a *cache miss*—then and only then is the runner dispatched. Upon his return, the data is given to the requesting process, but a copy is also entrusted to the librarian, who places it upon his shelves. The most recently used blocks are kept close at hand, while those that languish unrequested for an age are eventually returned to the void to make space for more popular items. This is the soul of the **buffer cache**: an intermediary establishment designed to satisfy the system's ravenous appetite for data with the speed of immediate memory, avoiding, whenever possible, the ponderous mechanics of physical I/O.

The Librarian's Ledger: The `struct buf`

Every block held within this library is accompanied by a meticulously kept ledger entry, a `struct buf`, which tracks its identity, status, and location. This structure is the single most important apparatus in the entire Block I/O subsystem, a master ticket that follows a block from the moment it is requested until it is finally released.

The Buffer Ticket Structure (sys/buf.h:27):

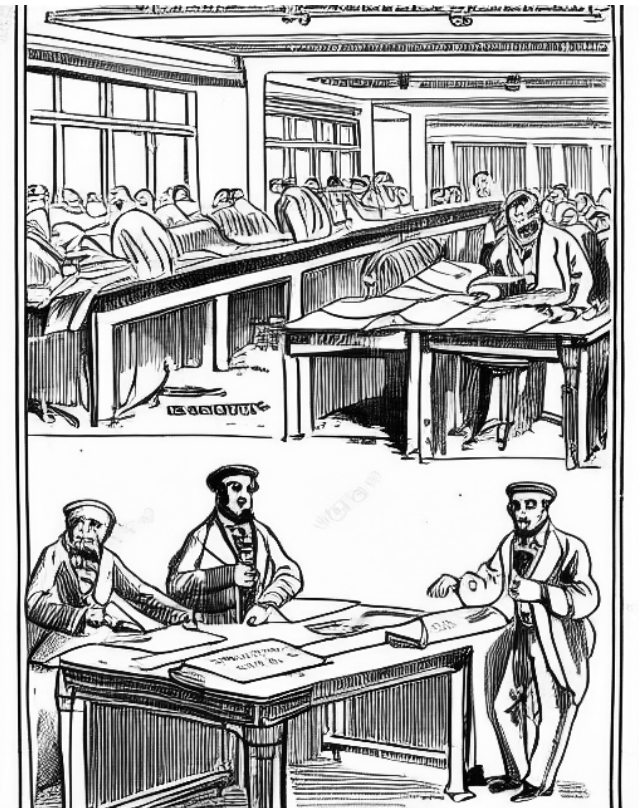
```
typedef struct  buf {
    uint        b_flags;                /* see defines below */
    struct      buf *b_forw;            /* headed by d_tab of conf.c */
    struct      buf *b_back;            /*  "  */
    struct      buf *av_forw;           /* position on free list, */
    struct      buf *av_back;           /*     if not BUSY*/
    o_dev_t     b_dev;                  /* major+minor device name */
    unsigned b_bcount;          /* transfer count */
    union {
        caddr_t b_addr;         /* low order core address */
        int     *b_words;               /* words for clearing */
        daddr_t *b_daddr;               /* disk blocks */
    } b_un;
    daddr_t     b_blkno;                /* block # on device */
    char        b_oerror;               /* OLD error field returned after
I/O */
    unsigned int b_resid;               /* words not transferred after
error */
    struct  proc  *b_proc;              /* process doing physical or swap
I/O */
    struct      page  *b_pages;         /* page list for PAGEIO */
    long        b_bufsize;              /* size of allocated buffer */
    int (*b_iodone)();          /* function called by iodone */
    struct      vnode *b_vp;            /* vnode associated with block */
    dev_t       b_edev;                 /* expanded dev field */
} buf_t;
```

This ledger entry contains all that is needed to manage the block:

- **`b_flags`** : A bitmask describing the buffer's state: is it being read (`B_READ`), being written, is the I/O complete (`B_DONE`), is it busy (`B_BUSY`), or is it a delayed write (`B_DELWRI`)?
- **`b_forw`**, **`b_back`** : Pointers that link the buffer into a hash queue, allowing the librarian to quickly find a block by its device and block number.
- **`av_forw`**, **`av_back`** : Pointers for the *available* list, a separate chain of buffers that are not currently busy and are candidates for reuse. This is the shelf of books ready for lending.
- **`b_dev`**, **`b_blkno`** : The precise identity of the data: its device and block number.
- **`b_un.b_addr`** : The memory address where the actual data resides.
- **`b_bcount`** : The size of the data block in bytes.

The dual-linking scheme is ingenious. Every buffer lives on a hash queue so it can be found quickly. If it is not busy, it *also* lives on the free list. This allows a buffer to be found by its identity

and then swiftly removed from the list of available buffers once it is claimed.



Block I/O - Postal Sorting Center

Checking Out a Block: `getblk` and `bread`

When a process requires a block, it calls upon `bread()` (Block Read), which serves as the public-facing clerk of the lending library. The `bread` function's primary duty is to acquire the block using `getblk()` and, if the block's data is not already valid, to initiate a read from the device.

The true heart of the allocation is `getblk()`, the master librarian who presides over the cache.

The Master Librarian's Search (os/bio.c:425):

```
struct buf *
getblk(dev, blkno, bsize)
    register dev_t dev;
    register daddr_t blkno;
    long bsize;
{
    register struct buf *bp;
    register struct buf *dp, *nbp = NULL;
    register int s;

    /* ... error checking ... */

    blkno = LTOPBLK(blkno, bsize);
    s = spl0();
loop:
    dp = bhash(dev, blkno);
    for (bp = dp->b_forw; bp != dp; bp = bp->b_forw) {
        if (bp->b_blkno != blkno || bp->b_edev != dev
          || bp->b_flags & B_STALE)
            continue;
        spl6();
        if (bp->b_flags & B_BUSY) {
            bp->b_flags |= B_WANTED;
            syswait.iowait++;
            sleep((caddr_t)bp, PRIBIO+1);
            syswait.iowait--;
            spl0();
            goto loop;
        }
        splx(s);
        notavail(bp);
        /* ... buffer resizing logic ... */
        return bp;
    }

    splx(s);
    if (nbp == NULL) {
        nbp = getfreeblk(bsize);
        spl0();
        goto loop;
    }
        bp = nbp;
    /* ... initialize new buffer ... */
    return bp;
}
```

The ritual performed by `getblk` is as follows:

1. **Consult the Index**: It computes a hash from the device and block number (`bhash(dev, blkno)`) to find the appropriate chain of buffers.
2. **Search the Shelf**: It walks the hash chain. If it finds a matching, valid buffer, it checks if it is busy.
3. **Wait if Busy**: If the buffer is found but is `B_BUSY` , another process is currently using it. Our process marks the buffer as `B_WANTED` and enters a deep slumber (`sleep`), to be awoken only when the buffer becomes free. Upon waking, it must restart its search from the beginning, as the state of the cache may have changed entirely.
4. **Claim the Buffer**: If the buffer is found and not busy, `getblk` marks it `B_BUSY` by calling `notavail()` , removing it from the free list, and returns it. This is a cache hit.
5. **Fetch from Afar**: If, after searching the entire chain, no matching buffer is found (a cache miss), `getblk` must procure a new buffer structure, either by finding a free one or allocating a new one via `getfreeblk()` . It then populates this new buffer with the details of the requested block and returns it, though its contents are not yet valid.

Once `getblk` returns a buffer, `bread` inspects it. If the `B_DONE` flag is not set, the data is stale or nonexistent. `bread` then sets the `B_READ` flag, sets the transfer count, and calls the device's *strategy* routine to queue the physical read operation. It then calls `biowait()` , another slumber, waiting for the I/O to complete before returning the freshly-populated buffer to the caller.

Settling the Account: `bwrite` and Delayed Writes

When a process has modified a buffer, it must return it to the library, settling its account. This is accomplished via `bwrite()` or its variants.

A simple `bwrite()` is a synchronous affair. The process hands the buffer over and waits patiently until the data has been safely written to the physical disk.

The Synchronous Write (os/bio.c:166):

```
void
bwrite(bp)
    register struct buf *bp;
{
    register flag;

    sysinfo.lwrite++;
    flag = bp->b_flags;
    bp->b_flags &= ~(B_READ | B_DONE | B_ERROR | B_DELWRI);
    u.u_iow++;
    sysinfo.bwrite++;
    (*bdevsw[getmajor(bp->b_edev)].d_strategy)(bp);
    if ((flag & B_ASYNC) == 0) {
        (void) biowait(bp);
        brelse(bp);
    } else
        basyncnt++;
}
```

The function clears any lingering flags, invokes the device strategy routine to begin the write, and, unless the `B_ASYNC` flag is set, it waits for completion with `biowait()` before finally releasing the buffer with `brelse()`.

More interesting is the case of the **delayed write**, invoked via `bdwrite()`. In this scenario, the process tells the librarian, “I have updated this block, but I may have more changes for it shortly. Do not trouble the disk runner just yet.” The librarian marks the buffer with the `B_DELWRI` flag and places it back on the available list. The physical write is deferred until the system is less busy, or until another process needs this specific buffer for a different purpose. This mechanism is a triumph of efficiency, consolidating multiple small writes into a single, larger I/O operation.

The Ghost of SVR4: A Unified Consciousness

My dear reader, the SVR4 buffer cache, a fine and noble apparatus, was a world unto itself. It was a cache for block devices, distinct and separate from the memory used for file pages. This duality was its defining feature and, in time, its obsolescence. A file's data might exist in the page cache for mapping into a process's address space, and *also* in the buffer cache if read via the `read()` system call. This redundancy, this double-caching, was a waste of our most precious resource: memory.

Modern Contrast (2026): By the year 2026, the very notion of a separate buffer cache has been relegated to the historical archives. The Linux kernel, for instance, employs a **unified page cache**. There is but one library. When a block is read from a disk, it enters the page cache. If a process `read()` s the file, the data is copied from the page cache to the user's buffer. If a process `mmap()` s the file, the very same pages are mapped directly into its address space. There is no duplication.

