Memory management and I/O Subsystem

Swapping

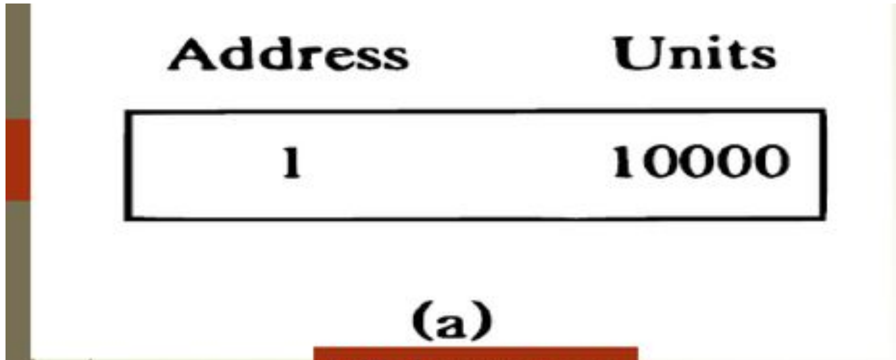
- Historically, UNIX System transferred entire processes between primary memory and the swap device, but not transfer parts of a process independently, except for shared text. Such a memory management policy is called as swapping.
- There are three parts to the description of the swapping algorithm:
  - Managing space on the swap device (allocation of the space on swap device)
  - Swapping processes out of main memory, and
  - Swapping processes into main memory

Allocation of the space on swap device

- The swap device is a block device in a configurable section of a disk
- The kernel allocates space for files one block at a time, it allocates space on the swap device in group of continuous blocks.
- As the allocation of swap device differs from the allocation of scheme for file system, the data structure that catalog free space differ too.
- The kernel maintains free space for the swap device in an in-core table, called a map.
- Maps, used for other resources besides the swap device (some device drivers), allocates a first fir allocation of continuous blocks of a resources.
- A map is an array where each entry consist of an address of an allocable resource and the number of resource available there; the kernel interprets the address and the unit according to the type of map.
- Initially, a map contains one entry that indicates the address and total number of resources.

Address	Units
1	10000

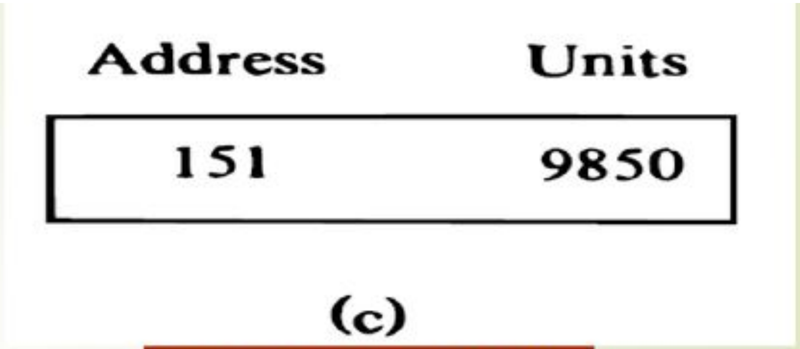
Figure 9.1. Initial Swap Map



Address	Units
1	10000

(a)

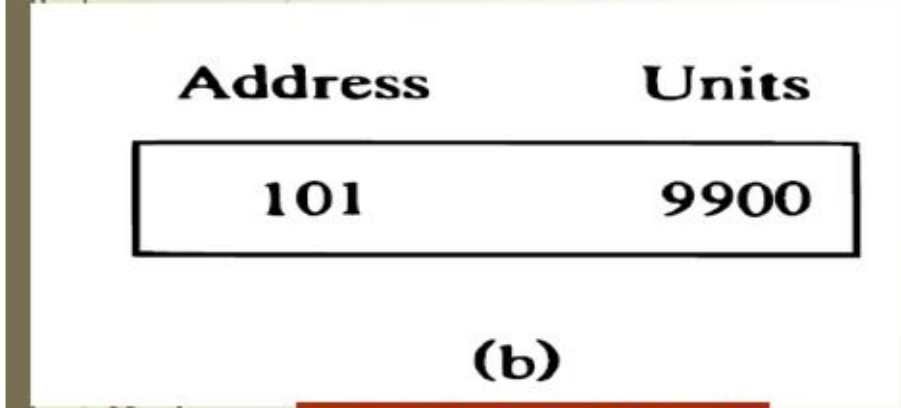
Initial swap map



Address	Units
151	9850

(c)

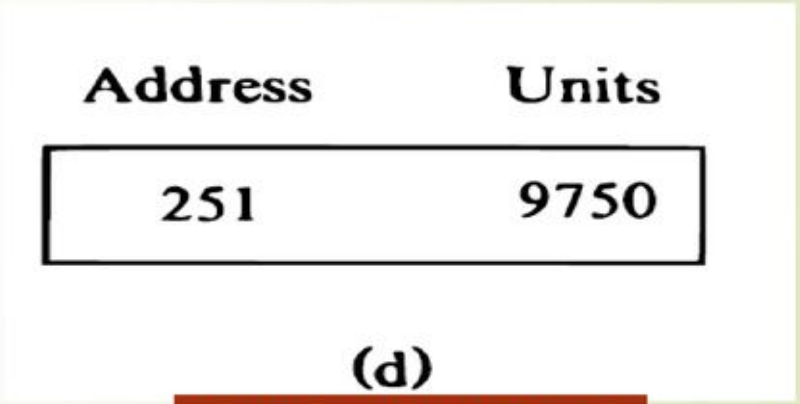
After allocating 50 units



Address	Units
101	9900

(b)

After allocating 100 units

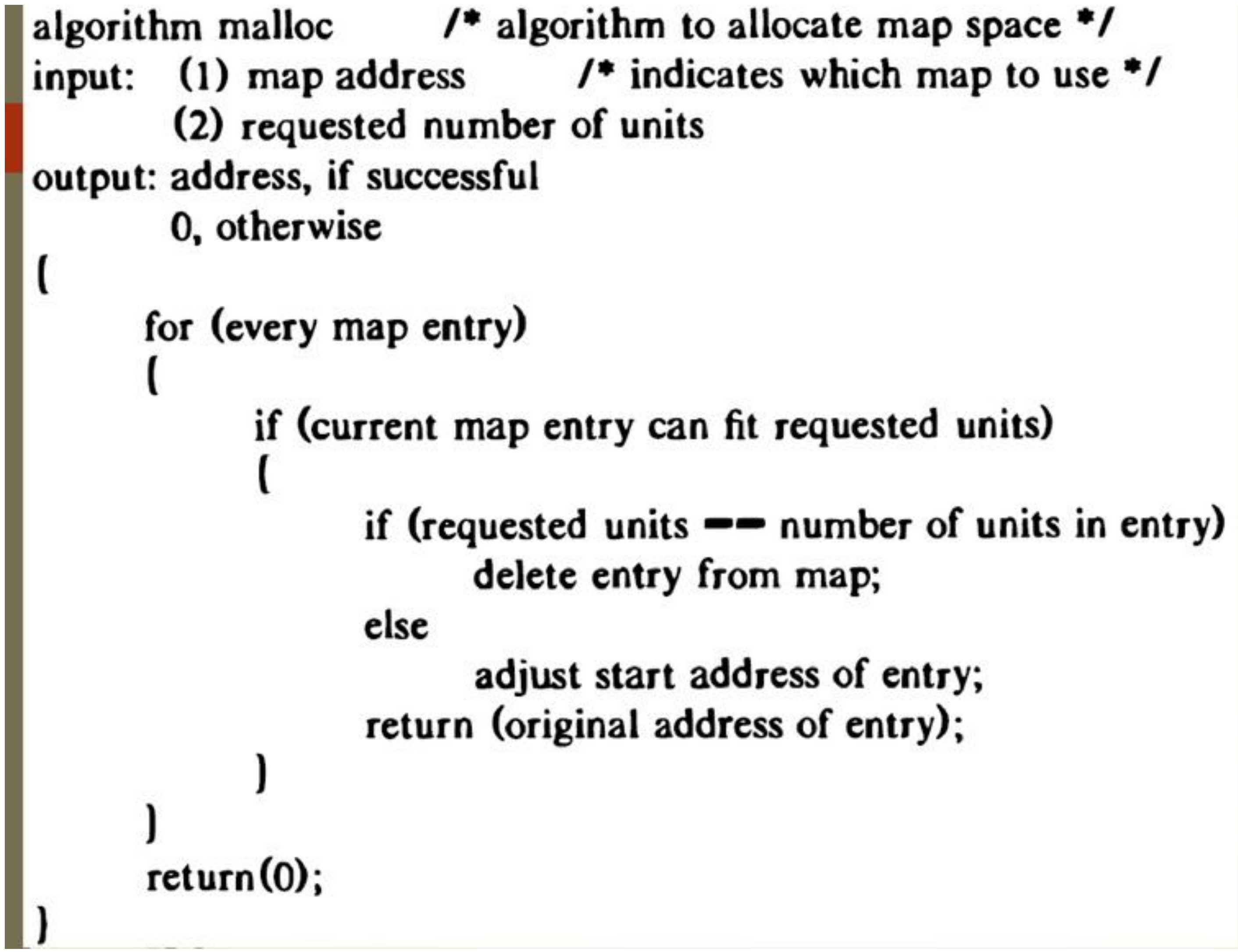


Address	Units
251	9750

(d)

After allocating 100 units

As the kernel allocates and frees resources, it updates the map so that it continues to contain accurate information about resources.



```
algorithm malloc        /* algorithm to allocate map space */
input:  (1) map address        /* indicates which map to use */
        (2) requested number of units
output: address, if successful
        0, otherwise
{
    for (every map entry)
    {
        if (current map entry can fit requested units)
        {
            if (requested units == number of units in entry)
                delete entry from map;
            else
                adjust start address of entry;
            return (original address of entry);
        }
    }
    return(0);
}
```

Algorithm to allocate map space

Freeing the resource

- When freeing the resource, the kernel finds their proper position in the map by address.
- Three cases are possible:
  - The freed resource completely fill a hole in the map. In this case, the kernel combines the newly freed resources and the existing two entries into one entry in the map.
  - The freed resources partially fill a hole in the map. If the address of the freed resources are continuous with the map entry that would immediately precede them or with the entry which immediately follow them, the kernel adjust the address and units fields of the appropriate entry to account for the resource just feed.
  - The freed resource partially fills the hole but are not contiguous to any resources in the map.

Address	Units
251	9750

(a)

After allocating 100 units

Address	Units
101	50
251	9750

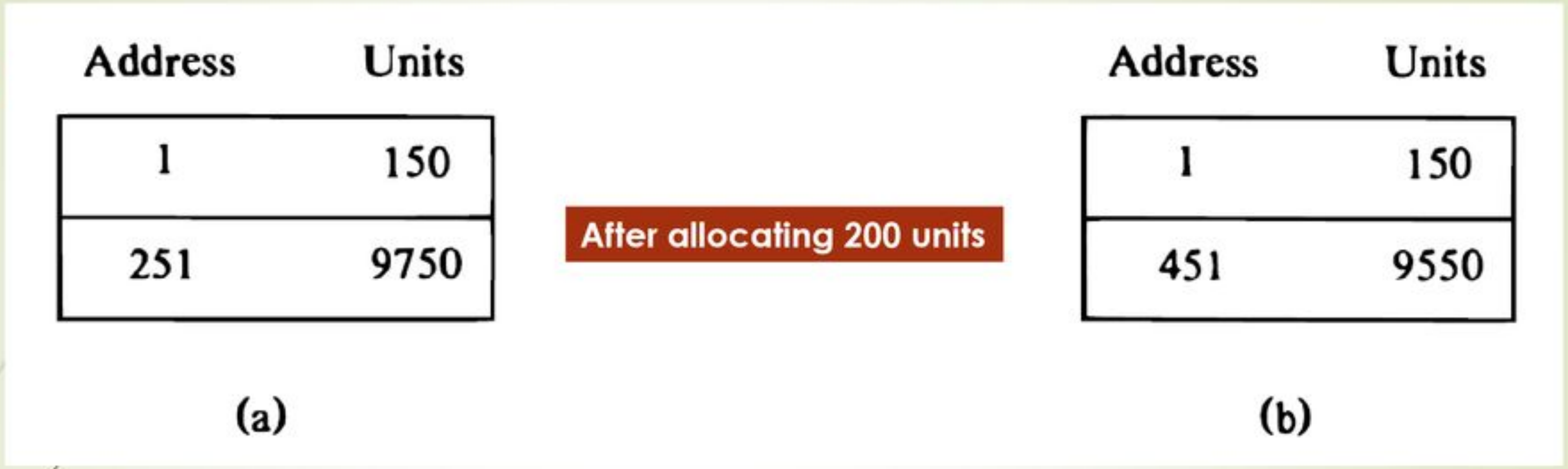
(b)

After freeing 50 units starting at 101

Address	Units
1	150
251	9750

(c)

After freeing 100 units starting at 1



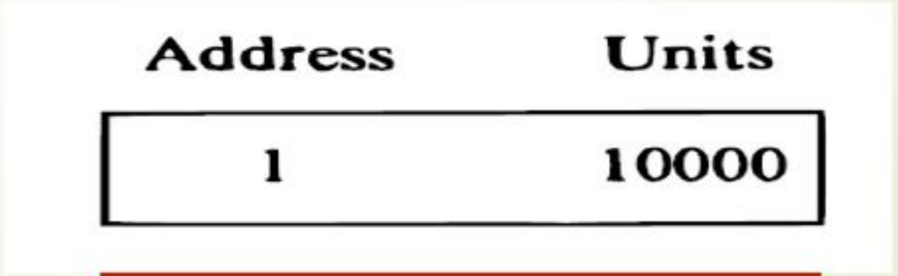
Address	Units
1	150
251	9750

(a)

After allocating 200 units

Address	Units
1	150
451	9550

(b)



Address	Units
1	10000

After freeing 350 units starting at address 151

Swapping process out

- The kernel swaps a process out if it needs space in memory, which may results from any of the following:
  - The **fork system call** must allocate space for a child process,
  - The **brk system call** increases the size of a process,
  - A **process becomes larger by the natural growth** of its stack,
  - The kernel wants to free space in memory for processes it had previously **swapped out and should now swap in.**
- When the kernel decides that a process is eligible for swapping from main memory, it **decrements the reference** count of **each region** in the **process** and **swaps** the **region out** if its reference **count** drops to **0**.
- The **allocates space** on a **swap device** and **locks** the **process** in the memory, **preventing swapper** from **swapping** it out while the current swap operation is in progress.
- The kernel **saves** the **swap address** of the **region** in **the region table entry**.

