





13 Memory Management (mm)

In the Intel 80X86 architecture system, the Linux kernel's memory management program uses paging management. It uses the page directory and page table structure to handle the application and release of memory from other parts of the kernel. Memory management is performed in units of memory pages, and a memory page refers to 4K bytes of physical memory with consecutive addresses. The page directory entry and page table entry allow you to address and manage the usage of the specified memory page. There are three files in the memory management directory of Linux 0.12, as shown in Listing 13-1:

List 13-1 Memory management subdirectory file list

	Filename	Size	Last Modified Time (GMT)	Desc
	Makefile	1221 bytes	1992-01-12 19:49:22	
	memory.c	13464 bytes	1992-01-13 22:57:04	
	page.s	508 bytes	1991-10-02 14:16:30	
	swap.c	5193 bytes	1992-01-13 15:46:41	

Among them, the `page.s` assembly file is relatively short, it only contains the memory page exception interrupt service program (INT 14), mainly to achieve the processing of page fault and page write protection. `Memory.c` is the core program for memory page management. It is used for memory initialization operations, page directory and page table management, and other parts of the kernel for memory application processing. The `swap.c` program is used for memory page exchange management, which mainly includes exchange mapping bitmap management functions and switching device access functions.

13.1 Main Functionalities

In the Intel 80X86 CPU, the program uses an address consisting of segment and intra-segment offset during the addressing process. This address is not directly used to address physical memory addresses and is therefore referred to as a virtual address. In order to address physical memory, an address translation mechanism is needed to map or transform virtual addresses into physical memory addresses. This address translation mechanism is one of the main functionalities of memory management (Another main function is the memory addressing protection mechanism). The virtual address is first transformed into an intermediate address form by the segment management mechanism—the 32-bit linear address of the CPU, and then this linear address is mapped to the physical address using the paging mechanism.

In order to understand how the Linux kernel manages memory operations, we need to understand how memory paging management works and understand its addressing mechanism. The purpose of paging management is to map physical memory pages to a linear address. When analyzing the memory management program in this chapter, it is necessary to clearly distinguish whether the given address refers to a linear address or an actual physical memory address.

See Chapter 4 for a detailed description of memory management in Intel 80X86 CPU Protected Mode. For

the convenience of reading, we will further explain the related contents of the memory paging management mechanism.

13.1.1 Memory paging mechanism

In the Intel 80X86 system, memory paging management is performed through a two-level tables consisting of a memory page directory table and page tables, as shown in Figure 13-1. The page directory table and the page table have the same structure, and the table item structure is also the same, as shown in Figure 13-4 below. Each entry in the page directory table (referred to as a page directory entry, 4 bytes) is used to address a page table, and each page table entry (4 bytes) is used to specify a page of physical memory pages. Therefore, when a page directory entry and a page table entry are specified, we can uniquely determine the corresponding physical memory page. The page directory table occupies one page of memory, so up to 1024 page tables can be addressed, and each page table also occupies one page of memory, so a page table can also address up to 1024 physical memory pages. Thus, in a 32-bit 80X86 CPU, all page tables addressed by a page directory table can address a total of $1024 \times 1024 \times 4096 = 4\text{G}$ memory space. In the Linux 0.12 kernel, all processes share a single page-directory table, and each process has its own page table. The kernel code and data segment length is specified as 16MB, using 4 page tables (ie 4 page directory entries). After segmentation mechanism transformation, the kernel code and data segment are located in the first 16MB range of the linear address space, and then transformed by the paging mechanism, which is directly mapped one by one to 16MB of physical memory. So for the kernel segment its linear address is the physical address.

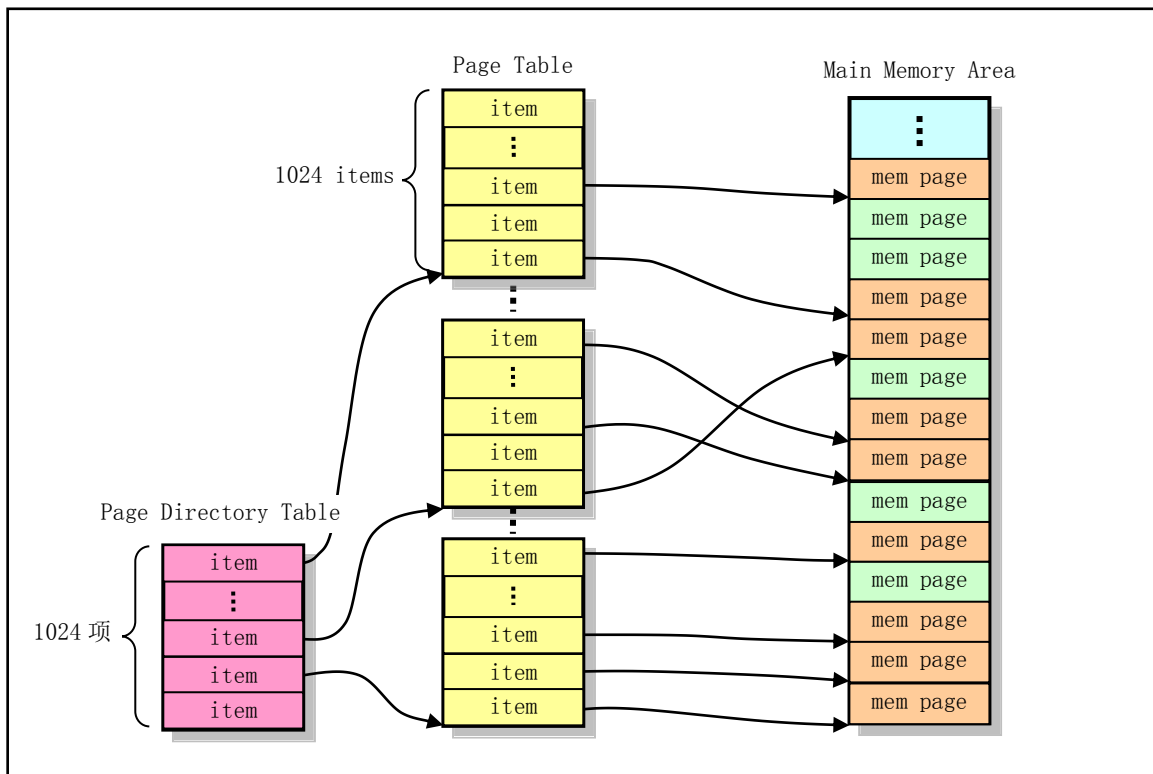


Figure 13-1 Page directory table and page table structure diagram

For user processes or other parts of the kernel, a linear address is used when applying for memory. So, how does a linear address use these two tables to map to a physical address? In order to use the paging mechanism, a 32-bit linear address is divided into three parts, which are used to specify a page directory entry, a page table

entry, and an offset address on the corresponding physical memory page, so that it can be indirectly addressed the physical memory location specified by the linear address, as shown in Figure 13-2.

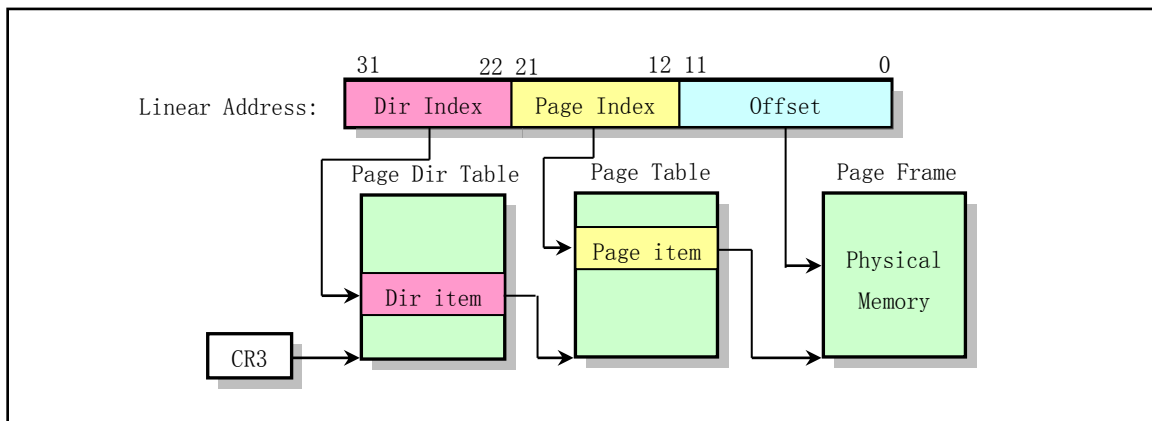


Figure 13-2 Schematic diagram of linear address transformation

Bits 31-22 of the linear address are used to determine the directory entries in the page directory; bits 21-12 are used to address the page table entries in the page table specified by the page directory entry, and the last 12 bits are used as the offset address in the physical memory page specified by the page entry.

In memory management functions, a large number of transform calculations from linear addresses to actual physical addresses are used. For the linear address of a given process, we can easily find the page directory entry corresponding to the linear address through the address translation relationship shown in Figure 13-2. If the directory entry is valid (used), the page frame address in the directory entry specifies the base address of a page table in physical memory. Then, in combination with the page table entry pointer in the linear address, if the page table entry is valid, based on the specified page frame address in the page table entry, we can finally determine the address of the actual physical memory page corresponding to the specified linear address. Conversely, if you need to find the corresponding linear address from a known physical memory page address, you need to search the entire page directory table and all page tables. If the physical memory page is shared, we may find multiple corresponding linear addresses. Figure 13-3 graphically shows how a given linear address is mapped onto a physical memory page. For the first process (task 0), its page table is after the page directory table, for a total of 4 pages. For an application's process, the memory used by its page table is applied to the memory manager when the process is created, so it is in the main memory area.

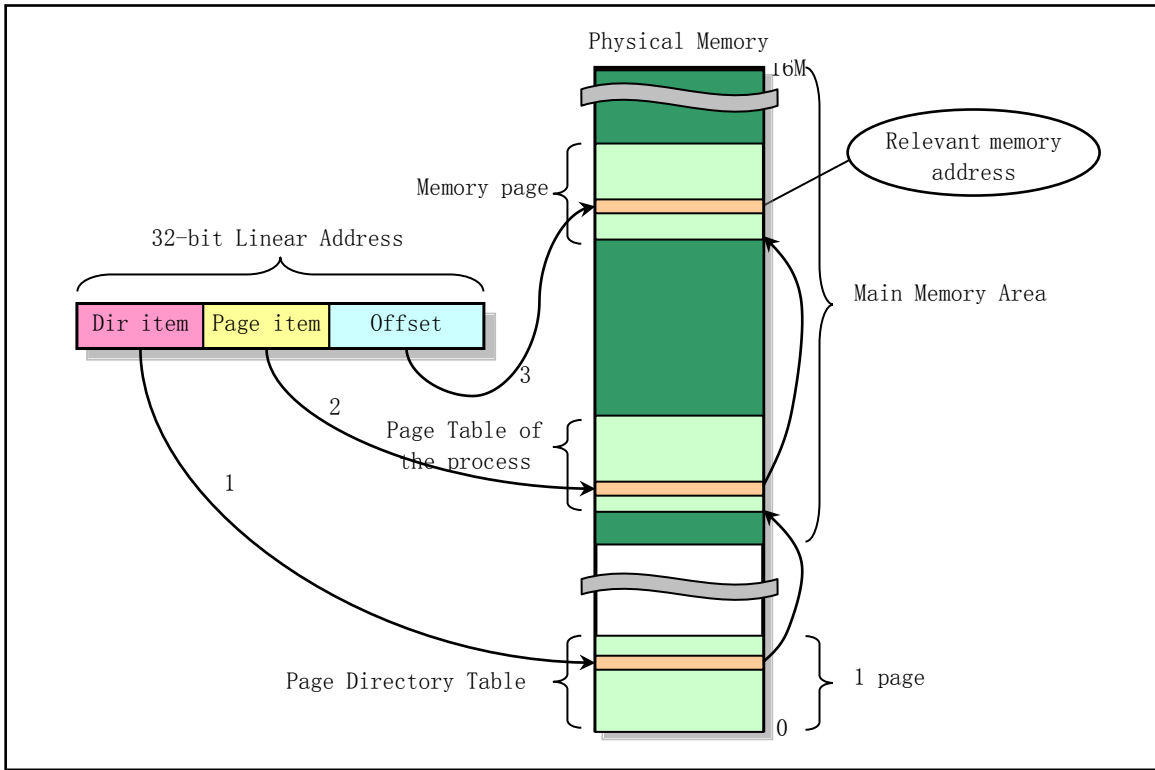


Figure 13-3 Physical address corresponding to the linear address

Multiple page directory tables can exist simultaneously in a system, and only one page directory table is available at a time. The currently used page directory table is determined by the CPU's register CR3, which stores the physical memory address of the current page directory table. But in the Linux kernel discussed in this book, the kernel code and all processes share a single page directory table.

In Figure 13-1, we see that the physical memory page corresponding to each page table entry is random within the 4G address range and is determined by the content of the page frame address in the page table entry set by the memory manager. Each page table entry consists of a page frame address, an access flag bit, a dirty (rewritten) flag bit, and a presence flag bit, as shown in Figure 13-4.

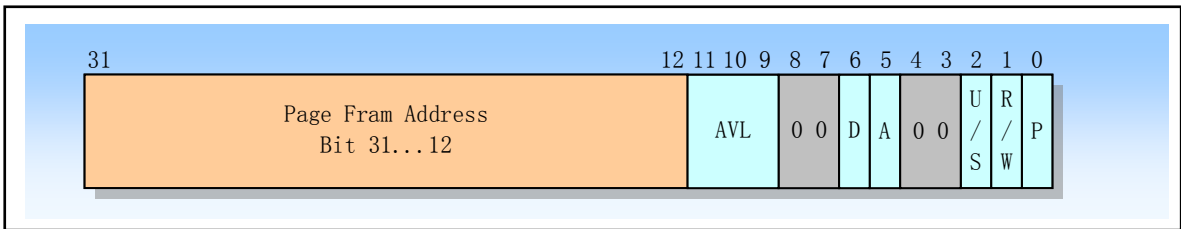


Figure 13-4 Page directory and page table entry structure

The page frame address specifies the physical start address of a page of memory. Because the memory page is on the 4K address boundary, its lower 12 bits are always 0, so the lower 12 bits of the entry can be used for other purposes. In a page directory table, the page frame address of the table entry is the start address of a page table; in the second level page table, the page frame address of the page table entry contains the physical memory page address of the desired memory operation.

The presence bit (P) in the figure determines whether a page table entry can be used for the address

translation process. $P=1$ means the entry is available. When the directory entry or the second-level entry has $P=0$, the entry is invalid and cannot be used in the address translation process. At this moment, all other bits of the entry are available to the program; the processor does not test these bits.

When the CPU attempts to use a page table entry for address translation, if $P=0$ of any one of the page table entries at this time, the processor will issue a page exception interrupt signal. At this point, the page fault interrupt exception handler can map and load the requested page into physical memory, and the instruction that caused the exception will be re-executed.

The accessed (A) and modified or dirty (D) bits are used to provide information about the use of the page. These bits are set by hardware, but are not reset, except for the modified bits in the page directory entry. The small difference between a page directory entry and a page table entry is that the page table entry has a dirty bit (D), while the page directory entry does not.

Before a read/write operation is performed on a page of memory, the CPU sets the accessed bits of the associated directory entry and secondary page table entry. Before writing to the address covered by a secondary page table entry, the processor sets the modified bit (D) of the secondary page table entry, and the bits (D) in the page directory entry are not used. When the required memory exceeds the actual amount of physical memory, the memory manager can use these bits to determine which pages can be taken from memory to make room. The memory manager is also responsible for detecting and resetting these bits.

The read/write bit (R/W) and the user/supervisor bit (U/S) are not used for address translation, but the protection mechanism for paging level is performed by the CPU during the address translation process at the same time.

13.1.2 Physical Memory Allocation and Management

With the above basic concepts, we can explain how the Linux system manages memory, but we also need to first understand the use of memory space by the Linux kernel. For the Linux 0.12 kernel, it supports up to 16M of physical memory by default. In an 80X86 computer system with 16MB of memory, the Linux kernel occupies the foremost part of physical memory, as shown in Figure 13-5. The label 'end' in the figure indicates the location where the kernel module ends. This is followed by a cache buffer with a maximum memory address of 4M. The cache buffer is divided into two sections by the display memory and the ROM BIOS. The remaining memory portion is called the main memory area. The main memory area is allocated and managed by the procedures in this chapter. If there is a RAM virtual disk in the system, the memory space occupied by the virtual disk needs to be deducted from the front of the main memory area. When you need to use the main memory area, you need to apply to the memory management program in this chapter. The basic unit of application is the memory page. Figure 13-5 shows the function of each part of the physical memory.

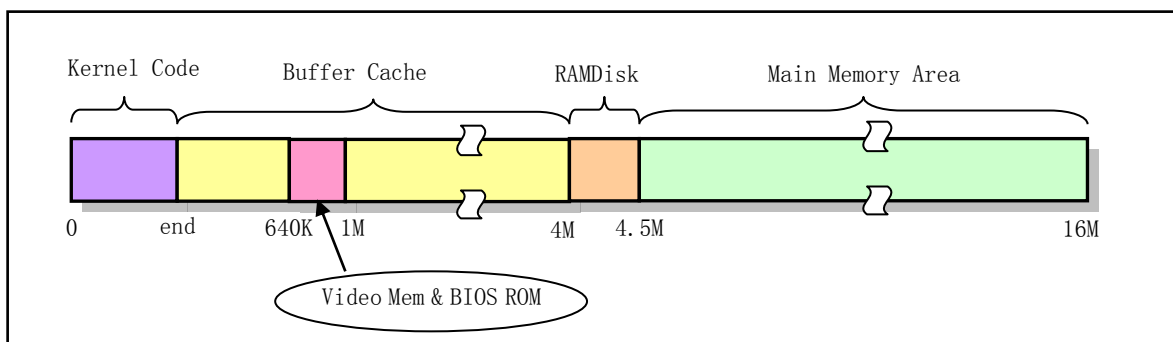


Figure 13-5 Main memory area schematic

In Chapter 6, Booting System, we already know that Linux's page directory and page table are set in the program head.s. The head.s program stores a page directory table at physical address 0, followed by four page tables. These four page tables will be used for mapping operations in the memory area occupied by the kernel. Since the code and data for task 0 are included in the kernel region, task 0 also uses these page tables. Other derived processes will request memory pages in the main memory area to store their own page tables. The two programs in this chapter are used to manage these tables to achieve the allocation of memory pages in the main memory area.

In order to save physical memory, when a new process is generated by calling `fork()`, the new process shares the same memory area with the original process. Only when one of the processes starts a write operation will the system allocate additional memory pages for it. This is the concept of copy-on-write.

The `page.s` program is used to implement the page fault or exception processing (INT 14). For interrupts caused by page faults and page write protection, the interrupt handler will call the `do_no_page()` and `do_wp_page()` functions in `memory.c` respectively. `do_no_page()` will take the required page from the block device to the memory specified location. In the case of shared memory pages, `do_wp_page()` copies the page being written (copy-on-write), which also cancels the sharing of the page.

13.1.3 Linear address space allocation

When reading the code in this chapter, we also need to understand the distribution of code and data in a program's logical address space, as shown in Figure 13-6.

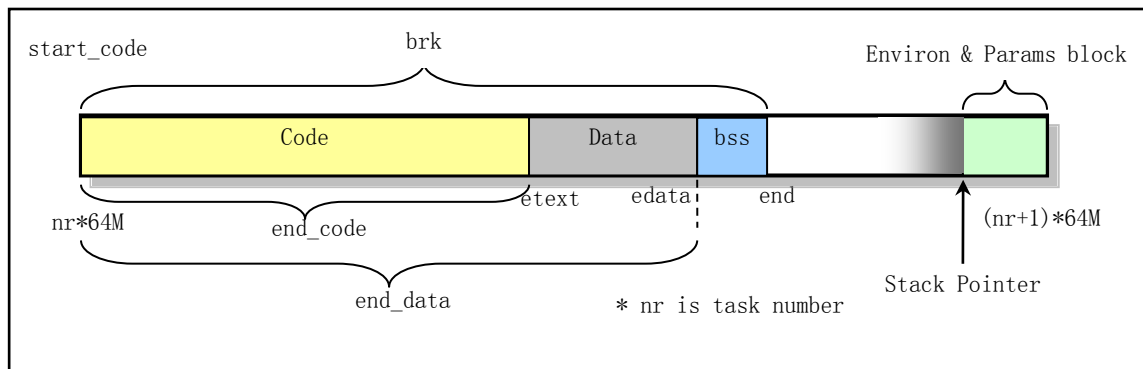


Figure 13-6 The distribution of processes in their logical address space

The logical address space occupied by each process starts from the location of `nr*64MB` in the linear address space (`nr` is the task number), and the logical address space occupies a range of 64MB. The last part of the environment parameter data block is up to 128K in length, and its left side is the starting stack pointer. 另外, In the figure, `bss` is the data segment that is not initialized by the process. The first page of the `bss` segment is initialized to all zeros when the process is created.

13.1.4 Transform between logical, linear, and physical address

In the kernel memory management code, the conversion operation between the program logical address (or virtual address), the linear address of the CPU, and the physical memory address is often encountered. For example, when copying a page table, we need to linearly translate a given page directory entry (PDE) to get the physical memory address of the corresponding page table (PT); when it comes to copy-on-write operations, it involves mapping of pages in linear address space to physical addresses; when attempting to share a page, it

involves mapping operations that map program logical address pages into CPU linear address space. Below we explain the conversion operation between them separately. Since the memory management in the kernel is usually operated in units of 4KB memory pages, let us first give the conversion method of the address to the page start address, and then explain how the pages in these different address spaces are converted.

1. Conversion of address to corresponding page address

Since the page address starts on the 4KB memory address boundary (ie, the lower 12 bits of the page address is 0), the address 'Page_addr' of a memory page containing the specified address 'addr' is:

```
Page_addr = addr & 0xfffff000;
```

In these three address spaces, their page address calculation process is the same. Below we explain the transformation calculation method for the page address.

2. Linear address and logical address decomposition

According to the paging mechanism of the CPU, a 32-bit linear address 'addr' can be decomposed into a page directory entry PDE number (bits 31-22), a page table entry PTE number (bits 21-12), and an in-page offset (bit 11- 0), see Figure 13-2. Therefore, the general formula for obtaining these three parts independently is as follows:

```
Page directory entry number: PDE_No = (addr >> 22);  
Page table entry number: PTE_No = (addr >> 12) & 0x3ff;  
Offset within the page: Offset = (addr & 0xfff);
```

Similarly, when performing the address translation operation, we can also regard the logical address 'Vaddr' in the process logical address space as being composed of these three parts "logically". Just when mapping them to the CPU linear address space, you need to add the corresponding part of the program code base address 'Base' in the linear address space:

```
Logic_PDE_No = (Vaddr >> 22);  
Logic_PTE_No = (Vaddr >> 12) & 0x3ff;  
Logic_Offset = (Vaddr & 0xfff);
```

Since in this kernel, each process allocates a linear address space in units of 64 MB in length, the page table entry number and the in-page offset value corresponding to the code base address of each process are both 0, so we only need to consider the page directory entry number of the code base address during the conversion process. In the actual code, the code base address 'Base' of the process 'p' is 'p->start_code', so when a logical address 'Vaddr' in the program corresponds to the linear address space, the three parts of the address are:

```
PDE_No = (Base >> 22) + Logic_PDE_No;  
        = (p->start_code >> 22) + (Vaddr >> 22);
```

```
PTE_No = Logic_PTE_No
        = (Vaddr >> 12) & 0x3ff;
Offset = Logic_Offset
        = (Vaddr & 0xfff);
```

3. Program page logical address to physical address conversion

The so-called logical page address means that the page address is the address calculated from the start address of the program process 'p' code, which is called the code base address, as shown in Figure 13-6. According to the structure of page directory entries and page table entries (see Figure 13-4), and the above decomposition of logical addresses, because each page directory entry and page table entry take up 4 bytes, we left shift the table entry number 2 bits to get the offset of entry in the table, plus the physical base address PDT_Base of the page directory table, we can get the directory entry address (pointer) PDE. For the Intel 80X86 CPU, its current page directory base address PDT_Base is stored in control register CR3. Because all programs in this kernel share only one page directory table, and the page directory table is stored at the beginning of physical memory 0, the page directory table base address PDT_Base=0. So we have the address (pointer) PDE of the page directory entry in physical memory:

```
PDE = PDT_Base + (PDE_No << 2);
    = 0 + (PDE_No << 2);
    = (((p->start_code >> 22) + (Vaddr >> 22)) << 2);
    = ((p->start_code >> 20) & 0xffc) + ((Vaddr >> 20) & 0xffc);
```

Referring to Figure 13-4, we can get the physical address PT of the corresponding page table from the contents of the directory entry, and add the offset of the entry in the page table to obtain the physical address (pointer) of the page table entry PTE. :

```
PT = (*PDE) & 0xfffff000;
PTE = PT + (PTE_No << 2)
    = PT + (((Vaddr >> 12) & 0x3ff) << 2);
    = ((*PDE) & 0xfffff000) + ((Vaddr >> 10) & 0xffc);
```

Bits 31-12 of the page table entry are the physical page frame addresses. Therefore, the physical page address corresponding to the logical address 'Vaddr' of the final program is:

```
PPaddr = (*PTE) & 0xfffff000;
```

4. Linear address to physical address conversion

According to the above decomposition of the linear address, for the linear address Laddr, its page directory entry number, page table entry number, and in-page offset value have been given at the beginning of the second point above. Correspondingly, the offset value PDE in the page directory table corresponding to the page directory entry number is:

```
PDE = (PDE_No << 2);
      = ((Laddr >> 22) << 2);
      = ((Laddr >> 20) & 0xffc);
```

We can get the physical address PT of the corresponding page table from the contents of the directory entry, and add the offset value of the entry in the page table to obtain the physical address PTE (pointer) of the page table entry:

```
PT = (*PDE) & 0xfffff000;
PTE = PT + (PTE_No << 2);
      = PT + ((Laddr >> 12) & 0x3ff) << 2);
      = (*PDE) & 0xfffff000 + ((Laddr >> 10) & 0xffc);
      = (*(Laddr >> 20) & 0xffc) & 0xfffff000 + ((Laddr >> 10) & 0xffc);
```

Therefore, the actual physical page address corresponding to the linear address 'Laddr' is 'PAddr', and the corresponding physical address 'Paddr' is the physical page address plus the offset within the page, as shown below:

```
PPAddr = (*PTE) & 0xfffff000;
Paddr = PPAddr + Laddr & 0xfff;
       = (*PTE) & 0xfffff000 + Laddr & 0xfff;
       = (((Laddr >> 10) & 0xffc) + (((Laddr >> 20) & 0xffc) & 0xfffff000)) + Laddr & 0xfff;
```

13.1.5 Page-fault exception handling

In the state where the paging mechanism (PG=1) is enabled, if the CPU detects the following conditions during the conversion of the linear address to the physical address, it will cause a page-fault exception interrupt INT 14:

- The presence bit (P) in the page directory entry or page table entry used in the address translation process is equal to 0, indicating that the page table or the page containing the operand does not exist in physical memory;
- The current execution code does not have sufficient privileges to access the specified page, or the user mode code writes a read-only page, and so on.