Furthermore, the simple, doubly-linked lists for hashing and free management have been supplanted by far more sophisticated structures. Linux employs radix trees to track a file's pages and highly advanced I/O schedulers (such as Budget Fair Queueing) that go far beyond our simple `disksort` elevator algorithm. These schedulers can prioritize interactive requests, merge adjacent requests, and ensure fairness among many competing processes. The `struct buf` has been replaced by the `struct bio` , a far more abstract representation of a block I/O request that can describe a collection of non-contiguous memory pages (a "scatter-gather list") to be transferred in a single operation. The soul of my buffer cache lives on, but its form has evolved into a far more integrated and conscious entity.

Conclusion: The Enduring Value of the Cache

The Block I/O subsystem's buffer cache, our metaphorical lending library, stands as a testament to a foundational principle of computing: the profound and inescapable disparity between the speed of the processor and the mechanical languor of the disk. By maintaining a small, intelligently managed collection of frequently used data blocks in fast memory, the kernel mitigates this disparity with remarkable effectiveness.

The mechanisms of `getblk`, `bread`, and `bwrite`, governed by the state chronicled in the `struct buf` ledger, form a complete system for the efficient mediation of data. Whether it is retrieving a block with the speed of a cache hit or deferring a write to a more opportune moment, the buffer cache ensures that the kernel's voracious need for data is met with decorum and dispatch, preventing the entire system from being bogged down in the muddy slowness of the physical disk. It is a simple, elegant, and utterly essential piece of civic infrastructure in the metropolis of the kernel.

Character I/O: A Pneumatic Tube Telegraph

Imagine, if you will, a sprawling government ministry in the age of steam and brass. To convey memoranda, forms, and directives between its myriad departments, the architects have installed a magnificent and intricate **pneumatic tube system**. An urgent request, sealed in a brass canister, is dispatched from the Foreign Office; it travels through a branching network of tubes to the Records Department, where a clerk retrieves it, appends the necessary file, and sends it on to the Ministerial Secretariat for approval. The canister's journey is not fixed; it can be routed through any number of intermediate departments, each of which may inspect, modify, or add to its contents.

This, in essence, is the philosophy of the SVR4 **Character I/O subsystem**, which is built almost entirely upon the elegant and powerful **STREAMS** framework. Where the Block I/O Subsystem deals with large, uniform blocks of data like a freight railway, Character I/O deals with streams of data of arbitrary length and format, like our versatile pneumatic tubes. It is the mechanism that drives terminals, network connections, and other byte-oriented devices, providing a flexible, modular framework for processing data as it flows between user processes and the hardware.

A "Stream" is the data path between a user process and a device driver. This path is constructed by linking together a series of processing modules, each a two-way junction in our tube network. Data, encapsulated in messages, flows up and down this chain, processed at each stage according to the module's function.

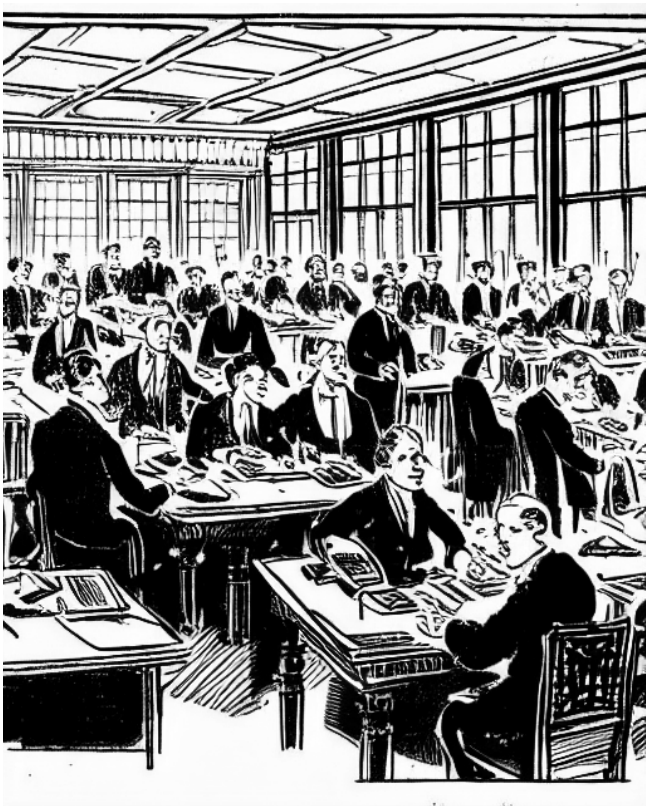
The Grand Directory of Stations: `cdevsw`

At the heart of the Character I/O system lies a grand directory, a master list of every "station" in the pneumatic tube network. This is the **Character Device Switch**, or `cdevsw`, an array defined in `sys/conf.h`. Each entry in this table represents a character device driver, providing the kernel with the initial entry points to establish a new Stream.

The Character Device Switch (sys/conf.h:31):

```
struct cdevsw {
    int (*d_open)();
    int (*d_close)();
    int (*d_read)();
    int (*d_write)();
    int (*d_ioctl)();
    int (*d_mmap)();
    int (*d_segmap)();
    int (*d_poll)();
    /* ... fields omitted for brevity ... */
    struct streamtab *d_str;
    char        *d_name;
    int *d_flag;
};
```

When a process opens a character device file, the kernel consults this table using the device's major number as an index. While there are pointers for direct `read` and `write` functions, for a STREAMS device, the most crucial field is `d_str` . If this pointer is not null, it signals that this device is a portal to the STREAMS framework. The kernel then uses the `streamtab` structure it points to for constructing the initial tube connection to the device driver.



Character I/O - Telegraph Office

The Blueprints of a Tube Line: `streamtab` and `queue`

The `d_str` field in `cdevsw` points to a `streamtab` structure, the blueprint for a STREAMS module or driver. It defines the processing procedures for both the upstream (read-side) and downstream (write-side) flow of data.

The Stream Blueprint (sys/stream.h:182):

```
struct streamtab {
    struct qinit *st_rdinit;
    struct qinit *st_wrinit;
    struct qinit *st_muxrinit;
    struct qinit *st_muxwinit;
};
```

This structure simply points to two `qinit` structures, one for the read queue (`st_rdinit`) and one for the write queue (`st_wrinit`). The `qinit` structure, in turn, contains the function pointers that do the actual work.

The fundamental unit of a Stream is the **queue**. Every module, and indeed the stream head and driver, consists of a pair of queues: a read queue for data flowing upstream (from the device to the user) and a write queue for data flowing downstream.

The Clerk's Workstation: `struct queue` (sys/stream.h:55):

```
struct  queue {
    struct      qinit   *q_qinfo;       /* procs and limits for queue */
    struct      msgb    *q_first;       /* first data block */
    struct      msgb    *q_last;        /* last data block */
    struct      queue   *q_next;        /* Q of next stream */
    struct      queue   *q_link;        /* to next Q for scheduling */
    _VOID               *q_ptr;         /* to private data structure */
    ulong               q_count;        /* number of bytes on Q */
    ulong               q_flag;         /* queue state */
    /* ... water marks and packet sizes ... */
};
```

Each queue acts as a clerk's workstation. It has:

- `q_qinfo` : A pointer to the `qinit` structure, defining the procedures this clerk knows how to perform.
- `q_first, q_last, q_count` : The message queue itself—the clerk's inbox.
- `q_next` : A pointer to the *next* queue in the stream. This is the crucial link that determines where to send the canister next.
- `q_ptr` : A private data pointer for the module to maintain its own state.

When a Stream is first opened, it consists of a **stream head** (the kernel's interface to the user process) connected directly to the **stream driver** (the kernel's interface to the hardware). Later, processing modules can be "pushed" onto the Stream, inserting them between the head and the driver, creating a processing pipeline of arbitrary complexity.

Sending a Canister: Message Passing with `putnext()`

Data and control information travel through a Stream in the form of messages (`mblk_t`). A message is a linked list of blocks, allowing for efficient, zero-copy processing. When a module's clerk has finished with a message, he does not need to know the intricate details of the entire tube network. He simply invokes `putnext()` .

The `putnext` macro is a masterpiece of indirection: `#define putnext(q, mp) ((*(q)->q_next->q_qinfo->qi_putp)((q)->q_next, (mp)))`

It is a simple yet powerful instruction: "Take this message (`mp`), look at *my* `q_next` pointer to find the next station, and deliver it to that station's `put` procedure." This single mechanism allows for the seamless flow of data through the chain of modules. Each module only needs to know about its immediate neighbor, allowing for a highly modular and extensible system. A terminal line discipline module can be pushed, followed by a JSON parsing module, followed by a data logging module, all without any module needing specific knowledge of the others.

The Ghost of SVR4: The Decline of a Grand Design

Ah, STREAMS. It was a grand, unifying theory. A single, beautiful framework intended to solve all of character I/O, from serial ports to the most complex networking protocols. We believed it was the future. Every character device, every TTY, every network interface was a stream. The modularity was intoxicating; one could construct intricate processing pipelines on the fly. We built TCP/IP, UDP, and even parts of the file system upon its elegant foundation.

Modern Contrast (2026): And yet, where is STREAMS now? Vanished, a ghost in the machine. Its very generality became its downfall. The framework, while flexible, imposed a significant overhead. The constant allocation and deallocation of message blocks, the multiple function calls to traverse the queue chain—it was the bureaucracy of our pneumatic tube system made manifest. For simple cases, it was overkill. For high-performance networking, it became a bottleneck.

The architects of Linux, observing from afar, chose a different path. They rejected the single, unified theory in favor of specialized, purpose-built tools. The terminal subsystem in Linux has its own intricate line discipline and TTY handling, optimized for that specific purpose. Networking is handled by the socket layer and the `netdevice` framework, which are brutally efficient and tailored for high-speed packet processing. The `struct file_operations` in a Linux character device driver now points directly to the functions that will handle I/O, with no intermediary queueing system unless the driver itself implements one. STREAMS' complexity, its many interlocking structures— `queue`, `qinit`, `mblk`, `dblk` —were deemed a liability. The grand, unified design was replaced by a pragmatic collection of disparate, but faster, mechanisms.