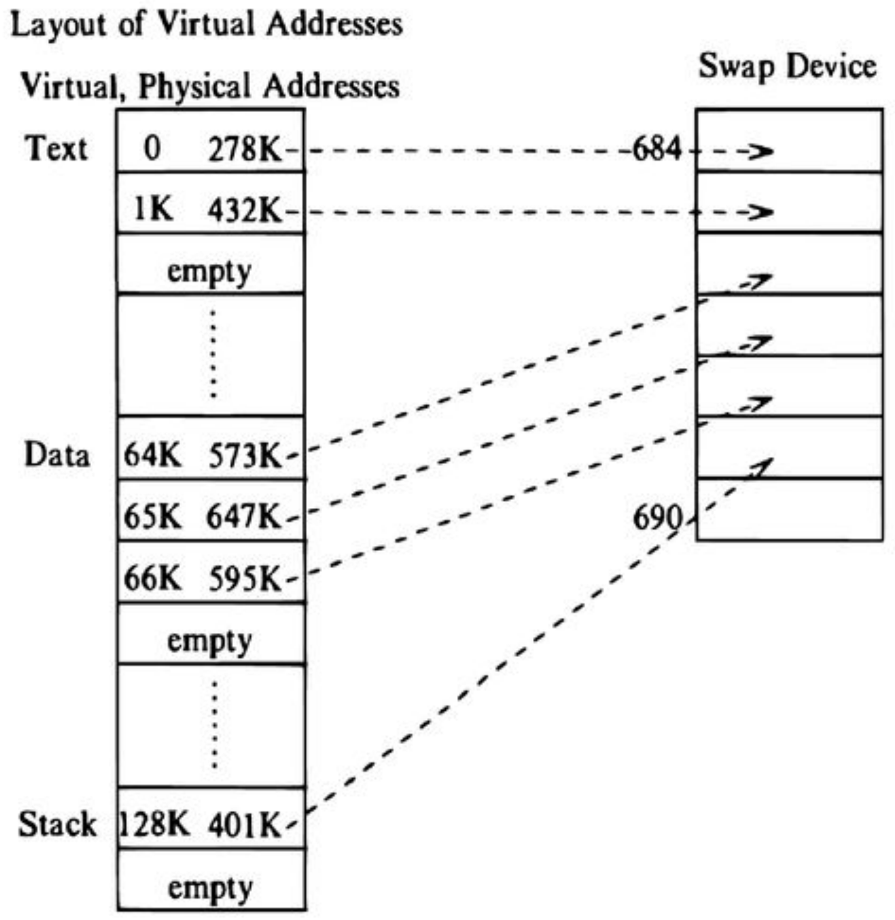


Figure 9.6. Mapping Process Space onto the Swap Device

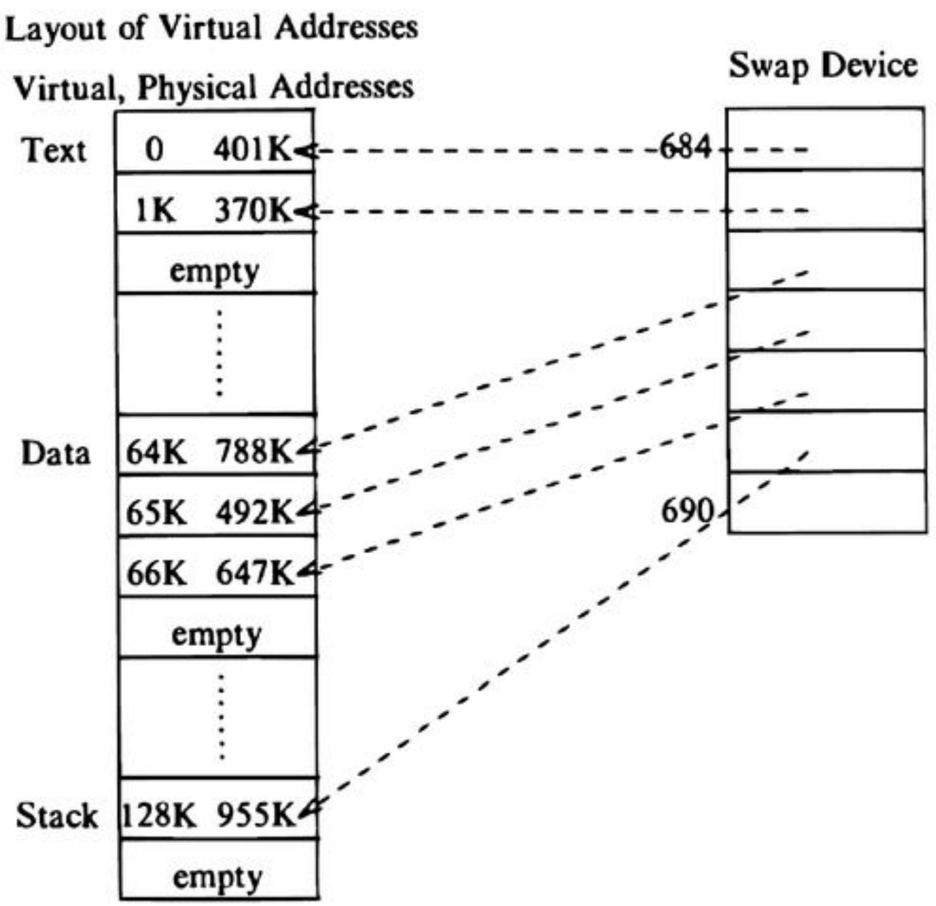


Figure 9.7. Swapping a Process into Memory

Swapping process in

- Process 0, the swapper process is the only process which swaps in the process from swapped device to memory and vice verse.
- At the conclusion of system initialization, the swapper goes to infinite loop, where its only task is to work as a swapper process.
- The swapper sleeps if there is no work to do, the kernel periodically wakes the process 0.
- The swapper makes no system calls, but uses internal functions for its execution or to do swapping.
- The clock handler measures the time that each process has been in core or swapped out.
- When the swapper wakes up to swap processes in, it examines all processes that are in the state "ready to run but swapped out" and select s the one that has been swapped out the longest.
- And repeats the procedure if enough memory available.

Swapping process in

- One of the following situations eventually arises:
  - No "ready run " processes exits on the swap device: the swapper goes to sleep
  - The swapper find an eligible process to swap in but the system does not contain the enough memory : the swapper attempts to swap another process out and, if successful, restart the swapping algorithm, searching for the process to swap in
- If the swapper must swap a process out, it examines every process in memory: following processes do not get swapped out
  - Zombie processes
  - Processes locked in main memory, doing region operations
- The kernel swaps out sleeping processes rather than "ready to run" because "ready to run" processes have a great chance of being scheduled soon.
- If there is no sleeping process in the memory, the ready to run process is swapped out checking nice value and time in memory.

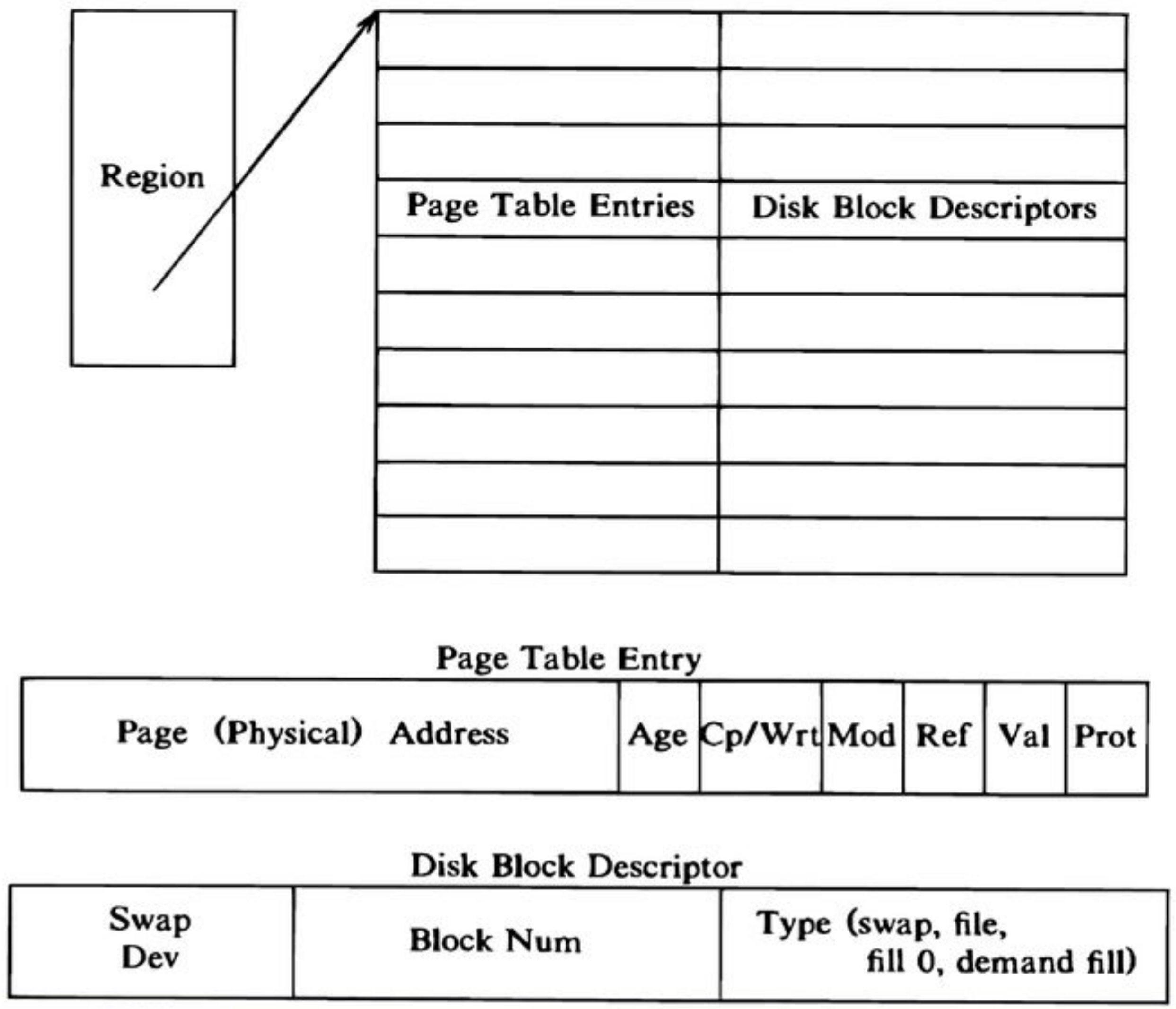
```
/* loop2: here in revised algorithm (see page 285) */
    for (all processes loaded in main memory, not zombie and not locked in memory)
    {
            if (there is a sleeping process)
                    choose process such that priority + residence time
                                    is numerically highest;
            else  /* no sleeping processes */
                    choose process such that residence time + nice
                                    is numerically highest;
    }
    if (chosen process not sleeping or residency requirements not
                                    satisfied)
            sleep (event must swap process in);
    else
            swap out process;
    goto loop;        /* goto loop2 in revised algorithm */
}
```

Demand Paging

- Machines whose memory architecture is based on pages and whose CPU has restartable instruction can support a kernel that implements a demand paging algorithm, swapping pages of memory between main memory and a swap device.
- Demand paging system free processes from size limitations, but the kernel still imposes a limit on he virtual size of a process, dependent on the amount of virtual memory the Machin can address.
- Processes tends to execute instructions in small portions of their text space, such as program loops and frequently called subroutines, and their data references tends to cluster in small subsets of the total data space of the process, this is known as the principle of "locality".
- **Working set** of a process, which is the set of pages that the process has referenced in its last n memory references; the number n is called the **window** of the working set
- When a process addresses a page that is not in its working set, it incurs a **validity page fault**; in handling the fault, the kernel **updates** the **working set**, **reading** in **pages** from a **secondary device if necessary**.

Data structure for demand paging

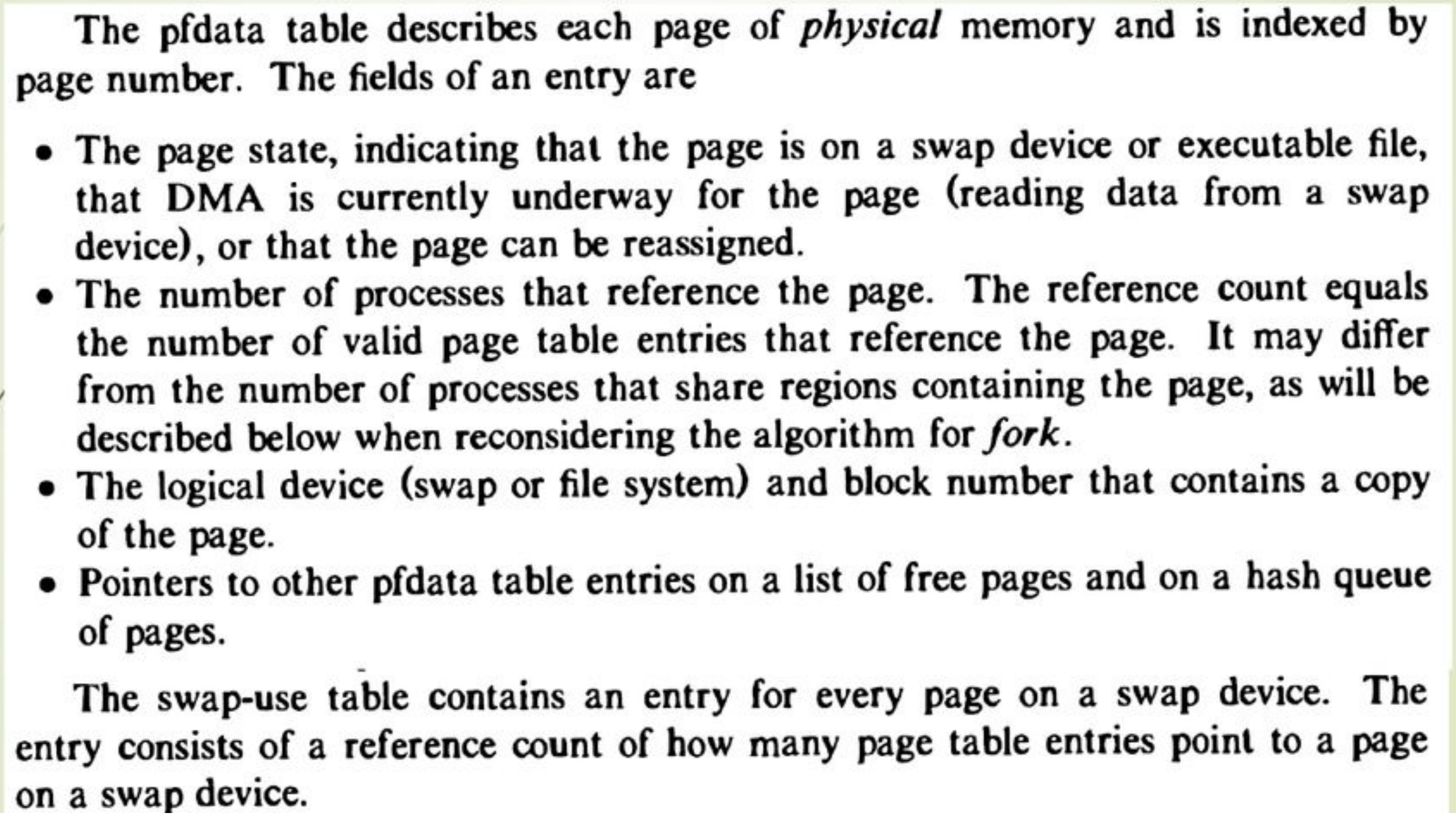
- The kernel contains 4 major data structures to support low level memory management functions and demand paging:
  - **Page table entries**
  - **Disk block descriptors**
  - **Page frame data table (pfdata table)**
  - **Swap use table**
- The kernel allocates space for the pfdata table once for the lifetime of the system but allocates memory pages for the other structures dynamically.