The page exception handling procedure can recover and restart an interrupted program or process from the page-not-present conditions and does not affect the continuity of program execution. It can also restart a program or task after a privilege violation, but the problem that caused the privilege violation may not be corrected. At this point, the CPU provides the following two aspects to the page-fault exception handler to assist in diagnosing and correcting the error:

- An error code on the stack. The format of the error code is a 32-bit long word, but only the lowest 3 bits are useful. Their names are the same as the last three bits in the page table entry (U/S, W/R, P). Their meanings and roles are:
 - ◆ Bit 0 (P), the exception is caused by a not-present page or violating access privileges. P=0, indicating that the page does not exist; P=1 indicates that the page-level protection privilege is

violated.

- ◆ Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
- ◆ Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, the CPU was executing at supervisor mode; U/S=1, CPU was executing at user mode.
- The linear address in control register CR2. The CPU will store the linear address that generated the exception in CR2. The page fault exception handler can use this address to locate the relevant page directory and page table entry. If another page exception is allowed to occur during the execution of the page exception handler, the handler should push CR2 onto the stack.

The `page.s` program that will be described later uses the above information to distinguish between a page fault exception or a write protection exception, thereby determining whether to call the page fault processing function `do_no_page()` or the write protect function `do_wp_page()` in the `memory.c` program.

13.1.6 Copy-on-write mechanism

Copy-on-write is a way to defer or avoid copying data. At this point, the kernel does not copy the data in the entire address space of the process, but allows the parent and child processes to share the same copy. When process A creates a child process B using the `fork()`, since child process B is actually a copy of parent process A, it will have the same physical page as the parent process. That is, in order to save memory and speed up the process of creating a process, the `fork()` function will let the child process B share the physical page of the parent A in a read-only manner, and also set the access rights of the parent A to these physical pages to only read mode too (see the `copy_page_tables()` in the `memory.c` program). In this way, when either parent A or child B performs a write operation on these shared physical pages, a page-fault exception (INT 14) is generated. At this point, the CPU will execute the system-provided exception handler `do_wp_page()` to try to resolve the exception. This is the copy-on-write mechanism.

The `do_wp_page()` function un-shares the physical page that caused the write exception interrupt (by calling the `un_wp_page()` function) and copies a new physical page for the write process, thus making the parent A and the child B each has a physical page with the same content. And the physical page that will perform the write operation is marked as write-accessible, and then the copy operation is actually performed (only this physical page is copied). Finally, when returning from the exception handler, the CPU will re-execute the write operation that caused the exception, allowing the process to continue.

Therefore, when a process writes within its own virtual address range, the passive copy-on-write operation above is used, namely: write operation -> page fault exception -> handle write protection exception -> re-execute the write operation instruction. For kernel code, when a write operation is performed within the virtual address range of a process, for example, a process invokes a system-call, if the system-call copies data into the buffer of the process, the kernel will call function `verify_area()`. This function first actively calls the memory page verification function `write_verify()` to determine whether there is a page sharing condition. If there is, the copy-on-write operation of the page is performed.

In addition, it is worth noting that in the Linux 0.12 kernel, the copy-on-write technique was not used when executing the `fork()` to create the process in the kernel code address space (linear address <1MB) . So when process 0 (ie idle process) creates process 1 (init process) in kernel space, it will use the same piece of code and data segments. However, since the page table entry copied by process 1 is also read-only, when the process 1 needs to perform a stack (write) operation, it also causes a page exception, so in this case the memory manager will allocate memory for the process in the main memory area. .

It can be seen that copy-on-write will delay the copying operation of the memory page until the actual

write operation, and the page copy operation may not be performed at all when the page is not written. For example, when `fork()` creates a process and immediately calls `execve()` to execute a new program. So this technique can avoid the overhead of unnecessary memory page copying.

13.1.7 Load on demand mechanism

When using the `execve()` system call to load an executable image file on the file system, the kernel allocates 64MB of contiguous space to the corresponding process in the 4G linear address space of the CPU, and applies and allocates a certain amount physical memory pages for its environment and command line parameters. Other than that, there is actually no other physical memory page allocated to the executable. Of course, it is also impossible to load the code and data from the execution image file from the file system. Therefore, once the program starts running from the entry execution point, it will immediately cause the CPU to generate a page fault exception (the memory page where the execution pointer is located does not exist). At this point, the kernel's page fault exception handler will load the relevant code page in the executable file from the file system into the physical memory page according to the specific linear address causing the page fault exception, and map to the page position specified in the process logical address. When the exception handler returns, the CPU re-executes the instruction that caused the exception, allowing the executable program to continue execution.

If the program needs to run to another page that has not been loaded during execution, or the code instruction needs to access data that has not been loaded, then the CPU will also generate a page fault exception interrupt, and the kernel will then load the other corresponding page content into memory and execute the program again. In this way, only the code or data pages that are run to (used) in the executable file are loaded into the physical memory by the kernel. This method of loading a page in an executable file only when it is actually needed is called a load on demand technique or a demand-paging technique.

One obvious advantage of using a demand-loading technique is that it allows the executable program to start running immediately after calling the `execve()` system-call, without having to wait for multiple block device I/O operations to load the entire executable file image into memory before it starts running. Therefore, the execution speed of the system to load the execution program will be greatly improved. However, this technique has certain requirements on the format in which the object file is loaded. It requires that the file object format being executed be of the ZMAGIC type, that is, the object file format of the demand paging format. In this object file format, the code segment and data segment of the program are stored from the page boundary to accommodate the kernel to read the code or data content in units of one page.

13.2 memory.c

13.2.1 Function

The `memory.c` program manages the paging of the memory, realizing the dynamic allocation and recycling of the memory pages in the main memory area. For physical memory areas other than the memory occupied by the kernel (above 1MB address), the kernel uses a byte array `mem_map[]` to indicate the state of the physical memory page. Each byte entry describes the occupancy status of a physical memory page. The value indicates the number of times occupied, and 0 indicates that the corresponding physical memory is idle. When applying a page of physical memory, the value of the corresponding byte is incremented by one. When the byte value is 100, it means that it is completely occupied and can no longer be allocated any more.

In the course of memory management initialization, the kernel code first calculates the number of memory

pages (PAGING_PAGES) corresponding to the memory area above 1MB, and sets value of all items of `mem_map[]` to 100 (occupied), and then the `mem_map[]` items corresponding to the main memory area are all cleared (zeroed), as shown in Figure 13-7. Therefore, the buffer cache area above the 1MB address used by the kernel and the virtual disk area (if any) have been initialized to be in a full occupancy state. The items corresponding to the main memory area in `mem_map[]` are set or reset during system use. For example, for a machine with 16MB of physical memory and a 512KB virtual disk as shown in Figure 13-5, the `mem_map[]` array has a total of $(16\text{MB} - 1\text{MB}) / 4\text{KB} = 3840$ entries, which corresponds to 3840 pages. The number of pages in the main memory area is $(16\text{MB} - 4.5\text{MB}) / 4\text{KB} = 2944$, corresponding to the last 2944 items of the `mem_map[]` array, while the first 896 items correspond to the physical memory occupied by the cache buffer and virtual disk above the 1MB memory. Therefore, in the memory management initialization process, the first 896 items of `mem_map[]` are set to the occupied state (value is 100) and can no longer be allocated for use. The value of the 2944 entry is cleared to 0 and can be allocated by the memory manager.

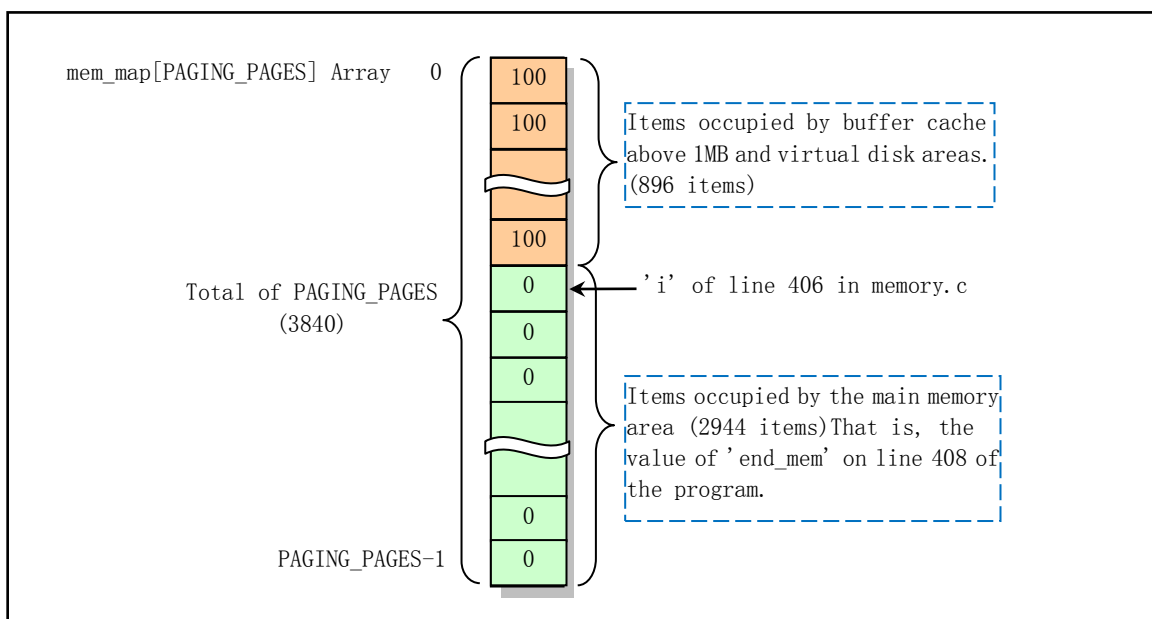


Figure 13-7 Mem_map[] array initialization with 16MB memory and 512KB vdisk

For the management of the process virtual address (or logical address), the kernel uses the segment management mechanism of the processor to implement, and the mapping relationship between the physical memory page and the linear address is handled by modifying the contents of the page directory and the page table entry. The following is a detailed description of several main functions provided in the program.

The `get_free_page()` and `free_page()` functions are specifically designed to manage the occupation and freeness of physical memory in the main memory area, regardless of the linear address of each process. The `get_free_page()` function is used to request a page of free memory in the main memory area and return the starting address of the physical memory page. It first scans the memory page bytemap array `mem_map[]`, looking for a byte entry with a value of 0 (corresponding to a free page). If not, it returns 0, indicating that the physical memory has been used up. If a byte with a value of 0 is found, it is set to 1 and the starting address of the corresponding free page is calculated. Then the memory page is cleared, and finally the physical memory start address of the free page is returned.

`free_page()` is used to release a page of physical memory at the specified address. It first determines if the given memory address is <1M, and if so, returns, because within 1M is kernel-specific; If the specified physical

memory address is greater than or equal to the actual memory highest address, an error message is displayed; Then, the page number is converted by the specified memory address: $(addr - 1M) / 4K$; then it is judged whether the `mem_map[]` byte item corresponding to the page number is 0, if not 0, then it is decremented by 1 and returned; Otherwise, the byte entry is cleared and an error message "Trying to release a free page" is displayed.

The `free_page_tables()` and `copy_page_tables()` functions release or copy the physical memory page blocks corresponding to the specified linear address and length (the number of page tables) in units of physical memory blocks (4M) corresponding to one page table. They not only modify the contents of the corresponding directory entries in the page directory and page table of the linear address, but also release or occupy the physical memory pages corresponding to all page table items in each page table.

The `free_page_tables()` function is used to release the physical memory page corresponding to the specified linear address and length (number of page tables). It first determines that the specified linear address should be on the 4M boundary and that the specified address value should be above the space occupied by the kernel and buffer cache. Then calculate the number of directory entries (ie, the number of page tables) occupied in the page directory table, and calculate the corresponding start directory entry number. Then, starting from the corresponding start directory entry, the occupied directory entry is released, and all page table entries and corresponding physical memory pages in the page table pointed to by the corresponding directory entry are released. Finally refresh the page transform cache.

The `copy_page_tables()` function is used to copy the page directory entry and page table corresponding to the specified linear address and length (page table count) memory, so that the copied page directory and the original physical memory area corresponding to the page table are shared. The function first verifies whether the specified source and destination linear addresses are both at the 4Mb memory boundary, and then calculates the corresponding start page directory entry (`from_dir`, `to_dir`) from the specified linear address; and calculates the number of page tables (page directory entries) occupied by the memory region to be copied. Then start copying the original directory entries and page table entries to the new free directory entries and page table entries, respectively. There is only one page directory table, and the page table of the new process needs to apply for a free memory page to store. Thereafter, the original and new page directories and page table entries are all set to read-only pages. When there is a write operation, the page exception handler is used to perform a copy-on-write operation. Finally, the byte array item corresponding to the shared physical memory page is incremented by one.

The `put_page()` function is used to map a specified physical memory page to a specified linear address. It first determines that the specified memory page address is within the range of 1M and the system's highest-end memory address, and then calculates the corresponding directory entry for the specified linear address in the page directory table. If the directory entry is valid, the address of the corresponding page table is taken, otherwise the free page is applied to the page table, and the attributes of the page table entry are set. Finally, the specified physical memory page address is still returned.

The `do_wp_page()` function is a page write protection procedure called in the page exception handler (implemented in `mm/page.s`). It first determines if the address is in the code area of the process and then performs a copy-on-write operation.

`do_no_page()` is a page fault function called during a page exception. It first determines the offset length value of the specified linear address relative to the process base address in the process space. If it is larger than the code plus data length, or the process is just starting to create, then apply for a page of physical memory and map it to the process linear address. Otherwise, try to perform a page sharing operation. If not, apply for a page of memory and read a page of content from the device. If the specified (linear address + 1 page) length exceeds

the length of the process code plus data when the content of the page is added, the excess is cleared. The page is then mapped to the specified linear address.

The `get_empty_page()` function is used to get a page of free physical memory and map to the specified linear address. It mainly uses the `get_free_page()` and `put_page()` functions to implement this function.

13.2.2 Code annotation

Program 13-1 linux/mm/memory.c

```

1  /*
2   *  linux/mm/memory.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  demand-loading started 01.12.91 - seems it is high on the list of
9   *  things wanted, and it should be easy to implement. - Linus
10  */
11
12  /*
13   *  Ok, demand-loading was easy, shared pages a little bit trickier. Shared
14   *  pages started 02.12.91, seems to work. - Linus.
15   *
16   *  Tested sharing by executing about 30 /bin/sh: under the old kernel it
17   *  would have taken more than the 6M I have free, but it worked well as
18   *  far as I could see.
19   *
20   *  Also corrected some "invalidate()"s - I wasn't doing enough of them.
21   */
22
23  /*
24   *  Real VM (paging to/from disk) started 18.12.91. Much more work and
25   *  thought has to go into this. Oh, well..
26   *  19.12.91 - works, somewhat. Sometimes I get faults, don't know why.
27   *              Found it. Everything seems to work now.
28   *  20.12.91 - Ok, making the swap-device changeable like the root.
29   */
30
31  // <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
32  //      manipulation function prototypes.
33  // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
34  //      descriptors/interrupt gates, etc. is defined.
35  // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
36  //      of the initial task 0, and some embedded assembly function macro statements about the
37  //      descriptor parameter settings and acquisition.
38  // <linux/head.h> Head header file. A simple structure for the segment descriptor is defined,
39  //      along with several selector constants.
40  // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
41  //      used functions of the kernel.
42  #include <signal.h>

```

```

33 #include <asm/system.h>
34
35 #include <linux/sched.h>
36 #include <linux/head.h>
37 #include <linux/kernel.h>
38
39 // CODE_SPACE(addr) (((addr)+0xfff)&~0xfff) < current->start_code + current->end_code)
40 // This macro is used to determine if a given linear address is within the code section of the
41 // current process. "(((addr)+4095)&~4095)" is used to obtain the end address of the memory
42 // page where the linear address 'addr' is located. See line 265.
43 #define CODE_SPACE(addr) (((addr)+4095)&~4095) < \
44 current->start_code + current->end_code)
45
46 // A global variable that holds the highest physical address of the actual physical memory.
47 unsigned long HIGH_MEMORY = 0;
48
49 // Copy 1 page of memory from 'from' to 'to' (4K bytes).
50 #define copy_page(from, to) \
51 __asm__ ("cld ; rep ; movsl"::"S" (from), "D" (to), "c" (1024):"cx", "di", "si")
52
53 // Physical memory mapped byte array (1 byte represents 1 page of memory). The corresponding
54 // byte of each page is used to mark the number of times the page is currently referenced
55 // (occupied). For machines with 16MB of physical memory, it can map up to 15Mb of memory. In
56 // the initialization function mem_init(), the position that cannot be used as the main memory
57 // area page is set to USED (100) in advance.
58 unsigned char mem_map [ PAGING_PAGES ] = {0,};
59
60 /*
61  * Free a page of memory at physical address 'addr'. Used by
62  * 'free_page_tables()'
63  */
64
65 // Free 1 page of memory starting with the physical address 'addr'.
66 // The memory space below 1MB of physical address is used for kernel programs and buffers, and
67 // cannot be used as memory space for allocation pages. Therefore the parameter 'addr' needs
68 // to be greater than 1MB.
69 void free_page(unsigned long addr)
70 {
71 // This function first determines the rationality of the physical address 'addr' given by the
72 // parameter. If the physical address 'addr' is less than the low end of the memory (1MB), it
73 // means it is in the kernel program or cache range, and will not be processed; if the 'addr'
74 // is greater than or equal to the highest end of the physical memory in the system, it will
75 // display an error message and the kernel stopped working.
76 if (addr < LOW_MEM) return; // LOW_MEM = 1MB, defined in include/linux/mm.h, 30.
77 if (addr >= HIGH_MEMORY)
78     panic("trying to free nonexistent page");
79 // If the verification of the parameter 'addr' is passed, the memory page number counted from
80 // the low end of the memory is calculated based on this physical address.
81 // "Page number = (addr - LOW_MEM) / 4096"
82 // It can be seen that the page number starts from 0, and the page number is stored in 'addr'.
83 // If the mapping byte corresponding to the page number is not equal to 0, it is returned after
84 // decrementing by 1. At this point, the mapped byte value should be 0, indicating that the
85 // page has been released. However, if the corresponding page byte is originally 0, it means
86 // that the physical page is originally idle, indicating that the kernel code is faulty. The

```

```

// error message is then displayed and the kernel is stopped.
58     addr -= LOW_MEM;
59     addr >>= 12;                // divided by 4096.
60     if (mem_map[addr]--) return;
61     mem_map[addr]=0;
62     panic("trying to free free page");
63 }
64
65 /*
66  * This function frees a continuous block of page tables, as needed
67  * by 'exit()'. As does copy_page_tables(), this handles only 4Mb blocks.
68  */
69
70 // According to the given linear address and limit length (number of page tables), free the
71 // memory block and set the table entries to be empty.
72 // The page directory table is located at the beginning of physical address 0. There are 1024
73 // entries, each of which is 4 bytes, which together account for 4K bytes. Each directory entry
74 // corresponds to one page table. The kernel page table starts at the physical address 0x1000
75 // (following the directory table space) and has 4 page tables. Each page table has 1024 entries,
76 // each of which is 4 bytes, and therefore also accounts for 4KB (1 page) of memory. Except
77 // for processes 0 and 1 in the kernel code, the pages occupied by the page tables of other
78 // processes are requested by the kernel from the main memory area when the process is created.
79 // Each page table entry corresponds to 1 page of physical memory, so a page table can map up
80 // to 4MB of physical memory.
81 // Parameters: from - the linear base address; size - the length of bytes to be freed.
82 int free_page_tables(unsigned long from, unsigned long size)
83 {
84     unsigned long *pg_table;
85     unsigned long *dir, nr;
86
87     // First check if the linear base address given by the parameter 'from' is at the 4MB boundary,
88     // because this function can only handle this situation. If from = 0, an error occurs. This
89     // indicates that the function is trying to free up space occupied by the kernel and buffer.
90     // Then calculate the number of page directory entries corresponding to the length given by
91     // the parameter 'size' (multiple of 4MB with carry), that is, the number of page tables occupied.
92     // Since 1 page table can manage 4MB of physical memory, the value of the memory length to be
93     // copied is divided by 4MB by shifting 22 bits to the right. Add 0x3ffff (ie 4Mb -1) to get
94     // the result of the integer multiple with carry, that is, add 1 if there is a remainder in
95     // the operation. For example, if the original size = 4.01Mb, then the result size = 2 is obtained.
96     // Next, the starting directory entry corresponding to the given linear base address is
97     // calculated.
98     // The corresponding directory entry number is equal to "from >> 20". Since each entry occupies
99     // 4 bytes, and since the page directory table is stored from physical address 0, the actual
100    // directory entry pointer is "directory entry number << 2", which is "(from >> 20)". "& 0xffc"
101    // makes sure the directory entry pointer is within the valid range.
102    if (from & 0x3ffff)
103        panic("free_page_tables called with wrong alignment");
104    if (!from)
105        panic("Trying to free up swapper memory space");
106    size = (size + 0x3ffff) >> 22;
107    dir = (unsigned long *) ((from >> 20) & 0xffc); /* _pg_dir = 0 */
108
109    // At this point 'size' is the number of page tables that need to be released, ie the number
110    // of page directory entries, and 'dir' is the starting directory entry pointer. Now, start

```



```
// a loop operation for each page directory entry, and then release the entries in each page
// table in turn. If the current directory entry is invalid (P bit = 0), indicating that the
// directory entry is not used (the corresponding page table does not exist), the next directory
// entry continues to be processed. Otherwise, the page table address 'pg_table' is taken from
// the directory entry, and 1024 entries in the page table are processed, and the physical memory
// page corresponding to the valid page table entry (P bit=1) is released, or the invalid page
// table entry (P bit = 0) is released from the swap device, that is, release the corresponding
// memory page in the swap device (because the page may have been swapped out). Then clear the
// page table entry and continue processing the next page entry. When all the entries in a page
// table have been processed, the memory page occupied by the page table itself is freed, and
// the next page directory entry is processed. Finally refreshes the page transform cache and
// returns 0.
```

```
80     for ( ; size-->0 ; dir++) {
81         if (!(1 & *dir))
82             continue;
83         pg_table = (unsigned long *) (0xfffff000 & *dir); // get page table addr.
84         for (nr=0 ; nr<1024 ; nr++) {
85             if (*pg_table) {
86                 if (1 & *pg_table) // free the page if valid.
87                     free\_page(0xfffff000 & *pg_table);
88                 else // free the page in swap device.
89                     swap\_free(*pg_table >> 1);
90                 *pg_table = 0; // reset the page content.
91             }
92             pg_table++; // points to next page.
93         }
94         free\_page(0xfffff000 & *dir); // free the page of the page table.
95         *dir = 0; // reset the directory entry.
96     }
97     invalidate(); // refresh CPU page transform cache.
98     return 0;
99 }
100
101 /*
102  * Well, here is one of the most complicated functions in mm. It
103  * copies a range of linear addresses by copying only the pages.
104  * Let's hope this is bug-free, 'cause this one I don't want to debug :-)
105  *
106  * Note! We don't copy just any chunks of memory - addresses have to
107  * be divisible by 4Mb (one page-directory entry), as this makes the
108  * function easier. It's used only by fork anyway.
109  *
110  * NOTE 2!! When from==0 we are copying kernel space for the first
111  * fork(). Then we DONT want to copy a full page-directory entry, as
112  * that would lead to some serious memory waste - we just copy the
113  * first 160 pages - 640kB. Even that is more than we need, but it
114  * doesn't take any more memory - we don't copy-on-write in the low
115  * 1 Mb-range, so the pages can be shared with the kernel. Thus the
116  * special case for nr=xxxx.
117  */
```

```
//// Copy page directory entries and page table entries.
// This function is used to copy the page directory entries and page table entries corresponding
// to the specified linear address and memory size, so that the physical memory area corresponding
```

```

// to them is shared by the two sets of page table mapping. When copying, we need to apply for
// a new page to store the new page table, the original physical memory area will be shared.
// After that, the two processes (the parent and its child processes) will share the memory
// area until the kernel performs a write operation, and the kernel allocates a new memory page
// (copy-on-write) for the write operation process. The parameters 'from' and 'to' are linear
// addresses, and 'size' is the length of memory that needs to be copied (shared) in bytes.
118 int copy\_page\_tables(unsigned long from, unsigned long to, long size)
119 {
120     unsigned long * from_page_table;
121     unsigned long * to_page_table;
122     unsigned long this_page;
123     unsigned long * from_dir, * to_dir;
124     unsigned long new_page;
125     unsigned long nr;
126
// The code first detects the validity of the source address 'from' and the destination address
// 'to' given by the parameter. Both the source and destination addresses need to be on the
// 4Mb memory boundary address. This requirement is given because 1024 entries in a page table
// can manage 4 Mb of memory. The source address and the destination address only satisfy this
// requirement to ensure that the page table entry is copied from the first entry of a page
// table, and all the original entries of the new page table are valid. Then get the start
// directory entry pointers (from_dir and to_dir) of the source and destination addresses. Then
// calculate the number of page tables (ie, the number of directory entries) occupied by the
// memory block to be copied according to the 'size' given by the parameter.
127     if ((from & 0x3ffff) || (to & 0x3ffff))
128         panic("copy_page_tables called with wrong alignment");
129     from_dir = (unsigned long *) ((from >> 20) & 0xffc); /* _pg_dir = 0 */
130     to_dir = (unsigned long *) ((to >> 20) & 0xffc);
131     size = ((unsigned) (size + 0x3ffff)) >> 22;

// After obtaining the source start directory entry pointer from_dir and the destination start
// directory entry pointer to_dir and the number of page tables to be copied, the following
// begins to apply one page of memory to each page directory entry to save the corresponding
// page table, and start page table entry copy operation. If the page table specified by the
// destination directory entry already exists (P=1), the kernel crashes. If the source directory
// entry is invalid, that is, the specified page table does not exist (P=0), the next page
// directory entry continues to be looped.
132     for( ; size-->0 ; from_dir++, to_dir++) {
133         if (1 & *to_dir)
134             panic("copy_page_tables: already exist");
135         if (!(1 & *from_dir))
136             continue;

// After verifying that the current source directory entry and destination entry are normal,
// we take the page table address from_page_table in the source directory entry. In order to
// save the page table corresponding to the destination directory entry, it is necessary to
// apply for one page of free memory page in the main memory area. If the function get_free_page()
// returns 0, it means that no free memory page is obtained, and there may be insufficient memory.
// Then return a value of -1 to exit.
137         from_page_table = (unsigned long *) (0xfffff000 & *from_dir);
138         if (!(to_page_table = (unsigned long *) get\_free\_page()))
139             return -1;          /* Out of memory, see freeing */