Conclusion: The Flexible Conduit

The Character I/O subsystem, through its embodiment of the STREAMS framework, provided SVR4 with a remarkably flexible and powerful tool. It treated all forms of character-based data flow as a journey through a dynamic network, capable of modification and extension at will. By breaking down complex processing tasks into a series of modular, interconnected clerks, it allowed for unprecedented code reuse and rapid development of new drivers and protocols.

While the performance demands of the modern era may have led to the ascendancy of more specialized subsystems, the vision of STREAMS—our pneumatic tube network—remains a compelling example of elegant design. It recognized that the journey of data is as important as its destination, and provided a framework to shape that journey with precision and modular grace.

Executable Formats: The Master Artisan's Workshop

Consider a master artisan in his workshop, a place of profound knowledge and skill. Upon his workbench is placed a complex automaton, a marvel of brass and clockwork, brought to him to be animated. The artisan does not simply wind it up; he knows that different makers use vastly different mechanisms. One might use steam pressure, another a series of descending weights, and a third a tightly coiled spring. To attempt to wind a steam-powered automaton would be folly.

Instead, the artisan first peers at the base of the machine, searching for a small, inscribed *maker's mark* or *magic number*. Upon finding it, he turns to a great wooden cabinet that lines the wall of his workshop. This cabinet contains many drawers, each labeled with the mark of a known maker. Inside each drawer is a specific set of tools and instructions—a `execsw` entry—perfectly suited to animating a machine from that particular maker. By matching the mark to the drawer, the artisan selects the correct tools and brings the automaton to life.

This is precisely the challenge and the solution employed by the SVR4 kernel when faced with the `exec()` system call. The kernel is a master artisan, and the various binary files it may be asked to execute—COFF, ELF, or even interpreted scripts—are the automata. It cannot assume a single method of activation. Instead, it relies on a modular system to identify and delegate the task to a specialized handler for that specific executable format.

The Cabinet of Toolsets: `struct execsw`

The kernel's "cabinet of toolsets" is the **executable switch table**, `execsw`, an array of `execsw` structures defined in `sys/exec.h`. This table is the kernel's registry of all known executable types. Each entry represents a complete, self-contained "toolset" for one format.

The Artisan's Toolset Drawer (sys/exec.h:48):

```
struct execsw {
    short *exec_magic;
    int   (*exec_func)();
    int   (*exec_core)();
};

extern int nexectype;
extern struct execsw execsw[];
```

Each entry in the `execsw` array is elegantly simple, containing three essential items:

- **`exec_magic`** : A pointer to the "maker's mark," a short integer (or array of integers) that uniquely identifies the binary format. For an ELF file, this is the number `0x7f45` (the characters `\177ELF`). For a COFF file, it is `0514` (octal).
- **`exec_func`** : A function pointer to the specific "artisan" who knows how to read, map, and prepare this type of executable. This is the heart of the mechanism, pointing to a function like `elfexec()` or `coffexec()` .
- **`exec_core`** : A function pointer to a routine that knows how to generate a core dump file in this specific format, should the process meet an untimely end.

The kernel maintains a global array, `execsw[]` , and a count of its entries, `nexectype` . The `exec()` logic need not know the details of any format; it must only know how to consult this table.

Executable Formats - Customs House

The Master Artisan's Method: `gexec()`

The master artisan who presides over this process is the `gexec()` function, found in `os/exec.c`. When a user calls `exec()`, the generic system call logic performs initial setup (locating the file, checking basic permissions) and then hands control to `gexec()` to perform the magic.

`gexec()`'s method is a disciplined traversal of the `execsw` cabinet.

The `gexec` Ritual (os/exec.c:380, simplified):

```
int
gexec(vpp, args, level, execsz)
    struct vnode **vpp;
    struct uarg *args;
    int level;
    long *execsz;
{
    /* ... variable setup ... */

    /* Read the first few bytes of the file to get the magic number */
    if ((error = exhd_getmap(&ehda, 0, 2, EXHD_NOALIGN, (caddr_t)&mcp)) !=
0) {
        exhd_release(&ehda);
        goto closevp;
    }
    magic = getexmag(mcp);

    /* ... permission and setuid checks ... */

    error = ENOEXEC;
    for (i = 0; i < nexectype; i++) {
        if (execsw[i].exec_magic && magic != *execsw[i].exec_magic)
            continue; /* This is not the right toolset, try the next */

        u.u_execsw = &execsw[i];
        /* We found a match! Invoke the specialist function. */
        error = (*execsw[i].exec_func)
                  (vp, args, level, execsz, &ehda, setid);

        if (error != ENOEXEC)
            break; /* Success or a fatal error, our work is done. */
    }

    /* ... cleanup ... */

    return error;
}
```

The procedure is methodical:

1. **Read the Maker's Mark**: `gexec` first reads the first two bytes of the file to retrieve its `magic` number.
2. **Consult the Cabinet**: It then enters a loop, iterating through every drawer (`execsw` entry) from `0` to `nexectype` .

3. **Find the Right Toolset**: In each iteration, it compares the file's magic number with the `exec_magic` number for that toolset. If they do not match, it `continue` s to the next drawer.
4. **Delegate to the Specialist**: When a match is found, it calls the specialist function pointed to by `exec_func` (e.g., `elfexec`), passing it the file's vnode and user arguments.
5. **Assess the Result**: If the specialist returns `ENOEXEC` , it means that despite the matching magic number, it was not a valid executable of that type (perhaps a corrupted file or a non-executable object file). `gexec` will then continue its search. If any other error (or success) is returned, the loop is broken.

The Specialized Toolsets: `elfexec` and `coffexec`

The functions `elfexec()` (in `exec/elf/elf.c`) and `coffexec()` (in `exec/coff/coff.c`) are the specialist artisans. Once invoked by `gexec` , their job is to understand the intricate internal structure of their respective formats.

They are responsible for:

- Reading the file's main header, program headers, and section headers.
- Mapping the text and data segments into the process's address space using `execmap()` .
- Handling requests for shared libraries, a complex dance of its own.
- Preparing the initial stack frame for the new process.
- Ultimately, preparing the `execenv` structure that tells the kernel how the new process image is laid out in memory.

This division of labor is the key to the system's extensibility. To add a new executable format to the kernel, a programmer would only need to write a new `myformatexec()` function and add a new entry to the `execsw` table, with no changes required to the core `gexec` logic.

The Ghost of SVR4: A Self-Describing Workshop

Our `execsw` table was a fine piece of engineering, but it was a static affair. The cabinet of toolsets was built, polished, and sealed at the factory—that is, when the kernel was compiled. To add support for a new executable format, one had to be a kernel artisan oneself, adding the new toolset to the master `conf.c` file and forging an entirely new kernel. This was a high barrier to entry, reserved for the most dedicated of engineers.

Modern Contrast (2026): The workshop of a 2026 Linux kernel is a far more dynamic and magical place. The `execsw` concept has evolved into `binfmt_misc`, a "miscellaneous binary format" handler. This is not a static, compiled-in table, but a fully dynamic interface exposed through the `/proc` filesystem. A system administrator, without even touching a compiler, can simply `echo` a specially formatted string into `/proc/sys/fs/binfmt_misc/register` to teach the kernel about a new executable type.

This string acts as a recipe, telling the kernel: "If you see a file whose first few bytes match this magic number (or whose filename ends in this extension, e.g., `.py`), do not try to load it yourself. Instead, invoke this user-space interpreter program (e.g., `/usr/bin/python`) and pass it the executable's path as an argument." This has allowed Linux to seamlessly execute Java JAR files, Python scripts, and even Windows `.exe` files via Wine, all without a single change to the core kernel code. The artisan's cabinet is no longer sealed; it is an open book, and any user with sufficient privilege can add a new chapter.

Conclusion: The Wisdom of Delegation

The `exec()` mechanism in SVR4 demonstrates a powerful design pattern: the dispatch table. By separating the generic task of *identifying* a format from the specific task of *interpreting* it, the kernel remains agile and extensible. The master artisan, `gexec`, does not need to be an expert in every type of automaton; he only needs to be an expert in identifying the maker's mark and delegating to the appropriate specialist.

This clear separation of concerns, embodied in the `execsw` table, allowed SVR4 to support both the legacy COFF and the modern ELF formats side-by-side, providing a smooth transition during a critical period in UNIX history. It created a workshop where new toolsets could be added with minimal disruption, ensuring that the kernel could adapt to new and unforeseen types of clockwork automata for years to come.

System Initialization: The Grand Exhibition

The dawn of a new day for the system is a momentous occasion, akin to the grand opening of a magnificent world's fair. Before the gates can be thrown open to the public, a master of ceremonies must ensure that every exhibit is prepared, every gas lamp is lit, and every steam engine is polished and ready. In the SVR4 kernel, this master of ceremonies is the `main()` function, residing in `os/main.c`. It is the very first C function to run after the low-level assembly language boot code has prepared the machine, and its duty is to orchestrate the transformation of a machine from a silent collection of circuits into a vibrant, living UNIX system.

The process is not a chaotic rush, but a meticulously ordered progression. The `main` function follows a strict protocol, first establishing the foundational infrastructure, then systematically bringing each of the kernel's great mechanisms to life. Only when every piece is in its proper place are the first processes created and the system finally opened for business, transitioning from its solitary setup phase into the bustling, multi-user world it was built for.

The Master of Ceremonies: `main()`

The `main()` function is the unwavering conductor of the startup orchestra. Upon being called, it inherits a machine with memory laid out and a provisional stack, but with interrupts disabled and the great subsystems of the kernel lying dormant. Its first actions are foundational.