Page table entries and disk block descriptor

- Valid
- Reference
- Modify
- Copy on write
- Age

Pfdata table and swap use table



The pfdata table describes each page of *physical* memory and is indexed by page number. The fields of an entry are

- The page state, indicating that the page is on a swap device or executable file, that DMA is currently underway for the page (reading data from a swap device), or that the page can be reassigned.
- The number of processes that reference the page. The reference count equals the number of valid page table entries that reference the page. It may differ from the number of processes that share regions containing the page, as will be described below when reconsidering the algorithm for *fork*.
- The logical device (swap or file system) and block number that contains a copy of the page.
- Pointers to other pfdata table entries on a list of free pages and on a hash queue of pages.

The swap-use table contains an entry for every page on a swap device. The entry consists of a reference count of how many page table entries point to a page on a swap device.

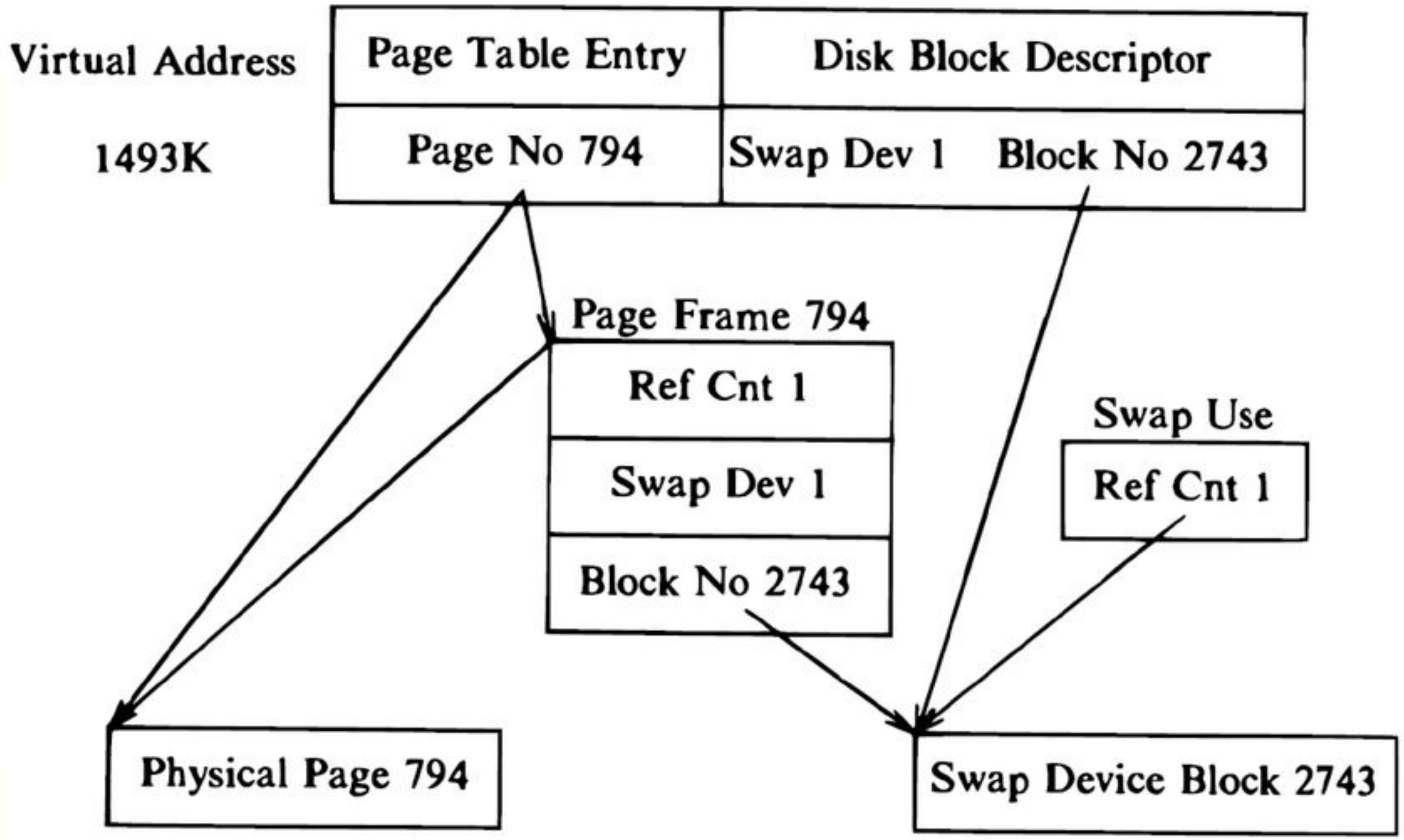


Figure 9.14. Relationship of Data Structures for Demand Paging

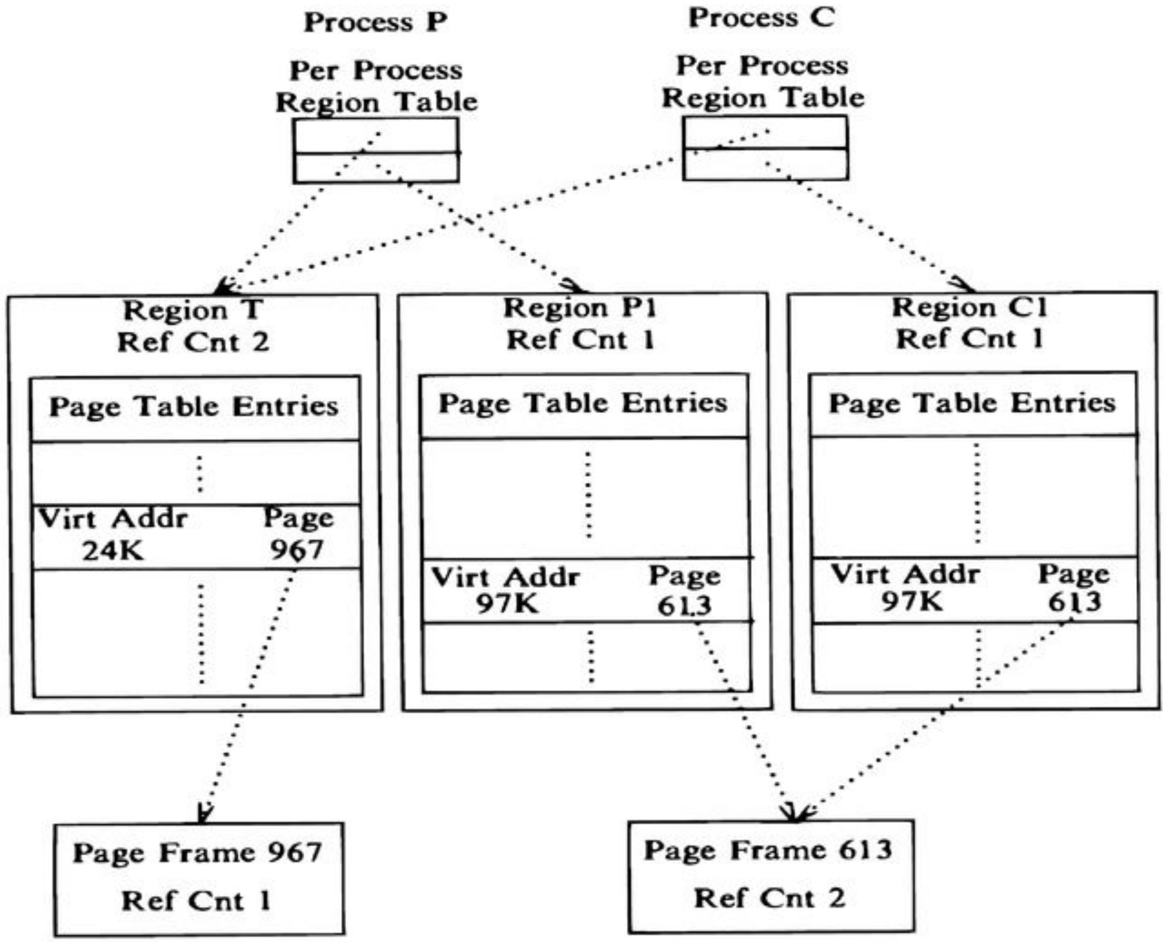
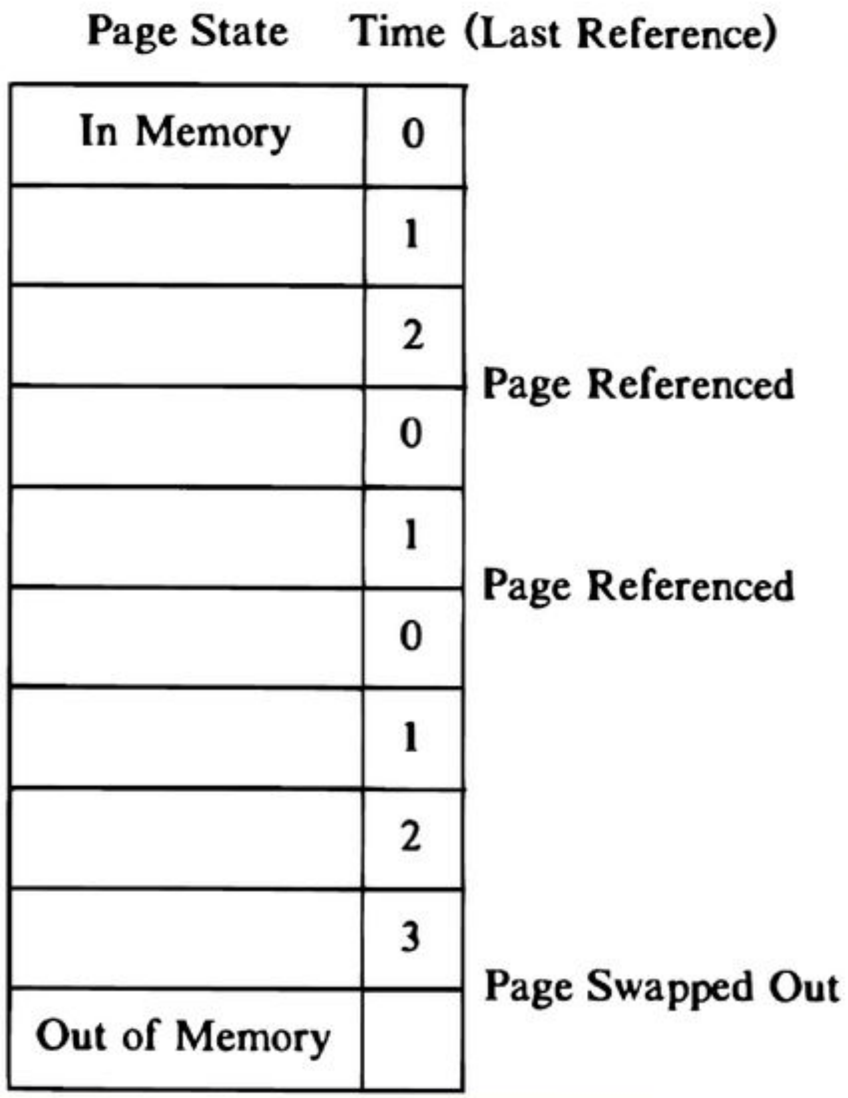
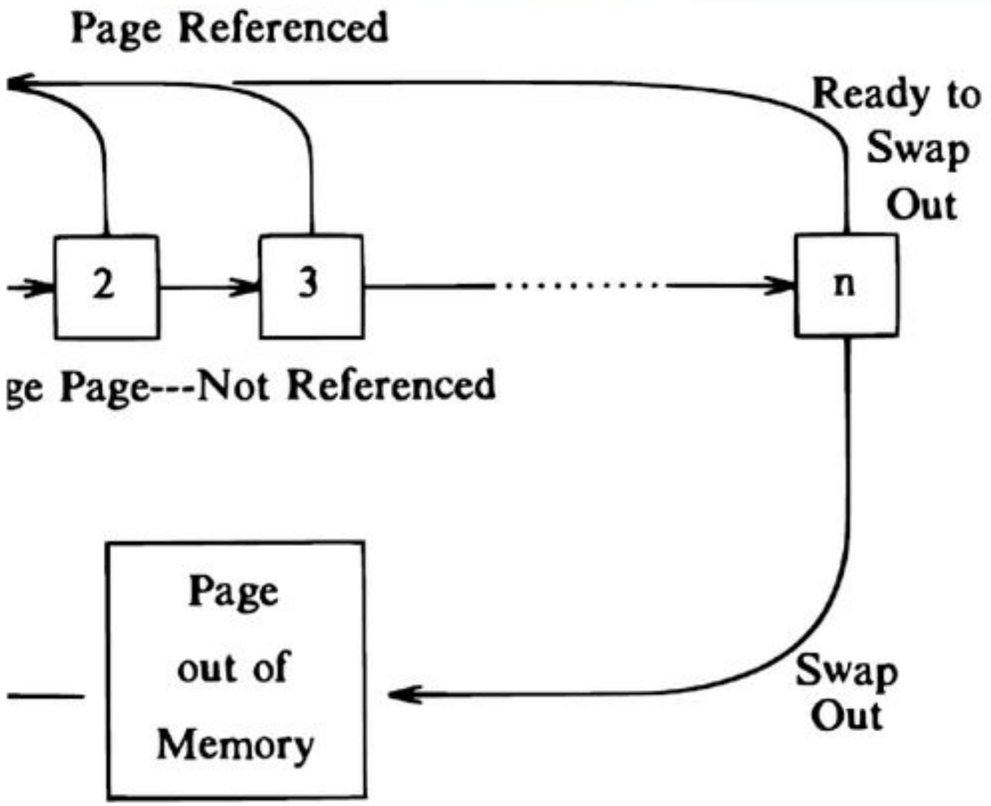


Figure 9.15. A Page in a Process that Forks

The page stealer process

- The page stealer is a kernel process that swaps out memory pages that are no longer part of the working set of a process.
- The kernel creates the page stealer during system initialization and invokes it through the lifetime of the system when low on free pages.
- It examines every active, unlocked region, skipping locked regions in the expectation of examining them during its next pass through the region list, and increments the age field of all valid pages.
- The kernel locks a region when a process faults on a page in a region, so that the page cannot steal the page being faulted in.
- There are two paging states for a page in memory:
- The page is aging and not yet eligible for swapping
- And the page is eligible for swapping and available for reassignment