```

```

// Next, we set the destination directory entry information: the last 3 bits are set, that is,
// the current destination directory entry "OR" 7, indicating that the memory page mapped to
// the corresponding page table is user-level, and can read, write, and exists (Usr, R/W,
// Present). (If the U/S bit is 0, then R/W has no effect. If U/S is 1, and R/W is 0, then the
// code running at the user level can only read the page. If U/S and R /W are all set, there
// is permission to read and write). Then set the number of page items to be copied for the
// page table corresponding to the currently processed page directory entry. If it is in kernel
// space, you only need to copy the corresponding page table entry of the first 160 pages (nr=160),
// corresponding to the beginning of 640KB physical memory, otherwise you need to copy all 1024
// entries in a page table (nr= 1024), which can map 4MB of physical memory.
140         *to_dir = ((unsigned long) to_page_table) | 7;
141         nr = (from==0)?0xA0:1024;                                // 0xA0 = 160

// At this point, for the current page table, we start to cyclically copy the specified 'nr'
// memory page table entries. We first take out the contents of the source page table entry.
// If the current source page is not used (the content is 0), then the table item is not copied
// and the next entry is processed.
142         for ( ; nr-- > 0 ; from_page_table++, to_page_table++) {
143             this_page = *from_page_table;
144             if (!this_page)
145                 continue;
// If the entry has content, but its presence bit P=0, the page corresponding to the entry may
// be in the swap device. So we apply for 1 page of memory and read the page from the swap device
// (if there is one in the swap device), and then copy the page table entry to the destination
// page table entry, and modify the source page table entry content to point to the new memory
// page, and set the table entry flag to "page dirty" plus 7. Then continue processing the next
// page entry. Otherwise, the R/W flag (bit 1 is set to 0) is reset in the page table entry,
// that is, the memory page corresponding to the page table entry is set to read-only, and then
// the page table entry is copied to the destination page table.
146             if (!(1 & this_page)) {
147                 if (!(new_page = get\_free\_page\(\)))
148                     return -1;
149                 read\_swap\_page(this_page>>1, (char *) new_page);
150                 *to_page_table = this_page;
151                 *from_page_table = new_page | (PAGE\_DIRTY | 7);
152                 continue;
153             }
154             this_page &= ~2;
155             *to_page_table = this_page;

// If the address of the physical page indicated by the page table entry is above 1MB, you need
// to set the memory page map array mem_map[]. Then calculate the page number and use it as an
// index to increase the number of references in the corresponding item of the page mapping
// array. For pages below 1MB, it is a kernel page, so there is no need to set mem_map[]. Because
// mem_map[] is only used to manage page usage in the main memory area. So for the kernel to
// move to task 0 and call fork() to create task 1 (used to run init()), since the copied pages
// are still in the kernel code area, the statements in the following judgment will not be
// executed. The page for task 0 can still be read and written at any time. The judgment statement
// will only be executed when the parent process code calling fork() is in the main memory area
// (page position is greater than 1MB). This happens only when the process calls execve() and
// loads the new program code.
// The meaning of the line 157 statement is that the memory page pointed to by the source page
// table entry is also read-only, because now two processes have shared memory areas. If one

```

```

// of the processes needs to perform a write operation, the page for the write operation can
// be allocated a new free page by the page exception write protection process, that is, a copy
// on write operation.
156         if (this_page > LOW\_MEM) {
157             *from_page_table = this_page; // set source read-only too.
158             this_page -= LOW\_MEM;
159             this_page >>= 12;
160             mem\_map[this_page]++;
161         }
162     }
163 }
164 invalidate(); // refresh CPU page transform cache.
165 return 0;
166 }
167
168 /*
169  * This function puts a page in memory at the wanted address.
170  * It returns the physical address of the page gotten, 0 if
171  * out of memory (either when trying to access page-table or
172  * page.)
173  */
174
175 // Map a physical memory page to a specified location in a linear address space.
176 // Or, the page at the specified address in the linear address space is mapped to the main memory
177 // area page. The main job of implementing this function is to set the information of the specified
178 // page in the relevant page directory entry and page table entry. This function is called in
179 // do_no_page() that handles page not present exceptions. For exceptions caused by page not
180 // present, when the page table is modified due to any page absence, it is not necessary to
181 // refresh the page translation buffer (or Translation Lookaside Buffer - TLB) of the CPU even
182 // if the page table entry flag P was changed from 0 to 1. Because invalid page entries are
183 // not buffered, there is no need to refresh when an invalid page table entry is modified, as
184 // shown here without calling the Invalidate() function. The parameter 'page' is a pointer to
185 // a page (page frame) in the allocated main memory area; 'address' is a linear address.
186
187 static unsigned long put\_page(unsigned long page, unsigned long address)
188 {
189     unsigned long tmp, *page_table;
190
191     /* NOTE !!! This uses the fact that _pg_dir=0 */
192
193     // Here first determine the validity of the physical memory page 'page' given by the parameter.
194     // A warning message is issued if the page position is lower than LOW_MEM or if the system actually
195     // contains the memory high-end HIGH_MEMORY. LOW_MEM=1MB, which is defined in the file
196     // include/linux/mm.h file, which is the lowest possible starting position of the main memory
197     // area. When the system physical memory is less than or equal to 6MB, the main memory area
198     // starts directly at LOW_MEM. Then check whether the page is the one that has been applied
199     // for, that is, whether the corresponding byte in the memory page map mem_map[] has been set.
200     // If not, a warning is required.
201     if (page < LOW\_MEM || page >= HIGH\_MEMORY)
202         printk("Trying to put page %p at %p\n", page, address);
203     if (mem\_map[(page-LOW\_MEM)>>12] != 1)
204         printk("mem_map disagrees with %p at %p\n", page, address);
205
206     // Then, according to the linear address given by the parameter, the corresponding entry pointer
207     // in the page directory table is calculated, and the page table address is obtained therefrom.

```

```

// If the directory entry is valid (P=1), that is, the specified page table is in memory, the
// specified page table address is taken from it and placed in the page_table variable. Otherwise,
// apply a free page to the page table, and set the corresponding flag (7 - User, U/S, R/W)
// in the corresponding directory entry, and then put the page table address into the page_table
// variable.
184     page_table = (unsigned long *) ((address>>20) & 0xffc);
185     if ((*page_table)&1)
186         page_table = (unsigned long *) (0xfffff000 & *page_table);
187     else {
188         if (!(tmp=get_free_page()))
189             return 0;
190         *page_table = tmp | 7;
191         page_table = (unsigned long *) tmp;
192     }
// Finally, the related entry content is set in the found page table page_table, that is, the
// address of the physical page is filled in the entry, and three flags (U/S, W/R, P) are set
// at the same time. The index value of the entry in the page table is equal to the 10-bit value
// consisting of linear address bits 21 - bits 12. There are a total of 1024 entries (0 -- 0x3ff)
// per page table.
193     page_table[(address>>12) & 0x3ff] = page | 7;
194 /* no need for invalidate */
195     return page; // Returns the physical page address.
196 }
197
198 /*
199  * The previous function doesn't work very well if you also want to mark
200  * the page dirty: exec.c wants this, as it has earlier changed the page,
201  * and we want the dirty-status to be correct (for VM). Thus the same
202  * routine, but this time we mark it dirty too.
203  */
204 unsigned long put_dirty_page(unsigned long page, unsigned long address)
205 {
206     unsigned long tmp, *page_table;
207
208 /* NOTE !!! This uses the fact that _pg_dir=0 */
209
210     if (page < LOW_MEM || page >= HIGH_MEMORY)
211         printk("Trying to put page %p at %p\n", page, address);
212     if (mem_map[(page-LOW_MEM)>>12] != 1)
213         printk("mem_map disagrees with %p at %p\n", page, address);
214     page_table = (unsigned long *) ((address>>20) & 0xffc);
215     if ((*page_table)&1)
216         page_table = (unsigned long *) (0xfffff000 & *page_table);
217     else {
218         if (!(tmp=get_free_page()))
219             return 0;
220         *page_table = tmp|7;
221         page_table = (unsigned long *) tmp;
222     }

```

```

223     page_table[(address>>12) & 0x3ff] = page | (PAGE\_DIRTY | 7);
224 /* no need for invalidate */
225     return page;
226 }
227
228 //// Cancel write page protection. Used for the processing of write protection exception
229 //// during page exceptions (copy-on-write).
230 //// When the kernel creates a process, the new process and the parent process are set to share
231 //// code and data memory pages, and all of these pages are set to read-only. When the new process
232 //// or the original process needs to write data to the memory page, the CPU will detect this
233 //// and generate a page write protection exception. So in this function the code will first
234 //// determine if the page to be written is shared. If not, set the page to be writable and then
235 //// exit; if the page is in a shared state, you need to re-apply a new page and copy the written
236 //// page content for the writing process to use separately. The share was therefore cancelled.
237 //// The input parameter is a page table entry pointer.
228 void un\_wp\_page(unsigned long * table_entry)
229 {
230     unsigned long old_page, new_page;
231
232     //// First take the physical page location (address) in the given page table entry of the parameter
233     //// and determine whether the page is a shared page. If the original page address is greater
234     //// than the low end of the memory LOW_MEM (in the main memory area), and its value in the page
235     //// mapping byte array is 1 (indicating that the page is only referenced once, the page is not
236     //// shared), then the R/W flag (writable) is set in the page table entry, and the page translation
237     //// cache is refreshed and then returned. That is, if the memory page is only used by one process
238     //// at this time, and is not a process in the kernel, the attribute can be directly changed to
239     //// writable, and there is no need to re-apply a new page.
232     old_page = 0xffff000 & *table_entry;    //// get physical page address in the entry.
233     if (old_page >= LOW\_MEM && mem\_map[MAP\_NR(old_page)]==1) {
234         *table_entry |= 2;                //// set R/W flag.
235         invalidate();
236         return;
237     }
238     //// Otherwise, it is necessary to apply for a free page in the main memory area for the process
239     //// of performing the write operation and cancel the page sharing. If the original page position
240     //// is greater than the low end of the memory (meaning mem_map[] > 1, the page is shared), the
241     //// page byte array value of the original page is decremented by 1. Then update the contents
242     //// of the specified page table entry to the new page address, and set the read and write flags
243     //// (U/S, R/W, P). After refreshing the page transformation cache (TLB), the original page content
244     //// is finally copied to the new page.
238     if (!(new_page=get\_free\_page()))
239         oom();                                //// Out of Memory.
240     if (old_page >= LOW\_MEM)
241         mem\_map[MAP\_NR(old_page)]--;
242     copy\_page(old_page, new_page);
243     *table_entry = new_page | 7;
244     invalidate();
245 }
246
247 /*
248 * This routine handles present pages, when users try to write
249 * to a shared page. It is done by copying the page to a new address
250 * and decrementing the shared-page counter for the old page.

```

```

251  *
252  * If it's in code space we exit with a segment error.
253  */
    /// Perform write protection processing on the shared page.
    // This is the C function called during the handling of page exception and will be called in
    // the page.s program. The function parameters error_code and address are automatically
    // generated by the CPU when the process writes a write-protected page. The parameter error_code
    // indicates the type of error; address is the page linear address that generated the exception.
    // The page needs to be copied (copy-on-write) when writing a shared page.
254 void do_wp_page(unsigned long error_code, unsigned long address)
255 {
    // The function first determines in what range the linear address given by the CPU control
    // register CR2 that caused the page exception. If address is less than TASK_SIZE (0x4000000,
    // or 64MB), it means that the exception page location is within the linear address range of
    // the kernel or task 0 or task 1, then issued a warning message; If (address - current process
    // code start address) is greater than the size of one process (64MB), indicating that the linear
    // address is not within the space of the process causing the exception, exit after issuing
    // an error message.
256     if (address < TASK_SIZE)
257         printk("\n|BAD! KERNEL MEMORY WP-ERR!|n|r");
258     if (address - current->start_code > TASK_SIZE) {
259         printk("Bad things happen: page error in do_wp_page|n|r");
260         do_exit(SIGSEGV);
261     }
262 #if 0
263  /* we cannot do this yet: the stdio library writes to code space */
264  /* stupid, stupid. I really want the libc.a from GNU */
    // If the linear address is in the code space of the process, the execution program needs to
    // be terminated. Because the code is read-only.
265     if (CODE_SPACE(address))
266         do_exit(SIGSEGV);
267 #endif
    // Then we call the above function un_wp_page() to handle the cancel page protection. But first
    // we need to prepare the parameters for it. Its parameter is the entry pointer in the page
    // table for the given linear address "address", which is calculated as:
    // (1) "((address>>10) & 0xffc)": Calculates the offset address of the page table entry in the
    // specified linear address in the page table. Because according to the linear address structure,
    // "(address>>12)" is the index of the page table entry, but each entry occupies 4 bytes, so
    // after multiplying 4: "(address>>12)<<2 = (address>>10)&0xffc", you can get the offset address
    // of the page table entry in the table. The "AND" operation &0xffc is used to limit the address
    // range within one page. Also, since only 10 bits are shifted, the last 2 bits are the highest
    // 2 of the lower 12 bits of the linear address and should be masked out. Therefore, the more
    // obvious way to find the offset of the page table entry in the linear address is:
    // "(((address>>12) & 0x3ff)<<2)".
    // (2) "(0xfffff000 & *((address>>20) & 0xffc))": Used to get the address of the page table in
    // the directory entry. Where "((address>>20) & 0xffc)" is used to get the offset of the entry
    // index in the directory table in the linear address. Because "address>>22" is the directory
    // entry index and each item is 4 bytes, multiply by 4: "(address>>22)<<2 = (address>>20)",
    // this is the offset address of the specified entry in the directory table. "&0xffc" is used
    // to mask out the last 2 bits of the directory entry index value, and " *((address>>20) & 0xffc)"
    // gets the physical address of the corresponding page table in the contents of the specified
    // directory entry. Finally, "AND" with 0xfffff000 is used to mask some of the flag bits in
    // the contents of the page directory entry (lower 12 bits of the directory entry). This can

```



```

// be more intuitively represented as:
//      "(0xffffffff000 & *((unsigned long *) ((address>>22) & 0x3ff)<<2)))".
// (3) A pointer (physical address) of the page table entry can be obtained by the offset address
// in the page table of (1), plus the address of the corresponding page table in the content
// of the directory entry in (2). Then we use it here to copy the shared page.
268     un_wp_page((unsigned long *)
269                (((address>>10) & 0xffc) + (0xfffff000 &
270                *((unsigned long *) ((address>>20) & 0xffc)))));
271 }
272 }
273
//// Write page verification.
// If the page is not writable, copy the page. This function will be called by the memory-verified
// generic function verify_area() on line 34 in fork.c.
// The parameter 'address' is the linear address of the specified page.
274 void write_verify(unsigned long address)
275 {
276     unsigned long page;
277
// First, take the page directory entry corresponding to the specified linear address, and judge
// whether the page table corresponding to the directory entry exists according to the existing
// bit (P) in the directory entry (bit P=1?), if not (P=0), return . This is because there is
// no sharing and copy-on-write for pages that do not exist, and if the program performs a write
// operation on a page that does not exist, the system will execute do_no_page() due to a page
// fault exception , and use the put_page() function to map a physical page for this place.
// Then the code fetches the page table address from the directory entry, plus the page table
// entry offset value of the specified page in the page table, and obtains the page table entry
// pointer corresponding to the address. The physical page corresponding to the given linear
// address is included in the entry.
278     if (!(page = *((unsigned long *) ((address>>20) & 0xffc)) & 1))
279         return;
280     page &= 0xfffff000;
281     page += ((address>>10) & 0xffc);
// Then, the bit 1 (R/W) and bit 0 (P) flags in the page table entry are checked. If the page
// is not writable (R/W = 0) and exists, then the share checking and page copy operations
// (copy-on-write) are performed, otherwise do nothing and quit directly.
282     if ((3 & *((unsigned long *) page) == 1)      /* non-writable, present */
283         un_wp_page((unsigned long *) page);
284     return;
285 }
286
//// Get a free memory page and map it to the specified linear address.
// The function get_free_page() is just an application to get a page of physical memory in the
// main memory area. This function not only gets a page of physical memory, but also calls
// put_page() to map the physical page to the specified linear address.
// The parameter 'address' is the linear address of the specified page.
287 void get_empty_page(unsigned long address)
288 {
289     unsigned long tmp;
290
// If a free page cannot be obtained, or if the page taken cannot be placed at the specified
// address, a message indicating that the memory is insufficient is displayed. The meaning of
// the comment on line 292 is: the parameter 'tmp' of the free_page() function can be 0, it

```



```

// doesn't matter, the function will ignore it and return normally.
291     if (!(tmp=get_free_page()) || !put_page(tmp,address)) {
292         free_page(tmp);           /* 0 is ok - ignored */
293         oom();
294     }
295 }
296
297 /*
298  * try_to_share() checks the page at address "address" in the task "p",
299  * to see if it exists, and if it is clean. If so, share it with the current
300  * task.
301  *
302  * NOTE! This assumes we have checked that p != current, and that they
303  * share the same executable or library.
304  */
305
306 // Try to share the page at the given address of the current process with the given process.
307 // The current process has the same execution code as the process 'p'. It can also be considered
308 // that the current process is a child process generated by the 'p' process executing the fork()
309 // operation, so their code content is the same at the beginning. If the content of the data
310 // segment has not been modified, their content should be the same too.
311 // The parameter 'address' is the logical address in the process, which is the logical page
312 // address that the current process wants to share with the 'p' process, and p is the process
313 // that will be shared. If the page at the 'address' of the 'p' process exists and has not been
314 // modified, let the current process share it with the p process. At the same time, we also
315 // need to verify whether the specified address has been applied for and got a page, and if
316 // so, the kernel goes wrong and crash. Returns 1 if page sharing succeeded, 0 failed.
317
318 static int try_to_share(unsigned long address, struct task_struct * p)
319 {
320     unsigned long from;
321     unsigned long to;
322     unsigned long from_page;    // page directory entry of process 'p'
323     unsigned long to_page;      // page directory entry of current process.
324     unsigned long phys_addr;
325
326     // First, the page directory entry location corresponding to the logical address 'address' in
327     // the given process p and the current process is respectively obtained. For the convenience
328     // of calculation, we first find out the "logical" page directory entry at the given address
329     // 'address', that is, the page directory entry locations calculated in the process space
330     // (0 - 64MB). This logical entry location plus the page directory entries corresponding to
331     // the start addresses of process p and current process in the CPU 4G linear address space can
332     // obtain the actual page directory entry addresses 'from_page' and 'to_page' corresponding
333     // to the page at the address 'address' respectively.
334     from_page = to_page = ((address>>20) & 0xffc);
335     from_page += ((p->start_code>>20) & 0xffc);    // page dir entry of process p.
336     to_page += ((current->start_code>>20) & 0xffc); // page dir entry of current.
337
338     // After obtaining the directory entries corresponding to the two processes, they are processed
339     // separately. The following is the operation of the entry of the p process. The purpose is
340     // to obtain the physical page address corresponding to the 'address' in the p process, and
341     // determine whether the physical page exists and is clean (not modified, not dirty). The method
342     // is to first take the content of the directory entry, and then get the physical address of
343     // the page table in it, thereby calculating the page table entry pointer corresponding to the
344     // logical address 'address', and extracting the content of the page table entry and temporarily

```

```

// saved in phys_addr.
316 /* is there a page-directory at from? */
317     from = *(unsigned long *) from_page;           // get content of page dir entry.
318     if (!(from & 1))                                // page table not exist ?
319         return 0;
320     from &= 0xffff000;                               // page table address.
321     from_page = from + ((address>>10) & 0xffc);    // page table entry pointer.
322     phys_addr = *(unsigned long *) from_page;       // content of the entry.

// Then look at the physical page of the page table entry mapped, whether it exists and is clean.
// The value '0x41' corresponds to the D (Dirty) and P (Present) flags in the page table entry.
// Returns if the page is dirty or not present. Then we get the physical page address from the
// entry and save it back to phys_addr. Finally, we need to check the validity of this physical
// page address. It should not exceed the machine's maximum physical address value, nor should
// it be less than the low end of the memory (1MB).
323 /* is the page clean and present? */
324     if ((phys_addr & 0x41) != 0x01)
325         return 0;
326     phys_addr &= 0xffff000;                          // physical page address.
327     if (phys_addr >= HIGH MEMORY || phys_addr < LOW MEM)
328         return 0;

// Similarly, the following code operates on the entry of the current process. The purpose is
// to obtain the address of the page table entry corresponding to 'address' in the current
// process, and it is determined that the page table entry has not been mapped to any physical
// page, that is, its P=0. First we take the current process page directory entry content to
// 'to'. If the entry is invalid (P=0), that is, the page table corresponding to the directory
// entry does not exist, then a free page is requested to store the new page table. And update
// the contents of the directory entry 'to_page' to point to the new page.
329     to = *(unsigned long *) to_page;                 // content of the entry of current.
330     if (!(to & 1))
331         if (to = get\_free\_page\(\))
332             *(unsigned long *) to_page = to | 7;
333     else
334         oom\(\);

// Then we take the page table address in the directory entry to 'to', plus the offset address
// of the entry in the table, and get the page table entry address into 'to_page'. For this
// page table entry, if we check that the corresponding physical page already exists (P=1),
// it means that we want to share the corresponding physical page in process p, but now we have
// it. Then the kernel is wrong, crash.
335     to &= 0xffff000;                                // page table address.
336     to_page = to + ((address>>10) & 0xffc);          // page table entry pointer.
337     if (1 & *(unsigned long *) to_page)
338         panic("try_to_share: to_page already exists");

// After finding the clean and existing physical page corresponding to the address 'address'
// in process p, and also determining the page table entry address in the current process, we
// now start to share them. The method used is very simple, that is, first modify the page table
// entry of the p process, set its write protection (R/W=0, read only) flag, and then let the
// current process copy the entry of the p process. At this point, the page at the current process
// address 'address' is mapped to the physical page mapped at the logical address 'address'
// of the p process.
339 /* share them: write-protect */

```

```

340     *(unsigned long *) from_page &= ~2;
341     *(unsigned long *) to_page = *(unsigned long *) from_page;
// We then refresh the page transform cache, calculate the page number of the physical page
// being manipulated, and increment the reference in the map byte array entry by one. Finally,
// return 1 if the sharing process is successful.
342     invalidate();
343     phys_addr -= LOW_MEM;
344     phys_addr >>= 12; // obtain the page number in main mem area.
345     mem_map[phys_addr]++;
346     return 1;
347 }
348
349 /*
350  * share_page() tries to find a process that could share a page with
351  * the current one. Address is the address of the wanted page relative
352  * to the current data space.
353  *
354  * We first check if it is at all feasible by checking executable->i_count.
355  * It should be >1 if there are other tasks sharing this inode.
356  */
// Look for the process running the same executable and try to share the page with it.
// When a page fault exception occurs, first check to see if you can share the page with other
// processes running the same executable file. The function first checks if there is another
// process in the system that is also running the same execution file as the current process.
// If so, look for such a task in all current tasks in the system. If we find such a task, try
// to share the page at the given address with it. If no other tasks in the system are running
// the same executable file as the current process, the preconditions for the shared page
// operation do not exist, so the function exits immediately.
// The way to determine if another process is executing the same executable file in the system
// is to use the executable field (or library field) in the process task structure, which points
// to the executable file's i-node in memory. We can make this judgment based on the reference
// number field i_count of the i-node. If the i_count of the node is greater than 1, it means
// that two or more processes in the system are running the same executable file, so we can
// compare executable field of all tasks in the task structure array to see if there is the
// processes running the same executable file.
// The parameter 'inode' is the i-node of the executable file of the process that wants to share
// the page. 'address' is the logical address of the page that the current process wants to
// share with a process. Returns 1 if the sharing operation is successful, 0 if it fails.
357 static int share_page(struct m_inode * inode, unsigned long address)
358 {
359     struct task_struct ** p;
360
// First check the memory i-node reference count value given by the parameter. If the i-node
// reference count value is equal to 1 (executable->i_count = 1) or the i-node pointer is empty,
// it means that only one process in the current system is running the execution file, or the
// provided i-node is invalid. So there is no sharing at all, just exit the function directly.
361     if (inode->i_count < 2 || !inode)
362         return 0;
// Otherwise, search all tasks in the task array, find the process that can share the page with
// the current process, that is, another process running the same executable file, and try to
// share the page with the specified address. If the process logical address 'address' is smaller
// than the start address LIBRARY_OFFSET of the process library file, it indicates that the
// shared page is within the logical address space corresponding to the process execution file.