The Opening Fanfare (os/main.c:113):

```
main(argc, argv)
    int argc;
    char        *argv[];
{
    register int        (**initptr)();
    /* ... */

    inituname();
        cred_init();
        dnlc_init();

    /* ... */

    clkstart();
    spl0();         /* enable interrupts */

    /* ... */
}
```

The ceremony begins:

1. **`inituname()`** : The system's identity—its name, release, and version—is established.
2. **`cred_init()`, `dnlc_init()`** : Foundational data structures for security credentials and filename caching are initialized.
3. **`clkstart()`** : The system's heart, the clock interrupt, begins to beat, giving the kernel its sense of time.
4. **`spl0()`** : With a flourish, the master of ceremonies enables interrupts on the processor. The system is now live and can respond to the outside world.

System Init - World's Fair Opening

The Grand Procession of Subsystems

With the foundations laid and the clock ticking, `main()` turns its attention to the great exhibits of the fair—the kernel's subsystems. It does not call each one by name. Instead, it defers to two great tables of function pointers, `io_init` and `init_tbl`, which constitute the guest list for the ceremony.

The Initialization Tables (os/main.c:196):

```
    for (initptr= &io_init[0]; *initptr; initptr++) {
    (**initptr)();
}

/* ... */

    for (initptr= &init_tbl[0]; *initptr; initptr++)
    (**initptr)();
```

This elegant design allows for great modularity. To add a new subsystem, a programmer need only add a pointer to its initialization function to one of these tables in the configuration files. The `main` function, like a true master of ceremonies, does not need to know the specifics of each exhibit, only that it must be called upon to set itself up. These loops are responsible for initializing every major part of the kernel: the buffer cache (`binit`), the STREAMS framework (`strinit`), device drivers, and filesystems.

After the core subsystems are running, the root filesystem is mounted (`vfs_mountroot()`), making the primary hierarchy of files and directories available for the first time.

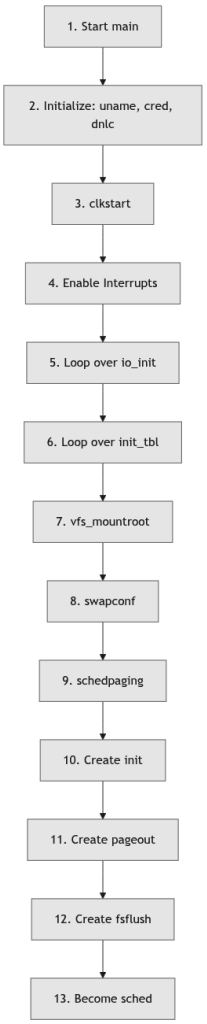


Figure 5.7.2: Flowchart of `main()` Initialization Sequence

The First Citizen: The `init` Process

With the fairgrounds fully prepared, it is time to admit the first citizen. `main()` calls `newproc()` to create the most important process of all: **Process 1**, known to all as `init`.

The Creation of `init` (os/main.c:267):

```
        if (newproc(NP_INIT, NULL, &error)) {
                /* ... setup for a new process ... */

                register proc_t *p = u.u_procp;
        proc_init = p;

        p->p_flag &= ~(SSYS | SLOCK);

                /* Set up the text region to do an exec of /sbin/init. */
                (void) as_map(p->p_as, UVTEXT, szicode, segvn_create,
zfod_argsp);
                if (copyout((caddr_t)icode, (caddr_t)(UVTEXT), szicode))
            cmn_err(CE_PANIC, "main - copyout of icode failed");

        /* ... allocate stack ... */

                return UVTEXT;
    }
```

This is a special case of process creation. The new process, which will have a process ID of 1, does not `fork` from an existing process in the usual way. Instead, `main()` constructs its user-space image from scratch. It maps a text segment and copies in a small piece of machine code known as `icode`. This `icode` is a miniature program whose only purpose is to execute the program `/sbin/init`.

Once this new process is created and made ready to run, the `main()` function's work in this context is done. It will return, and the scheduler will eventually run the newly-birthed `init` process, which will immediately `exec("/sbin/init")`. This user-space `init` program then takes over the task of bringing the system to its operational run-level by reading its configuration from `/etc/inittab` and spawning `getty` processes, daemons, and other essential services.

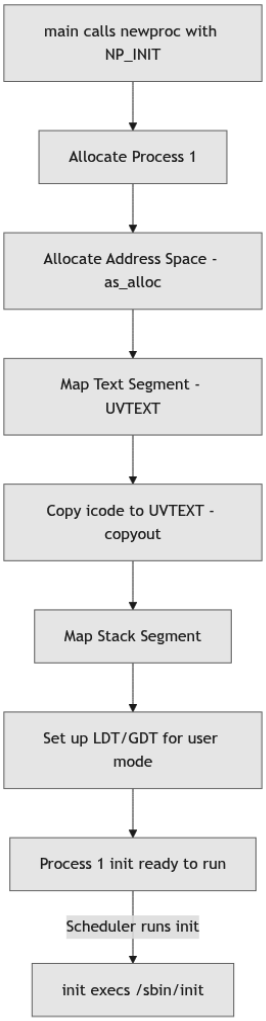


Figure 5.7.1: Flowchart for the Creation of the `init` Process (Process 1)

The Unseen Workforce: Kernel Daemons

Before its work is truly finished, `main()` creates a few more essential, but unseen, members of the exhibition staff. These are kernel-only processes, daemons that run perpetually in kernel mode to perform essential housekeeping tasks.

- **`sched`** : The original process, PID 0, having fulfilled its duty of creating all others, transforms itself into the scheduler, or “swapper.” It is the process that is run when no other

process is runnable. Its primary task is to manage memory, swapping processes out to disk when memory is scarce.

- **`pageout`** : Created via `newproc(NP_SYSPROC, ...)` and given the name “pageout,” this daemon is responsible for the page replacement algorithm, scanning memory for old, unused pages and freeing them to make room for new allocations.
- **`fsflush` (or `bdflush`)**: Another system process, this daemon periodically flushes “dirty” buffers from the block I/O cache to disk, ensuring that modified data is eventually made permanent.

These daemons are the tireless workforce of the system, laboring in the background to ensure the smooth operation of the grand exhibition.

The Ghost of SVR4: A Pre-ordained Ceremony

My dear reader, our initialization was a simple, linear, and deeply predictable affair. The `main` function called its initialization routines in a fixed order, `init` was born, and it, in turn, read `/etc/inittab` —a simple, sequential script of commands—to bring the system to life. `inittab` defined a set of "run-levels" and the processes to start or stop for each. It was robust, it was understandable, but it was not particularly swift. Each step waited for the last, a slow and deliberate procession.

Modern Contrast (2026): The opening ceremony of a 2026 Linux system is a different beast entirely. The simple, sequential `/sbin/init` has been almost universally replaced by `systemd` , a vastly more complex and powerful initialization system. `systemd` does not think in terms of sequential scripts, but in terms of "units"—services, mount points, sockets, and targets. It analyzes a vast web of dependencies between these units, determining that, for instance, the network filesystem service depends on the network being active, which in turn depends on the network card's driver being loaded.

Armed with this dependency graph, `systemd` performs a marvel of parallelization. It starts dozens, or even hundreds, of services simultaneously, activating only those that are not blocked by an unmet dependency. It uses socket activation, where a service is not even started until the first connection to its network socket arrives. The result is a boot process that is dramatically faster and more efficient than our stately, sequential march. The master of ceremonies no longer directs a simple procession; he conducts a complex and dazzling fireworks display, with dozens of rockets launching at once in a perfectly coordinated, yet parallel, spectacle.

Conclusion: Opening the Gates

The journey from a cold hardware state to a fully operational, multi-user system is a masterclass in orchestration. The `main` function, as the master of ceremonies, does not attempt to manage every detail itself. Instead, it relies on a well-defined protocol and a modular architecture, delegating the specifics of initialization to the individual subsystems. It erects the foundational pillars of the system, calls upon the exhibitors to prepare their displays, and, as its final and most important act, creates the first citizen, the `init` process, to whom it hands the keys to the city. With the unseen workforce of kernel daemons humming in the background, the grand exhibition is declared open, and the system comes to life.

Clock and Timer Management: The Keeper of the Great Clock

At the very center of the kernel-city stands a great clock tower. It does not merely tell time; it is the source of the city's pulse, the regular, rhythmic heartbeat that governs the pace of all activity. From this tower, a chime echoes at a constant, unwavering frequency. This chime—the **clock interrupt**—is the fundamental unit of time for the kernel, the metronome against which all process scheduling, time-slicing, and delays are measured.

The keeper of this clock is a sleepless entity, an interrupt handler that, upon every chime, rouses itself to perform the system's most essential temporal duties. It increments the system's notion of uptime, decrements the time slices of running processes, and, most importantly, consults a great bulletin board affixed to the tower's base. This board lists notices of tasks to be performed at specific future times. It is this mechanism, the `callout` table, that allows the kernel to remind itself to perform an action not now, but at some precise moment in the future.

The Heartbeat: `clkstart()` and the Clock Interrupt

Before the city can truly awaken, its heart must begin to beat. This is the responsibility of the `clkstart()` function, called from `main()` during the system's initialization. Its job is to program the machine's hardware—the Programmable Interval Timer (PIT)—to begin generating interrupts at a fixed frequency, typically 100 times per second (100 Hz).