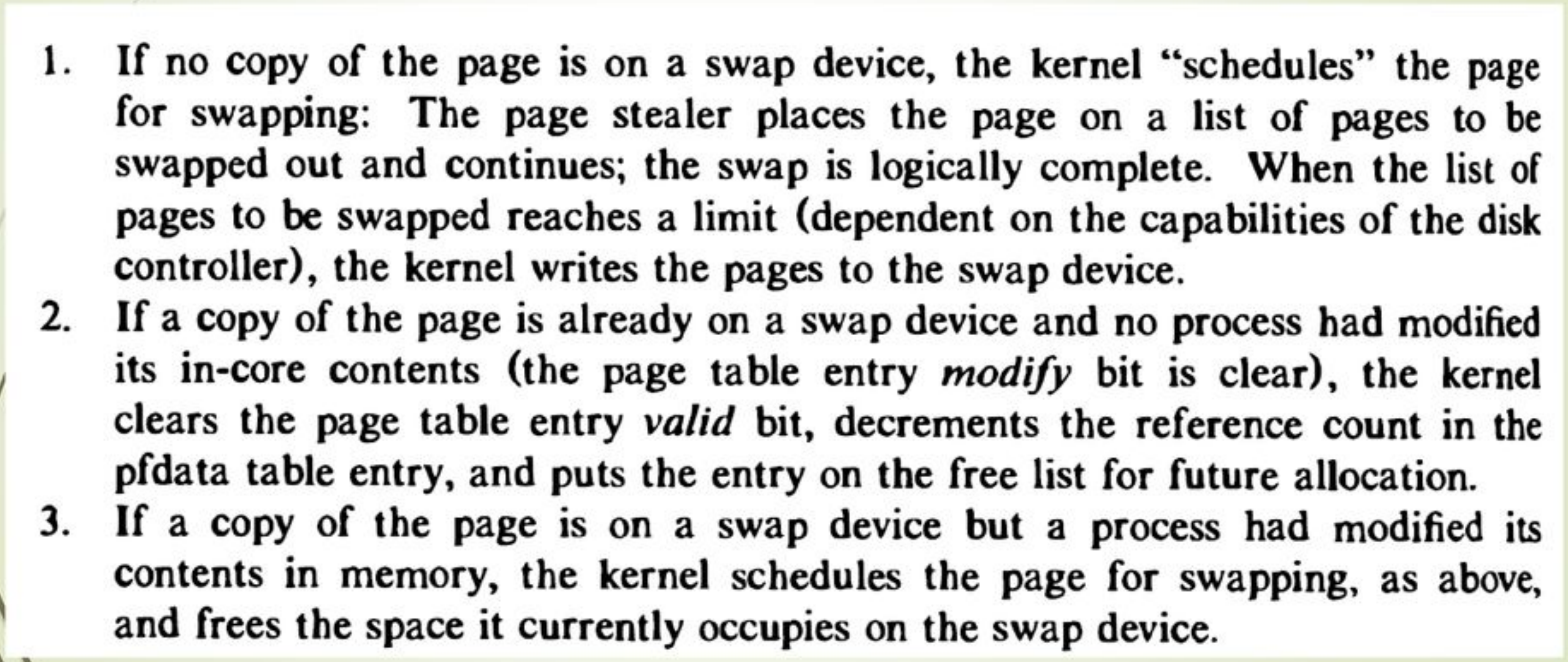




. State Diagram for Page Aging

The page stealer process

- When the page stealer decides to swap out a page, it considers weather a copy of the page is on a swap device. There are three possibilities



1. If no copy of the page is on a swap device, the kernel "schedules" the page for swapping: The page stealer places the page on a list of pages to be swapped out and continues; the swap is logically complete. When the list of pages to be swapped reaches a limit (dependent on the capabilities of the disk controller), the kernel writes the pages to the swap device.
2. If a copy of the page is already on a swap device and no process had modified its in-core contents (the page table entry *modify* bit is clear), the kernel clears the page table entry *valid* bit, decrements the reference count in the pfdata table entry, and puts the entry on the free list for future allocation.
3. If a copy of the page is on a swap device but a process had modified its contents in memory, the kernel schedules the page for swapping, as above, and frees the space it currently occupies on the swap device.

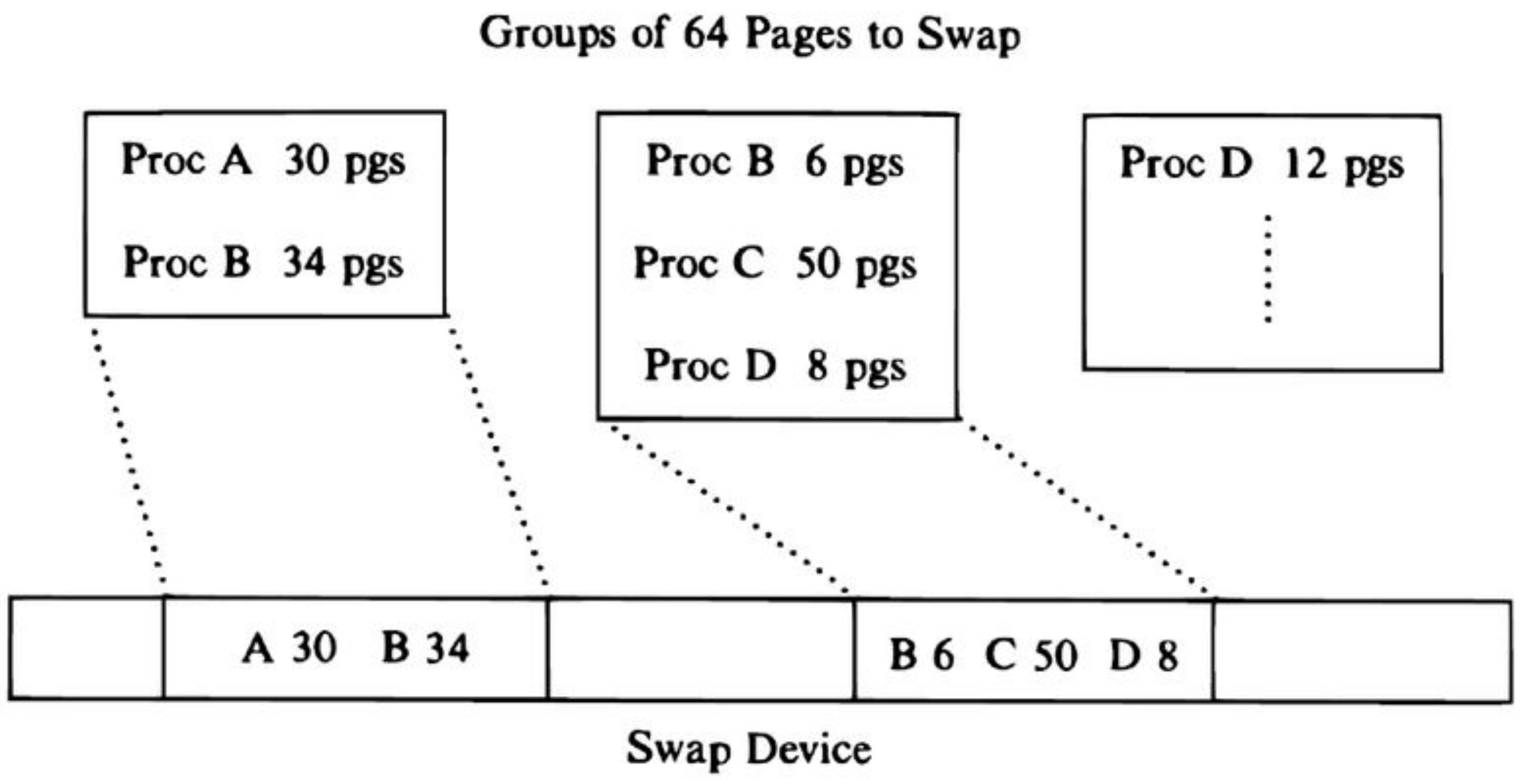


Figure 9.20. Allocation of Swap Space in Paging Scheme

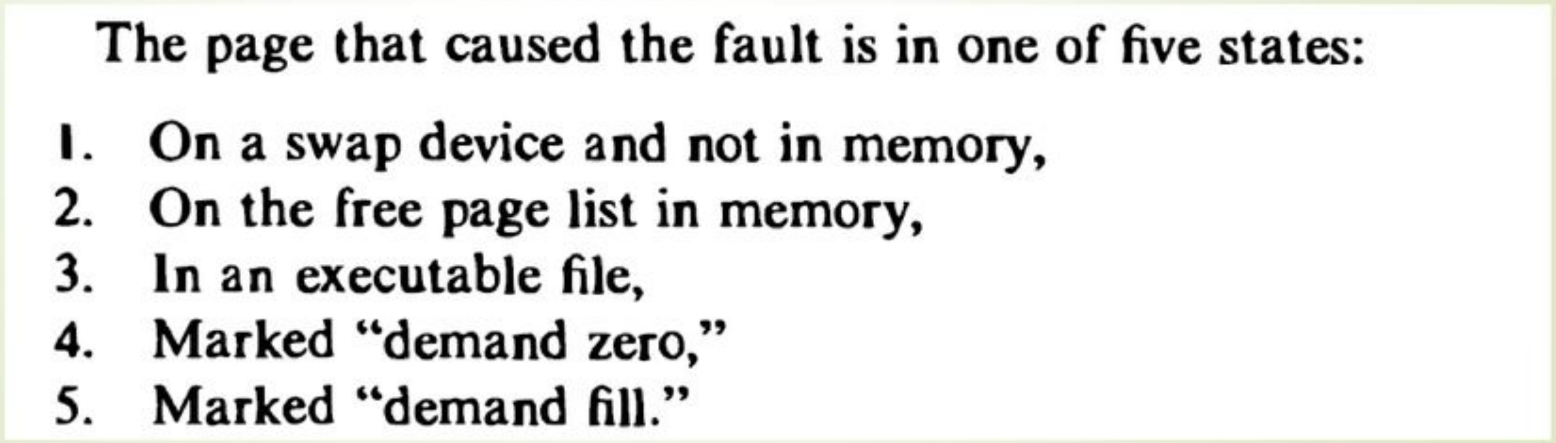
Suppose the page stealer swaps out 30,40,50, and 20 pages from process A, B,C, AND D, and that it writes 64 pages to the swap device in one disk write operation.

Page faults

- The system can incur two types of page faults:
  - Validity fault
  - Protection fault

Validity fault

- If a process attempts to access a page whose valid bit is not set, it incures a validity fault and the kernel invokes the validity fault handler.
- The hardware supplies the kernel with the virtual address that was accessed to cause the memory fault, and the kernel finds the page table entry and disk block descriptor for the page.



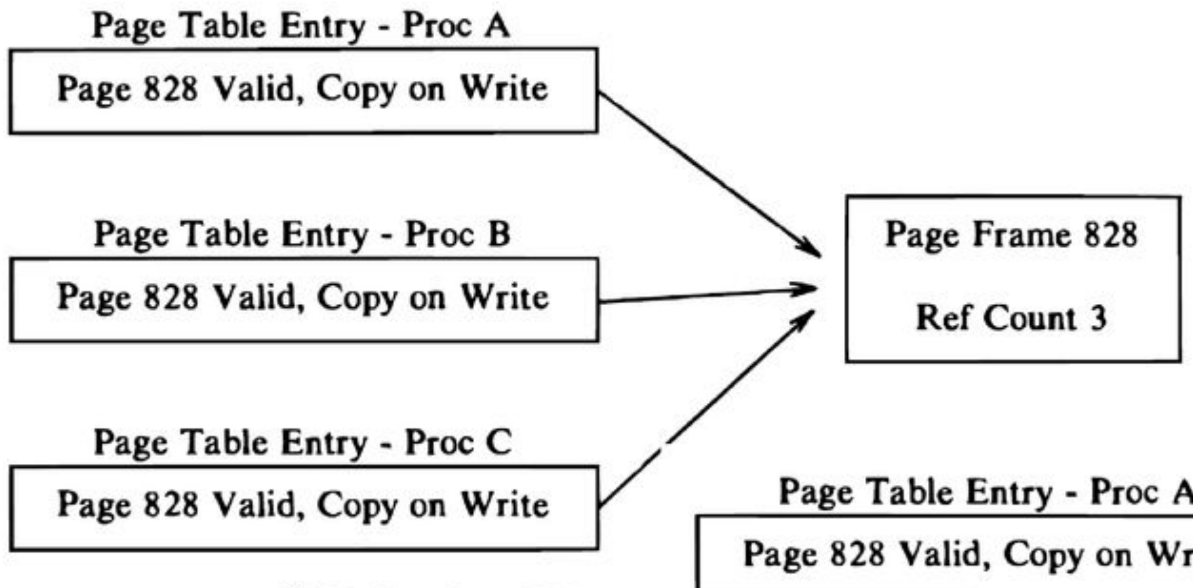
The page that caused the fault is in one of five states:

1. **On a swap device and not in memory,**
2. **On the free page list in memory,**
3. **In an executable file,**
4. **Marked "demand zero,"**
5. **Marked "demand fill."**

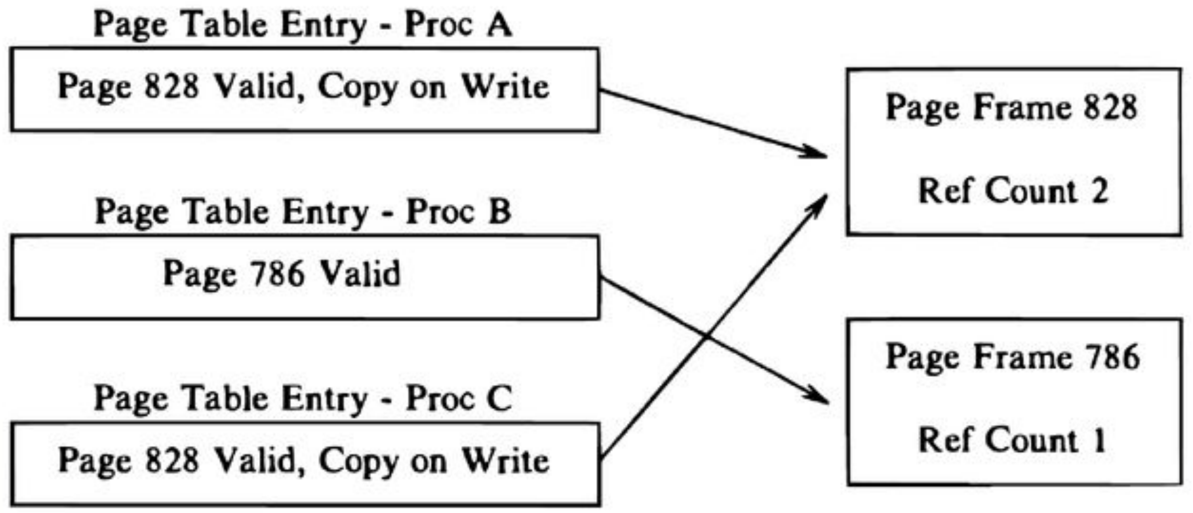
```
    else        /* page not in cache */
    {
            assign new page to region;

            put new page in cache, update pfdata entry;
            if (page not previously loaded and page "demand zero")
                    clear assigned page to 0;
            else
            {
                    read virtual page from swap dev or exec file;
                    sleep (event I/O done);
            }
            awaken processes (event page contents valid);
    }
    set page valid bit;
    clear page modify bit, page age;
    recalculate process priority;                                   t */
out:  unlock region;
}
```

Protection fault handler

- The second kind of memory fault that a process can incur is a protection fault, meaning that the process accessed a valid page but the permission bits associated with the page did not permit access.
- A process also incurs a protection fault when it attempts to write a page whose copy on write bit was set during fork system call.
- The kernel must determine whether permission was denied because the page requires a copy on write or whether something truly illegal happened
- The hardware supplies the protection fault handler with the virtual address where the fault occurred, and the fault handler finds the appropriate region and page table entry.
- It locks the region so that the page stealer cannot steal the page while the protection fault handler operates on it.
- If the fault handler determines that the fault was caused because the copy on write bit was set, and if the page is shared, the kernel allocates a new page and copies the contents of old page to it.
- Update the page table entries to new page number and decrease the reference count of old one
- If the copy on write bit is set but no other processes share the page, the kernel allows the process to reuse the physical page
- It turn off the copy on write bit and disassociate the page from its disk copy.