```

```

// Then check if the given i-node is the same as the executable file i-node of the process
// (that is, the executable field of the process). If it is not the same, continue to search.
// If the logical address 'address' is greater than or equal to the start address LIBRARY_OFFSET
// of the process library file, it indicates that the page to be shared is in the library file.
// So we check if the specified 'inode' is the same as the library file i-node of the process,
// if it is not the same, continue to search.
// If a process 'p' is found whose 'executable' or 'library' field is the same as the specified
// 'inode', the function try_to_share() is called to try page sharing. The function returns
// 1 if the sharing operation is successful, otherwise it returns 0.
363     for (p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
364         if (!*p) // continue search if the item is null.
365             continue;
366         if (current == *p) // continue search if it's the current.
367             continue;
368         if (address < LIBRARY_OFFSET) {
369             if (inode != (*p)->executable) // i-node of the executable file.
370                 continue;
371         } else {
372             if (inode != (*p)->library) // i-node of the library file.
373                 continue;
374         }
375         if (try_to_share(address,*p))
376             return 1;
377     }
378     return 0;
379 }
380
//// Page-not-present processing.
// This function is called during page exception interrupt processing and is called in the page.s
// program. The parameters 'error_code' and 'address' are automatically generated by the CPU
// when the process accesses the page due to a page-not-present fault. 'error_code' indicates
// the type of error; 'address' is the page linear address that generated the exception.
// The function first checks to see if the missing page is in the swap device, and if so, swaps
// in. Otherwise, try to share the page with the same file that has already been loaded, or
// just map a page of physical memory pages simply because the process dynamically requests
// the memory page. If the sharing operation is unsuccessful, the missing data page can only
// be read from the corresponding file to the specified linear address.
381 void do_no_page(unsigned long error_code,unsigned long address)
382 {
383     int nr[4];
384     unsigned long tmp;
385     unsigned long page;
386     int block,i;
387     struct m_inode * inode;
388
// The function first determines in what range the linear address given by the CPU control
// register CR2 that caused the page exception. If address is less than TASK_SIZE (0x4000000,
// or 64MB), it means that the exception page location is within the linear address range of
// the kernel or task 0 or task 1, then issued a warning message; If (address - current process
// code start address) is greater than the size of one process (64MB), indicating that the linear
// address is not within the space of the process causing the exception, exit after issuing
// an error message.
389     if (address < TASK_SIZE)

```

```

390         printk("\n\rBAD!! KERNEL PAGE MISSING\n\r");
391     if (address - current->start_code > TASK\_SIZE) {
392         printk("Bad things happen: nonexistent page error in do_no_page\n\r");
393         do\_exit(SIGSEGV);
394     }
    // Then, according to the specified linear address 'address', the corresponding secondary page
    // table entry pointer is obtained, and according to the content of the entry, it is determined
    // whether the page at the 'address' is in the swap device. If yes, swap in the page and exit.
    // The method is to first obtain the content of the page directory item corresponding to the
    // specified linear address 'address', and then take out the address of the page table therein,
    // and add the page table entry offset to obtain the corresponding page table entry pointer,
    // thereby obtaining the content of the page table entry. If the content of the entry is not
    // 0, but bit P=0, it indicates that the physical page specified by the page table entry should
    // be in the swap device. The function then exits after the specified page is loaded from the
    // swap device.
395     page = *(unsigned long *) ((address >> 20) & 0xffc); // content of page dir entry.
396     if (page & 1) {
397         page &= 0xffff000; // page table address.
398         page += (address >> 10) & 0xffc; // page table entry pointer.
399         tmp = *(unsigned long *) page; // content of page entry.
400         if (tmp && !(1 & tmp)) {
401             swap\_in((unsigned long *) page); // read in from swap device.
402             return;
403         }
404     }
    // Otherwise, we take the page address at the given linear address 'address' and calculate the
    // offset 'tmp' of the address in the process space relative to the process base address, ie
    // the corresponding logical address. Thus we can calculate the specific starting block number
    // of the missing page in the execution file image or in the library file.
405     address &= 0xffff000; // page address
406     tmp = address - current->start_code; // logical address of the page.

    // If the logical address 'tmp' is greater than the starting position of the library image file
    // in the process logical space, the missing page is in the library image file. Therefore, the
    // i-node 'library' of the library image file can be obtained from the current process task
    // structure, and the starting block number 'block' of the missing page in the library file
    // is calculated. If the logical address 'tmp' is smaller than the end of the execution image
    // file of the process, then the missing page is in the process execution file image, so the
    // i-node number 'executable' of the execution file can be obtained from the current process
    // task structure, and calculate the starting block number 'block' of the missing page in the
    // execution file image. If the logical address 'tmp' is neither in the address range of the
    // execution file nor in the library file space, the page not present is caused by the process
    // accessing the dynamically requested memory page data, so there is no corresponding i-node
    // and data block number (both set to be NULL).
    // Since the first block of the image file stored on the block device is the program header
    // structure, the first block of data needs to be skipped when the file is read. Because each
    // block of data has a length of BLOCK_SIZE = 1KB, one page of memory can store 4 blocks of
    // data. The process logical address 'tmp' is divided by the data block size plus one to get
    // the starting block number 'block' of the missing page in the execution image file.
407     if (tmp >= LIBRARY\_OFFSET) {
408         inode = current->library; // inode & block no. of library file.
409         block = 1 + (tmp - LIBRARY\_OFFSET) / BLOCK\_SIZE;
410     } else if (tmp < current->end_data) {

```

```

411         inode = current->executable;           // inode & block no of executable file.
412         block = 1 + tmp / BLOCK\_SIZE;
413     } else {
414         inode = NULL;                           // for dynamically applied page.
415         block = 0;
416     }
    // If the process accesses the page of its dynamic application or the page fault exception caused
    // by storing the stack information, then we directly apply for a page of physical memory page
    // and map it to the linear address 'address'. Otherwise, the missing page is within the scope
    // of the process execution file or library file, so try to share the page operation, and exit
    // if successful. If it is unsuccessful, you need to apply for a page of physical memory page,
    // then read the corresponding page in the execution file from the device and place (map) it
    // to the process page logical address 'tmp'.
417     if (!inode) {                               // it's a dynamic applied page.
418         get\_empty\_page(address);
419         return;
420     }
421     if (share\_page(inode,tmp))                   // try to share the page at 'tmp'.
422         return;
423     if (!(page = get\_free\_page()))               // apply for a free page.
424         oom();
425 /* remember that 1 block is used for header */
    // According to this block number and the i-node of the execution file, we can find the device
    // logic block number (stored in nr[] array) in the corresponding block device from the mapping
    // bitmap, and use the bread\_page() to read the 4 logical blocks into the physical page.
426     for (i=0 ; i<4 ; block++,i++)
427         nr[i] = bmap(inode,block);
428     bread\_page(page, inode->i_dev, nr);

    // When reading a device logic block, there may be a situation where the read page position
    // may be less than 1 page long from the end of the file. Therefore, some useless information
    // may be read. The following operation is to clear this part of the part beyond the execution
    // file 'end_data'. Of course, if the page is more than 1 page from the end, it is not read
    // from the file in the executable file image, but is read from the library file, so there is
    // no need to perform the clear operation.
429     i = tmp + 4096 - current->end_data;           // the excess byte length.
430     if (i>4095)                                   // more than 1 page from the end.
431         i = 0;
432     tmp = page + 4096;                             // points to the end of the page.
433     while (i-- > 0) {                               // i bytes cleared start from page end.
434         tmp--;
435         *(char *)tmp = 0;
436     }
    // Finally, a page causing the page fault exception is mapped to the specified linear address
    // address. If the operation is successful, it will return, otherwise the memory page will be
    // released, indicating that the memory is not enough.
437     if (put\_page(page,address))
438         return;
439     free\_page(page);
440     oom();
441 }
442
    //// Memory management initialization.

```

```

// This function initializes the physical memory area above the 1MB address. The kernel manages
// and accesses memory in pages, with a page length of 4KB. This function divides all physical
// memory above 1MB into pages and manages these pages using a page-mapped byte array
// mem_map[]. For machines with 16MB of memory, the array has 3840 items ((16MB - 1MB) / 4KB)
// and can manage 3840 physical pages. Whenever a memory page is occupied, the corresponding
// byte item in mem_map[] is incremented by one; if a page is released, the corresponding byte
// value is decremented by one. If the byte item is 0, it indicates that the corresponding page
// is idle; if the byte value is greater than or equal to 1, it indicates that the page is occupied
// or shared by multiple processes.
// Since the kernel buffer cache and some devices need to use a certain amount of memory, the
// amount of memory that the system actually allocates for use is reduced. We refer to the memory
// area that can be actually allocated for use by the kernel as the "main memory area" (MMA),
// and its start position is represented by the variable 'start_mem', and the end address is
// represented by 'end_mem'. For a PC system with 16 MB of memory, 'start_mem' is typically
// 4 MB and 'end_mem' is 16 MB. Therefore, the main memory area is [4MB-16MB] at this time,
// and a total of 3072 physical pages are available for allocation. The range 0 - 1MB memory
// area is reserved for the kernel.
// The parameter 'start_mem' is the starting address of the main memory area that can be used
// for page allocation (the memory space occupied by RAMDISK has been removed). 'end_mem' is
// the actual physical memory maximum address. The address range [start_mem, end_mem] is the
// main memory area.
443 void mem_init(long start_mem, long end_mem)
444 {
445     int i;
446
// The function first sets the memory mapped byte array items corresponding to all pages in
// the range of 1MB to 16MB to the occupied state, that is, all bytes are set to USED (100).
// PAGING_PAGES is defined as (PAGING_MEMORY>>12), which is the number of all physical memory
// pages above 1MB (15MB/4KB = 3840).
447     HIGH_MEMORY = end_mem; // set the memory top (16MB).
448     for (i=0 ; i<PAGING_PAGES ; i++) // PAGING_PAGES = 3840.
449         mem_map[i] = USED;
// Then find the item number 'i' in the byte array corresponding to the page at the start address
// 'start_mem', and calculate the number of pages in the main memory area. At this point, the
// i-th item of the mem_map[] corresponds to the first page in the main memory area. Finally,
// the array item corresponding to the pages in the main memory area are cleared (indicating
// idle). For systems with 16MB of physical memory, the bytes corresponding 4MB - 16MB main
// memory area in mem_map[] is cleared.
450     i = MAP_NR(start_mem); // page number at the beginning of the MMA.
451     end_mem -= start_mem;
452     end_mem >>= 12; // total number of pages in MMA.
453     while (end_mem-->0)
454         mem_map[i++]=0; // bytes of MMA pages are reset.
455 }
456
// Display system memory information.
// The number of memory pages used in the system and the total number of physical memory pages
// in the main memory area are counted according to the information in the mem_map[], and the
// contents of page directory and page table. This function is called on line 186 of the
// chr_drv/keyboard.S program, which displays system memory statistics when the "Shift + Scroll
// Lock" key is pressed.
457 void show_mem(void)
458 {

```



```

459     int i, j, k, free=0, total=0;
460     int shared=0;
461     unsigned long * pg_tbl;
462
    // First, according to the byte array mem_map[], we count the total number of pages in the main
    // memory area 'total', and the number of free pages 'free' and the number of shared pages
    // 'shared', and display these information.
463     printk("Mem-info: \n\r");
464     for(i=0 ; i<PAGING_PAGES ; i++) {
465         if (mem_map[i] == USED)                // pages not for allocation.
466             continue;
467         total++;
468         if (!mem_map[i])
469             free++;                            // free pages in the main mem ares.
470         else
471             shared += mem_map[i]-1;            // shared pages (byte value > 1).
472     }
473     printk("%d free pages of %d\n\r", free, total);
474     printk("%d pages shared\n\r", shared);

    // Then we count the number of paging management logic pages of the CPU. The first four items
    // in the page directory table are used by the kernel code and are not listed as statistical
    // ranges. The method is to loop through all page directory entries starting with item 5. If
    // the corresponding secondary page table exists, the memory page occupied by the secondary
    // page table itself is first counted (line 484), and then the physical memory page corresponding
    // to all page entries in the page table is counted.
475     k = 0;                                    // statistics of pages occupied by a process.
476     for(i=4 ; i<1024 ;) {
477         if (1&pg_dir[i]) {
    // (If the address of the corresponding page table in the page directory entry is greater than
    // the highest physical memory address of the machine, HIGH_MEMORY, there is a problem with
    // the directory entry. The directory entry information is displayed and the next directory
    // entry continues to be processed.)
478             if (pg_dir[i]>HIGH_MEMORY) {        // content is abnormal.
479                 printk("page directory[%d]: %08X\n\r",
480                     i, pg_dir[i]);
481                 continue;                        // needs "i++;" before it.
482             }
    // If the "address" of the page table in the page directory entry is greater than LOW_MEM (that
    // is, 1MB), the physical memory page statistics value 'k' of one process is incremented by
    // one, and the statistics value 'free' of all physical memory pages occupied by the system
    // is incremented by one. Then take the corresponding page table address 'pg_tbl' and count
    // the contents of all page table items in the page table. If the physical page indicated by
    // the current page table entry exists and the physical page "address" is greater than LOW_MEM,
    // then the corresponding page of the page table entry is included in the statistical value.
483             if (pg_dir[i]>LOW_MEM)
484                 free++, k++;                    // pages occupied in the page table.
485             pg_tbl=(unsigned long *) (0xfffff000 & pg_dir[i]);
486             for(j=0 ; j<1024 ; j++)
487                 if ((pg_tbl[j]&1) && pg_tbl[j]>LOW_MEM)
    // (If the physical page address is greater than the highest physical memory address of the
    // machine, HIGH_MEMORY, it indicates that there is a problem with the content of the page table
    // item, so the content of the page table item is displayed. Otherwise, the corresponding page

```



```
// of the page table item is included in the statistical value.)
488         if (pg_tbl[j]>HIGH MEMORY)
489             printk("page_dir[%d][%d]: %08X\n\r",
490                 i, j, pg_tbl[j]);
491         else
492             k++, free++; // pages of page table items.
493     }
// Since the linear space size of each task is 64MB, one task occupies 16 page directory entries.
// Therefore, every 16 directory entries are counted here, the page table occupied by the process
// task structure is counted. If k=0 at this time, it means that the process corresponding to
// the current 16 directory entries does not exist in the system (not created or terminated).
// Then, after the corresponding process number and the physical page statistics value k are
// displayed, k is cleared to count the pages occupied by the next process.
494     i++;
495     if (!(i&15) && k) { // k !=0 indicates that the process exists.
496         k++, free++; // one page/process for task_struct */
497         printk("Process %d: %d pages\n\r", (i>>4)-1, k);
498         k = 0;
499     }
500 }
// Finally, the memory page being used in the system and the total number of pages in the main
// memory area are displayed.
501     printk("Memory found: %d (%d) \n\r", free-shared, total);
502 }
503
```

13.3 page.s

13.3.1 Function

The page.s file includes page fault exception interrupt handler (interrupt 14), which is handled primarily in two cases. First, due to page fault exceptions caused by page not present, this needs to be handled by calling `do_no_page(error_code, address)`; second, page exceptions caused by page write protection. At this point, the page write protection handler `do_wp_page(error_code, address)` is called for processing. The error code (error_code) in the function parameter is automatically generated by the CPU and pushed onto the stack. The linear address (address) accessed when an exception occurs is taken from the control register CR2. CR2 is specifically used to store linear addresses when a page fails.

13.3.2 Code annotation

Program 13-2 linux/mm/page.s

```
1 /*
2  * linux/mm/page.s
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
```

```

8  * page.s contains the low-level page-exception code.
9  * the real work is done in mm.c           // memory.c
10 */
11
12 // This variable will be used in kernel/traps.c to set the page exception descriptor.
13 .globl _page_fault           # declared as a global variable.
14 _page_fault:
15     xchgl %eax, (%esp)      # take the error code to EAX.
16     pushl %ecx
17     pushl %edx
18     push %ds
19     push %es
20     push %fs
21     movl $0x10, %edx       # set the kernel data segment selector.
22     mov %dx, %ds
23     mov %dx, %es
24     mov %dx, %fs
25     movl %cr2, %edx        # get the linear address that caused the page exception.
26     pushl %edx             # the linear address and error code are pushed onto the stack
27     pushl %eax             # as arguments to the function to be called.
28 // Test the "page present" flag P (bit 0), and call the do_no_page() function if it is an exception
29 // caused by a page fault. Otherwise, the page write protection function do_wp_page() is called.
30     testl $1, %eax         # Test flag P (bit 0), jump if it is set.
31     jne 1f
32     call _do_no_page       # mm/memory.c, line 381.
33     jmp 2f
34 1:     call _do_wp_page     # mm/memory.c, line 254.
35 // Discard the two arguments pushed onto the stack, pop the registers and exit the interrupt.
36 2:     addl $8, %esp
37     pop %fs
38     pop %es
39     pop %ds
40     popl %edx
41     popl %ecx
42     popl %eax
43     iret

```

13.4 swap.c

13.4.1 Function

Starting with version 0.12, Linux has added virtual memory swapping functionality to the kernel. This function is mainly implemented by this program. When the physical memory capacity is limited and the usage is tight, the program saves the temporarily unused memory page contents to the disk (switching device) to free up memory space for the programs that are in urgent need. If we later need to use the memory page content already stored on the swap device again in the future, the program is responsible for taking them back and putting them back into memory. Memory swap management uses a mapping technique similar to that of main memory area management, using bit maps to determine the specific save location and map location of the

swapped memory pages.

When compiling the kernel, if we defined the swap device number SWAP_DEV, then the compiled kernel has the memory swap functionality. For Linux 0.12, the switching device uses a designated independent partition on the hard disk that does not contain a file system. When the swapping program is initialized, it first reads page 0 on the swap device. This page is the swap area management page, which contains the bitmap used by the swapping page management. The 10 characters starting from the 4068th byte are the swap device feature string "SWAP-SPACE". If the feature string is not on the partition, the given partition is not a valid swap device.

The swap.c program mainly includes swap mapping bitmap management functions and swap device access functions. The `get_swap_page()` and `swap_free()` functions are respectively used to apply a swap page and release the specified page in the swap device based on the swapping bitmap; the `swap_out()` and `swap_in()` functions are respectively used to output/input the memory page information to/from the swap device. The latter two functions use the `read_swap_page()` and `write_swap_page()` functions to access the specified swap device. These two functions are defined in the `include/linux/mm.h` header file in the form of macros:

```
#define read_swap_page(nr, buffer)    ll_rw_page(READ, SWAP_DEV, (nr), (buffer));
#define write_swap_page(nr, buffer)   ll_rw_page(WRITE, SWAP_DEV, (nr), (buffer));
```

The `ll_rw_page()` function is a low-level page read and write function of the block device. The code is implemented in the `kernel/blk_drv/ll_rw_blk.c` file. It can be seen that the swap device access functions are essentially the device page access functions that specifies the device number.

During the kernel initialization process, if the system defines the swap device number SWA_DEV, the kernel executes the swap processing initialization function `init_swapping()`. This function first checks if the system does have a swap device based on the array of blocks in the system, and if the swap partition of the device is valid. It then requests a memory page and reads the first swap management page (page 0) on the swap partition into the memory page. Page 0 stores the swap page bitmap mapping information, where each bit represents an swap page. If a bit is 0, it indicates that the swap page on the corresponding device has been used (occupied) or unavailable; if the bit is 1, indicating that the corresponding swap page is available. Since a page has a total of SWAP_BITS ($4096 \times 8 = 32768$) bits, the swap partition can manage up to 32,768 pages.

In a running Linux, if a block device partitioner (such as `fdisk`) initializes the swap partition to have 'swap_size' swap pages, the first page (page 0) of the swap partition on the device will be used for swap management (occupied), so the first bit in the swap bitmap should also be zero. Therefore, the number of swap pages actually available on the device is 'swap_size - 1', and their page number ranges are `[1 -- swap_size-1]`, so their corresponding bits in the bitmap are all 1s (idle). Bits in the range of `[swap_size -- SWAP_BITS]` in the bitmap are initialized to an unavailable state (all 0s) because there is no corresponding swap page on the device. Therefore, in the initial normal case, the bitmap bit status should be as shown below (the shaded portion is not available for swapping):

bits in the bitmap:	0, 1, 1, 1, ... , 1,	0, ..., 0.
swap page number :	0, 1, 2, 3, ... , swap_size-1,	swap_size, ..., SWAP_BITS-1.

13.4.2 Code annotation

Program 13-3 linux/mm/swap.c

```

1  /*
2   *  linux/mm/swap. c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  /*
8   *  This file should contain most things doing the swapping from/to disk.
9   *  Started 18. 12. 91
10  */
11
12 // <string.h> String header file. Defines some embedded functions about string operations.
13 // <linux/mm.h> Memory management header file. Contains page size definitions and some page
14 //   release function prototypes.
15 // <linux/sched.h> The scheduler header file defines the task structure task_struct, the data
16 //   of the initial task 0, and some embedded assembly function macro statements about the
17 //   descriptor parameter settings and acquisition.
18 // <linux/head.h> Head header file. A simple structure for the segment descriptor is defined,
19 //   along with several selector constants.
20 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
21 //   used functions of the kernel.
22 #include <string.h>
23
24 #include <linux/mm.h>
25 #include <linux/sched.h>
26 #include <linux/head.h>
27 #include <linux/kernel.h>
28
29 // Each byte has 8 bits, so a page (4096 bytes) has a total of 32768 bits. If one bit corresponds
30 // to one page of memory, the bitmap of one page can manage up to 32,768 pages, corresponding
31 // to 128 MB of memory capacity. If the bit is set, it indicates that the corresponding swap
32 // page is idle.
33 #define SWAP_BITS (4096<<3)           // define swap total bits in a page (32768).
34
35 // bitop() is a bit manipulation macro. By giving a different "op", we can define three
36 // operations for testing, setting, or clearing specified bit.
37 // The parameter 'addr' of the inline assembly function specifies the linear address; 'nr' is
38 // the bit offset at the specified address. The macro places the value of the 'nr'th bit at
39 // the given address 'addr' into the carry flag, sets or resets the bit and returns the carry
40 // flag value (ie, returns the original bit value).
41 // The first instruction on line 25 combines with the "op" to form different instructions:
42 // "op" = "", the instruction bt - Bit test, and sets the carry with the original value.
43 // "op" = "s", the instuction bts - Bit test and set, and the original bit is sets to the carry.
44 // "op" = "r", instruction btr - Bit test and reset, and the original bit is set to the carry.
45 // Input: %0 - (return value), %1 - bit offset (nr); %2 - base address (addr); %3 - plus
46 // operation register initial value (0).
47 // The inline assembly code saves the bit specified by the base address (%2) and the bit offset
48 // (%1) to the carry flag CF, and then sets (resets) the bit. The instruction ADCL is loaded
49 // with a carry bit to set the operand (%0) according to the carry bit CF. If CF = 1, the return

```

```

// register value = 1, otherwise the return register value = 0.
21 #define bitop(name, op) \
22 static inline int name(char * addr, unsigned int nr) \
23 { \
24 int __res; \
25 __asm__ __volatile__ ("bt" op " %1, %2; adcl $0, %0" \
26 : "=g" (__res) \
27 : "r" (nr), "m" (*(addr)), "0" (0)); \
28 return __res; \
29 }
30
// Here we define 3 inline functions according to different op characters.
31 bitop(bit, "") // define bit(char * addr, unsigned int nr)
32 bitop(setbit, "s") // define setbit(char * addr, unsigned int nr)
33 bitop(clrbit, "r") // define clrbit(char * addr, unsigned int nr)
34
35 static char * swap_bitmap = NULL;
36 int SWAP_DEV = 0; // The swap device number set when the kernel is initialized.
37
38 /*
39  * We never page the pages in task[0] - kernel memory.
40  * We page all other pages.
41  */
// The first virtual memory page, the virtual memory page starting at the end of task 0 (64MB).
42 #define FIRST_VM_PAGE (TASK_SIZE>>12) // = 64MB/4KB = 16384
43 #define LAST_VM_PAGE (1024*1024) // = 4GB/4KB = 1048576
44 #define VM_PAGES (LAST_VM_PAGE - FIRST_VM_PAGE) // = 1032192 (count from 0).
45
//// Apply and get a swap page number.
// Scans the entire swapping bitmap (except for bit 0 of the bitmap page itself), resets the
// first found bit, and returns its position, which is the current free swap page number. Returns
// the swap page number if the operation is successful, otherwise returns 0.
46 static int get_swap_page(void)
47 {
48     int nr;
49
50     if (!swap_bitmap)
51         return 0;
52     for (nr = 1; nr < 32768 ; nr++)
53         if (clrbit(swap_bitmap, nr))
54             return nr; // return the current free swap page number.
55     return 0;
56 }
57
// Release the sawp page specified in the swap device.
// Set the bit in the swap bitmap corresponding to the specified page number by the parameter.
// In a swap bitmap, if a bit is 1, it means that the corresponding swap page is idle. Therefore,
// if the original bit is equal to 1, it means that the original page of the swap device is
// not occupied, or the bitmap is in error. Then an error message is displayed and returned.
58 void swap_free(int swap_nr)
59 {
60     if (!swap_nr)
61         return;

```

```

62     if (swap\_bitmap && swap_nr < SWAP\_BITS)
63         if (!setbit(swap\_bitmap, swap_nr))
64             return;
65     printk("Swap-space bad (swap\_free\(\))\n|r");
66     return;
67 }
68
// Swap the specified page into memory.
// The page of the specified page table entry is read from the swap device into the newly
// requested memory page. At the same time, modify the corresponding bit in the swap bitmap
// (set), and modify the contents of the page table entry, let it point to the memory page,
// and set the corresponding flag.
69 void swap\_in(unsigned long *table_ptr)
70 {
71     int swap_nr;
72     unsigned long page;
73
// The function first checks the validity of the swap bitmap and parameters. If the swap bitmap
// does not exist, or the page corresponding to the specified page table entry already exists
// in the memory, or the swapping page number is 0, a warning message is displayed and exits.
// For the memory page that has been placed in the swap device, the corresponding page table
// entry should be the swap page number*2, ie (swap_nr << 1). See the description of line 111
// in the function try_to_swap_out() below.
74     if (!swap\_bitmap) {
75         printk("Trying to swap in without swap bit-map");
76         return;
77     }
78     if (1 & *table_ptr) {
79         printk("trying to swap in present page\n|r");
80         return;
81     }
82     swap_nr = *table_ptr >> 1;
83     if (!swap_nr) {
84         printk("No swap page in swap_in\n|r");
85         return;
86     }
// Then apply for a memory page and read the page with the page number swap_nr from the swap
// device. After the page is swapped in using the read_swap_page(), the corresponding bit in
// the swap bitmap is set. If it is originally set, it means that the same page is read again
// from the switching device, so the warning message is displayed. Finally, let the page table
// entry point to the physical page, and set the page modified, user readable and writable and
// presence flags (Dirty, U/S, R/W, P).
87     if (!(page = get\_free\_page()))
88         oom();
89     read\_swap\_page(swap_nr, (char *) page);    // defined in file nclude/linux/mm.h
90     if (setbit(swap\_bitmap, swap_nr))
91         printk("swapping in multiply from same page\n|r");
92     *table_ptr = page | (PAGE\_DIRTY | 7);
93 }
94
// Try to swap out the page.
// If the memory page has not been modified, it does not need to be saved in the swap device,
// because the corresponding page can also be read directly from the corresponding image file.