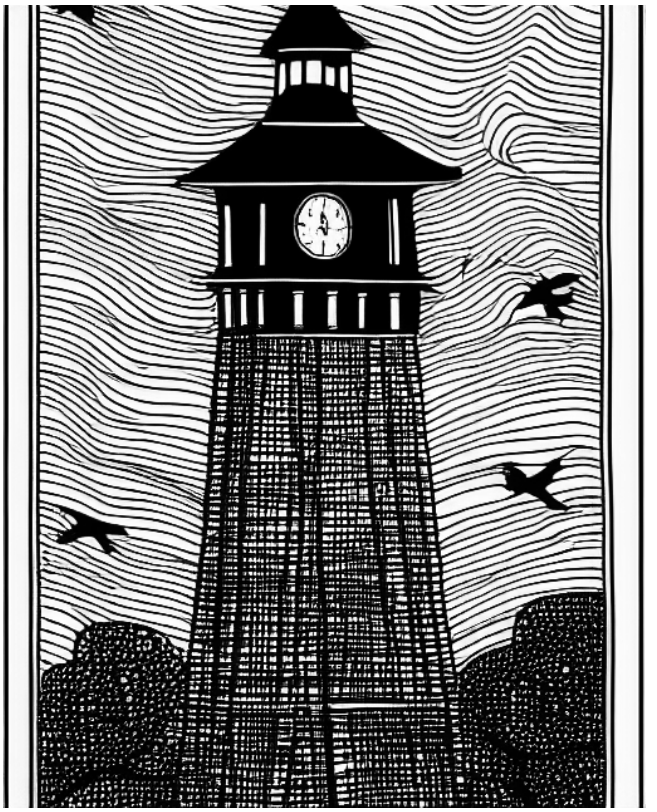
Winding the Great Clock (ml/pit.c:54):

```
clkstart()
{
    unsigned int        flags;
    unsigned char       byte;

    /* ... hardware-specific calculations ... */

    intr_disable();         /* disable interrupts */
    /* We use only timer 0, so we program that. */
    outb(pitctl_port, pit0_mode);
    byte = clknumb;
    outb(pitctr0_port, byte);
    byte = clknumb>>8;
    outb(pitctr0_port, byte);

    /* ... initialize high-resolution timer variables ... */

    intr_restore();         /* restore interrupt state */
}
```

Once `clkstart()` has run, the hardware is armed. At every tick (e.g., every 10 milliseconds), it sends an electrical signal to the processor, triggering an interrupt. The processor immediately stops its current activity, saves its state, and jumps to the `clock()` function in `os/clock.c`, the interrupt service routine.

This `clock()` function is the tireless keeper of time. On every invocation, it:

- Increments the global `lbolt` counter, a variable that tracks the number of ticks since the system booted.
- Updates the user and system time accounting for the currently running process.
- Checks if any process's alarm timer (`p_clktim`) has expired and, if so, posts a `SIGALRM` signal.
- Decrements the time slice of the current process, marking it for rescheduling if its quantum is exhausted.
- Crucially, it checks if any notices posted on the time-ordered bulletin board are now due.



Clock and Timers - City Clock Tower

The Bulletin Board: The `callout` Table

Affixed to the base of the clock tower is a public bulletin board where any part of the kernel can post a notice for a future action. This is the **`callout` table**, a simple array of `callo` structures defined in `sys/callo.h`.

A Notice for Future Action (sys/callo.h:17):

```
struct  callo
{
    int c_time;         /* incremental time */
    int c_id;           /* timeout id */
    caddr_t     c_arg;          /* argument to routine */
    void        (*c_func)();    /* routine */
};
extern  struct  callo   callout[];
```

Each `callo` entry, or “notice,” contains the four elements needed for a delayed function call:

- **`c_time`** : The absolute `lbolt` value at which the function should be executed.
- **`c_func`** : A pointer to the function that is to be called.
- **`c_arg`** : The argument to be passed to that function.
- **`c_id`** : A unique identifier for this specific request, so that it might be cancelled before it occurs.

The `callout` array is managed as a **heap**, a data structure that allows for very efficient retrieval of the entry with the smallest `c_time` . This means the notice that is due to expire soonest is always at the very top of the heap (at `callout[0]`), allowing the `clock()` function to check for due notices with extreme efficiency. It need only look at the first entry; if its time has not yet come, then no other entry’s time has come either.

Posting a Notice: `timeout()`

When a driver or subsystem needs to schedule an action for the future—for instance, to check for a response from a slow device in half a second—it uses the `timeout()` function.

Pinning a Notice to the Board (os/clock.c:380):

```
int
timeout(fun, arg, tim)
    void (*fun)();
    caddr_t arg;
    long tim;
{
    register struct     callo   *p1;
    register int        j;
    int t;
    int id;
    int s;

    t = lbolt + tim;            /* absolute time in the future */

    s = spl7();

    if ((j = calllimit + 1) == v.v_call)
        cmn_err(CE_PANIC,"Timeout table overflow");

    /* Add the new entry and then restore the heap property. */
    calllimit = j;
    j = heap_up(t, j);

    p1          = &callout[j];
    p1->c_time  = t;
    p1->c_func  = fun;
    p1->c_arg   = arg;
    p1->c_id    = id = ++timeid;

    splx(s);
    return id;
}
```

The `timeout` function's procedure is straightforward:

1. It calculates the absolute time `t` in the future when the function should run by adding the requested delay `tim` to the current `lbolt`.
2. It raises the interrupt priority level (`spl7()`) to ensure the `callout` table is not modified by an interrupt while it is being manipulated.
3. It finds the next available slot in the `callout` array.
4. It calls `heap_up()` to take this new notice and percolate it up the heap until it finds its correct, time-ordered position.
5. It fills in the function pointer, argument, and unique ID, and returns the ID to the caller.

This ID can later be passed to `untimeout()` to find and remove the notice from the board before it has a chance to execute, cancelling the pending action.

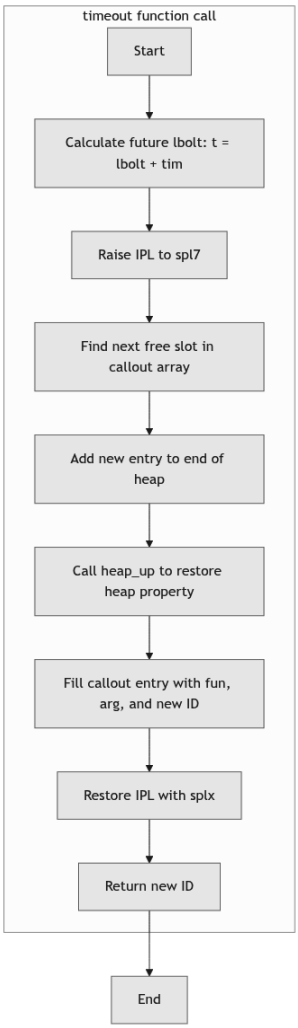


Figure 5.8.1: Flowchart for `timeout()` Function

The Ghost of SVR4: The Humble Heap

In my day, the `callout` table, organized as a heap, was a perfectly serviceable mechanism. With a system-wide `HZ` of 100, we could only schedule events with a granularity of 10 milliseconds, which was more than sufficient for the hardware of the era. The heap structure ensured that adding and removing timers was reasonably efficient, with a cost proportional to the logarithm of the number of pending timeouts—a great improvement over the simple, unsorted array of my ancestors, which had to be scanned linearly on every clock tick!

Modern Contrast (2026): The great clock in a 2026 Linux kernel is a far more sophisticated instrument. The simple heap has been replaced by hierarchical **timer wheels**. A timer wheel is an ingenious data structure that groups timers into buckets based on when they are set to expire. Timers due in the very near future are in a “fast” wheel with fine-grained slots; timers due far in the future are in a “slow” wheel with coarse-grained slots. As time progresses, entire buckets of timers from the coarser wheels are “cascaded” down into the finer ones. This allows the kernel to manage hundreds of thousands of pending timers with near-constant-time overhead for insertion, deletion, and expiration checking.

Furthermore, modern systems are no longer bound by a single, slow `HZ` . They support **high-resolution timers (hrtimers)** and a “tickless” or “dynamic tick” kernel. If there are no immediate events to service, the kernel can tell the hardware to stop sending interrupts altogether, allowing the CPU to enter deep sleep states to conserve power. It will then program the clock hardware to fire a single interrupt at the precise nanosecond the very next timer is due to expire. The city’s great clock no longer chimes ceaselessly; it has learned to chime only when a notice on the bulletin board is truly ready for action.

Conclusion: The City’s Pacemaker

The clock and timer mechanisms are the unsung heroes of the operating system, the essential pacemakers that ensure the orderly progression of time and the timely execution of deferred work. The clock interrupt provides the fundamental rhythm, while the `callout` table and its associated `timeout()` function provide a robust and efficient means for any part of the kernel to schedule its own future. Like the great clock tower at the city’s heart, this system provides a central, reliable point of temporal coordination, allowing the complex, asynchronous world of a multitasking kernel to function with precision and grace.

DMA and Buffer Management: The Architect's Drafting Table

Let us imagine a grand architectural firm where plans are drawn up on an enormous drafting table, representing the system's memory. The master architect—the CPU—can reach any part of this vast surface with ease. However, to assist with the tedious task of copying and moving sections of the blueprints, the firm employs a team of junior drafters. These juniors are the **Direct Memory Access (DMA)** controllers, specialized hardware that can move data between memory and I/O devices without involving the master architect, freeing him for more complex work.

A difficulty arises. Some of the most reliable and time-tested of these junior drafters, the venerable ISA-bus devices, are of a shorter stature. They cannot reach the upper portions of the enormous table; their access is physically limited to the lower 16 megabytes of the surface. When a blueprint located in the upper reaches of the table needs to be copied by one of these drafters, a special procedure is required. This is the central challenge of DMA and buffer management in a 32-bit system that must maintain compatibility with 24-bit-address devices: how to manage data transfers when the source or destination lies beyond the reach of the hardware assigned to the task.

The Universal Work Order: `struct dma_cb`

To bring order to this process, every DMA request, regardless of the drafter who will fulfill it, is described by a universal work order. This is the DMA Command Block, or `dma_cb`, defined in `sys/dma.h`. This structure contains every parameter needed to specify a data transfer.