Page Table Entry - Proc A
Page 828 Valid, Copy on Write
Page Table Entry - Proc B
Page 828 Valid, Copy on Write
Page Table Entry - Proc C
Page 828 Valid, Copy on Write
Page Frame 828
Ref Count 3

(a) Before Proc B Incurs


Page Table Entry - Proc A
Page 828 Valid, Copy on Write
Page Table Entry - Proc B
Page 786 Valid
Page Table Entry - Proc C
Page 828 Valid, Copy on Write
Page Frame 828
Ref Count 2
Page Frame 786
Ref Count 1

(b) After Protection Fault Handler Runs for Proc B

```
        else        /* "steal" page, since nobody else is using it */
        {
                if (copy of page exists on swap device)
                        free space on swap device, break page association;
                if (page is on page hash queue)
                        remove from hash queue;
        }
        set modify bit, clear copy on write bit in page table entry;
        recalculate process priority;
        check for signals;
  out:  unlock region;
}
                decrement old page frame reference count;
                update page table entry to point to new physical page;
        }
```

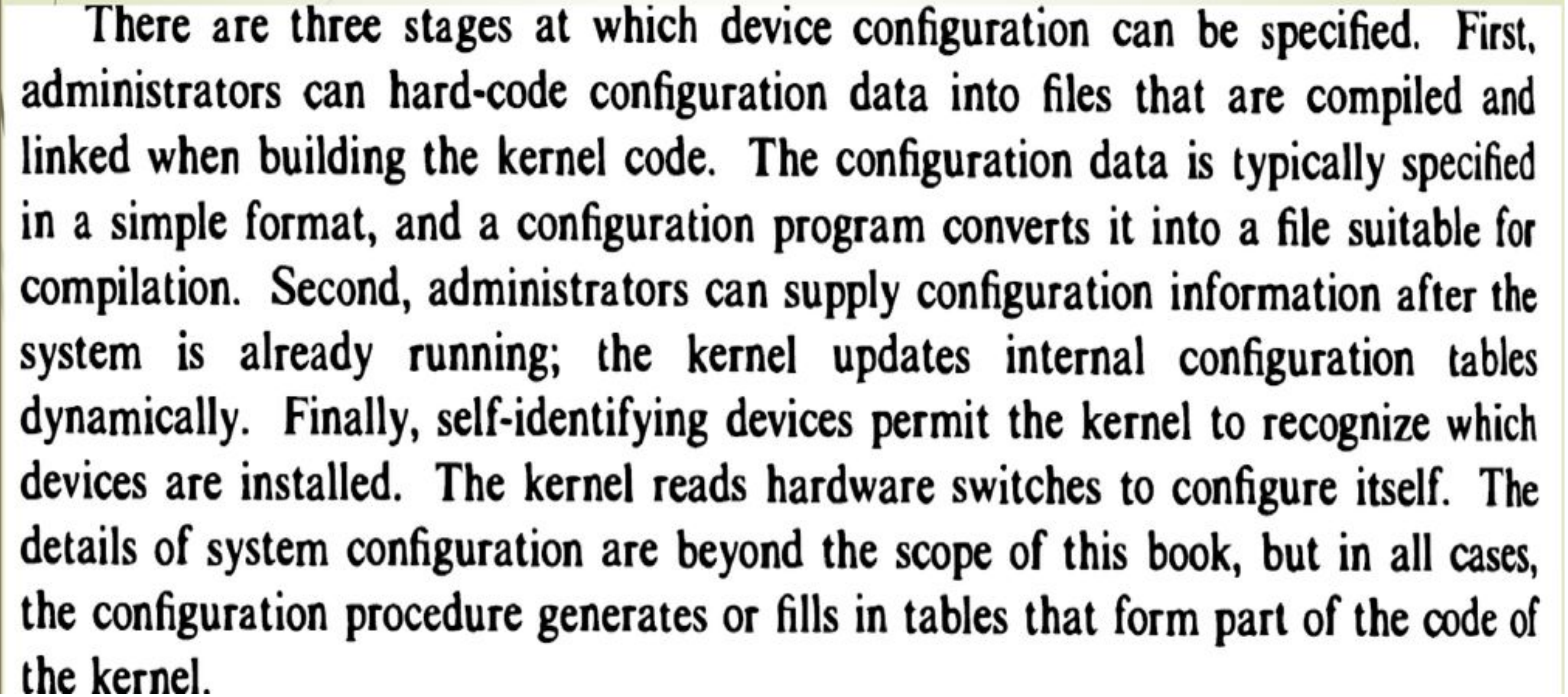
The I/O SubSystem

Driver interfaces, disk drives, terminal drivers, Streams.

Driver Interface

- The UNIX system contains **two types** of devices, **block devices** and raw or **character devices**.
- **Block** devices, such as **disk and tapes**, look like random access storage devices to the rest of the system
- **Character** devices include all other devices such as **terminals and network media.**
- **Block devices may have a character device interface, too.**
- **The user interface** to **devices** goes through the **file system** : every **device has a name** that look like a **file name** and is **accessed** like a **file.**
- The device special file has a **inode** and occupies a **node** in the **directory** hierarchy of the **file system.**
- The **device** file is **separated** from the **other files** by the **file type** stored in its inode, either "block" or "Character special"

System Configuration



There are three stages at which device configuration can be specified. First, administrators can hard-code configuration data into files that are compiled and linked when building the kernel code. The configuration data is typically specified in a simple format, and a configuration program converts it into a file suitable for compilation. Second, administrators can supply configuration information after the system is already running; the kernel updates internal configuration tables dynamically. Finally, self-identifying devices permit the kernel to recognize which devices are installed. The kernel reads hardware switches to configure itself. The details of system configuration are beyond the scope of this book, but in all cases, the configuration procedure generates or fills in tables that form part of the code of the kernel.

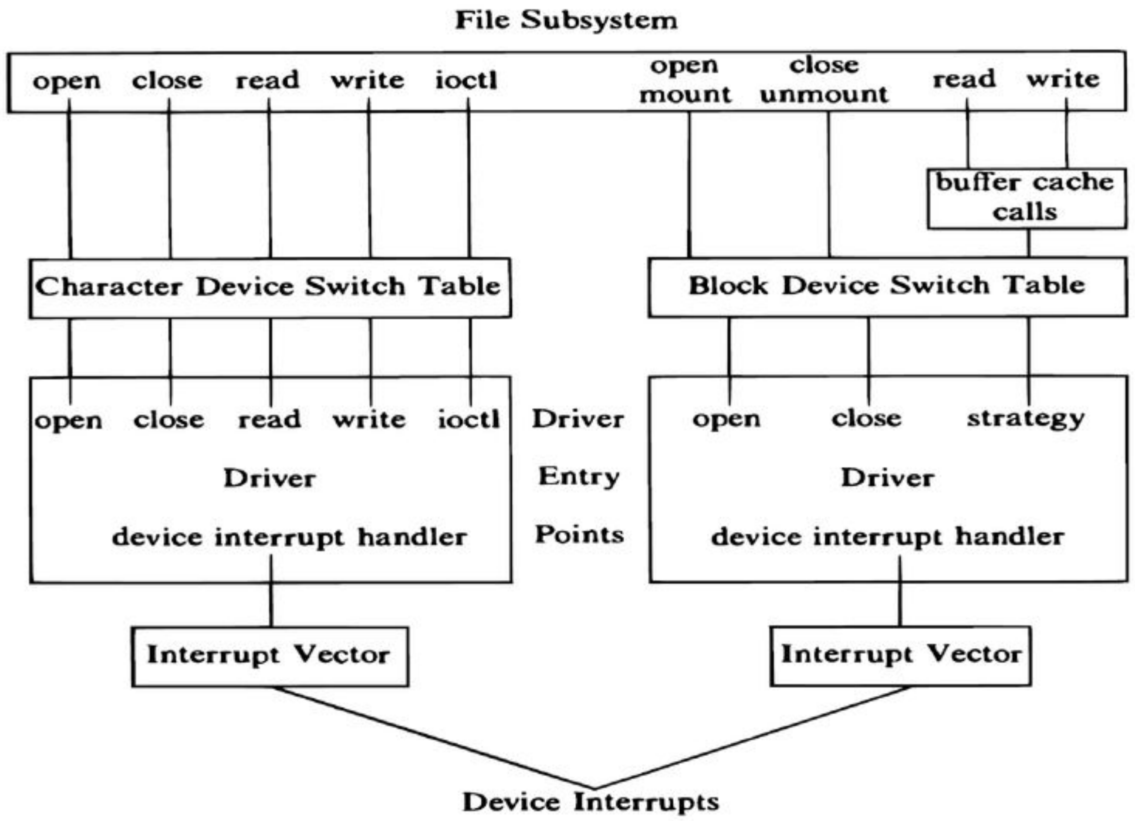
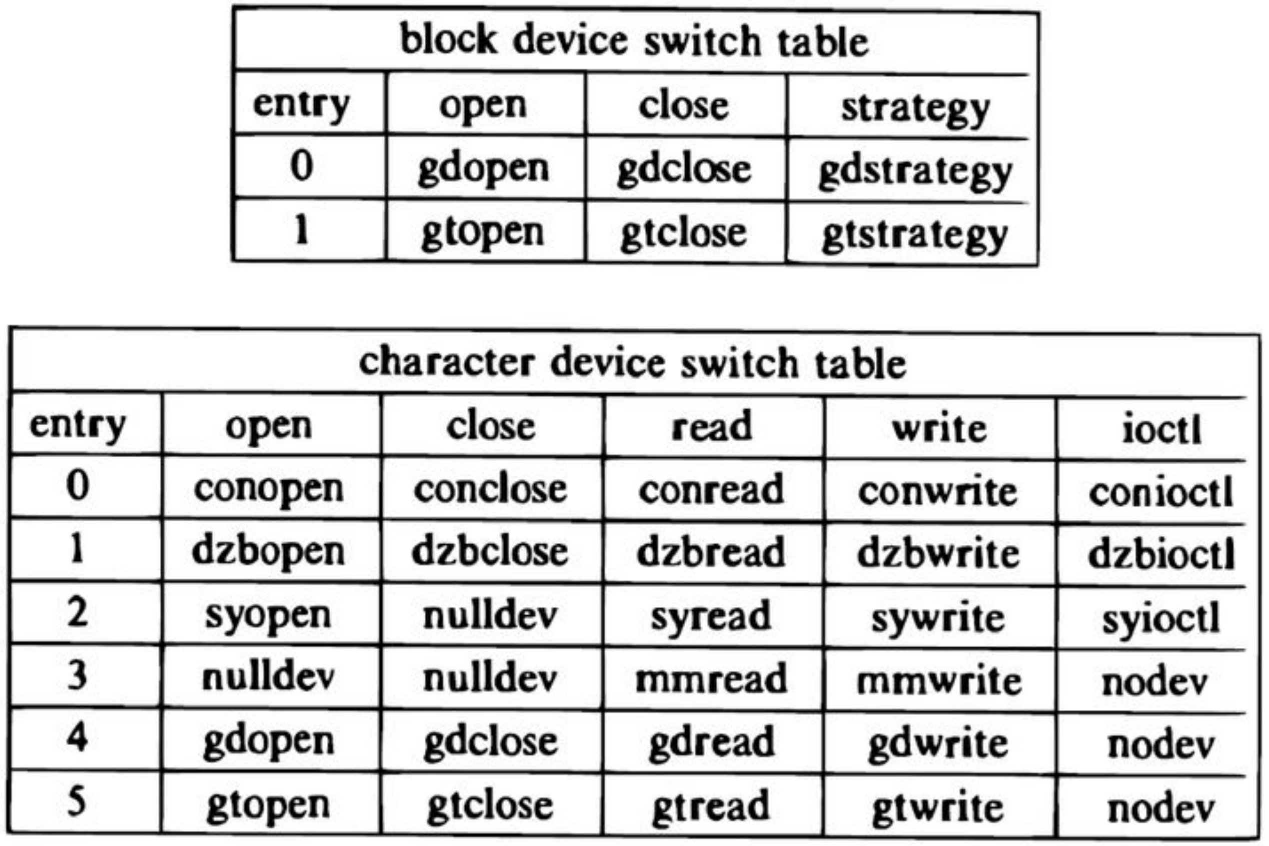


Figure 10.1. Driver Entry Points

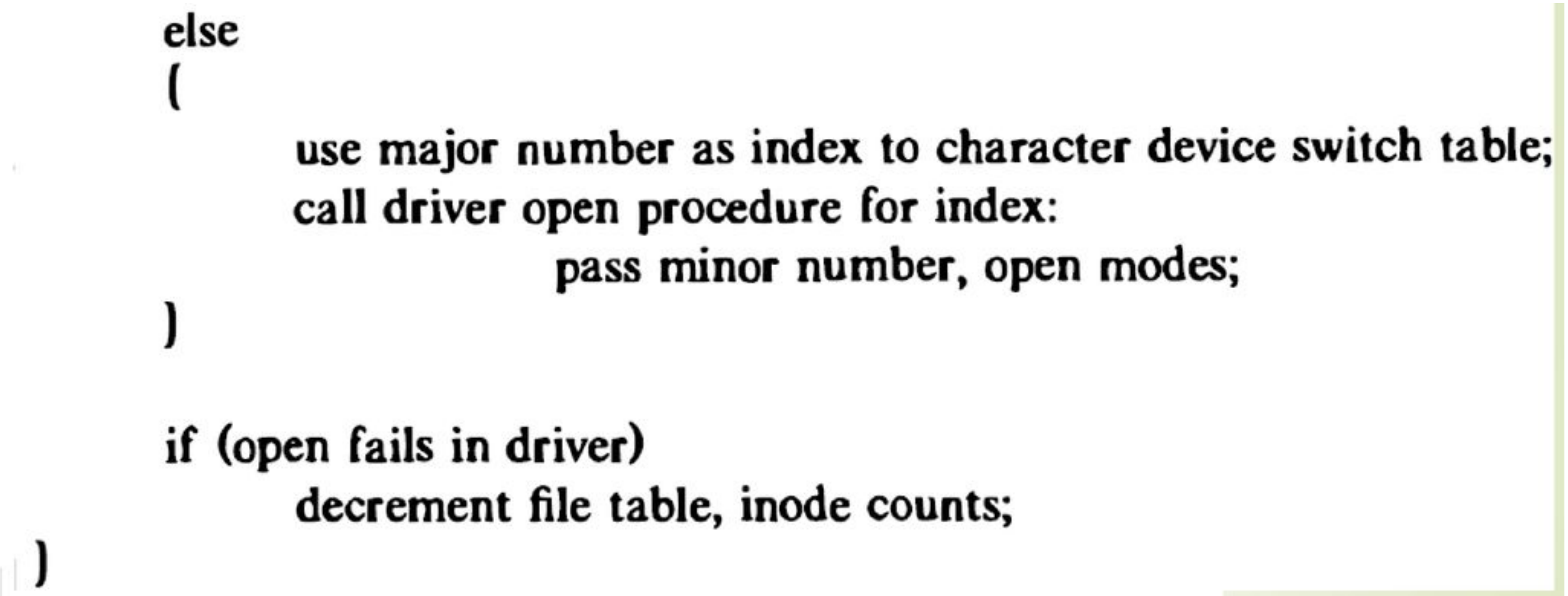


block device switch table			
entry	open	close	strategy
0	gdopen	gdclose	gdstrategy
1	gtopen	gtclose	gtstrategy