```

```

// So you can directly release the corresponding physical page. Otherwise, apply for an swap
// page number and then swap out the page. At this time, the swap page number is to be saved
// in the corresponding page table entry, and it is still necessary to keep the page table entry
// bit P = 0. The parameter 'table_ptr' is a pointer to a page table entry. Returns 1 if the
// page is swapped or released successfully, otherwise returns 0.
95 int try\_to\_swap\_out(unsigned long * table_ptr)
96 {
97     unsigned long page;
98     unsigned long swap_nr;
99
// The function first determines the validity of the parameter. If the memory page that needs
// to be swapped out does not exist (or is invalid), the code exits; If the physical page address
// specified by the page table entry is greater than the memory-managed high-end PAGING_MEMORY
// (15MB), it also exits.
100     page = *table_ptr;
101     if (!(PAGE\_PRESENT & page))
102         return 0;
103     if (page - LOW\_MEM > PAGING\_MEMORY)
104         return 0;
// If the memory page has been modified, but the page is shared, then in order to improve the
// efficiency of the operation, such pages should not be swapped out, so the function returns
// 0 and exists. Otherwise we get a swap page number and save it in the page table entry, then
// swap the page out and release the corresponding physical memory page.
105     if (PAGE\_DIRTY & page) {
106         page &= 0xffff000; // get physical page address.
107         if (mem\_map[MAP\_NR(page)] != 1)
108             return 0;
109         if (!(swap_nr = get\_swap\_page())) // get a swap page number.
110             return 0;
// For pages to be swapped, the swap page number (swap_nr << 1) is stored in the corresponding
// page table entry. The reason of multiplying 2 is to free the presence bit (P, bit 0) of the
// page table entry. Only pages with a bit P=0 and a page table entry content other than 0 will
// be in the swap device. The Intel manual clearly states that when a table entry has a
// P = 0 (invalid page), all other bits (bits 31 - 1) are free to use. The write page function
// write_swap_page(nr, buffer) is defined as ll_rw_page(WRITE, SWAP_DEV, (nr), (buffer)), see
// line 12 of the linux/mm.h file.
111         *table_ptr = swap_nr<<1;
112         invalidate(); // refresh the CPU transform cache.
113         write\_swap\_page(swap_nr, (char *) page);
114         free\_page(page);
115         return 1;
116     }
// Otherwise, it means that the page has not been modified, so you don't have to swap it out,
// but you can release it directly.
117     *table_ptr = 0;
118     invalidate();
119     free\_page(page);
120     return 1;
121 }
122
123 /*
124  * Ok, this has a rather intricate logic - the idea is to make good
125  * and fast machine code. If we didn't worry about that, things would

```

```

126  * be easier.
127  */
    // Put the memory page in the swap device.
    // Starting from the page directory entry (FIRST_VM_PAGE>>10) corresponding to the linear
    // address 64MB, the entire 4GB linear space is searched. During this time we tried to swap
    // the corresponding memory page to the swap device. Returns 1 if the page is successfully swapped
    // out, otherwise returns 0. The two static local variables are used to temporarily stored the
    // current search point for the beginning of the next searching. This function will be called
    // in get_free_page() (defined at line 172).
128 int swap_out(void)
129 {
130     static int dir_entry = FIRST_VM_PAGE>>10; // 16, first directory entry of task 1.
131     static int page_entry = -1;
132     int counter = VM_PAGES; // 1032192, see line 44.
133     int pg_table;
134
    // First we loop through the page directory table to find the page directory entry pg_table
    // containing the valid secondary page table, and exit the loop if found. , Otherwise, we adjust
    // the number of remaining secondary page tables 'counter' of the directory entry, and then
    // continue to check the next directory entry. If the appropriate (existing) page directory
    // entry has not been found after all searches, the code returns with 0.
135     while (counter>0) {
136         pg_table = pg_dir[dir_entry]; // content of the directory entry.
137         if (pg_table & 1) // exit the loop if the table exists.
138             break;
139         counter -= 1024; // One page table has 1024 items.
140         dir_entry++; // next directory entry.
141         if (dir_entry >= 1024)
142             dir_entry = FIRST_VM_PAGE>>10;
143     }
    // After getting the page table pointer in the current directory entry, the swap function
    // try_to_swap_out() is called one by one for all 1024 pages in the page table to try to swap
    // out it. Returns 1 if a page is successfully swapped out to the swap device. If all page tables
    // for all directory entries have failed, the warning message is displayed and returns 0.
144     pg_table &= 0xfffff000; // page table pointer (address).
145     while (counter-- > 0) {
146         page_entry++; // Page table entry (initially -1).
    // If we have tried to process all the items in the current page table, but still can not
    // successfully find a page that can be swapped out, that is, the page table item index number
    // is greater than or equal to 1024, then use the same way as the previous 135 - 143 lines To
    // select the secondary page table in the next page directory entry.
147         if (page_entry >= 1024) {
148             page_entry = 0;
149             repeat:
150                 dir_entry++;
151                 if (dir_entry >= 1024)
152                     dir_entry = FIRST_VM_PAGE>>10;
153                 pg_table = pg_dir[dir_entry]; // content of page directory entry.
154                 if (!(pg_table&1))
155                     if ((counter -= 1024) > 0)
156                         goto repeat;
157                 else
158                     break;

```



```

159         pg_table &= 0xffff000;           // get page table pointer.
160     }
161     if (try to swap out(page_entry + (unsigned long *) pg_table))
162         return 1;
163 }
164 printk("Out of swap-memory\n\r");
165 return 0;
166 }
167
168 /*
169  * Get physical address of first (actually last :- ) free page, and mark it
170  * used. If no free pages left, return 0.
171  */
172
173     /// Get a free physical page in the main memory area.
174     // If there is no physical memory page available, then we perform page swap processing and then
175     // apply for and try to get the page again.
176     // Input: %1(ax = 0) ; %2(LOW_MEM) Memory start position managed by byte bitmap;
177     // %3(cx = PAGING_PAGES); %4(edi = mem_map + PAGING_PAGES - 1).
178     // The function returns the address of the new page in %0(ax = physical page start address).
179     // The above %4 register actually points to the last byte of the memory byte bitmap mem_map[]. This
180     // function scans all page flags backwards from the end of the bitmap (the total number of pages
181     // is PAGING_PAGES), and returns the page address if the page is free (bitmap byte is 0).
182     // NOTE! This function simply points out a free physical page in the main memory area, but it
183     // is not mapped to the address space of any process. The put_page() function in the memory.c
184     // program is used to map a specified page into the address space of a process. Of course, using
185     // this function for the kernel does not require the use of put_page() for mapping because the
186     // kernel code and data space (16MB) are mapped to the physical address space peer-to-peer.
187     // Line 174 defines a local register variable. This variable will be saved in the eax register
188     // for efficient access and operation. This method of defining variables is mainly used in inline
189     // assembly files.
190
191     unsigned long get\_free\_page(void)
192     {
193         register unsigned long __res asm("ax");
194
195         // First we look up the byte with a value 0 in the memory byte bitmap and then clear the
196         // corresponding physical memory page. If the resulting page address is larger than the actual
197         // physical memory capacity, look for it again. If no free page is found, perform the swap
198         // operation and then look it up again. Finally, the free physical page address is returned.
199
200     repeat:
201         // Find a free page using the memory byte bitmap mem_map[].
202         __asm__( "std ; repne ; scasb\n\t"           // al(0) compared with content of each page (di).
203                 "jne 1f\n\t"                       // jump to label 1 if none of it equals 0.
204                 "movb $1, 1(%%edi) \n\t"           // set byte [1+edi] to 1 of the page.
205                 "sall $12, %%ecx\n\t"              // pages * 4K = relative address of the page.
206                 "addl %2, %%ecx\n\t"               // plus LOW_MEM to get the absolute page address.
207
208                 // Zero the memory page.
209                 "movl %%ecx, %%edx\n\t"            // load edx register with page start address.
210                 "movl $1024, %%ecx\n\t"           // load ecx with counting number 1024.
211                 "leal 4092(%%edx), %%edi\n\t"      // load edi with page end address (4092 + edx).
212                 "rep ; stosl\n\t"                  // reset (clear) each byte in the page.
213                 "movl %%edx, %%eax\n\t"           // load eax with the page start address.
214                 "1:"
215                 : "=a" (__res)

```

```

189 : "0" (0), "i" (LOW MEM), "c" (PAGING PAGES),
190 "D" (mem_map+PAGING PAGES-1)
191 : "di", "cx", "dx";
192 if (__res >= HIGH MEMORY) // search again if the page is out of main page area.
193     goto repeat;
194 if (!__res && swap_out()) // do swapping if no free page found and search again.
195     goto repeat;
196 return __res; // return the address of the free page.
197 }
198
// Memory page swapping initialization.
// The function first checks if the device has a swap partition based on the device's partition
// array (array of blocks) and checks if the swap partition is valid. Then apply to get a page
// of memory to store swap page bitmap array swap_bitmap[], and read the swap management page
// (the first page) from the device's swap partition into the swap bitmap array. Then carefully
// check whether each bit in the swap bitmap array and finally return.
199 void init_swapping(void)
200 {
// Blk_size[] is a pointer array to the number of blocks of the block device specified by the
// major device number. Each of its items corresponds to the total number of data blocks owned
// by one sub-device, and each sub-device corresponds to one partition of the block device.
// If no swap device number is defined in the system, the code returns; if the swap device does
// not set an array of blocks, a warning message is displayed and returned.
201     extern int *blk_size[]; // defined in blk_drv/ll_rw_blk.c, line 49.
202     int swap_size, i, j;
203
204     if (!SWAP_DEV)
205         return;
206     if (!blk_size[MAJOR(SWAP_DEV)]) {
207         printk("Unable to get size of swap device\n\r");
208         return;
209     }
// Then get and check the total number of blocks 'swap_size' in the swap partition of the swap
// device. If it is 0, it will return; if the total number of blocks is less than 100, a warning
// message will be displayed and then exit.
210     swap_size = blk_size[MAJOR(SWAP_DEV)][MINOR(SWAP_DEV)];
211     if (!swap_size)
212         return;
213     if (swap_size < 100) {
214         printk("Swap device too small (%d blocks)\n\r", swap_size);
215         return;
216     }
// Then we convert the total number of swap blocks into the corresponding total number of swap
// pages (blocks/4). This value cannot be greater than the number of pages that SWAP_BITS can
// represent (32768). Then get a free memory page to store the swap bitmap array 'swap_bitmap',
// where each bit represents one swap page.
217     swap_size >>= 2;
218     if (swap_size > SWAP_BITS)
219         swap_size = SWAP_BITS;
220     swap_bitmap = (char *) get_free_page();
221     if (!swap_bitmap) {
222         printk("Unable to start swapping: out of memory :-)\n\r");
223         return;

```

```

224     }
    // Then we read page 0 on the device swap partition into the swap_bitmap page, which is the
    // swap area management page. Among them, the beginning of the 4086th byte contains the swap
    // device feature string "SWAP-SPACE". If the feature string is not found, it is not a valid
    // swap device. So we display a warning message, release the memory page we just got and exit
    // the function. Otherwise, the feature string (10 bytes) in the memory page is cleared.
    // The macro read_swap_page(nr, buffer) is defined in file include/linux/mm.h, line 11.
225     read_swap_page(0, swap_bitmap);
226     if (strncmp("SWAP-SPACE", swap_bitmap+4086, 10)) {
227         printk("Unable to find swap-space signature\n\r");
228         free_page((long) swap_bitmap);
229         swap_bitmap = NULL;
230         return;
231     }
232     memset(swap_bitmap+4086, 0, 10);
    // Then we check the read swap bitmap, which contains a total of 32,768 bits. If the bit of
    // the bitmap is 0, it means that the corresponding swap page on the device is used (occupied).
    // If the bit is 1, it indicates that the corresponding swap page is available (idle). So for
    // the swap partition on the device, the first page (page 0) is used for swap management and
    // is already occupied (bit 0 is 0). The swap page [1 -- swap_size-1] is available, so their
    // corresponding bits in the bitmap should be 1 (idle). Bits in the [swap_size -- SWAP_BITS]
    // range in the bitmap should also be initialized to 0 (occupied) because there is no
    // corresponding swap pages. Below, when checking the bitmap, the bitmap is checked in two steps
    // based on the unavailable and available parts.
    // First check the bitmap bits of the unavailable swap page, they should all be 0 (occupied).
    // If any of these bits is 1 (idle), there is a problem with the bitmap. The error message is
    // then displayed, the page occupied by the bitmap is released, and the function is exited.
233     for (i = 0 ; i < SWAP_BITS ; i++) {
234         if (i == 1)
235             i = swap_size;
236         if (bit(swap_bitmap, i)) {
237             printk("Bad swap-space bit-map\n\r");
238             free_page((long) swap_bitmap);
239             swap_bitmap = NULL;
240             return;
241         }
242     }
    // Then check and count whether all bits between [1 to swap_size-1] are 1s (idle). If the
    // statistics show that there is no free swap page, it means that there is a problem with the
    // swap function, so the page occupied by the bitmap is released and the function is exited.
    // Otherwise, the swap device works ok and the number of swap pages and the total bytes of swap
    // space are displayed.
243     j = 0;
244     for (i = 1 ; i < swap_size ; i++)
245         if (bit(swap_bitmap, i))
246             j++;
247     if (!j) {
248         free_page((long) swap_bitmap);
249         swap_bitmap = NULL;
250         return;
251     }
252     printk("Swap device ok: %d pages (%d bytes) swap-space\n\r", j, j*4096);
253 }

```

13.5 Summary

This chapter describes how the kernel manages and accesses physical and virtual memory in the system. It focuses on the memory allocation management mechanism of Intel CPU, and how the Linux kernel divides the memory space. At the same time, it gives the principle of copy-on-write mechanism and demand loading mechanism. Finally, we illustrate and describe the bitmap-swap page management and processing mechanism of the swap partition on the device.

In the next chapter, we fully describe all the header files included in the Linux kernel source code, including important data structures and macro definitions in almost all kernel code.

14 Header Files (include)

The program should declare the function first before using it. For ease of use, it is common practice to put the same type of functions or data structures and declarations of constants in a header file. Any related type definitions and macro definitions can also be included in the header file. The preprocessor directive `"#include"` is used in the program source file to reference the relevant header file.

A control line statement as shown below in the program will cause the line to be replaced by the contents of the file 'filename':

```
# include <filename>
```

Of course, the filename 'filename' cannot contain `>` and newline characters, as well as the `"`, `'`, `\`, or `/*` characters. The compiler will search for this file in a set of pre-set places. Similarly, the following form of control line will cause the compiler to first search for the 'filename' file in the directory where the source program is located:

```
# include "filename"
```

If the file is not found, the compiler will perform the same search process as above. In this form, the file name 'filename' also cannot contain newline characters and `"`, `'`, `\`, or `/*` characters, but the `>` character is allowed.

In general application source code, the header files are inextricably linked to the library files in the development environment. Each function in the library needs to be declared in the relevant header file. The header files in the application development environment (usually placed in the `/usr/include/` directory) can be thought of as an integral part of the functions in the libraries they provide (eg `libc.a`), and are instructions or interface statements for library functions. After the compiler converts the source code program into an object module, the linker combines all the object modules of the program, including the modules in any library files used, to form an executable program.

For the standard C library, there are about 15 basic header files. Each header file represents a functional description or structure definition of a particular class of functions, such as I/O manipulation functions, character handling functions, and so on. A detailed description of the standard library and its implementation can be found in the book "The Standard C Library" by Mr. Plauger.

For the kernel source code described in this book, the header files involved can be seen as a summary of the services provided by the kernel and its libraries, and are the header files specific to the kernel and its related programs. In these header files, all the data structures, initialization data, constants, and macro definitions used by the kernel are mainly described, as well as a small amount of program code. In addition to several specialized header files (such as the block device header file `blk.h`), the header files used in the Linux 0.12 kernel are placed in the `include/` directory of the kernel source tree. Therefore, compiling this Linux kernel does not require the use of any header files located in the `/usr/include/` directory provided by the development environment. Of course, except for the `tools/build.c` program. Because although this program is included in the kernel source tree, it is just a utility or application for creating a kernel image file that will not be linked into the

kernel code.

Starting with kernel version 0.95, the header files in the kernel source tree need to be copied to the `/usr/include/linux` directory to compile the kernel smoothly. That is, starting with this version of the kernel, the header files have been merged with the header files used by the Linux development environment.

14.1 Files in the include/ directory







The header files used by the kernel are stored in the `include/` directory of the kernel source tree. The files in this directory are shown in Listing 14-1. One point to note here is that for ease of use and compatibility, Linus uses a similar naming convention for standard kernel C header files when compiling kernel program header files. The names of many header files, and even some of the contents of the files, are basically the same as those of the standard C library. But these header files are still kernel-specific or program-specific that is closely tied to the kernel. On a Linux system, they coexist with the header files of the standard library. The usual practice is to place these header files in a subdirectory of the standard library header file directory for use by programs that require kernel data structures or constants.











In addition, due to copyright issues, Linus also attempted to rewrite some header files to replace the header files of the standard C library with copyright restrictions. So the header files in these kernel sources have some overlap with the header files in the development environment. In Linux systems, for application development, the kernel header files in the `asm/`, `linux/`, and `sys/` subdirectories in Listing 14-1 usually need to be copied to the directory where the standard C library header files are located (`/usr/include`), while other files do not conflict with the standard library header files, you can directly put them in the standard library header file directory, or change them to the three subdirectories here.

The `asm/` directory is primarily used to store header files for function declarations or data structures that are closely related to the computer architecture being used. For example, the Intel CPU port IO assembly macro file `io.h`, the interrupt descriptor set assembly macro header file `system.h`, and so on. The `linux/` directory contains some header files used by the Linux kernel program, including the header file `sched.h` used by the scheduler, the memory management header file `mm.h`, and the terminal management data structure file `tty.h`. The `sys/` directory contains several header files related to kernel resources. The `sys/` directory stores several header files related to kernel resources. However, starting from version 0.98, the header files in the `sys/` directory under the kernel directory tree are all moved to the `linux/` directory.

There are 36 header files (*.h) in the Linux 0.12 kernel, including 4 in the `asm/` subdirectory, 11 in the `linux/` subdirectory, and 8 in the `sys/` subdirectory. Starting from the next section, we first describe the 13 header files in the `include/` directory, and then describe the files in each subdirectory in turn. The order of description is sorted by file name.

List 14-1 Files in the linux/include/ directory

Filename	Size	Last Modified Time	Description
 <code>asm/</code>		1992-01-09 16:46:04	
 <code>linux/</code>		1992-01-12 19:43:55	
 <code>sys/</code>		1992-01-09 16:46:03	
 <code>a.out.h</code>	6047 bytes	1991-09-17 15:10:49	
 <code>const.h</code>	321 bytes	1991-09-17 15:12:39	
 <code>ctype.h</code>	1049 bytes	1991-11-07 17:30:47	

	errno.h	1364 bytes	1992-01-03 18:52:20
	fcntl.h	1374 bytes	1991-09-17 15:12:39
	signal.h	1974 bytes	1992-01-04 14:54:10
	stdarg.h	780 bytes	1991-09-17 15:02:23
	stddef.h	285 bytes	1991-12-28 03:19:05
	string.h	7881 bytes	1991-09-17 15:04:09
	termios.h	5268 bytes	1992-01-14 13:53:25
	time.h	874 bytes	1992-01-04 14:58:17
	unistd.h	7300 bytes	1992-01-13 22:48:52
	utime.h	225 bytes	1991-09-17 15:03:38

14.2 a.out.h

14.2.1 Function

In the Linux kernel, the a.out.h file is used to define the executable file structure loaded in a.out format, mainly used in executable file loader program fs/exec.c. This file is not part of the standard C library, but a kernel-specific header file. However, since there is no conflict with the header file name of the standard library, it can generally be placed in the /usr/include/ directory on Linux systems for use by programs that involve related content. This header file defines an a.out (Assembly out) format for the object file. This object file format is used for .o files and executables used in Linux 0.12 systems.

The a.out.h file contains three data structure definitions and some related macro definitions, so the file can be divided into three parts accordingly:

- Lines 1-108 give and describe the object header structure and associated macro definitions;
- Lines 109-185 are definitions and descriptions of the structure of symbol entries;
- Lines 186-217 define and describe the structure of the relocation table entry.

Due to the large amount of content in the file, a detailed description of three data structures and associated macro definitions is placed after the program list. Here is a brief introduction to the exec structure in the a.out format file.

An a.out executable file consists of an exec header section at the beginning and subsequent code, data, and other parts. The header part of a executable file mainly contains an exec structure, which contains information such as the magic number field, code and data length, symbol table length, and code execution start position. The kernel uses this information to load the executable into memory and execute it, and the linker (ld) uses these parameters to combine some of the module files into one executable. This is the only necessary component of a object file.

Starting with the 0.96 kernel, the Linux system directly uses GNU's header file a.out.h. As a result, programs compiled under Linux 0.9x cannot run on Linux 0.1x systems. Below we analyze the differences between the two a.out header files, and explain how to make some executable files compiled under 0.9x without using dynamic link library can also run under 0.1x.

The main difference between the a.out.h file used by Linux 0.12 and the GNU file of the same name is the first field a_magic of the exec structure. The file field name of GNU is a_info, and the field is further divided

into three sub-domains: Flags, Machine Type, and Magic Number. At the same time, the corresponding macros N_MACHTYPE and N_FLAGS are defined for the machine type field, as shown in Figure 14-1.

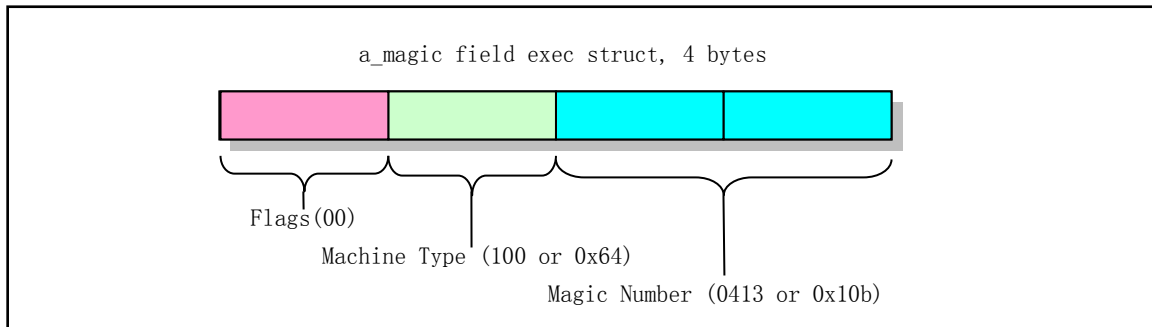


Figure 14-1 The first field in the exec structure a_magic(a_info)

In the Linux 0.9x system, for executable files linked with static libraries, the values in parentheses in the figure are the default values for each field. The 4 bytes at the beginning of this binary executable file are:

```
0x0b, 0x01, 0x64, 0x00
```

The a.out header file in this kernel only defines the magic number field. Therefore, the first 4 bytes of a binary executable file in a.out format on a Linux 0.1x system are:

```
0x0b, 0x01, 0x00, 0x00
```

It can be seen that the difference between the executable file in the a.out format of GNU and the executable file compiled on the Linux 0.1x system is only in the machine type field. So we can clear the machine type field (3rd byte) of the a.out format executable file on Linux 0.9x and run it on the 0.1x system. As long as the system-call invoked by the ported executable file is already implemented in the 0.1x system. The author used this approach when starting to rebuild many of the commands in the Linux 0.1x root file system. In other respects, GNU's a.out.h header file is no different from a.out.h here.

14.2.2 Code annotation

Program 14-1 linux/include/a.out.h

```

1 #ifndef A_OUT_H
2 #define A_OUT_H
3
4 #define __GNU_EXEC_MACROS__
5
6 // Lines 6-108 are the first part of the document, which mainly defines the execution structure
7 // of the object file and the macros of related operations.
8 // Below is the object file header structure, see the detailed description after the program.
9 struct exec {
10     unsigned long a_magic;          /* Use macros N_MAGIC, etc for access */
11     unsigned a_text;               /* length of text, in bytes */

```

```

9  unsigned a_data;                /* length of data, in bytes */
10 unsigned a_bss;                /* length of uninitialized data area for file, in bytes */
11 unsigned a_syms;                /* length of symbol table data in file, in bytes */
12 unsigned a_entry;              /* start address */
13 unsigned a_trsize;              /* length of relocation info for text, in bytes */
14 unsigned a_drsize;              /* length of relocation info for data, in bytes */
15 };
16
17 // Macro definition, used to take the magic number in the above exec structure.
18 #ifndef N_MAGIC
19 #define N_MAGIC(exec) ((exec).a_magic)
20 #endif
21 #ifndef OMAGIC
22 /* Code indicating object file or impure executable. */
23 // Historically, on the PDP-11 computer, the magic number (magic number) was octal number 0407
24 // (0x107). Historically, on the PDP-11 computer, the magic number (magic number) was octal
25 // number 0407 (0x107), which was at the beginning of the executable file header structure.
26 // It was originally a jump instruction of the PDP-11, indicating that it jumped to the beginning
27 // of the code after the next 7 words. In this way, the loader can jump directly to the beginning
28 // of the instruction after putting the executable file into memory. There is no program to
29 // use this method, but this octal number is retained as a flag (magic number) for identifying
30 // the file type. 'OMAGIC' can be considered as 'Old Magic'.
31 #define OMAGIC 0407
32 /* Code indicating pure executable. */
33 // NMAGIC (New Magic), used after 1975. It involves the virtual storage mechanism.
34 #define NMAGIC 0410 // 0410 == 0x108
35 /* Code indicating demand-paged executable. */
36 // This type of header structure occupies 1KB of space at the beginning of the file.
37 #define ZMAGIC 0413 // 0413 == 0x10b
38 #endif /* not OMAGIC */
39 // There is also a QMAGIC, which is used to save disk capacity and store the header structure
40 // and code of the file on the disk in a compact manner.
41 // The following macro (Bad Magic) is used to determine the correctness of the magic number
42 // field. Returns true if the magic number cannot be recognized.
43 #ifndef N_BADMAG
44 #define N_BADMAG(x) \
45     (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC \
46      && N_MAGIC(x) != ZMAGIC)
47 #endif
48 #define N_BADMAG(x) \
49     (N_MAGIC(x) != OMAGIC && N_MAGIC(x) != NMAGIC \
50      && N_MAGIC(x) != ZMAGIC)
51
52 // A macro definition that gives the remaining length between the end of the header structure
53 // and the 1KB position.
54 #define N_HDROFF(x) (SEGMENT_SIZE - sizeof (struct exec))
55
56 // The following macros are used to manipulate the contents of the object file, including .o
57 // module files and executable files.
58
59 // The starting offset of the code portion.