The DMA Command Block (sys/dma.h:54):

```
struct dma_cb {
    struct dma_cb  *next;       /* free list link */
    struct dma_buf *targbufs;   /* list of target data buffers */
    struct dma_buf *reqrbufs;   /* list of requestor data buffers */
    unsigned char  command;     /* Read/Write/Translate/Verify */
    unsigned char  targ_type;   /* Memory/IO */
    unsigned char  reqr_type;   /* Memory/IO */
    unsigned char  targ_step;   /* Inc/Dec/Hold */
    unsigned char  reqr_step;   /* Inc/Dec/Hold */
    unsigned char  trans_type;  /* Single/Demand/Block/Cascade */
    unsigned char  targ_path;   /* 8/16/32 */
    unsigned char  reqr_path;   /* 8/16/32 */
    unsigned char  cycles;      /* 1 or 2 */
    unsigned char  bufprocess;  /* Single/Chain/Auto-Init */
    /* ... */
    int            (*proc)();   /* address of application call routine */
};
```

This comprehensive work order specifies the source (`reqrbufs`), the destination (`targbufs`), the direction (`command` : Read or Write), and the precise hardware parameters of the transfer, such as the data path width and transfer mode. By abstracting the request into this common structure, the kernel can use a generalized set of functions to manage DMA for a wide variety of devices.

DMA Buffers - Warehouse Loading Docks

The Scribe’s Assistant: `dma_breakup`

The most common and vexing problem is a single I/O request that is too large to be handled in one go, especially if it crosses a memory boundary that the hardware cannot traverse (such as the 16MB boundary for an ISA device). In this case, a scribe’s assistant must intervene to break the large, contiguous request into a series of smaller, manageable transfers. This assistant is the `dma_breakup()` function.

When a block device driver initiates a transfer on a buffer (`buf_t`) that is known to have such limitations, it does not call the hardware strategy routine directly. Instead, it calls `dma_breakup()` , passing its own strategy routine as an argument.

The Breakup Logic (io/physdsk.c:74-171, excerpt):

```
void
dma_breakup(strat, bp)
void (*strat)();
register struct buf *bp;
{
    register int iocount;
    register char *va;
    register int cc, rw, left;
    register int firsttime;

    rw = bp->b_flags & B_READ;
    firsttime = 1;
    iocount = bp->b_bcount;
    if (dbbnd(bp->b_un.b_addr) < NBPSCTR) {
        char *iovaddr;

        iovaddr = bp->b_un.b_addr;
        va = kseg(1);
        if (va == NULL) {
            bp->b_flags |= B_ERROR | B_DONE;
            bp->b_error = EAGAIN;
            return;
        }
        bp->b_un.b_addr = va;
        do {
            bp->b_bcount = cc = min(iocount, NBPP);
            left = iocount - cc;
            bp->b_flags &= ~B_DONE;
            if (rw == B_READ) {
                (*strat)(bp);
                spl6();
                while ((bp->b_flags & B_DONE) == 0) {
                    bp->b_flags |= B_WANTED;
                    SLEEP(bp);
                }
                spl0();
                if (bp->b_flags & B_KERNBUF)
                    bcopy(va, iovaddr, bp->b_bcount - bp->b_resid);
                else
                    copyout(va, iovaddr, bp->b_bcount - bp->b_resid);
                iovaddr += (bp->b_bcount - bp->b_resid);
                iocount -= (bp->b_bcount - bp->b_resid);
            } else {
                if (bp->b_flags & B_KERNBUF)
                    bcopy(iovaddr, va, cc);
                else
                    copyin(iovaddr, va, cc);
```

```
                (*strat)(bp);
                spl6();
                while ((bp->b_flags & B_DONE) == 0) {
                    bp->b_flags |= B_WANTED;
                    SLEEP(bp);
                }
                spl0();

                iovaddr += (bp->b_bcount - bp->b_resid);
                iocount -= (bp->b_bcount - bp->b_resid);
            }
        } while (iocount > 0);
        unkseg(va);
    }
}
```

`dma_breakup` is a master of delegation. It creates a temporary, secondary buffer header and uses it to submit a series of smaller I/O requests to the underlying driver's strategy routine, one for each physically contiguous chunk of the original request. It meticulously tracks the progress, waiting for each small piece to complete before submitting the next, until the entire original request has been satisfied.

The Lowly Drafting Stool: `dmaable` Buffers

Sometimes, breaking up the request is not enough. If the data's final destination is in a memory region that the junior drafter simply cannot reach, an intermediate staging area is required. This is the purpose of the special pool of **DMA-able memory**.

At boot time, the system identifies all physical memory that lies below the 16-megabyte line—the part of the drafting table everyone can reach. A special pool of memory pages, `dmaable_pages`, is reserved from this region. When a device needs to perform DMA to or from a high memory address, the kernel intervenes:

1. It allocates a temporary buffer from the special DMA-able pool.
2. **On a write:** It copies the user's data *from* the high-memory buffer *to* the temporary DMA-able buffer. It then instructs the DMA controller to write from this temporary buffer to the

device.

3. **On a read:** It instructs the DMA controller to read from the device *into* the temporary DMA-able buffer. Once the DMA is complete, the kernel copies the data *from* the temporary buffer *to* the user's final destination in high memory.

This process, known as "double-buffering" or creating a "bounce buffer," is managed by functions like `dmaable_rawio()` in `io/dmacheck.c`. It is inefficient—it requires an extra copy operation by the CPU—but it is the essential accommodation that allows modern, high-memory systems to work harmoniously with older, address-limited hardware. It is the architect providing a lowly drafting stool for the junior drafter to stand on, enabling him to complete his work.

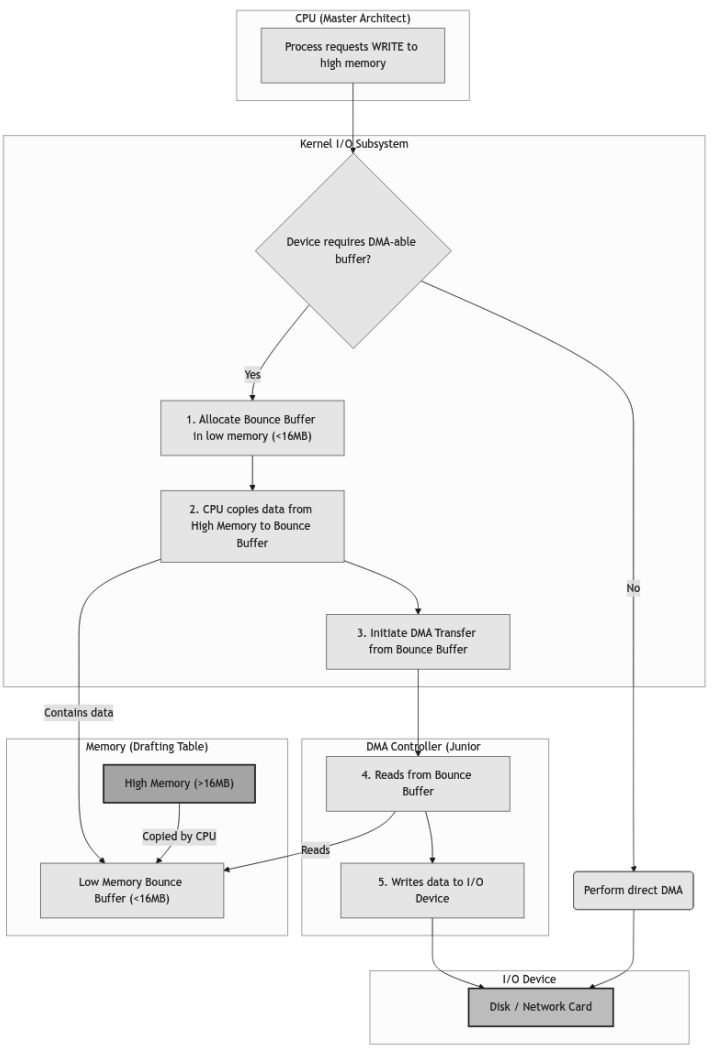


Figure 5.9.1: Data Flow with a DMA Bounce Buffer

The Ghost of SVR4: The Magic of the IOMMU

We went to such great lengths to accommodate our less capable devices! The logic in `dmacheck.c` is a testament to this, a complex web of buffer checks, page allocations, and data copying, all to manually bridge the gap between the 24-bit world of ISA and the 32-bit world of the CPU. We maintained separate free lists for DMA-able pages and non-DMA-able pages, and the `dma_breakup` function was a constant companion for any driver author. It was a source of great complexity, but it was the price of compatibility.

Modern Contrast (2026): The modern architect's workshop has been graced with a magical new piece of furniture: the **IOMMU (Input/Output Memory Management Unit)**. This piece of hardware sits between the I/O bus and the main memory bus and acts as a universal translator for addresses. When a device attempts to perform DMA, the IOMMU intercepts the address. It maintains its own set of page tables, conceptually similar to the CPU's MMU, which map the device's limited "I/O Virtual Addresses" to arbitrary physical memory addresses.

A 32-bit device can be instructed to write to address `0x100000`, and the IOMMU will translate that on the fly to physical address `0x8C000000` in high memory. The device believes it is operating in a simple, contiguous memory space, while the IOMMU is transparently and efficiently scattering its reads and writes all over the physical RAM. Bounce buffers are no longer needed. The `dma_breakup` function becomes largely obsolete. The complex software logic for managing DMA-able pools vanishes, replaced by the simple act of programming the IOMMU's translation tables. The lowly drafting stool has been replaced by a magical, self-adjusting floor that brings any part of the great table effortlessly within reach.

Conclusion: Bridging the Divide

DMA buffer management in SVR4 is a pragmatic and resourceful solution to a difficult problem. It acknowledges the physical limitations of hardware and provides a robust software framework to bridge the divide between devices of different capabilities. The universal language of the DMA Command Block allows for standardized requests, while the `dma_breakup` function and the management of a dedicated `dmaable` memory pool provide the necessary accommodations for older devices. This system, while complex, is a perfect illustration of the kernel's role as a master mediator, creating a harmonious and functional whole from disparate and sometimes challenging parts. It ensures that every drafter in the firm, no matter their stature, can contribute to the final blueprint.