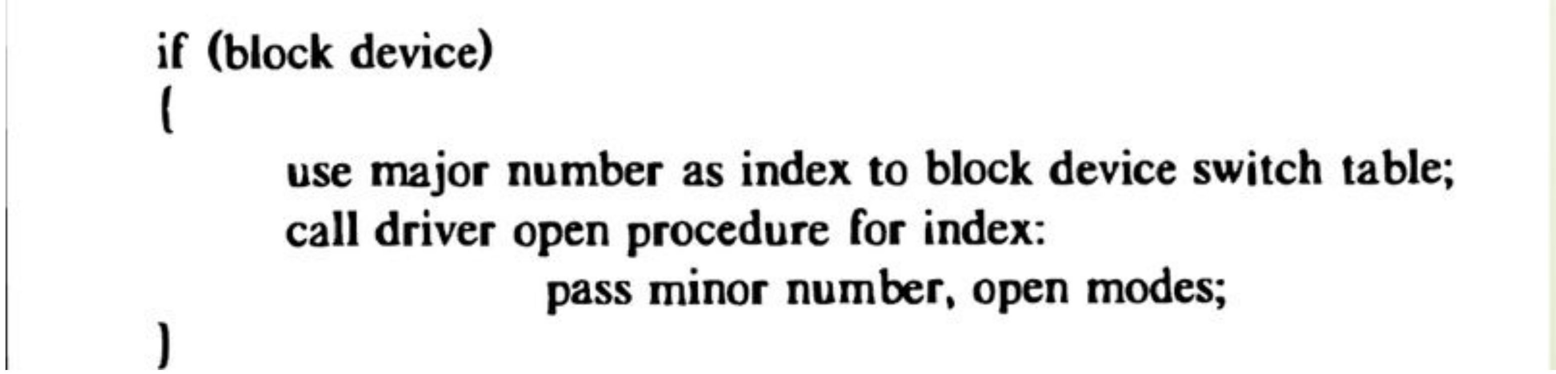
character device switch table					
entry	open	close	read	write	ioctl
0	conopen	conclose	conread	conwrite	conioctl
1	dzbopen	dzbclose	dzbread	dzbwrite	dzbioctl
2	syopen	nulldev	syread	sywrite	syioctl
3	nulldev	nulldev	mmread	mmwrite	nodev
4	gdopen	gdclose	gdread	gdwrite	nodev
5	gtopen	gtclose	gtread	gtwrite	nodev

Figure 10.2. Sample Block and Character Device Switch Tables

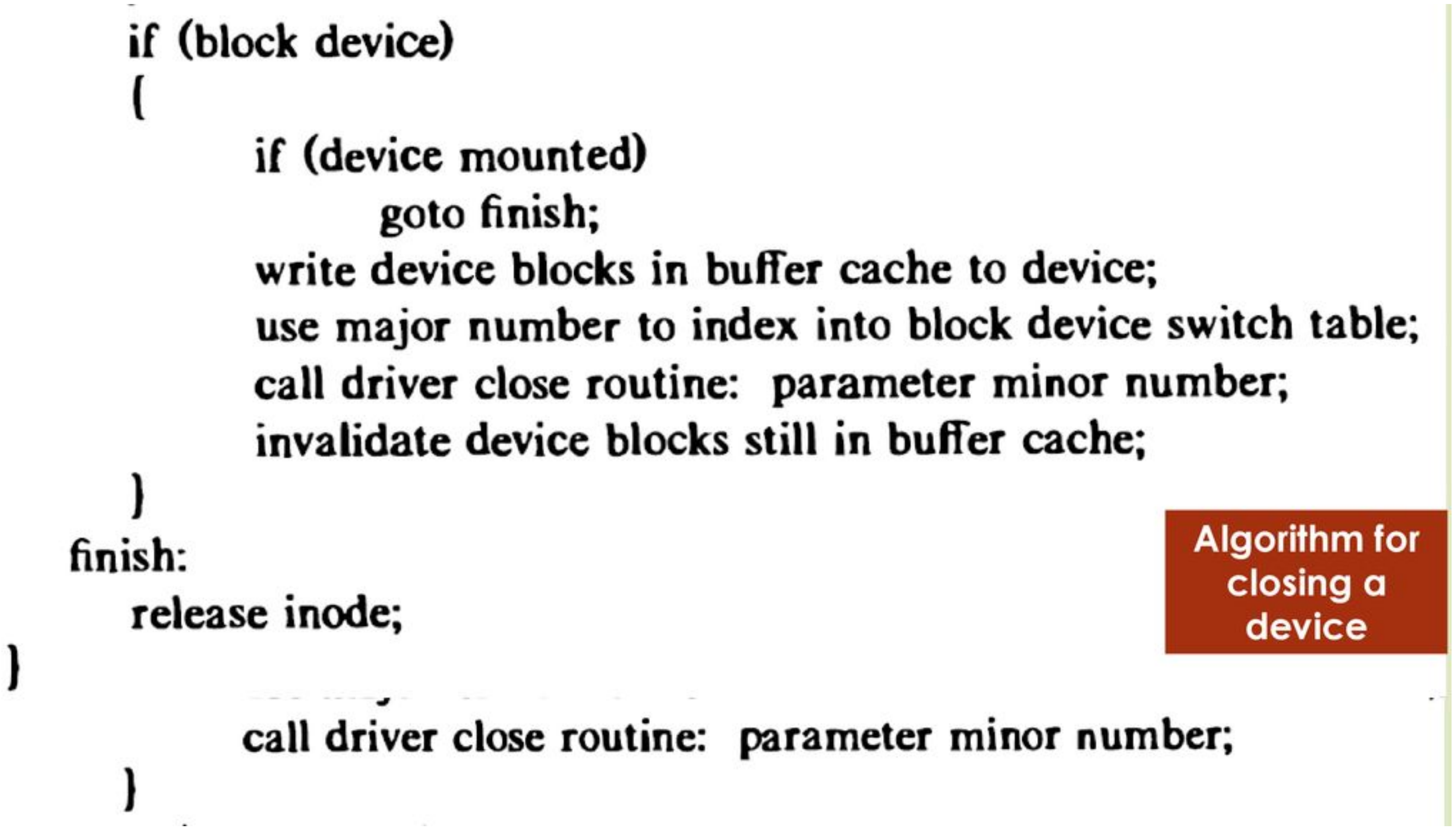
Algorithm for opening a device



```
    else
    {
            use major number as index to character device switch table;
            call driver open procedure for index:
                        pass minor number, open modes;
    }

    if (open fails in driver)
            decrement file table, inode counts;
}
```



```
    if (block device)
    {
            use major number as index to block device switch table;
            call driver open procedure for index:
                        pass minor number, open modes;
    }
```



```
    if (block device)
    {
          if (device mounted)
                goto finish;
          write device blocks in buffer cache to device;
          use major number to index into block device switch table;
          call driver close routine:  parameter minor number;
          invalidate device blocks still in buffer cache;
    }
  finish:
    release inode;
}
          call driver close routine:  parameter minor number;
    }
```

Algorithm for closing a device

Strategy interface

- The **kernel** uses the **strategy interface** to **transmit** data **between** the **buffer** cache and a **device**, although the read and write procedures of character devices sometimes uses their strategy procedure to transfer data directly between the device and the user address space.
- The strategy procedures may queue I/O jobs for a device on a work list or do more sophisticated processing to schedule I/O jobs.
- The kernel passes a buffer header address to the driver strategy procedure; the header contains a list of page addresses and size for transmission of data to or from the device.

Ioctl system call

- The ioctl system call is a generalization of the terminal-specific stty (set terminal settings) and gtty (get terminal settings) system calls available in earlier versions of the UNIX system.
- It provides a general, catch-all entry point for device specific commands, allowing a process to set hardware options associated with a device and software option associated with a device driver.
- The specific actions specified by the ioctl call vary per device and are defined by the device driver.
- The syntax of the system call is **Ioctl(fd, command, arg)**
- Where **fd** is the file descriptor returned by a prior open system call, **command** is a request of the driver to do a particular action, and **arg** is a parameter fro the command
- Commands are driver specific; hence, each driver interprets commands according to internal specification, and the format of the data structure arg depends on the command

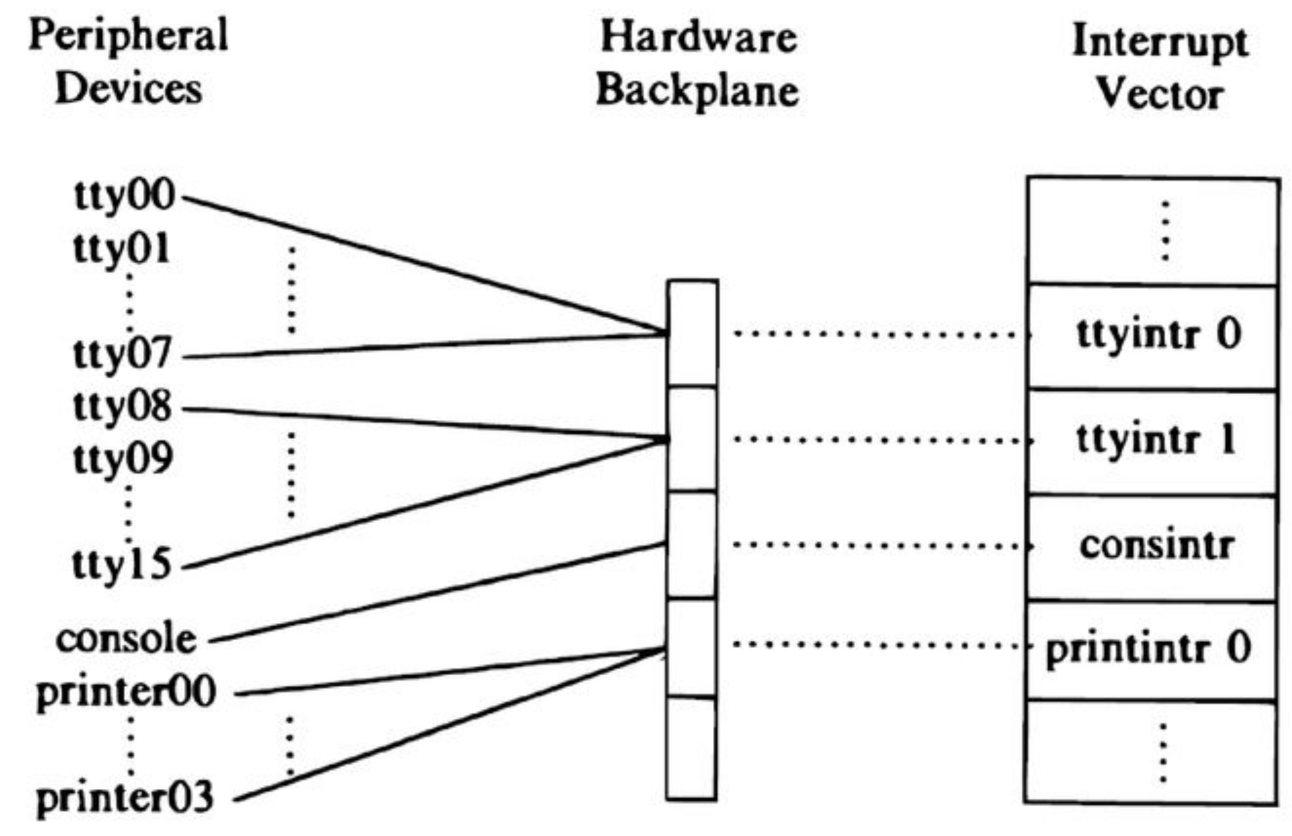


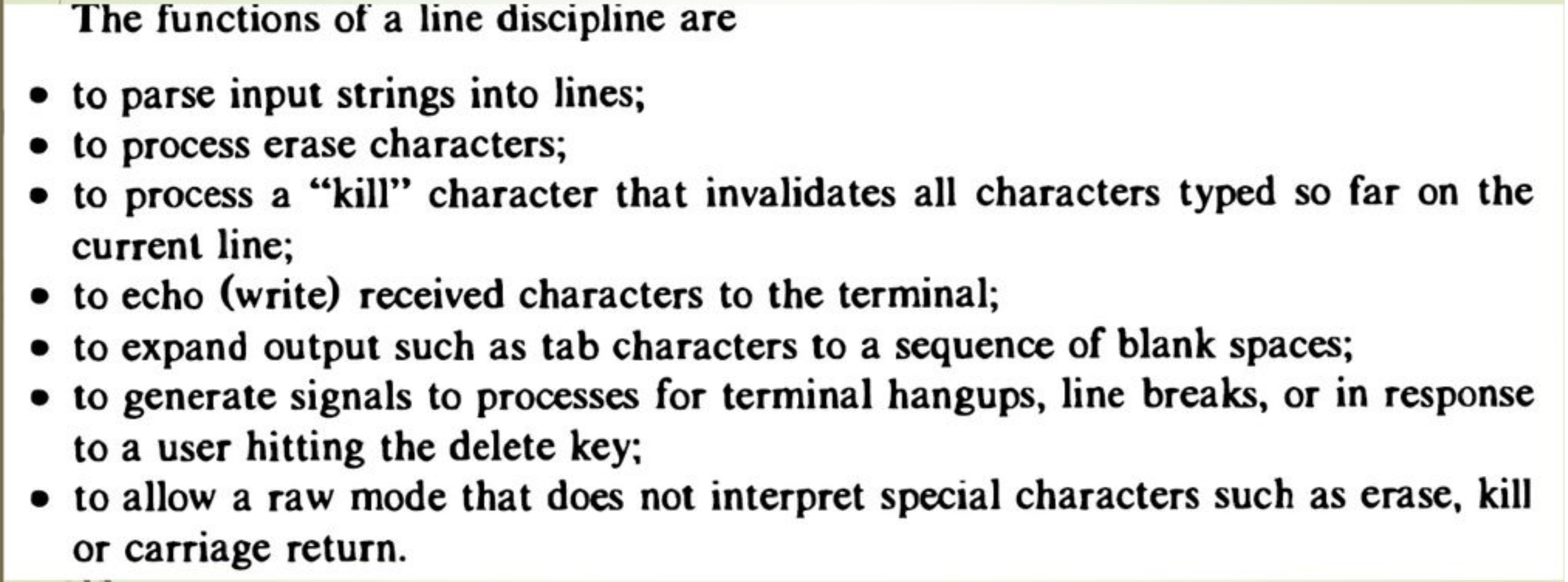
Figure 10.6. Device Interrupts

Disk Drivers

- The disk units on UNIX system have been configured into sections that contains individual file systems, allowing "the disk to be broken up into more manageable pieces".
- For instance, if a disk contains four manageable systems, an administrator may leave one unmounted, mount another "read only" and mount the last two "read-write".
- Even though all the unmounted file system coexist on one physical unit, users cannot access files in the unmounted file system using access methods.
- The disk driver translates a file system address, consisting of a logical device number and block number, to a particular sector on the disk.
- The driver gets the address in one of two ways:
- Either the strategy procedure uses a buffer from the buffer pool and the buffer header contains the device and block number, or the read and write procedures are passed the logical device number as a parameter

Terminal drivers

- Terminal drivers have the same function as other drivers: to control the transmission of data to and from the terminals.
- Terminals are special because they user's interface to the system.
- To accommodate interactive use of the UNIX system, terminal drivers contain an internal interface to line discipline module, which interprets input and output.
- In canonical mode, the line discipline converts raw data sequences typed at the keyboard to a canonical form(what the user really means) befor sending the data to receiving process.
- The line discipline also converts raw output sequences written by a process to a format that the user expects.
- In raw mode, the line discipline passes data between processes and the terminal without such conversions



The functions of a line discipline are

- to parse input strings into lines;
- to process erase characters;
- to process a "kill" character that invalidates all characters typed so far on the current line;
- to echo (write) received characters to the terminal;
- to expand output such as tab characters to a sequence of blank spaces;
- to generate signals to processes for terminal hangups, line breaks, or in response to a user hitting the delete key;
- to allow a raw mode that does not interpret special characters such as erase, kill or carriage return.