```

```

// If the file is of type ZMAGIC, ie an executable file, the code portion begins at a 1024-byte
// offset from the execution file; otherwise the code portion begins with the end of the execution
// header structure (32 bytes), ie the file is a module File (OMAGIC type).
42 #ifndef N_TXTOFF
43 #define N_TXTOFF(x) \
44 (N_MAGIC(x) == ZMAGIC ? N_HDROFF((x)) + sizeof (struct exec) : sizeof (struct exec))
45 #endif
46
// The data part start offset, starting at the end of the code section.
47 #ifndef N_DATOFF
48 #define N_DATOFF(x) (N_TXTOFF(x) + (x).a_text)
49 #endif
50
// The code relocation info offset, starting at the end of the data section.
51 #ifndef N_TRELOFF
52 #define N_TRELOFF(x) (N_DATOFF(x) + (x).a_data)
53 #endif
54
// The data relocation info offset, starting at the end of the code relocation info.
55 #ifndef N_DRELOFF
56 #define N_DRELOFF(x) (N_TRELOFF(x) + (x).a_trsize)
57 #endif
58
// The symbol table offset, start at the end of the data segment relocation table.
59 #ifndef N_SYMOFF
60 #define N_SYMOFF(x) (N_DRELOFF(x) + (x).a_drsize)
61 #endif
62
// The string information offset, after the symbol table.
63 #ifndef N_STROFF
64 #define N_STROFF(x) (N_SYMOFF(x) + (x).a_syms)
65 #endif
66
// The following is location operation where a executable is loaded into logical space.
67 /* Address of text segment in memory after it is loaded. */
68 #ifndef N_TXTADDR
69 #define N_TXTADDR(x) 0 // The code segment begins at address 0.
70 #endif
71
72 /* Address of data segment in memory after it is loaded.
73 Note that it is up to you to define SEGMENT_SIZE
74 on machines not listed here. */
75 #if defined(vax) || defined(hp300) || defined(pyr)
76 #define SEGMENT_SIZE PAGE_SIZE
77 #endif
78 #ifdef hp300
79 #define PAGE_SIZE 4096
80 #endif
81 #ifdef sony
82 #define SEGMENT_SIZE 0x2000
83 #endif /* Sony. */
84 #ifdef is68k
85 #define SEGMENT_SIZE 0x20000

```

```

86 #endif
87 #if defined(m68k) && defined(PORTAR)
88 #define PAGE_SIZE 0x400
89 #define SEGMENT_SIZE PAGE_SIZE
90 #endif
91
92 // Here, the kernel defines the memory page as 4KB, and the segment size is defined as 1KB,
93 // so the above definition is not used.
94 #define PAGE_SIZE 4096
95 #define SEGMENT_SIZE 1024
96
97 // The size defined by the segment (considering the carry).
98 #define _N_SEGMENT_ROUND(x) (((x) + SEGMENT_SIZE - 1) & ~(SEGMENT_SIZE - 1))
99
100 // Code segment end address.
101 #define N_TXTENDADDR(x) (N_TXTADDR(x) + (x).a_text)
102
103 // Data segment start address.
104 // If the file is of the OMAGIC type, the data segment immediately follows the code segment.
105 // Otherwise the data segment address starts from the segment boundary after the code segment
106 // (1KB boundary alignment), for example for a ZMAGIC type file.
107 #ifndef N_DATADDR
108 #define N_DATADDR(x) \
109     (N_MAGIC(x) == OMAGIC ? (_N_TXTENDADDR(x)) \
110      : (_N_SEGMENT_ROUND (_N_TXTENDADDR(x))))
111 #endif
112
113 /* Address of bss segment in memory after it is loaded. */
114 // The uninitialized data segment bbs is located and follows the data segment.
115 #ifndef N_BSSADDR
116 #define N_BSSADDR(x) (N_DATADDR(x) + (x).a_data)
117 #endif
118
119 // Lines 110--185 are part 2. It describes the symbol table entries in the object file and defines
120 // related operation macros. See the detailed instructions after the program listing.
121 // The symbol table entry (record) structure in the a.out object file.
122 #ifndef N_NLIST_DECLARED
123 struct nlist {
124     union {
125         char *n_name;
126         struct nlist *n_next;
127         long n_strx;
128     } n_un;
129     unsigned char n_type;           // The byte is divided into 3 fields, and
130     char n_other;                  // lines 146-154 are the mask code for each field.
131     short n_desc;
132     unsigned long n_value;
133 };
134 #endif
135
136 // The constants for the n_type field in the nlist structure are defined below.
137 #ifndef N_UNDF
138 #define N_UNDF 0

```

```

126 #endif
127 #ifndef N\_ABS
128 #define N\_ABS 2
129 #endif
130 #ifndef N\_TEXT
131 #define N\_TEXT 4
132 #endif
133 #ifndef N\_DATA
134 #define N\_DATA 6
135 #endif
136 #ifndef N\_BSS
137 #define N\_BSS 8
138 #endif
139 #ifndef N\_COMM
140 #define N\_COMM 18
141 #endif
142 #ifndef N\_FN
143 #define N\_FN 15
144 #endif
145
146 // The following 3 constants are the mask (octal) of the n_type field in the nlist structure.
147 #ifndef N\_EXT
148 #define N\_EXT 1 // 0x01 (0b0000,0001) symbol is external (global) ?.
149 #endif
150 #ifndef N\_TYPE
151 #define N\_TYPE 036 // 0x1e (0b0001,1110) The type bits of the symbol.
152 #endif
153 #ifndef N\_STAB
154 #define N\_STAB 0340 // 0xe0 (0b1110,0000) These bits are used for symbol debugger.
155 #endif
156 /* The following type indicates the definition of a symbol as being
157    an indirect reference to another symbol. The other symbol
158    appears as an undefined reference, immediately following this symbol.
159
160    Indirection is asymmetrical. The other symbol's value will be used
161    to satisfy requests for the indirect symbol, but not vice versa.
162    If the other symbol does not have a definition, libraries will
163    be searched to find a definition. */
164 #define N\_INDR 0xa
165
166 /* The following symbols refer to set elements.
167    All the N\_SET[ATDB] symbols with the same name form one set.
168    Space is allocated for the set in the text section, and each set
169    element's value is stored into one word of the space.
170    The first word of the space is the length of the set (number of elements).
171
172    The address of the set is made into an N\_SETV symbol
173    whose name is the same as the name of the set.
174    This symbol acts like a N\_DATA global symbol
175    in that it can satisfy undefined external references. */
176
177 /* These appear as input to LD, in a .o file. */

```

```
178 #define N_SETA 0x14      /* Absolute set element symbol */
179 #define N_SETT 0x16      /* Text set element symbol */
180 #define N_SETD 0x18      /* Data set element symbol */
181 #define N_SETB 0x1A      /* Bss set element symbol */
182
183 /* This is output from LD. */
184 #define N_SETV 0x1C      /* Pointer to set vector in data area. */
185
186 #ifndef N_RELOCATION_INFO_DECLARED
187
188 /* This structure describes a single relocation to be performed.
189  * The text-relocation section of the file is a vector of these structures,
190  * all of which apply to the text section.
191  * Likewise, the data-relocation section applies to the data section. */
192
193 // The code and data relocation info structure in a object file of a.out.
194 struct relocation_info
195 {
196     /* Address (within segment) to be relocated. */
197     int r_address;
198     /* The meaning of r_symbolnum depends on r_extern. */
199     unsigned int r_symbolnum:24;
200     /* Nonzero means value is a pc-relative offset
201      * and it should be relocated for changes in its own address
202      * as well as for changes in the symbol or section specified. */
203     unsigned int r_pcrel:1;
204     /* Length (as exponent of 2) of the field to be relocated.
205      * Thus, a value of 2 indicates 1<<2 bytes. */
206     unsigned int r_length:2;
207     /* 1 => relocate with value of symbol.
208      * r_symbolnum is the index of the symbol
209      * in file's the symbol table.
210      * 0 => relocate with the address of a segment.
211      * r_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
212      * (the N_EXT bit may be set also, but signifies nothing). */
213     unsigned int r_extern:1;
214     /* Four bits that aren't used, but when writing an object file
215      * it is desirable to clear them. */
216     unsigned int r_pad:4;
217 };
218 #endif /* no N_RELOCATION_INFO_DECLARED. */
219
220 #endif /* __A_OUT_GNU_H__ */
221
```

14.2.3 Information

14.2.3.1 a.out executable file format

The Linux kernel version 0.12 only supports the format of a.out (Assembler output) executable files and object files. Although this format has been gradually not used, but the ELF (Executable and Link Format) format is more fully used, due to its simplicity, it is suitable as a material for learning. Below we give a comprehensive introduction to the a.out format.

Three data structures and some macros are declared in the header file a.out.h. These data structures describe the structure of the object files on the system. In the Linux 0.12 system, the compiled object module file (referred to as the module file) and the binary executable file generated by the linker are in the a.out format. Here we collectively refer to them as object files. An object file can consist of up to seven parts (sections). They are in order:

- a) **exec header** -- This section contains some parameters (exec structure) that the kernel uses to load the executable file into memory and execute it, and the linker (ld) uses these parameters to combine some of the module files into one executable file. This is the only necessary component of the object file.
- b) **text segment** -- Contains the instruction code (text) and related data that is loaded into memory when the program is executed. It can be loaded in read-only form.
- c) **data segment** -- This section contains data that has already been initialized and is always loaded into readable and writable memory.
- d) **text relocations** -- This section contains records data for use by the linker. Used to locate and update a pointer or address in a code segment when combining object module files.
- e) **data relocation** -- Similar to the role of the code relocation section, but for the relocation of pointers in the data segment.
- f) **symbol table** -- This section also contains records data for use by the linker to cross-reference named variables and functions (symbols) between binary object module files.
- g) **string table** -- This part contains character strings corresponding to the symbol names.

Each object or binary file begins with an execution data structure (exec structure) in the form of:

```
struct exec {
    unsigned long a_magic      // Use macros for access, such as N_MAGIC.
    unsigned a_text           // length of code, in bytes.
    unsigned a_data           // length of data, in bytes.
    unsigned a_bss            // length of the uninitialized data area for file, in bytes.
    unsigned a_syms           // length of the symbol table data in the file, in bytes.
    unsigned a_entry          // Execution start address.
    unsigned a_trsize         // length of the relocation info for text, in bytes.
    unsigned a_drsiz          // length of the relocation info for data, in bytes.
};
```

The functionality of each field is as follows:

- **a_magic** -- This field contains three subfields, the flag field, the machine type id field, and the magic number field, as shown in Figure 14-1. However, for the Linux 0.12 system, its object file only uses the magic number subfield and is accessed using the macro N_MAGIC(), which uniquely determines the difference between the binary executable file and other loaded files. This subfield must contain one of the following values:
 - ◆ **OMAGIC** -- Indicates that the text and data segments are immediately following the execution header and are stored consecutively. The kernel loads both text and data segments into readable and writable memory. The magic number of the object file compiled by the compiler is OMAGIC (octal 0407).
 - ◆ **NMAGIC** -- Like OMAGIC, text and data segments follow the execution header and are stored continuously. However, the kernel loads the text into read-only memory and loads the data

segment into writable memory at the next page boundary after the text.

- ◆ ZMAGIC -- The kernel loads separate pages from the binary executable when necessary. The execution header, text segment, and data segment are all processed by the linker into blocks of multiple page sizes. The text page loaded by the kernel is read-only, and the page of the data segment is writable. The magic number of the executable file generated by the linker is ZMAGIC (0413, ie 0x10b).
- a_text -- This field contains the size of the text segment, the number of bytes.
- a_data -- This field contains the size of the data segment, the number of bytes.
- a_bss -- Contains the length of the 'bss segment' that the kernel uses to set the initial break(brk) after the data segment. When the kernel is loading the program, this writable memory appears to be behind the data segment and is initially all zeros.
- a_syms -- Contains the size in bytes of the symbol table section.
- a_entry -- The memory address of the program execution start point after the kernel has loaded the executable file into memory.
- a_trsize -- This field contains the size of the text relocation table, in bytes.
- a_drsize -- This field contains the size of the data relocation table, in bytes.

Several macros are defined in the a.out.h header file. These macros use the exec structure to test for consistency or to locate various section offsets in the executable file. These macros are:

N_BADMAG(exec)	Returns a non-zero value if the a_magic field cannot be recognized.
N_TXTOFF(exec)	The starting byte offset of the code segment.
N_DATOFF(exec)	The starting byte offset of the data segment.
N_DRELOFF(exec)	The starting byte offset of the data relocation table.
N_TRELOFF(exec)	The starting byte offset of the text relocation table.
N_SYMOFF(exec)	The starting byte offset of the symbol table.
N_STROFF(exec)	The starting byte offset of the string table.

The relocation record has a standard format, which is described using a relocation information (relocation_info) structure, as shown below.

```

struct relocation_info
{
    int r_address;           // The address that needs to be relocated within the segment.
    unsigned int r_symbolnum:24; // The meaning is related to r_extern.
                                // Specifies a symbol or a segment in the symbol table.
    unsigned int r_pcrel:1;   // A pc-related flag.
    unsigned int r_length:2;  // Length (as exponent of 2) of the field to be relocated.
    unsigned int r_extern:1;  // 1:relocate with value of symbol, 0:with address of segment.
    unsigned int r_pad:4;     // 4 bits are not used, but it is best to clear them.
};

```

The meaning of each field in the structure is as follows:

- r_address -- This field contains the byte offset of the pointer that the linker needs to process (edit). The offset of the text relocation is counted from the beginning of the text segment, and the offset of the data relocation is calculated from the beginning of the data segment. The linker adds the value

already stored at the offset to the new value calculated using the relocation record.

- **r_symbolnum** -- This field contains the ordinal number (not the byte offset) of a symbol structure in the symbol table. After the linker calculates the absolute address of the symbol, it adds the address to the pointer being relocated. (If the **r_extern** bit is 0, then the situation is different, see below.)
- **r_pcrel** -- If this bit is set, the linker considers that a pointer is being updated, which uses the pc-related addressing mode and is part of the machine code instruction. When the running program uses this relocated pointer, the address of the pointer is implicitly added to the pointer.
- **r_length** -- This field contains the power of 2 of the length of the pointer: 0 means 1 byte long, 1 means 2 bytes long, 2 means 4 bytes long.
- **r_extern** -- If set, it indicates that the relocation requires an external reference; the linker must use a symbol address to update the pointer. When the bit is 0, the relocation is "local"; the linker updates the pointer to reflect the changes in the load addresses of the various segments, rather than reflecting changes in the value of a symbol. In this case, the contents of the **r_symbolnum** field are an **n_type** value; such field tells the linker what segment the relocated pointer points into.
- **r_pad** -- These 4 bits are not used in Linux systems. It is best to set all 0 when writing a object file.

Symbols map names to addresses (or more generally, strings are mapped to values). Due to the linker's adjustment of the address, the name of a symbol must be used to indicate its address until it has been assigned an absolute address value. A symbol consists of a fixed-length record in the symbol table and a variable-length name in the string table. The symbol table is an array of **nlist** structures, as shown below.

```
struct nlist {
    union {
        char      *n_name;
        struct nlist *n_next;
        long       n_strx;
    } n_un;
    unsigned char n_type;           // divided into 3 fields, and lines 146-154 are their masks.
    char          n_other;
    short         n_desc;
    unsigned long n_value;
};
```

The meaning of each field is:

- **n_un.n_strx** -- Contains the byte offset of the symbol name in the string table. When a program accesses a symbol table using the **nlist()** function, the field is replaced with the **n_un.n_name** field, which is a pointer to a string in memory.
- **n_type** -- Used by the linker to determine how to update the value of the symbol. The 8-bit wide **n_type** field can be divided into three subfields using the bitmasks code starting at line 146--154, as shown in Figure 14-2. For symbols of **N_EXT** type location bits, the linker treats them as "external" symbols and allows other binary object files to reference them. The **N_TYPE** mask is used to select the bits of interest to the linker:
 - ◆ **N_UNDF** -- An undefined symbol. The linker must locate an external symbol with the same name in another binary object file to determine the absolute data value of the symbol. In special cases, if the **n_type** field is non-zero and no binary file defines this symbol, the linker resolves the symbol into an address in the BSS segment, reserving bytes of length equal to **n_value**. If the

symbol is not defined in more than one binary object file and the binary object files do not match their length values, the linker will select the longest value found across all binary object files.

- ◆ **N_ABS** -- An absolute symbol. The linker does not update an absolute symbol.
- ◆ **N_TEXT** -- A text (code) symbol. The value of this symbol is the text address, and the linker updates its value when it merges the binary object files.
- ◆ **N_DATA** -- A data symbol; similar to **N_TEXT**, but for data addresses. The value of the corresponding text and data symbol is not the offset of the file but the address; In order to find the offset of the file, it is necessary to determine the address at which the relevant section starts loading and subtract it, and then add the offset of the section.
- ◆ **N_BSS** -- A BSS symbol; similar to a text or data symbol, but without a corresponding offset in the binary object file.
- ◆ **N_FN** -- A file name symbol. When merging a binary object file, the linker inserts the symbol before the symbol in the binary file. The name of the symbol is the file name given to the linker, and its value is the address of the first text segment in the binary file. File name symbols are not required for linking and loading, but are very useful for debugging programs.
- ◆ **N_STAB** -- The mask code is used to select the bits of interest to the symbolic debugger (eg gdb); its value is specified in `stab()`.
- **n_other** -- This field provides symbol independence information about the symbol relocation operation according to the segment determined by **n_type**. Currently, the lowest 4 bits of the **n_other** field contain one of two values: **AUX_FUNC** and **AUX_OBJECT**. **AUX_FUNC** associates symbols with callable functions, and **AUX_OBJECT** associates symbols with data, regardless of whether they are in text segments or data segments. This field is mainly used by the linker `ld` for the creation of dynamic executable programs.
- **n_desc** -- Reserved for use by the debugger; the linker does not process it. Different debuggers use this field for different purposes.
- **n_value** -- Contains the value of the symbol. For text, data, and BSS symbols, this is an address; for other symbols (such as debugger symbols), the value can be arbitrary.

A string table consists of symbol strings of length unsigned long followed by a null. The length represents the byte size of the entire table, so the minimum value (that is, the offset of the first string) on a 32-bit machine is always 4.

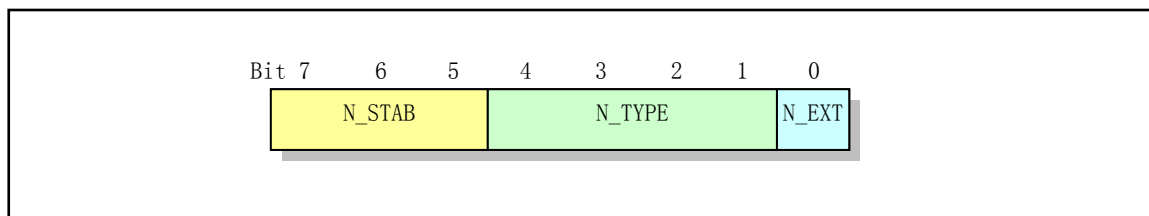


Figure 14-2 Symbol type field **n_type**

14.3 const.h

14.3.1 Function

The const.h file defines some of the flag constants used by the file modes and type field `i_mode` in the file `i-node`.

14.3.2 Code annotation

Program 14-2 linux/include/const.h

```

1 #ifndef CONST_H
2 #define CONST_H
3
4 #define BUFFER_END 0x200000          // the memory end used by the buffer (not used).
5
6 // Each flag bit of the i_mode field in the i-node structure.
7 #define I_TYPE          0170000      // Indicate the i-node type (type mask).
8 #define I_DIRECTORY    0040000      // Is a directory file.
9 #define I_REGULAR      0100000      // Is a regular file, not a dir or a special file.
10 #define I_BLOCK_SPECIAL 0060000     // Is a block device special file.
11 #define I_CHAR_SPECIAL 0020000     // Is a character device special file.
12 #define I_NAMED_PIPE   0010000     // Is a named pipe node.
13 #define I_SET_UID_BIT   0004000     // Set efficient user ID type at execution time.
14 #define I_SET_GID_BIT   0002000     // Set efficient group ID type at execution time.
15 #endif
16

```

14.4 ctype.h

14.4.1 Function

The ctype.h file is a header file for character testing and processing, and is one of the header files for the standard C library. It defines some macros for character type checking and conversion. It defines some macros for character type checking and conversion. For example, determine if a character `c` is a numeric character (`isdigit(c)`) or a space character (`isspace(c)`). During the check process, an array or table (defined in `lib/ctype.c`) is used, which defines the properties and types of all the characters in the ASCII table. When a macro is used, the character code is taken as an index value in the table `_ctype[]`, and a byte is obtained from the table, so that the relevant bit is obtained.

In addition, macro names that begin with two underscores or begin with an underscore followed by an uppercase letter are usually reserved for the header writer, such as the names `__abc` and `_SP`.

14.4.2 Code annotation

Program 14-3 linux/include/ctype.h

```

1 #ifndef CTYPE_H
2 #define CTYPE_H
3
4 #define U      0x01    /* upper */           // used for uppercase characters [A-Z].
5 #define L      0x02    /* lower */           // used for lowercase characters [a-z].
6 #define D      0x04    /* digit */           // used for digitals [0-9].
7 #define C      0x08    /* cntrl */           // used for control characters.
8 #define P      0x10    /* punct */           // used for punctuation characters.
9 #define S      0x20    /* white space (space/lf/tab) */
10 #define X      0x40    /* hex digit */           // used for hexadecimal digits.
11 #define SP     0x80    /* hard space (0x20) */    // used for the space character (0x20).
12
13 // An array (table) of character attributes that defines the attributes of each character.
14 // The other is a temporary character variable. They are all defined in lib/ctype.c file.
15 extern unsigned char ctype[];
16 extern char ctmp;
17
18 // Below are some macros that determine the character type.
19 #define isalnum(c) ((ctype+1)[c]&(U|L|D))    // is a character or digital.
20 #define isalpha(c) ((ctype+1)[c]&(U|L))        // is a character.
21 #define iscntrl(c) ((ctype+1)[c]&(C))          // is a control character.
22 #define isdigit(c) ((ctype+1)[c]&(D))          // is a digital.
23 #define isgraph(c) ((ctype+1)[c]&(P|U|L|D)) // is a graphic character.
24 #define islower(c) ((ctype+1)[c]&(L))          // is a lowercase character.
25 #define isprint(c) ((ctype+1)[c]&(P|U|L|D|SP)) // is a printable character.
26 #define ispunct(c) ((ctype+1)[c]&(P))          // is a punctuation mark.
27 #define isspace(c) ((ctype+1)[c]&(S))          // is a space, \f, \n, \r, \t, \v.
28 #define isupper(c) ((ctype+1)[c]&(U))          // is an uppercase character.
29 #define isxdigit(c) ((ctype+1)[c]&(D|X))      // is a hexadecimal number.
30
31 // In the following two macro definitions, the macro parameter is prefixed (unsigned), so 'c'
32 // should be bracketed, which means it is (c). Because 'c' may be a complex expression in the
33 // program. For example, if the argument is a + b, without parentheses, it becomes: (unsigned)
34 // a + b in the macro definition, which is obviously wrong. After bracketing, it can be correctly
35 // represented as (unsigned) (a + b).
36 #define isascii(c) (((unsigned) c)<=0x7f)        // is an ASCII character.
37 #define toascii(c) (((unsigned) c)&0x7f)          // convert to ASCII character.
38
39 // The reason for using a temporary variable 'ctmp' in the following two macro definitions
40 // is that the macro's parameters can only be used once in the macro definition. But this is
41 // not safe for multithreading because two or more threads may use this public temporary variable
42 // at the same time. So from kernel 2.2.x, these two macro definitions are changed to use two
43 // functions.
44 #define tolower(c) (ctmp=c, isupper(ctmp)? ctmp-'A'-'a': ctmp) // to lowercase char.
45 #define toupper(c) (ctmp=c, islower(ctmp)? ctmp-'a'-'A': ctmp) // to uppercase char.
46
47 #endif
48

```

14.5 errno.h

14.5.1 Function

There is a variable called 'errno' in the UNIX type system or the standard C language. Whether this variable is needed in the C standard has caused a lot of controversy in the C standardization organization (X3J11). But the result of the debate was that the 'errno' was not removed, instead a header file named "errno.h" was created. Because the standardization organization wants each library function or data object to be declared in a corresponding standard header file.

The main reason for the debate is that for each system-call in the kernel, if the return value is the result of the system-call, it is difficult to report the error. If we let each function return a true/false indication value and the resulting value returns separately, we can't easily get the result of the system-call. One solution is to combine these two methods: For a particular system-call, you can specify an error return value that is different from the range of valid result values. For example, a pointer can take a null value, and for a pid it can return a value of -1. In many other cases, '-1' can be used to indicate an error value as long as it does not conflict with the resulting value. However, the standard C library function return value only tells whether an error has occurred, and the type of error must be known from other places, so the variable 'errno' is used.

In order to be compatible with the design mechanism of the standard C library, the library file in the Linux kernel also uses this processing method. Therefore, this header file of the standard C library is also borrowed. See the lib/open.c program and the system-call macro definition in unistd.h for examples. In some cases, although the program knows the error from the returned '-1' value, but wants to know the specific error number, you can determine the error number of the last error by reading the value of 'errno'.