Console and Terminal I/O: The Telegraph Office

In the bustling kernel-city, communication with the outside world—with the user at their keyboard—is handled by a specialized and highly structured institution: the Telegraph Office. This is not the simple freight depot of the Block I/O Subsystem, which deals in large, uniform crates of data. The Telegraph Office deals in the nuanced and often irregular flow of individual characters, the dots and dashes of conversation that must be received, interpreted, formatted, and dispatched with precision. This is the domain of the SVR4 **Terminal I/O Subsystem**.

At its core, this system is a masterclass in the **STREAMS** framework. A terminal is not a single, monolithic driver, but a vertical assembly of message-passing modules, a processing pipeline that transforms raw electrical signals into the polished, line-oriented text that user programs expect. Each keystroke is a telegraphic signal, sent from the hardware up through a series of clerks, each performing a specific duty, until a complete, coherent message is ready for delivery to the application.

The Telegraph Operator: The `asy` Driver

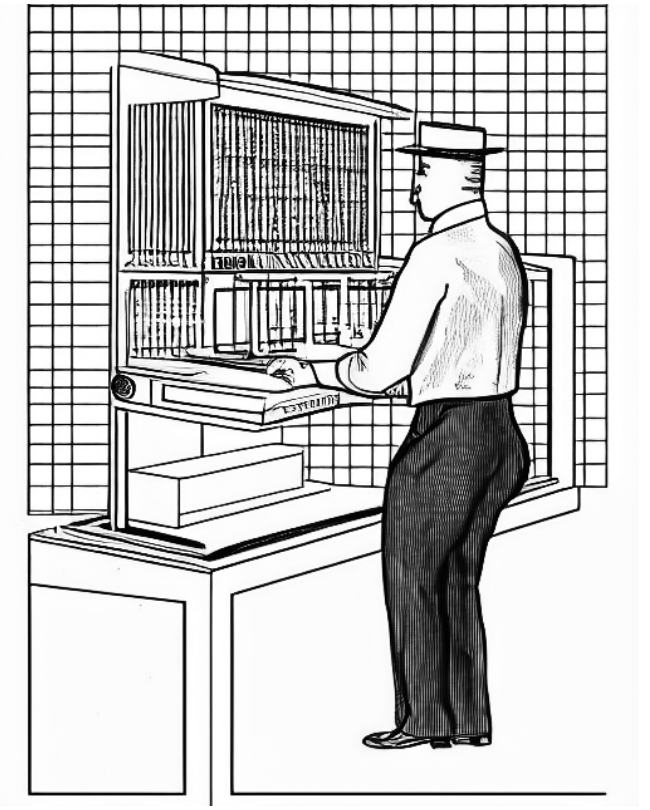
At the lowest level, closest to the hardware, sits the telegraph operator. This is the **asynchronous serial communications driver**, or `asy`. This driver's sole concern is the physical manipulation of the serial port hardware—the UART chip—translating electrical voltages into bytes, and bytes back into voltages. It knows nothing of lines, words, or erase characters; it knows only of data registers, status bits, and interrupts.

The operator's workstation is defined by the `asy` structure, which holds the hardware port addresses for a given serial line.

The Operator's Workstation (sys/asy.h:147):

```
struct asy{
    int         asy_flags;
    unsigned    asy_dat;        /* data register port */
    unsigned    asy_icr;        /* interrupt control register port */
    unsigned    asy_isr;        /* interrupt status register port */
    unsigned    asy_lcr;        /* line control register port */
    unsigned    asy_mcr;        /* modem control register port */
    unsigned    asy_lsr;        /* line status register port */
    unsigned    asy_msr;        /* modem status register port */
    /* ... */
};
```

When a character arrives at the serial port, an interrupt is generated. The `asy` driver's interrupt handler awakens, reads the byte from the `asy_dat` port, packages it into a STREAMS message (`mblk_t`), and sends it "upstream" to the next clerk in the pipeline. It is a simple, mechanical task, performed with speed and no interpretation.



Console Terminal - Telegraph Station

The Formatting Clerk: The `ldterm` Module

The raw bytes sent up from the `asy` operator are not yet fit for consumption by a user application. A user typing “h”, “e”, “l”, “l”, “o”, “backspace”, “p” expects the application to read the line “help”. The task of interpreting these raw keystrokes, handling line-editing conventions, processing special characters, and assembling complete lines falls to the formatting clerk: the **line discipline module**, `ldterm`.

`ldterm` is a STREAMS module that is “pushed” on top of the `asy` driver. It intercepts the stream of characters and buffers them, a process known as *canonical processing*. It maintains a complex state machine for each terminal session, defined in `struct ldterm_mod`.

The Clerk’s Ledger (sys/ldterm.h:50):

```
typedef struct ldterm_mod {
    mblk_t *t_savbp;            /* Chars saved up for a single message */
    mblk_t *t_echomp;   /* Characters waiting to be echoed */
    int t_msglen;               /* # of chars in saved message */
    long t_state;               /* internal state of ldterm module */
    struct termios t_modes;     /* Current terminal modes */
    unsigned char *t_eucp;      /* Ptr to euc width structure */
    /* ... many other state fields ... */
} ldtermstd_state_t;
```

When `ldterm`’s “put” procedure (`ldtermrput`) receives a data message from the driver, it does not immediately pass it on. Instead, it processes each character according to the terminal’s current modes (`t_modes`):

- If it’s a normal character, it is echoed back down the stream (if `ECHO` is set) and added to the line buffer (`t_savbp`).
- If it’s the `ERASE` character (e.g., backspace), `ldterm` removes the previous character from its buffer and sends the appropriate erasure sequence (e.g., backspace-space-backspace) back down to the terminal for display.
- If it’s the `KILL` character, the entire line buffer is cleared.
- If it’s the `EOF` or newline character, the contents of the line buffer are packaged into a single message and finally sent upstream to the stream head, where the user’s `read()` call will be satisfied.

This entire formatting service is provided by the `ldterm` module, which is defined by its STREAMS entry points.

The Line Discipline's `qinit` Structure (io/ldterm.c:104):

```
static struct qinit ldtermrinit = {
    ldtermrput, /* The 'put' procedure for the read-side */
    ldtermrsrv, /* The 'service' procedure for the read-side */
    ldtermopen,
    ldtermclose,
    NULL,
    &ldtermmiinfo
};

struct streamtab ldterm_info = {
    &ldtermrinit,       /* Read-side processing */
    &ldtermwinit,       /* Write-side processing */
    NULL,
    NULL
};
```

This structure is the key to its identity as a STREAMS module, providing the kernel with the function pointers needed to plumb it into the stream.

Connecting the Office: The STREAMS Pipeline

The true power of this model is how these independent components are connected. When a user opens a terminal device like `/dev/tty01`, the kernel sees that the `asy` driver has a `d_str` pointer in its `cdevsw` entry. This tells the kernel to create a STREAMS pipeline. Initially, the **stream head** (the kernel's interface for the user's `read` and `write` calls) is connected directly to the `asy` driver.

Then, through an `I_PUSH ioctl()` call, the `ldterm` module is pushed onto the stream. The kernel re-wires the pointers: the stream head now points to `ldterm`, and `ldterm` points to `asy`. A two-way communication path is formed:

- **Upstream (Read):** `asy` -> `ldterm` -> Stream Head -> `read()`

- **Downstream (Write):** `write()` -> Stream Head -> `ldterm` -> `asy`

This modular pipeline allows for immense flexibility. One could, for instance, push a “JSON formatting” module on top of `ldterm` to automatically parse terminal input, or a network module below it to create a remote terminal session, all without modifying the other components.

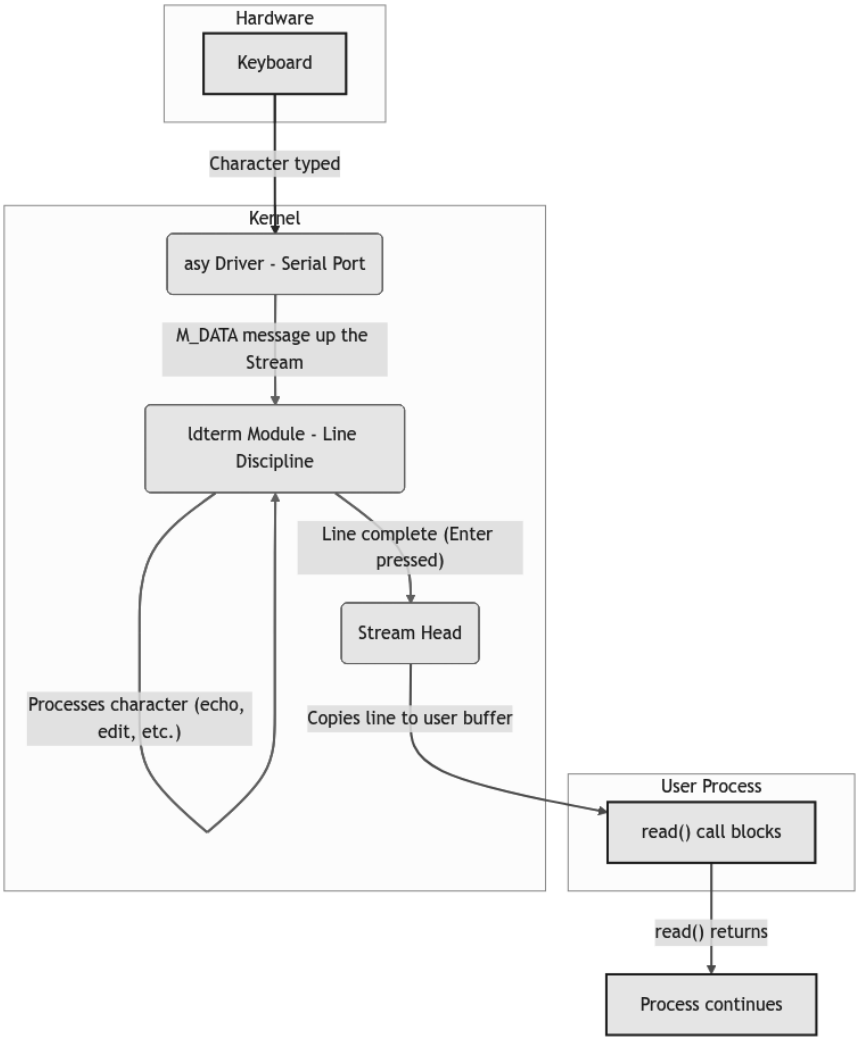


Figure 5.10.1: TTY STREAMS Data Flow from Hardware to User Process

The Ghost of SVR4: The Universal Erector Set

We saw in STREAMS a universal building block, an Erector Set for I/O. The TTY subsystem was its most perfect expression. We built everything with it. The console, serial ports, network pseudo-terminals for `rlogin` and `telnet` —they were all just different stacks of STREAMS modules. A pseudo-terminal (`ptem`) driver on the bottom, `ldterm` in the middle, and the stream head on top. This modularity was the pinnacle of our design philosophy.