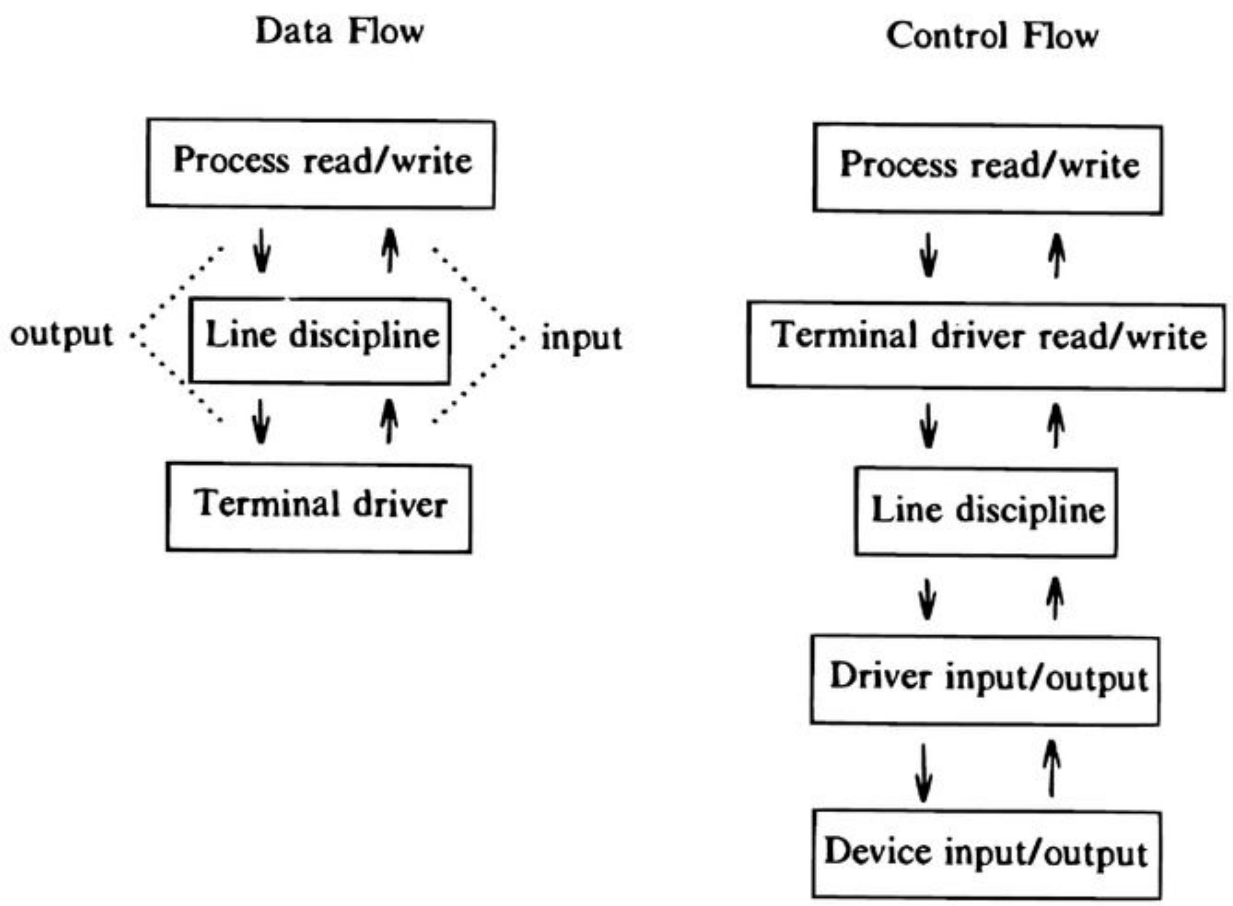
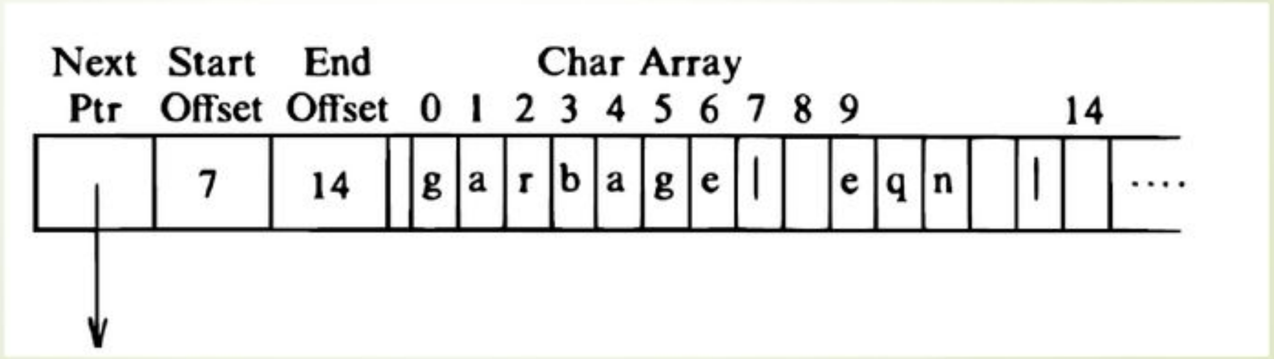


Figure 10.9. Call Sequence and Data Flow through Line Discipline

Clists

- Line discipline manipulates data on the clists.
- A clist, or character list, is a variable length linked list of cblocks with a count of the number of characters on the list.
- a cblock contains **a pointer** to the next **cblock** o the linked list, **a small character array** to contain **data**, and a set of **offsets** indicating the **position of the valid data in the cblock**
- The **start offset** indicates the first location of valid data in the array, and the **end offset** indicates the first location of non valid data.



The kernel maintains a linked list of free cblocks and has six operations on clists and cblocks.

1. It has an operation to assign a cblock from the free list to a driver.
2. It also has an operation to return a cblock to the free list.
3. The kernel can retrieve the first character from a clist: It removes the first character from the first cblock on the clist and adjusts the clist character count and the indices into the cblock so that subsequent operations will not retrieve the same character. If a retrieval operation consumes the last character of a cblock, the kernel places the empty cblock on the free list and adjusts the clist pointers. If a clist contains no characters when a retrieval operation is done, the kernel returns the null character.
4. The kernel can place a character onto the end of a clist by finding the last cblock on the clist, putting the character onto it, and adjusting the offset values. If the cblock is full, the kernel allocates a new cblock, links it onto the end of the clist, and places the character into the new cblock.
5. The kernel can remove a group of characters from the beginning of a clist one cblock at a time, the operation being equivalent to removing all the characters in the cblock one at a time.
6. The kernel can place a cblock of characters onto the end of a clist.

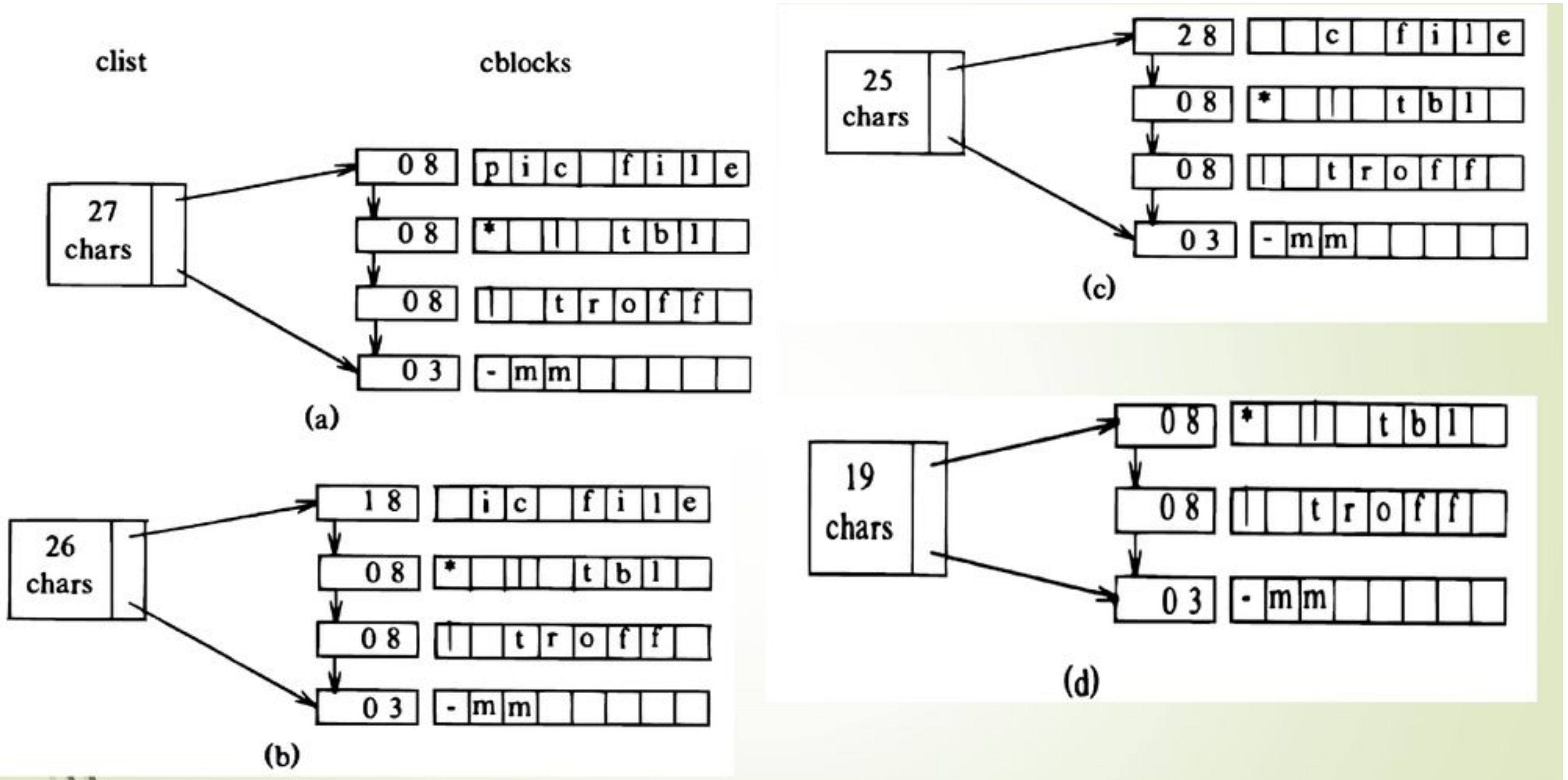


Figure 10.11. Removing Characters from a Clist

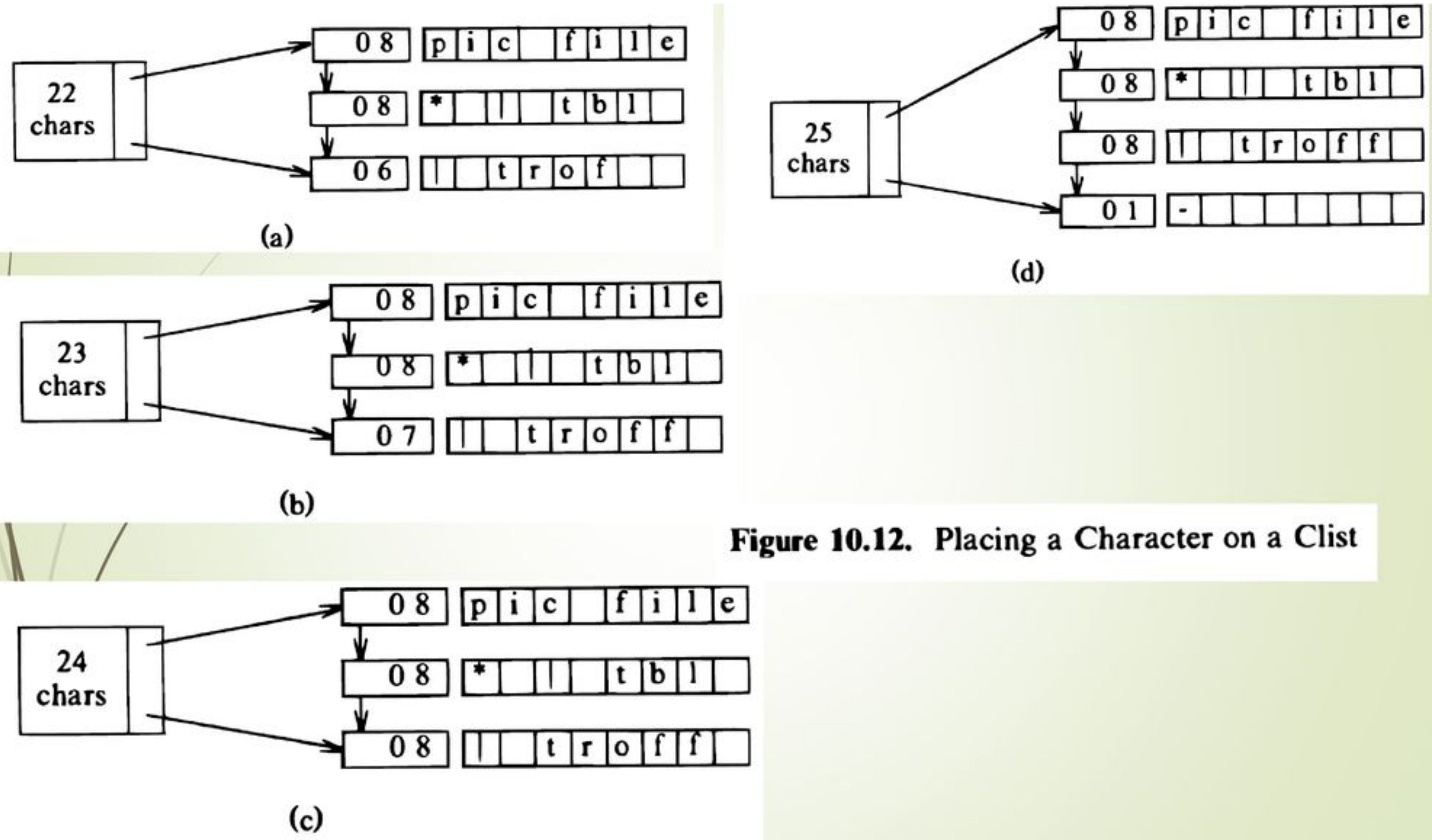


Figure 10.12. Placing a Character on a Clist

10.3.2 The Terminal Driver in Canonical Mode

The data structures for terminal drivers have three clists associated with them: a clist to store data for output to the terminal, a clist to store "raw" input data provided by the terminal interrupt handler as the user typed it in, and a clist to store "cooked" input data, after the line discipline converts special characters in the raw clist, such as the erase and kill characters.