14.5.2 Code annotation

Program 14-4 linux/include/errno.h

```
1 #ifndef ERRNO\_H
2 #define ERRNO\_H
3
4 /*
5  * ok, as I hadn't got any other source of information about
6  * possible error numbers, I was forced to use the same numbers
7  * as minix.
8  * Hopefully these are posix or something. I wouldn't know (and posix
9  * isn't telling me - they want $$$ for their f***ing standard).
10 *
11 * We don't use the _SIGN cludge of minix, so kernel returns must
12 * see to the sign by themselves.
13 *
14 * NOTE! Remember to change strerror() if you change this file!
15 */
16
17 // System-calls and many library functions return a special value to indicate an operation
18 // failure or an error. This value is usually chosen to be '-1', or some other specific value.
19 // But this return value only indicates that an error has occurred. If we need to know the type
20 // of error, we need to look at the variable 'errno' which represents the system error number.
21 // This variable is declared in the errno.h file and is initialized to 0 when the program begins
```

```

    // execution.
17 extern int errno;
18
    // In the event of an error, the system-call puts the error number in the variable 'errno'
    // (negative value) and returns -1. Therefore, if the program needs to know the specific error
    // number, you need to check the value of 'errno'.
19 #define ERROR          99          // General error.
20 #define EPERM          1          // The operation is not permitted.
21 #define ENOENT        2          // The file or directory does not exist.
22 #define ESRCH          3          // The specified process does not exist.
23 #define EINTR          4          // Interrupted system-call.
24 #define EIO            5          // Input/output error.
25 #define ENXIO          6          // Specified device or address doesn't exist.
26 #define E2BIG          7          // The parameter list is too long.
27 #define ENOEXEC       8          // The format of executable file is incorrect.
28 #define EBADF          9          // File handle (descriptor) is incorrect.
29 #define ECHILD        10         // The child process does not exist.
30 #define EAGAIN        11         // The resource is temporarily unavailable.
31 #define ENOMEM       12         // No enough memory.
32 #define EACCES       13         // No access permissions.
33 #define EFAULT       14         // The address is wrong.
34 #define ENOTBLK      15         // Not a block device file.
35 #define EBUSY        16         // The resource is busy.
36 #define EEXIST       17         // File already exists.
37 #define EXDEV        18         // Illegal connection to device.
38 #define ENODEV       19         // The device does not exist.
39 #define ENOTDIR      20         // Not a directory file.
40 #define EISDIR       21         // Is a directory file.
41 #define EINVAL       22         // Invalid argument.
42 #define ENFILE       23         // The system has too many open files.
43 #define EMFILE       24         // Too many open files.
44 #define ENOTTY       25         // Inappropriate IO (no tty terminal).
45 #define ETXTBSY      26         // (No longer use).
46 #define EFBIG        27         // File size too big.
47 #define ENOSPC       28         // The device is full (the device has no space).
48 #define ESPIPE       29         // Invalid file pointer relocation.
49 #define EROFS        30         // The file system is read only.
50 #define EMLINK       31         // Too many links.
51 #define EPIPE        32         // The pipe is wrong.
52 #define EDOM        33         // Domain error.
53 #define ERANGE       34         // The result range error.
54 #define EDEADLK      35         // Resource deadlocks.
55 #define ENAMETOOLONG 36         // The filename is too long.
56 #define ENOLCK       37         // No locks are available.
57 #define ENOSYS       38         // Not yet implemented.
58 #define ENOTEMPTY    39         // The directory is not empty.
59
60 /* Should never be seen by user programs */
61 #define ERESTARTSYS    512        // Re-execute the system-call.
62 #define ERESTARTNOINTR 513        // Re-execute the system-call, no interrupt.
63
64 #endif
65

```

14.6 fcntl.h

14.6.1 Function

The fcntl.h file is the file control option header file. It mainly defines the file control function fcntl() and some of the options used in the file creation or open function. The fcntl() function is implemented in the linux/fs/fcntl.c file, and is used to perform various specified operations on the file descriptor (handle). The specific operation is specified by the function parameter cmd (command).

14.6.2 Code annotation

Program 14-5 linux/include/fcntl.h

```

1 #ifndef FCNTL\_H
2 #define FCNTL\_H
3
4 #include <sys/types.h>          // Type header file. The basic system data types are defined.
5
6 /* open/fcntl - NOCTTY, NDELAY isn't implemented yet */
7 #define O\_ACCMODE          00003          // File access mode mask.
8 // File access modes used by the open() and fcntl(), and only one of the three can be used.
9 #define O\_RDONLY            00          // Open file in read-only mode.
10 #define O\_WRONLY           01          // Open file in write-only mode.
11 #define O\_RDWR            02          // Open file in read-write mode.
12 // Below are the file creation and operation flags for open(). Can be used with the above
13 // access mode in a 'bit or' mode.
14 #define O\_CREAT            00100      /* not fcntl */ // Create if file does not exist.
15 #define O\_EXCL            00200      /* not fcntl */ // Exclusive use of file.
16 #define O\_NOCTTY          00400      /* not fcntl */ // No control terminal.
17 #define O\_TRUNC           01000      /* not fcntl */ // truncated to zero if write operation.
18 #define O\_APPEND          02000          // Open file in append mode.
19 #define O\_NONBLOCK        04000      /* not fcntl */ // Open file in non-blocking manner.
20 #define O\_NDELAY          O\_NONBLOCK
21
22 /* Defines for fcntl-commands. Note that currently
23  * locking isn't supported, and other things aren't really
24  * tested.
25  */
26 // The command (cmd) used by the file descriptor's operation function fcntl().
27 #define F\_DUPFD            0          /* dup */ // duplicate a file handle.
28 #define F\_GETFD            1          /* get f_flags */ // get file handle flags (FD_CLOEXEC).
29 #define F\_SETFD            2          /* set f_flags */ // set file handle flags.
30 #define F\_GETFL            3          /* more flags (cloexec) */
31 #define F\_SETFL            4          // get/set file state flag & access mode.
32 // Below are file lock commands. The parameter 'lock' of fcntl() points to flock structure.
33 #define F\_GETLK            5          /* not implemented */ // get flock that blocks the lock.
34 #define F\_SETLK            6          // Set (F_RDLCK or F_WRLCK) or clear lock (F_UNLCK).
35 #define F\_SETLKW           7          // Set or clear the lock in wait mode.

```

```
31
32 /* for F_[GET|SET]FL */
    // The file handle needs to be closed when executing the exec() function.
33 #define FD_CLOEXEC      1      /* actually anything with low bit set goes */
34
35 /* Ok, these are locking features, and aren't implemented at any
36  * level. POSIX wants them.
37  */
38 #define F_RDLCK          0      // Share or read file lock.
39 #define F_WRLCK          1      // Exclusive or write file lock.
40 #define F_UNLCK          2      // File unlock.
41
42 /* Once again - not implemented, but ... */
    // The following is a file lock data structure that describes the type (l_type) of the affected
    // file segment, the start offset (l_whence), the relative offset (l_start), the lock length
    // (l_len), and the pid that implements the lock.
43 struct flock {
44     short l_type;           // Lock type (F_RDLCK, F_WRLCK, F_UNLCK).
45     short l_whence;        // Start offset (SEEK_SET, SEEK_CUR or SEEK_END).
46     off_t l_start;         // The beginning of the lock. Relative offset (in bytes).
47     off_t l_len;           // The size of the lock; if 0, it is the end of the file.
48     pid_t l_pid;           // The process id of the lock.
49 };
50
    // The following are function prototypes using the above flags or commands.
    // Create a new file or rewrite an existing file. The parameter 'filename' is the file name
    // of the file to be created, and 'mode' is the property (see include/sys/stat.h).
51 extern int creat(const char * filename, mode_t mode);
    // File handle operation function. They can affect file open operations. The parameter 'fildes'
    // is the file handle (descriptor), and 'cmd' is the operation command, see lines 23-30 above.
    // The function can be in the following forms:
    // int fcntl(int fildes, int cmd);
    // int fcntl(int fildes, int cmd, long arg);
    // int fcntl(int fildes, int cmd, struct flock *lock);
52 extern int fcntl(int fildes, int cmd, ...);
    // open a file. Used to establish a connection between a file and a file handle.
    // The parameter 'flags' is a combination of the flags on lines 7-17 above.
53 extern int open(const char * filename, int flags, ...);
54
55 #endif
56
```

14.7 signal.h

14.7.1 Functionality

Signals provide a way to handle asynchronous events, and signals are also known as soft interrupts. By sending a signal to a process, we can control the execution state of the process (pause, resume, or terminated). The signal.h file defines the names and basic operational functions of all the signals used in the kernel. The

most important function are the functions `signal()` and `sigaction()` that change the way the signal is processed.

As you can see from the header file, the Linux kernel implements all 20 signals required by POSIX.1. So we can say that Linux was completely designed with compatibility with the standard at the outset. The specific implementation of the function can be found in the program `kernel/signal.c`.

14.7.2 Code annotation

Program 14-6 linux/include/signal.h

```

1 #ifndef SIGNAL_H
2 #define SIGNAL_H
3
4 #include <sys/types.h>          // Type header file. The basic system data types are defined.
5
6 typedef int sig_atomic_t;      // define signal atomic operation type.
7 typedef unsigned int sigset_t; /* 32 bits */ // define signal set type.
8
9 #define NSIG          32        // number of signals.
10 #define NSIG          NSIG    // NSIG = _NSIG
11
12 // The following are the signals defined in the Linux 0.12 kernel. This includes all 20 signals
13 // required by POSIX.1.
14 #define SIGHUP          1        // Hang Up      -- Hang up the control terminal or process.
15 #define SIGINT          2        // Interrupt  -- Interrupt from the keyboard.
16 #define SIGQUIT         3        // Quit       -- Exit command from the keyboard.
17 #define SIGILL          4        // Illeagle   -- Illegal instruction.
18 #define SIGTRAP         5        // Trap       -- Track breakpoints.
19 #define SIGABRT         6        // Abort      -- 
20 #define SIGIOT          6        // IO Trap    -- 
21 #define SIGUNUSED       7        // Unused     -- 
22 #define SIGFPE          8        // FPE        -- Coprocessor error.
23 #define SIGKILL         9        // Kill       -- Force the process to terminate.
24 #define SIGUSR1        10       // User1      -- User defined signal 1.
25 #define SIGSEGV        11       // Segment Violation -- Invalid memory reference.
26 #define SIGUSR2        12       // User2      -- User defined signal 2.
27 #define SIGPIPE        13       // Pipe       -- Pipe write error, no reader.
28 #define SIGALRM        14       // Alarm      -- Real-time timer alarm.
29 #define SIGTERM        15       // Terminate  -- Process terminated.
30 #define SIGSTKFLT      16       // Stack Fault -- Stack error (coprocessor).
31 #define SIGCHLD        17       // Child      -- Child process is stopped or terminated.
32 #define SIGCONT        18       // Continue   -- Resume the execution of process.
33 #define SIGSTOP        19       // Stop       -- Stop the execution of process.
34 #define SIGTSTP        20       // TTY Stop   -- Stop sig sent by process, can be ignored.
35 #define SIGTTIN        21       // TTY In     -- Background process requests input.
36 #define SIGTTOU        22       // TTY Out    -- Background process requests output.
37
38 /* Ok, I haven't implemented sigactions, but trying to keep headers POSIX */
39 // The above comment is obsolete because sigaction() has been implemented in the 0.12 kernel.
40 // The following is the symbolic constant that can be taken from the 'sa_flags' flag field in
41 // the sigaction structure.
42 // SA_NOCLDSTOP - When child is in stopped state, the SIGCHLD signal is not processed.
43 // SA_INTERRUPT - The system call is not restarted after it is interrupted by the signal.
44 // SA_NOMASK - This signal is not blocked from being received in its signal handler.

```

```

// SA_ONESHOT - The signal handler is restored to its default one once it has been called.
37 #define SA_NOCLDSTOP    1
38 #define SA_INTERRUPT    0x20000000
39 #define SA_NOMASK       0x40000000
40 #define SA_ONESHOT      0x80000000
41
// The following constants are used for sigprocmask(how, ) -- to add/remove a given signal
// to/from the blocking signal set, and change the blocking signal set (mask code).
// Used to change the behavior of this function.
42 #define SIG_BLOCK        0 /* for blocking signals */
43 #define SIG_UNBLOCK      1 /* for unblocking signals */
44 #define SIG_SETMASK      2 /* for setting the signal mask */
45
// The following three constant symbols all represent function pointers that have no return
// value and have an INT integer parameter. These three pointers are logically the addresses
// of functions that are practically impossible. They can be used as the second parameter of
// the signal() function below to tell the kernel to let the kernel process the signal, ignore
// the processing of the signal, or signal processing returns an error. For how to use them,
// see kernel/signal.c, lines 156--158.
46 #define SIG_DFL          ((void (*)(int))0) /* default signal handling */
47 #define SIG_IGN          ((void (*)(int))1) /* ignore signal */
48 #define SIG_ERR          ((void (*)(int))-1) /* error return from signal */
49
// The following defines a macro for initial setting the sigaction structure signal mask.
50 #ifndef notdef
51 #define sigemptyset(mask) ((*mask) = 0), 1 // Clear mask.
52 #define sigfillset(mask) ((*mask) = ~0), 1 // All bits of the mask are set.
53 #endif
54
// The following is the sigaction data structure, where the meaning of each field is:
// 'sa_handler' is the action that is specified for a signal. This signal can be ignored with
// SIG_DFL, or SIG_IGN above, or it can be a pointer to a function that handles the signal.
// 'sa_mask' gives the masking code for the signal, which will block the processing of these
// signals when the signal program is executed.
// 'sa_flags' specifies the set of signals that change the signal processing, which is defined
// by the bit flags of lines 37-40.
// 'sa_restorer' is a recovery function pointer, provided by the function library Libc, used
// to clean up the user stack. See signal.c.
// In addition, the signal that causes the trigger signal processing will also be blocked unless
// the SA_NOMASK flag is used.
55 struct sigaction {
56     void (*sa_handler)(int);
57     sigset_t sa_mask;
58     int sa_flags;
59     void (*sa_restorer)(void);
60 };
61
// The signal() function below is used to install a new signal handler for signal _sig, similar
// to sigaction(). This function takes two arguments: one is to specify the signal _sig to be
// captured, and the other is a function pointer _func with one argument and no return value.
// The return value of this function is also a function pointer with an int argument (the last
// (int)) and no return value, which is the original handle of the signal.
62 void (*signal(int _sig, void (*_func)(int)))(int);

```

```
// The following two functions are used to send signals. kill() is used to send a signal to
// any process or process group; raise() is used to send a signal to the current process itself,
// which is equivalent to kill(getpid(), sig). See kernel/exit.c, line 205.
63 int raise(int sig);
64 int kill(pid_t pid, int sig);
// In the task structure of the process, in addition to a 32-bit signal field 'signal' indicating
// the signal to be processed of the current process, there is a 32-bit blocking signal set
// field 'blocked' for masking the signal, each bit representing a corresponding blocked signal.
// Modifying the mask signal set can block or unblock the specified signal. The following five
// functions are used to manipulate the process to mask the signal set. Although it is very
// simple to implement, it is not implemented in this version of the kernel.
// The functions sigaddset() and sigdelset() are used to add, delete, and modify signals in
// the signal set. Sigaddset() is used to add the given signal signo to the signal set pointed
// to by the mask, sigdelset() is the opposite. The functions sigemptyset() and sigfillset()
// are used to initialize the process masking signal set. Before using the signal set, each
// program needs to initialize the mask signal set using one of these two functions. Sigemptyset()
// is used to clear all masked signals, that is, to respond to all signals; sigfillset() puts
// all signals into the signal set, ie masks all signals. Of course SIGINT and SIGSTOP cannot
// be blocked. The sigismember() function is used to test if a specified signal is in the signal
// set (1 - yes, 0 - no, -1 - error).
65 int sigaddset(sigset_t *mask, int signo);
66 int sigdelset(sigset_t *mask, int signo);
67 int sigemptyset(sigset_t *mask);
68 int sigfillset(sigset_t *mask);
69 int sigismember(sigset_t *mask, int signo); /* 1 - is, 0 - not, -1 error */
// Check the signal in 'set' to see if there is a pending signal. And in 'set' returns the currently
// blocked signal set in the process.
70 int sigpending(sigset_t *set);
// The following function is used to change the signal set that the process is currently blocking.
// If 'oldset' is not NULL, then it returns the current masked signal set. If the 'set' pointer
// is not NULL, modify the process mask signal set according to the 'how' (line 42-44).
71 int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
// The following function temporarily replaces the signal mask of the process with 'sigmask'
// and then pauses the process until it receives a signal. If a signal is captured and returned
// from the signal handler, the function returns and the signal mask is restored to the value
// before the call.
72 int sigsuspend(sigset_t *sigmask);
// The sigaction() function is used to change the action taken by the process when it receives
// signal, that is, to change the processing handler of the signal. See the description in the
// kernel/signal.c program.
73 int sigaction(int sig, struct sigaction *act, struct sigaction *oldact);
74
75 #endif /* _SIGNAL_H */
76
```

14.8 stdarg.h

14.8.1 Function

One of the biggest features of the C language is that it allows programmers to customize functions with

variable number of parameters. In order to access the parameters in these variable parameter lists, you need to use the macros in the stdarg.h file. The stdarg.h header file is modified by the C standardization organization according to the varargs.h file of the BSD system.

Stdarg.h is a standard parameter header file that defines a list of variable parameters in the form of macros. It mainly describes a type (va_list) and three macros (va_start, va_arg and va_end) for the vsprintf, vprintf, vfprintf functions. When reading this file, we need to first understand how to use the variable parameter function. See the description after the kernel/vsprintf.c list.

14.8.2 Code annotation

Program 14-7 linux/include/stdarg.h

```

1  #ifndef  STDARG\_H
2  #define  STDARG\_H
3
4  typedef char *va\_list;  // Defining va_list is a character pointer type.
5
6  /* Amount of space required in an argument list for an arg of type TYPE.
7   TYPE may alternatively be an expression whose type is used.  */
8
9  // The following sentence defines the byte length of the rounded TYPE type, which is a multiple
10 // of the int length (4).
11 #define __va_rounded_size(TYPE)  \
12 (((sizeof (TYPE) + sizeof (int) - 1) / sizeof (int)) * sizeof (int))
13
14 // The following macro initializes the pointer AP to point to the first argument passed to the
15 // function's variable argument list. Before calling va_arg or va_end for the first time, you
16 // must first call the va_start macro. The parameter LASTARG is the identifier of the rightmost
17 // parameter in the function definition, ie an identifier to the left of '...'. The AP is a
18 // variable parameter table parameter pointer, and LASTARG is the last specified parameter.
19 // &(LASTARG) is used to get its address (ie its pointer) and the pointer is a character type.
20 // After adding the width value of LASTARG, the AP is a pointer to the first parameter in the
21 // variable parameter list. This macro has no return value.
22 // The function __builtin_saveregs() on line 17 is defined in gcc's library program libgcc2.c
23 // and is used to hold registers. See the "Implementing the Varargs Macros" section of the gcc
24 // manual "Target Description Macros" for instructions.
25
26 #ifndef __sparc__
27 #define va\_start(AP, LASTARG) \
28 (AP = ((char *) &(LASTARG) + va\_rounded\_size (LASTARG)))
29 #else
30 #define va\_start(AP, LASTARG) \
31 (__builtin_saveregs (), \
32 AP = ((char *) &(LASTARG) + va\_rounded\_size (LASTARG)))
33 #endif
34
35 // The macro below is used by the called function to complete a normal return. va_end can modify
36 // the AP to not be used until va_start is called again. va_end must be called after va_arg
37 // has read all the parameters.
38
39 void va\_end (va\_list);          /* Defined in gnu lib */
40 #define va\_end(AP)
41
42 // The following macro is used to extend the expression to have the same type and value as the
43 // next passed parameter. For default values, va_arg can be used with characters, unsigned

```

```
// characters, and floating point types. The first time you use va_arg, it returns the first
// argument in the table, and each subsequent call returns the next argument in the table. This
// is done by first accessing the AP and then increasing its value to point to the next item.
// va_arg uses TYPE to complete access and locate the next item. Each time va_arg is called,
// it modifies the AP to indicate the next parameter in the table.
24 #define va_arg(AP, TYPE) \
25 (AP += __va_rounded_size (TYPE), \
26 *((TYPE *) (AP - __va_rounded_size (TYPE))))
27
28 #endif /* _STDARG_H */
29
```

14.9 stddef.h

14.9.1 Functionality

The `stddef.h` header file is created by the C standardization organization (X3J11), and the meaning of the file name is the standard (std) definition (def). It is mainly used to store "standard definitions" such as some commonly used constant symbols and function prototypes in UNIX-like systems. Another confusing header file is `stdlib.h`, which is also created by the standardization organization, and is mainly used to declare various function prototype declarations that are not related to other header file types. But the contents of these two header files often make it impossible to figure out which declarations are in which header file.

Some members of the standardization organization believe that C should also be a useful programming language in a stand-alone environment that does not fully support the standard C library. For a standalone environment, the C standard requires that it provide all the attributes of the C language. For the standard C library, such an implementation only needs to provide support for the functions in the four header files: `float.h`, `limits.h`, `stdarg.h`, and `stddef.h`. This requirement clarifies what the `stddef.h` file should contain, and the other three header files are basically used for more specific aspects:

- `float.h` Describe the floating point representation feature;
- `limits.h` Describe the integer representation characteristics;
- `stdarg.h` Provides macro definitions for accessing variable parameter list.

Any other type or macro definition used in a standalone environment should be placed in the `stddef.h` file, but later organizational members have relaxed these restrictions, causing some definitions to appear in multiple header files. For example, the `NULL` macro definition also appears in the other four header files. Therefore, in order to prevent conflicts, the `stddef.h` file first uses the `undef` command to cancel the original definition when defining `NULL` (line 14). In addition, the types and macros defined in this file have one thing in common: they have been tried to be included in the features of the C language, but since various compilers have defined this information in their own way, it's very hard to write code that replaces all of these definitions. This file is rarely used in the Linux 0.12 kernel.

14.9.2 Code annotation

Program 14-8 linux/include/stddef.h

```
1 #ifndef _STDDEF_H
```

```
2 #define _STDDEF_H
3
4 #ifndef _PTRDIFF_T
5 #define _PTRDIFF_T
6 typedef long ptrdiff_t;           // The type of result of subtracting two pointers.
7 #endif
8
9 #ifndef _SIZE_T
10 #define _SIZE_T
11 typedef unsigned long size_t;     // The type of result returned by sizeof().
12 #endif
13
14 #undef NULL
15 #define NULL ((void *)0)          // null pointer.
16
17 // The following defines a macro that calculates the offset position of a member in a type.
18 // By using this macro, you can determine the byte offset of a member (field) from the beginning
19 // of the structure in the structure type that contains it. The result of the macro is an integer
20 // constant expression of type size_t. Here is a trick usage: ((TYPE *)0) means that an integer
21 // type 0 is cast into a data object pointer type, and then the operation is performed on the
22 // result.
23 #define offsetof(TYPE, MEMBER) ((size_t) &((TYPE *)0)->MEMBER)
24
25 #endif
26
```

14.10 string.h

14.10.1 Functionality

The string.h header file defines all string manipulation functions as inline functions, and uses inline assembly language to improve execution speed. In addition, a NULL macro and a SIZE_T type are defined at the beginning of the file.

A header file of the same name is also provided in the standard C library, but the function implementation is in the standard C library, and its corresponding header file contains only the declaration of the relevant function. For the string.h file listed here, Linus gives the implementation of each function, but each function has the 'extern' and 'inline' keyword prefixes, that is, some inline functions are defined. Therefore, for a program containing this header file, if the inline function used for some reason cannot be embedded in the calling code, the same name function defined in the kernel function library lib/ directory will be used, see the lib/string.c program. In that string.c, the program first defines 'extern' and 'inline' to be empty, and then contains the string.h header file. Therefore, the string.c program actually contains the function implementation code declared in the string.h header file.

To make it easy to write comments on one line, some abbreviations are used here in the source without causing confusion, for example: string - str; character - char; pointer - ptr; address - addr. source - src; destination - dest; length - len.

14.10.2 Code annotation

Program 14-9 linux/include/string.h

```

1  #ifndef STRING\_H
2  #define STRING\_H
3
4  #ifndef NULL
5  #define NULL ((void *) 0)
6  #endif
7
8  #ifndef SIZE\_T
9  #define SIZE\_T
10 typedef unsigned int size\_t;
11 #endif
12
13 extern char * strerror(int errno);
14
15 /*
16  * This string-include defines all string functions as inline
17  * functions. Use gcc. It also assumes ds=es=data space, this should be
18  * normal. Most of the string-functions are rather heavily hand-optimized,
19  * see especially strtok, strstr, str[c]spn. They should work, but are not
20  * very easy to understand. Everything is done entirely within the register
21  * set, making the functions fast and clean. String instructions have been
22  * used through-out, making for "slightly" unclear code :-)
23  *
24  * (C) 1991 Linus Torvalds
25  */
26
27  /// Copy a source string to destination string until it encounters a NULL character.
28  // Params: dest - the destination str pointer, src - the source str pointer.
29  // In the embedded assembly code, %0 - register ESI(src), %1 - EDI(dest).
30  extern inline char * strcpy(char * dest, const char *src)
31  {
32  __asm__( "cld\n" // Clear direction flag.
33          "l:|t lodsb\n|t" // Load 1 byte from DS:[ESI] into AL and update ESI.
34          "stosb\n|t" // Store the byte in AL to ES:[EDI] and update EDI.
35          "testb %%al, %%al\n|t" // The byte just stored is 0?
36          "jne 1b" // If not, jump backward to label 1, else end.
37          :: "S" (src), "D" (dest): "si", "di", "ax");
38  return dest; // Returns the destination str pointer.
39  }
40
41  /// Copy count bytes of the source string to the destination.
42  // If the source str length is less than count bytes, null chars are appended to the dest string.
43  // Parameters: dest - the destination str pointer, src - the source str pointer, count - number
44  // of bytes copied. %0 - esi(src), %1 - edi(dest), %2 - ecx(count).
45  extern inline char * strncpy(char * dest, const char *src, int count)
46  {
47  __asm__( "cld\n" // Clear direction flag.
48          "l:|t decl %2\n|t" // Register ECX-- (count--).
49          "js 2f\n|t" // If count<0 then jump forward to label 2 and end.