Modern Contrast (2026): The Linux kernel, in its relentless pursuit of performance, viewed our elegant Erector Set as cumbersome. The overhead of allocating message blocks for every character, the function call chain to pass them up and down the stream—it was, frankly, slow. For a 9600-baud serial line, this was of no consequence. For a modern gigabit network connection masquerading as a terminal via SSH, it is a significant bottleneck. The modern Linux TTY subsystem, while still supporting the same `termios` interface, is a far more monolithic affair.

The line discipline is no longer a swappable STREAMS module but a tightly integrated set of functions (the `n_tty.c` discipline) compiled directly into the kernel's TTY core. The complex machinery of `mblk_t` messages is replaced by a simpler `tty_buffer` and `flip_buffer` mechanism. While pseudo-terminals still exist, they are a specialized driver, not a generic STREAMS component. The result is a system that is significantly faster for terminal I/O, but which has sacrificed the universal modularity we so prized. The specialized workshops have triumphed over the general-purpose Telegraph Office.

Conclusion: From Raw Signals to Polished Lines

The SVR4 console and terminal subsystem is the quintessential example of the power and elegance of the STREAMS architecture. It takes the raw, uninterpreted signals from the hardware and, through a disciplined chain of clerks, transforms them into the structured, line-oriented data that programs expect. The `asy` driver works the wire, and the `ldterm` module works the language.

This separation of concerns—physical transmission from logical formatting—creates a clean, powerful, and extensible system. Like a well-run telegraph office, it ensures that every message is not only received but is also understood, corrected, and properly formatted before it reaches its final destination, providing the seamless conversational interface that is the hallmark of a UNIX system.

Conclusion

This technical guide has examined the SVR4 i386 kernel implementation across five major subsystems: process management, memory management, file systems, networking, and I/O device management. The analysis reveals a mature, well-architected Unix kernel that balances performance, portability, and maintainability.

Key Architectural Principles

The SVR4 kernel demonstrates several enduring design principles:

Separation of Mechanism and Policy: The scheduling class framework separates the dispatcher mechanism from class-specific policies, enabling flexible scheduling strategies without modifying core code.

Layered Abstraction: The Virtual File System layer decouples file operations from specific filesystem implementations, while the Hardware Address Translation (HAT) layer abstracts memory management from architecture-specific details.

Copy-on-Write Optimization: Process creation and memory management extensively use COW techniques to minimize copying overhead and conserve memory.

Asynchronous I/O: The STREAMS framework and buffer cache enable efficient asynchronous I/O processing, improving system throughput.

Implementation Insights

Several implementation patterns recur throughout the codebase:

Reference Counting: VNodes, credentials, and address spaces use reference counting for safe resource sharing and deallocation.

Linked List Management: Dispatch queues, hash chains, and process relationships rely on carefully maintained linked lists with sentinel checks.

Bitmap Operations: Priority queues and resource allocation use bitmaps for O(1) operations on sparse data structures.

State Machines: Signal handling, process lifecycle, and scheduling employ explicit state machines with well-defined transitions.

Historical Context and Modern Relevance

The SVR4 architecture influenced many contemporary Unix-like systems:

- **Linux**: Borrowed concepts like VFS, signal handling patterns, and process structure
- **BSD**: Shared common ancestry with similar approaches to VM and networking
- **Solaris**: Direct descendant with many architectural elements preserved

Modern kernels have evolved beyond SVR4 in areas such as:

- **Scalability**: Per-CPU data structures and RCU synchronization for multicore systems
- **Security**: Mandatory access controls, capability systems, and kernel hardening
- **Virtualization**: Hardware-assisted virtualization and container support
- **Real-time**: Preemptible kernels and constant-time scheduling algorithms

Yet the foundational concepts—process abstraction, virtual memory, filesystem layering, and I/O management—remain remarkably consistent.

Further Study

Readers interested in deepening their understanding might explore:

- **Source Code**: The full SVR4 source at `https://github.com/calmsacibis995/svr4-src` provides extensive detail
- **Modern Kernels**: Compare with Linux kernel source and FreeBSD to see evolutionary changes
- **Academic Papers**: Classic works like “The Design of the UNIX Operating System” by Maurice Bach

- **Implementation Projects**: xv6, a teaching kernel that demonstrates similar principles in miniature

The SVR4 kernel represents a significant milestone in operating system design, and its study provides valuable insights into both historical development and contemporary systems programming.

Glossary

Operating System (OS)
: Software that manages computer hardware and software resources and provides common services for computer programs.

Unix
: A family of multitasking, multi-user computer operating systems that derive from the original AT&T Unix.

Kernel
: The core component of an operating system, managing system resources and acting as a bridge between hardware and applications.

Process
: An instance of a computer program that is being executed. It contains the program code and its current activity.

Process ID (PID)
: A unique identifier assigned to each process by the operating system kernel.

Fork
: A system call that creates a new process (child process) by duplicating an existing process (parent process).

Exec
: A family of system calls that loads and executes a new program within the context of an existing process.

Zombie Process
: A process that has completed execution but still has an entry in the process table to allow its parent process to read its exit status.

Orphan Process
: A process whose parent has terminated without waiting for the child to exit. Orphaned processes are adopted by the `init` process.

Context Switch
: The process of storing the state of a process or thread, so that it can be restored and resume execution later.

Scheduler
: The part of the operating system that selects which ready process will be executed by the CPU next.

System Call
: A programmatic way in which a computer program requests a service from the kernel of the operating system.

Virtual Memory

A memory management technique that provides an idealized abstraction of the storage resources that are actually available on a given machine, making it seem like a process has continuous, private memory.

Physical Memory
The actual RAM (Random Access Memory) installed in the computer, directly accessible by the CPU.

Page
A fixed-size block of virtual memory.

Page Frame
A fixed-size block of physical memory, typically the same size as a virtual memory page.

Paging
The process of moving data between physical memory and secondary storage (like a hard drive) in fixed-size blocks (pages).

Swap Space
A dedicated area on a hard disk used as an extension of physical memory when RAM is full.

Address Space
The range of memory addresses that a process can refer to. For a virtual memory system, this is the virtual address space.

Memory Management Unit (MMU)
A hardware component that translates virtual memory addresses used by the CPU into physical memory addresses.

File System
A method and data structure that an operating system uses to control how data is stored and retrieved on storage devices.

Inode
A data structure in a Unix-style file system that describes a file-system object such as a file or a directory. It stores metadata about the file.

File Descriptor
An abstract indicator (handle) used to access a file or other input/output resource, such as a pipe or network socket.

VFS (Virtual File System)
An abstraction layer on top of a more concrete file system. The purpose of VFS is to allow client applications to access different types of concrete file systems in a uniform way.

Mount Point
A directory in a file system where another file system is attached.

Buffer Cache
A region of main memory used by the operating system to store frequently accessed disk blocks, improving I/O performance.

Networking Stack

A set of network protocol layers that work together to provide network communication services.

Socket
An endpoint of a communication flow across a computer network. Sockets provide a standardized way for applications to send and receive data.

TCP (Transmission Control Protocol)
A core protocol of the Internet Protocol Suite, providing reliable, ordered, and error-checked delivery of a stream of octets between applications.

UDP (User Datagram Protocol)
A minimal, connectionless protocol that provides an unreliable, unordered, and non-guaranteed delivery of data between applications.

IP (Internet Protocol)
The primary protocol in the Internet Layer of the Internet Protocol Suite, used for delivering datagrams (packets) across network boundaries.

Signal
A limited form of inter-process communication (IPC) used in Unix-like operating systems to notify a process about an event.

Interrupt
A signal to the processor emitted by hardware or software indicating an event that needs immediate attention.

Device Driver
A computer program that operates or controls a particular type of device that is attached to a computer.

I/O (Input/Output)
The communication between an information processing system and the outside world, e.g., reading from a keyboard or writing to a screen.

DMA (Direct Memory Access)
A feature of computer systems that allows certain hardware subsystems to access main system memory (RAM) independently of the central processing unit (CPU).

Boot Process
The sequence of operations that a computer performs from power-on until it is ready to execute user applications.

Semaphore
A variable or abstract data type used to control access to a common resource by multiple processes and avoid critical section problems.

Message Queue
A form of asynchronous inter-process communication where processes exchange messages by passing them through a queue.

Shared Memory
A method of inter-process communication that allows multiple processes to access the same region of memory, enabling efficient data exchange.

Acknowledgements

This document utilizes the **Alegreya font**, Copyright 2011 Huerta Tipográfica (juan@huertatipografica.com.ar), licensed under the SIL Open Font License, Version 1.1.

The full license text for Alegreya is available at: `http://scripts.sil.org/OFL`

It also uses the **Space Mono font**, Copyright 2016 Google LLC (https://github.com/googlefonts/spacemono), licensed under the SIL Open Font License, Version 1.1.

The full license text for Space Mono is available at: `https://scripts.sil.org/OFL`

This work is a hybrid collaboration between Lee Hanken and Large Language Models. While generative systems were used to synthesize technical data and initial drafts, the narrative structure, technical verification, and final editorial curation were performed by the human author. The author assumes full responsibility for the accuracy and integrity of this work.