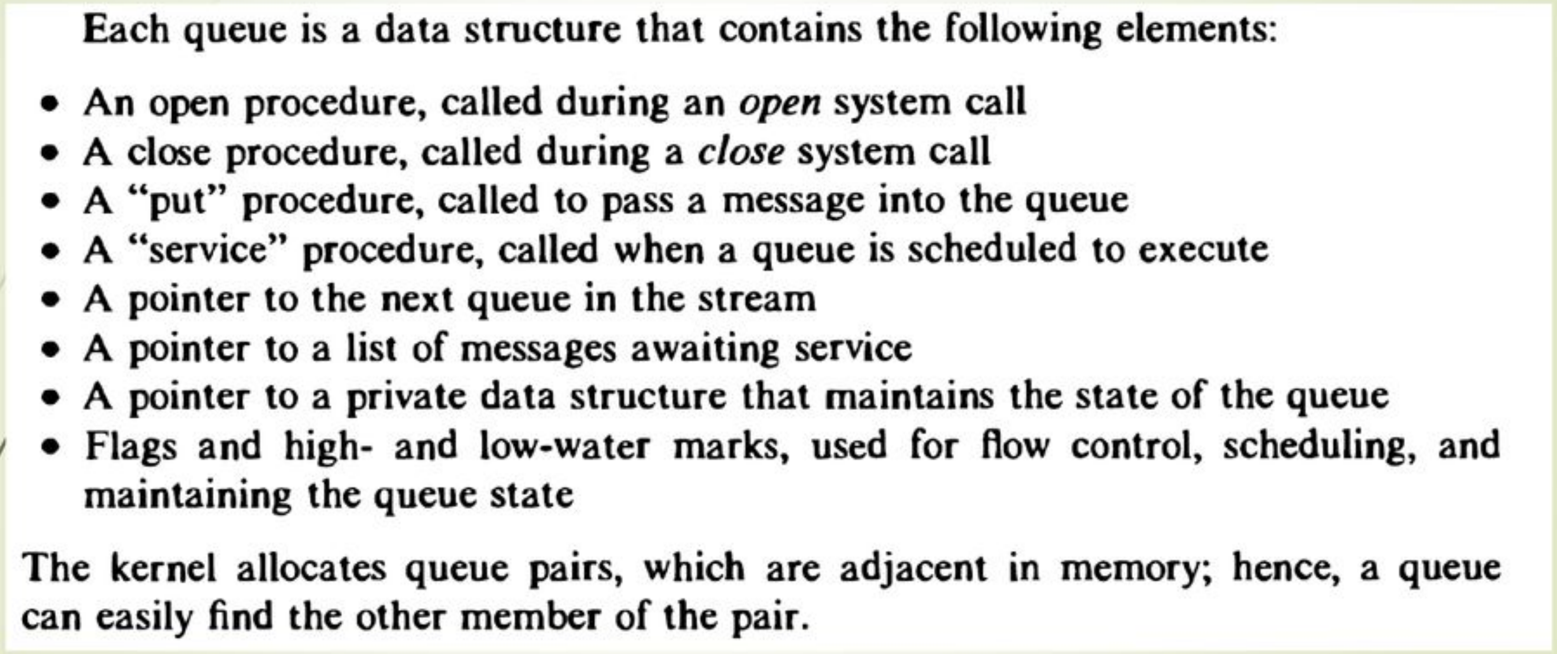
```
algorithm terminal_write
{
    while (more data to be copied from user space)
    {
        if (tty flooded with output data)
        {
            start write operation to hardware with data
                                          on output clist;
            sleep (event: tty can accept more data);
            continue;        /* back to while loop */
        }
        copy cblock size of data from user space to output clist:
                line discipline converts tab characters, etc;
    }

    start write operation to hardware with data on output clist;
}
```

Algorithm for writing data to the terminal

Streams

- A stream is a full-duplex connection between a process and a device driver
- It consists of a set of linearly linked queue pairs, one member of each pair for input and the other for output.
- When the kernel writes data to a stream, the kernel sends the data down the output queue; when a device driver receives input data, it sends the data up the input queues to a reading process.
- The queues pass messages to neighboring queue to a well defined interface.
- Each queue pair is associated with an instance of a kernel module, such as a driver, line discipline, or protocol, and the modules manipulate data passed through its queues.

Streams



Each queue is a data structure that contains the following elements:

- An open procedure, called during an *open* system call
- A close procedure, called during a *close* system call
- A "put" procedure, called to pass a message into the queue
- A "service" procedure, called when a queue is scheduled to execute
- A pointer to the next queue in the stream
- A pointer to a list of messages awaiting service
- A pointer to a private data structure that maintains the state of the queue
- Flags and high- and low-water marks, used for flow control, scheduling, and maintaining the queue state

The kernel allocates queue pairs, which are adjacent in memory; hence, a queue can easily find the other member of the pair.

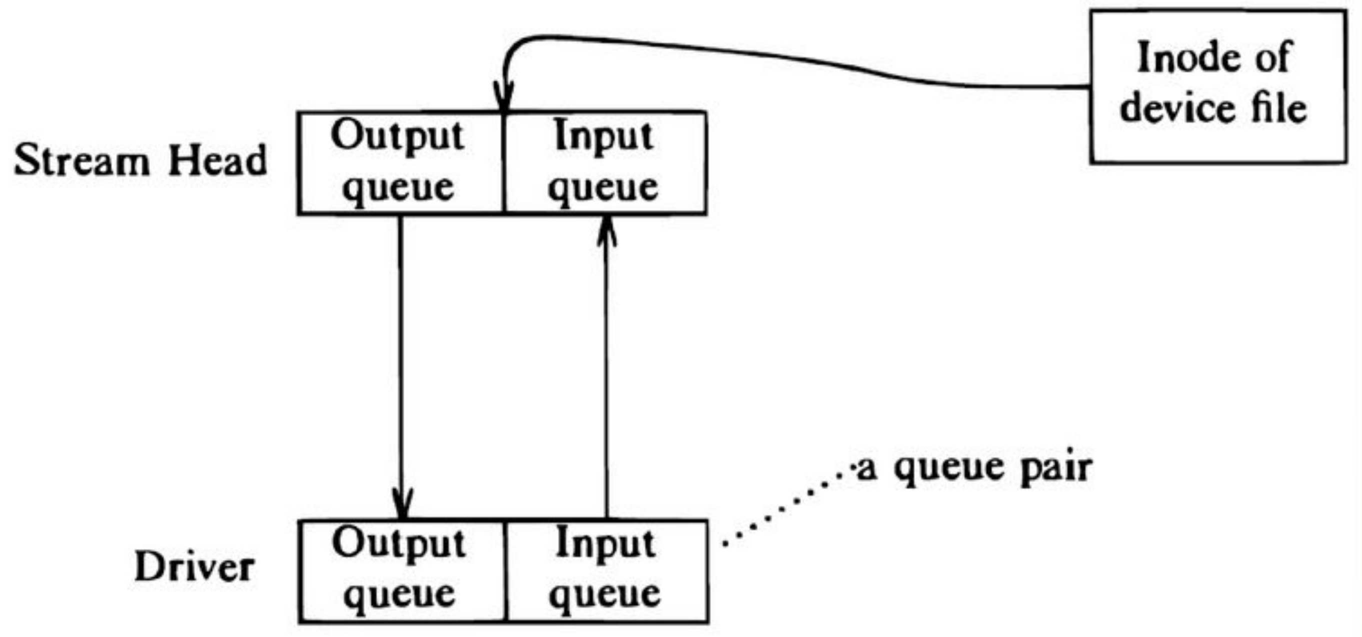


Figure 10.20. A Stream after Open

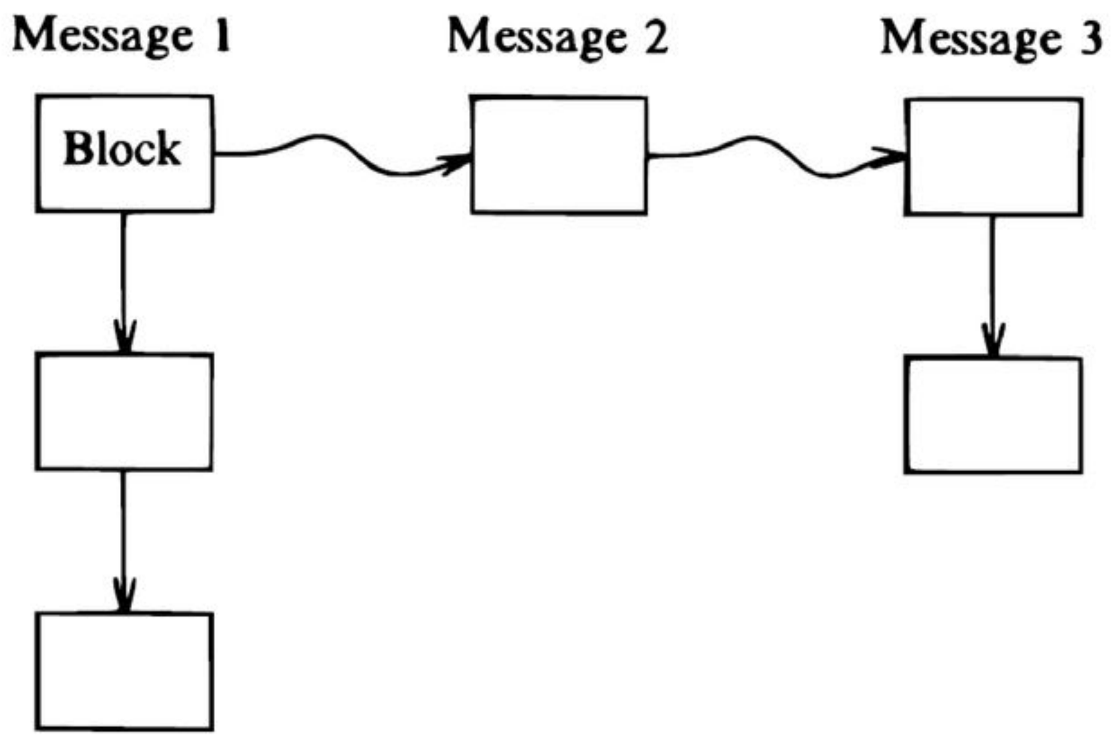


Figure 10.21. Streams Messages

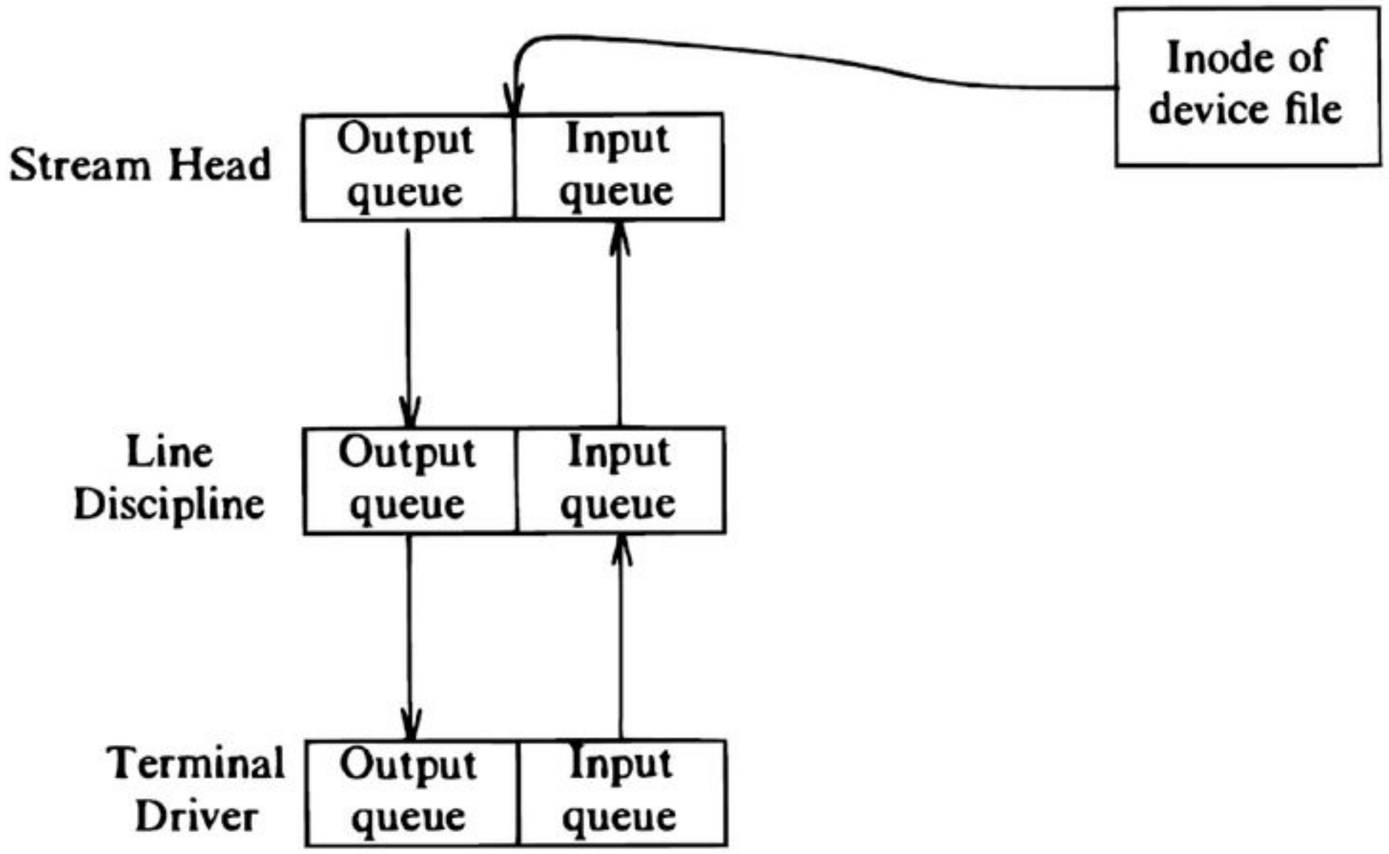


Figure 10.22. Pushing a Module onto a Stream