```

```

43     "lodsbl\n\t"           // Get 1 byte from DS:[ESI] to AL, and ESI++.
44     "stosbl\n\t"           // Store the byte to ES:[EDI], and EDI++.
45     "testb %%al,%%al\n\t"   // Is this byte 0?
46     "jne 1b\n\t"           // If not, jump forward to label 1 to continue copy.
47     "rep\n\t"               // Else, store remain number of NULLs into dest str.
48     "stosb\n\t"
49     "2:"
50     :: "S" (src), "D" (dest), "c" (count): "si", "di", "ax", "cx");
51 return dest;               // Returns the dest string pointer.
52 }
53
54     /// Copy the source string to the end of the destination string.
55     // Parameters: dest - the destination str pointer, src - the source str pointer.
56     // In the assembly code, %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1).
57 extern inline char * strcat(char * dest, const char * src)
58 {
59     __asm__( "cld\n\t"
60             "repne\n\t"           // Compare AL and ES:[EDI] byte and update EDI++,
61             "scasb\n\t"           // Until byte in dest is 0, the EDI points to last byte.
62             "decl %l\n\t"         // Let ES:[EDI] points to the location of 0.
63             "1:\tlodsb\n\t"       // Take source str byte DS:[ESI] to AL, and ESI++.
64             "stosb\n\t"           // Store this byte to ES:[EDI] and EDI++.
65             "testb %%al,%%al\n\t" // Is this byte 0?
66             "jne 1b"             // If not, jump back to label 1 to continue, else end.
67             :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff): "si", "di", "ax", "cx");
68 return dest;               // Returns the dest str pointer.
69 }
70
71     /// Copy count bytes of source str to the end of the dest str, and finally add a null char.
72     // dest - the destination str, src - the source str, count - number of bytes to copy.
73     // In the assembly code, %0 - esi(src), %1 - edi(dest), %2 - eax(0), %3 - ecx(-1), %4 - (count).
74 extern inline char * strncat(char * dest, const char * src, int count)
75 {
76     __asm__( "cld\n\t"
77             "repne\n\t"           // Compare AL and ES:[EDI], and update EDI++,
78             "scasb\n\t"           // Until 0 in dest, the EDI points to last byte.
79             "decl %l\n\t"         // Let ES:[EDI] points to the location of 0.
80             "movl %4,%3\n\t"      // Place the bytes to be copied into ECX.
81             "1:\tdecl %3\n\t"     // ECX-- (counts from 0).
82             "js 2f\n\t"           // Ecx < 0 ?, yes, jump forward to the label 2.
83             "lodsbl\n\t"         // Otherwise take the bytes at DS:[ESI] to AL, ESI++.
84             "stosb\n\t"           // Store to ES:[EDI], and EDI++.
85             "testb %%al,%%al\n\t" // The byte value is 0?
86             "jne 1b\n\t"         // If not, jump back to label 1 and continue copying.
87             "2:\txorl %2,%2\n\t"  // Clear AL to zero.
88             "stosb"              // Save to ES:[EDI].
89             :: "S" (src), "D" (dest), "a" (0), "c" (0xffffffff), "g" (count)
90             : "si", "di", "ax", "cx");
91 return dest;               // Returns the dest string pointer.
92 }
93
94     /// Compare two strings.
95     // Parameters: cs - string 1, ct - string 2.

```

```

// %0 - eax(__res) return value, %1 - edi(cs) str1 pointer, %2 - esi(ct) str2 pointer.
// If str1 > str2, returns 1; str1 = str2 returns 0; str1 < str2 returns -1.
88 extern inline int strcmp(const char * cs,const char * ct)
89 {
90     register int __res __asm__( "ax");    // __res is a register variable (eax).
91     __asm__( "cld\n"
92             "1:\tlodsb\n\t"                // Get str2 byte DS:[ESI] to AL, and ESI++.
93             "scasb\n\t"                    // Compare AL with str1 byte ES:[EDI], and EDI++.
94             "jne 2f\n\t"                    // If they are not equal, jump forward to label 2.
95             "testb %%al,%%al\n\t"          // Is this byte 0 (end of string)?
96             "jne 1b\n\t"                    // No, jump back to label 1, and continue comparison.
97             "xorl %%eax,%%eax\n\t"          // Yes, EAX is cleared, jumps forward to label 3,end.
98             "jmp 3f\n\t"
99             "2:\tmovl $1,%%eax\n\t"          // EAX is set to 1.
100            "jl 3f\n\t"                      // If str2 < str1, returns a positive value, end.
101            "negl %%eax\n\t"                  // Else EAX = -EAX, return a negative value, and end.
102            "3:"
103            : "=a" (__res): "D" (cs), "S" (ct): "si", "di");
104     return __res;                          // Returns the comparison result.
105 }
106
//// String 1 is compared to the first count characters of string 2.
// Parameters: cs - str1, ct - str2, count - the number of characters to compare.
// %0 - eax(__res), %1 - edi(cs) str1 pointer, %2 - esi(ct) str2 pointer, %3 - ecx(count).
// If str1 > str2, returns 1; str1 = str2 returns 0; str1 < str2 returns -1.
107 extern inline int strncmp(const char * cs,const char * ct,int count)
108 {
109     register int __res __asm__( "ax");    // __res is a register variable (eax).
110     __asm__( "cld\n"
111             "1:\tdecl %3\n\t"                // Register ECX-- (count--).
112             "js 2f\n\t"                      // If count<0, then jump forward to label 2.
113             "lodsb\n\t"                      // Load str2 char DS:[ESI] to AL, and ESI++.
114             "scasb\n\t"                      // Compare AL with str1 char ES:[EDI], and EDI++.
115             "jne 3f\n\t"                      // If they are not equal, jump forward to label 3.
116             "testb %%al,%%al\n\t"            // Is it a NULL char?
117             "jne 1b\n\t"                      // No, jump back to label 1,continue the comparison.
118             "2:\txorl %%eax,%%eax\n\t"        // Yes, then EAX is cleared (return value).
119             "jmp 4f\n\t"                      // Jump forward to label 4 and end.
120             "3:\tmovl $1,%%eax\n\t"          // Set EAX to 1.
121             "jl 4f\n\t"                      // If the result is str2 < str1, 1 is returned.
122             "negl %%eax\n\t"                  // Else EAX = -EAX, return a negative value and end.
123             "4:"
124             : "=a" (__res): "D" (cs), "S" (ct), "c" (count): "si", "di", "cx");
125     return __res;                          // Returns the comparison result.
126 }
127
//// Look for the first matching character in the string.
// Parameters: s - the string, c - the character to be found.
// In the assembly code, %0 - eax(__res), %1 - esi (str pointer s), %2 - eax (char c).
// Returns a pointer to the first occurrence of a matching character in the string. If no
// matching characters are found, a null pointer is returned.
128 extern inline char * strchr(const char * s,char c)
129 {

```



```

130 register char * __res __asm__( "ax" );
131 __asm__( "cld\n\t"
132         "movb %%al, %%ah\n\t"           // Move the char to be compared to AH.
133         "1:\tlodsb\n\t"                 // Get a string char DS:[ESI] to AL, and ESI++.
134         "cmpb %%ah, %%al\n\t"           // The char AL is compared with the specified char AH.
135         "je 2f\n\t"                     // If they are the same, jump forward to the label 2.
136         "testb %%al, %%al\n\t"          // Is the char in AL a NULL char? (End of string?)
137         "jne 1b\n\t"                     // If not, back to label 1 and continue comparison.
138         "movl $1, %1\n\t"               // If yes, no match char is found, and set ESI to 1.
139         "2:\tmovl %1, %0\n\t"           // Put pointer at the next byte of match char into EAX.
140         "decl %0"                       // Adjust the pointer to point to the matching char.
141         : "=a" (__res): "S" (s), "0" (c): "si" );
142 return __res;                          // Returns the pointer.
143 }
144
145 // Look for the last occurrence of the given character in the string. (reverse search string)
146 // Parameters: s - the string, c - the character to be found.
147 // In the assembly code, %0 - eax(__res), %1 - esi (string pointer s), %2 - eax (char c).
148 // Returns the pointer to the last occurrence of a matching character in the string. If no
149 // matching characters are found, a null pointer is returned.
150 extern inline char * strrchr(const char * s, char c)
151 {
152     register char * __res __asm__( "dx" );
153     __asm__( "cld\n\t"
154             "movb %%al, %%ah\n\t"           // Move the char to be compared to AH.
155             "1:\tlodsb\n\t"                 // Get a string char DS:[ESI] to AL, and ESI++.
156             "cmpb %%ah, %%al\n\t"           // The char AL is compared with the specified char AH.
157             "jne 2f\n\t"                     // If not the same, jump forward to the label 2.
158             "movl %%esi, %0\n\t"           // Store the char pointer to EDX.
159             "decl %0\n\t"                   // Adjust the pointer to point to the matching char.
160             "2:\ttestb %%al, %%al\n\t"      // Is the char in AL a NULL char? (End of string?)
161             "jne 1b"                       // If not, back to label 1 and continue comparison.
162             : "=d" (__res): "0" (0), "S" (s), "a" (c): "ax", "si" );
163 return __res;                          // Returns the pointer.
164 }
165
166 // Find the first sequence of chars in string1, any char in the sequence is contained in string2.
167 // Parameters: cs - string1 pointer, ct - string2 pointer.
168 // %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi(str1 ptr cs), %4 - (str2 ptr ct).
169 // Returns the length of the first sequence of chars in str1 contained in str2.
170 extern inline int strspn(const char * cs, const char * ct)
171 {
172     register char * __res __asm__( "si" );
173     __asm__( "cld\n\t"
174             "movl %4, %%edi\n\t"           // Calc the len of str2, store its pointer to EDI.
175             "repne\n\t"                   // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
176             "scasb\n\t"                   // If they are not equal, continue to compare (ECX--).
177             "notl %%ecx\n\t"              // Each bit in ECX is inversed.
178             "decl %%ecx\n\t"              // ECX--, The length of str2 is obtained.
179             "movl %%ecx, %%edx\n\t"       // Put the length of str2 into EDX temporarily.
180             "1:\tlodsb\n\t"               // Put the str1 char DS:[ESI] to AL, and ESI++.
181             "testb %%al, %%al\n\t"        // Is the char equal to 0 (end of str1)?
182             "je 2f\n\t"                   // If yes, jump forward to label 2 and end.

```

```

174     "movl %4,%%edi|n|t"           // Get the str2 pointer into EDI.
175     "movl %%edx,%%ecx|n|t"       // Put the str2 length into ECX.
176     "repne|n|t"                  // Compare AL and str2 char ES:[EDI], and EDI++.
177     "scasb|n|t"                  // If they are equal, continue to compare.
178     "je 1b|n"                    // If they are equal, jump backward to label 1.
179     "2:|tdecl %0"                // ESI--, points to the char in str1 contained in str2.
180     : "=S" (__res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
181     : "ax", "cx", "dx", "di");
182 return __res-cs;                  // Returns the length of the sequence of chars.
183 }
184
185 // Search the first char sequence in string1 that does not contain any of the chars in string2.
186 // Parameters: cs - string1 pointer, ct - string2 pointer.
187 // %0 - esi(__res), %1 - eax(0), %2 - ecx(-1), %3 - esi (str1 ptr cs), %4 - (str2 ptr ct).
188 // Returns the length of first char sequence in str1 that doesn't contain any of chars in str2.
189 extern inline int strcspn(const char * cs, const char * ct)
190 {
191     register char * __res __asm__("si");
192     __asm__("cld|n|t"
193         "movl %4,%%edi|n|t"       // Calc the len of str2, store its pointer to EDI.
194         "repne|n|t"               // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
195         "scasb|n|t"               // If they are not equal, continue to compare (ECX--).
196         "notl %%ecx|n|t"          // Calc the length of str2 by inverting each bits
197         "decl %%ecx|n|t"          // in ECX and decreasing ECX by one.
198         "movl %%ecx,%%edx|n"      // Put str2 length into EDX temporarily.
199         "1:|tlodsb|n|t"           // Take the str1 char DS:[ESI] to AL, and ESI++.
200         "testb %%al,%%al|n|t"    // Is the char equal to 0 (end of str1)?
201         "je 2f|n|t"              // If yes, jump forward to label 2 and end.
202         "movl %4,%%edi|n|t"       // Get the str2 pointer into EDI.
203         "movl %%edx,%%ecx|n|t"    // Put the str2 length into ECX.
204         "repne|n|t"               // Compare AL and str2 char ES:[EDI], and EDI++.
205         "scasb|n|t"               // If they not are equal, continue to compare.
206         "jne 1b|n"                // If they not are equal, jump backward to label 1.
207         "2:|tdecl %0"            // ESI--, points to the char in str1 contained in str2.
208         : "=S" (__res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
209         : "ax", "cx", "dx", "di");
210 return __res-cs;                  // Returns the length of the sequence of chars.
211 }
212
213 // In string1, look for any first character contained in string2.
214 // Parameters: cs - string1 pointer, ct - string2 pointer.
215 // %0 -esi(__res), %1 -eax(0), %2 -ecx(0xffffffff), %3 -esi(str1 ptr cs), %4 -(str2 ptr ct).
216 // Returns the pointer in string1 containing the first character in string2.
217 extern inline char * strpbrk(const char * cs,const char * ct)
218 {
219     register char * __res __asm__("si");
220     __asm__("cld|n|t"
221         "movl %4,%%edi|n|t"       // Calc the len of str2, store its pointer to EDI.
222         "repne|n|t"               // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
223         "scasb|n|t"               // If they are not equal, continue to compare (ECX--).
224         "notl %%ecx|n|t"          // Calc the length of str2 by inverting each bits
225         "decl %%ecx|n|t"          // in ECX and decreasing ECX by one.
226         "movl %%ecx,%%edx|n"      // Store str2 length into EDX temporarily.

```

```

219     "1:\tlodsb\n\t"           // Get the str1 char DS:[ESI] to AL, and ESI++.
220     "testb %%al,%%al\n\t"     // Is the char equal to 0 (end of str1)?
221     "je 2f\n\t"               // If yes, jump forward to label 2.
222     "movl %4,%%edi\n\t"       // Get the str2 pointer into EDI.
223     "movl %%edx,%%ecx\n\t"    // Put the str2 length into ECX.
224     "repne\n\t"               // Compare AL and str2 char ES:[EDI], and EDI++.
225     "scasb\n\t"               // If they not are equal, continue to compare.
226     "jne 1b\n\t"              // If they not are equal, jump backward to label 1.
227     "decl %0\n\t"             // ESI--, points to a char in str1 contained in str2.
228     "jmp 3f\n\t"              // Jump forward to label 3 and end.
229     "2:\txorl %0,%0\n\t"      // If no match is found, it will return NULL.
230     "3:"
231     : "=S" (__res): "a" (0), "c" (0xffffffff), "0" (cs), "g" (ct)
232     : "ax", "cx", "dx", "di";
233 return __res;                 // Returns the pointer in str1.
234 }
235
236 // In string1, look for the first substring that matches the entire string2.
237 // Parameters: cs - string1 pointer, ct - string2 pointer.
238 // %0 - eax(__res), %1 - eax(0), %2 - ecx(0xffffffff), %3 - esi(str1 ptr cs), %4 - (str2 ptr ct).
239 // Returns the first substring pointer in string1 that matches entire string2.
240 extern inline char * strstr(const char * cs,const char * ct)
241 {
242     register char * __res __asm__ ("ax");
243     __asm__ ("cld\n\t" \
244             "movl %4,%%edi\n\t"           // Calc the len of str2, store its pointer to EDI.
245             "repne\n\t"                   // Compare AL(0) with char in str2 (ES:[EDI]), EDI++.
246             "scasb\n\t"                   // If they are not equal, continue to compare (ECX--).
247             "notl %%ecx\n\t"              // Calc str2 len by inversing all bits and dec 1 in ECX.
248             "decl %%ecx\n\t"              /* NOTE! This also sets Z if searchstring='' */
249             "movl %%ecx,%%edx\n\t"        // Store str2 length into EDX temporarily.
250             "1:\tmovl %4,%%edi\n\t"       // Get the str2 pointer into EDI.
251             "movl %%esi,%%eax\n\t"        // Copy the str1 pointer to EAX.
252             "movl %%edx,%%ecx\n\t"        // Put the str2 length into ECX.
253             "repe\n\t"                   // Comp str1,str2 char DS:[ESI],ES:[EDI], ESI++,EDI++.
254             "cmpsb\n\t"                   // If the chars are equal, they continue to compare.
255                                           // If they are all equal, jump to label 2, end.
256             "je 2f\n\t"                  /* also works for empty string, see above */
257             "xchgl %%eax,%%esi\n\t"       // str1 ptr => ESI, comparison result str1 ptr => EAX.
258             "incl %%esi\n\t"              // str1 pointer points to the next char.
259             "cmpb $0,-1(%%eax)\n\t"       // Is the byte pointed to by ptr in str1 (EAX-1) 0?
260             "jne 1b\n\t"                  // If not, go labell, continue comp from 2nd char of str1.
261             "xorl %%eax,%%eax\n\t"        // Clear EAX, indicating that no match was found.
262             "2:"
263             : "=a" (__res): "0" (0), "c" (0xffffffff), "S" (cs), "g" (ct)
264             : "cx", "dx", "di", "si";
265 return __res;                 // Returns the result.
266 }
267
268 // Calculate the length of a string.
269 // Parameter: s - a string.
270 // %0 - ecx(__res), %1 - edi (string pointer s), %2 - eax(0), %3 - ecx(0xffffffff).
271 // Returns the length of the string.

```

```

263 extern inline int strlen(const char * s)
264 {
265     register int __res __asm__("cx");    // __res is a register variable (ECX).
266     __asm__("cld\n\t"                  // Clear the direction flag.
267            "repne\n\t"                  // AL(0) is compared with char es:[edi] in the string,
268            "scasb\n\t"                  // If they are not equal, continue to compare.
269            "notl %0\n\t"                // Inverse each bit in ECX.
270            "decl %0"                   // ECX--, the length of the string is obtained.
271            : "=c" (__res) : "D" (s), "a" (0), "0" (0xffffffff) : "di");
272     return __res;                      // Returns the string length.
273 }
274
// a temporary string pointer, used to store the pointer to the parsed string (s) below.
275 extern char * \_\_strtok;                // string token.
276
///// Using the characters in string2, string1 is segmented into a token sequence.
// String1 is considered to be a sequence containing zero or more tokens and separated by one
// or more characters in the separator string2. When strtok() is called for the first time,
// a pointer to the first char of the first token in string1 is returned, and a null character
// is written to the separator when the token is returned. Subsequent calls to strtok() with
// null as the first argument will continue to scan string1 in this way until there is no token.
// The split string2 can be different during different invocations.
// Params: s - the string1 to be processed, ct - the string2 containing each separator.
// %0=ebx(__res), %1=esi(__strtok); %2=ebx(__strtok), %3=esi(str1 ptr s), %4=(str2 ptr ct).
// Returns a token in the string s, or a null pointer if no token is found. Subsequent calls
// to strtok() with a null string s will search for the next token in the original string s.
277 extern inline char * strtok(char * s, const char * ct)
278 {
279     register char * __res __asm__("si");
280     __asm__("testl %1, %1\n\t"          // First test if ESI (str1 pointer s) is NULL.
281            "jne 1f\n\t"                // If not, means the first call, and jump label 1.
282            "testl %0, %0\n\t"          // If NULL, means subsequent call, test EBX(__strtok).
283            "je 8f\n\t"                // If EBX is NULL, it cannot be processed, jump to end.
284            "movl %0, %1\n\t"          // Copy the EBX pointer(current str1 ptr) to ESI.
285            "l:\txorl %0, %0\n\t"        // Clear EBX.
286            "movl $-1, %%ecx\n\t"       // Set ECX = 0xffffffff.
287            "xorl %%eax, %%eax\n\t"     // Clear EAX.
288            "cld\n\t"                  // Clear direction flag.
289            "movl %4, %%edi\n\t"        // Let's find the str2 length. EDI points to str2.
290            "repne\n\t"                // Compare AL(0) with char at ES:[EDI], and EDI++.
291            "scasb\n\t"                // Until end of str2 (null char), or count ECX = 0.
292            "notl %%ecx\n\t"            // Inverse ECX and subtract 1, to get str2 length.
293            "decl %%ecx\n\t"            // If str2 len is 0, go forward to label 7.
294            "je 7f\n\t"                /* empty delimiter-string */
295            "movl %%ecx, %%edx\n\t"     // Store str2 length to EDX temporarily.
296            "2:\tlodsb\n\t"             // Get str1 char DS:[ESI] to AL, and ESI++.
297            "testb %%al, %%al\n\t"      // Is the char 0 (end of str1)?
298            "je 7f\n\t"                // If yes, jump forward to label 7.
299            "movl %4, %%edi\n\t"        // EDI again points to first char of the str2.
300            "movl %%edx, %%ecx\n\t"     // Copy str2 length into the counter ECX.
301            "repne\n\t"                // Compare str1 char AL with all chars in str2,
302            "scasb\n\t"                // to check if the char is a separator.
303            "je 2b\n\t"                // If found it in str2, jump backward to label 2.

```

```

304     "decl %1\n\t" // Else the str1 pointer ESI points to the char now.
305     "cmpb $0, (%1)\n\t" // Is this a NULL character?
306     "je 7f\n\t" // If yes, jump forward to label 7.
307     "movl %1, %0\n\t" // The char pointer ESI is stored in EBX.
308     "3:\t\tlods b\n\t" // Get next char of str1 to DS:[ESI], and ESI++.
309     "testb %%al, %%al\n\t" // Is the char 0 (end of str1)?
310     "je 5f\n\t" // If yes, means str1 ends and jump forward to label 5.
311     "movl %4, %%edi\n\t" // EDI again points to first char of the str2.
312     "movl %%edx, %%ecx\n\t" // Copy str2 length into the counter ECX.
313     "repne\n\t" // Compare str1 char AL with all chars in str2,
314     "scas b\n\t" // to check if the char is a separator.
315     "jne 3b\n\t" // If not, jump to label3 to detect the next char in str1.
316     "decl %1\n\t" // If it is a separator, ESI--, points to the separator.
317     "cmpb $0, (%1)\n\t" // Is this a NULL character?
318     "je 5f\n\t" // If yes, jump forward to label 5.
319     "movb $0, (%1)\n\t" // If not, replaced the separator with a NULL char.
320     "incl %1\n\t" // ESI points to next char in str1 (remaining str).
321     "jmp 6f\n\t" // Jump forward to label 6.
322     "5:\txorl %1, %1\n\t" // Clear ESI.
323     "6:\tcmpb $0, (%0)\n\t" // Does the EBX pointer point to a NULL char?
324     "jne 7f\n\t" // If not, jump forward to label 7.
325     "xorl %0, %0\n\t" // If yes, let the EBX = NULL. (end of str1)
326     "7:\ttestl %0, %0\n\t" // EBX is NULL?
327     "jne 8f\n\t" // If not, jump forward to label 8, end return.
328     "movl %0, %1\n\t" // Set ESI to NULL, means no more token in str1.
329     "8:"
330     : "=b" (__res), "=S" (__strtok)
331     : "0" (__strtok), "1" (s), "g" (ct)
332     : "ax", "cx", "dx", "di");
333 return __res; // Returns pointer to the new token.
334 }
335
336 /// Memory copy. Copy n bytes from source address src to the destination address dest.
337 /// Params: dest - the destination address, src - the source address, n - number of bytes.
338 /// %0 - ecx(n), %1 - esi(src), %2 - edi(dest).
339 extern inline void * memcpy(void * dest, const void * src, int n)
340 {
341     __asm__( "cld\n\t" // Clear direction flag.
342             "rep\n\t" // Repeatedly copying ECX bytes,
343             "movsb" // From DS:[ESI++] to ES:[EDI++].
344             :: "c" (n), "S" (src), "D" (dest)
345             : "cx", "si", "di");
346 return dest; // Returns the dest address.
347 }
348
349 /// The memory moves. Same as memory copy operation, but consider the direction of the move.
350 /// Params: dest - the destination address, src - the source address, n - number of bytes.
351 /// If (dest < src) then: %0 - ecx(n), %1 - esi(src), %2 - edi(dest);
352 /// otherwise: %0 - ecx(n), %1 - esi(src + n - 1), %2 - edi(dest + n - 1).
353 /// This directional operation is to prevent data overlapping during copying.
354 extern inline void * memmove(void * dest, const void * src, int n)
355 {
356 if (dest < src)

```

```

349 __asm__( "cld\n\t" // Clear direction flag.
350         "rep\n\t" // From DS:[ESI++] to ES:[EDI++],
351         "movsb" // Repeatedly copying ECX bytes.
352         :: "c" (n), "S" (src), "D" (dest)
353         : "cx", "si", "di");
354 else
355 __asm__( "std\n\t" // Set the direction and start copying from the end.
356         "rep\n\t" // From DS:[ESI--] to ES:[EDI--],
357         "movsb" // Repeatedly copying ECX bytes.
358         :: "c" (n), "S" (src+n-1), "D" (dest+n-1)
359         : "cx", "si", "di");
360 return dest; // Returns the dest address.
361 }
362
363 // Compare n bytes of two memory blocks, and don't stop comparing even if encounter NULL char.
364 // Params: cs - mem block1 address, ct - mem block2 address, count - number of bytes compared.
365 // %0 - eax(__res), %1 - eax(0), %2 - edi (mem block1), %3 - esi (mem block2), %4 - ecx(count).
366 // If block1 > block2 returns 1; block1 < block2, returns -1; block1 == block2, returns 0.
367 extern inline int memcmp(const void * cs, const void * ct, int count)
368 {
369     register int __res __asm__("ax");
370     __asm__( "cld\n\t" // Compare DS:[ESI++] with ES:[EDI++].
371             "repe\n\t" // If equal, continue comparing. repeat ECX times.
372             "cmpsb\n\t" // If all the same, jump to label 1, return 0 (EAX).
373             "je 1f\n\t" // else set EAX to 1.
374             "movl $1, %%eax\n\t" // If value of mem block2 < block1, jump to label 1.
375             "jl 1f\n\t" // else invert EAX value.
376             "negl %%eax\n\t"
377             "1:"
378             : "=a" (__res) : "0" (0), "D" (cs), "S" (ct), "c" (count)
379             : "si", "di", "cx");
380 return __res; // Returns the compare result (in EAX).
381 }
382
383 // Look for the specified character in a memory block of n bytes.
384 // Params: cs - the memory block address, c - the specified char, count - the compare size.
385 // %0 - edi(__res), %1 - eax (char c), %2 - edi (mem block address cs), %3 - ecx (num of bytes).
386 // Returns the pointer to the first matching location, or NULL if not found.
387 extern inline void * memchr(const void * cs, char c, int count)
388 {
389     register void * __res __asm__("di");
390     if (!count) // If memory block size is 0, then NULL is returned.
391         return NULL;
392     __asm__( "cld\n\t" // Compare the char in AL with the ES:[EDI++],
393             "repne\n\t" // Repeat if it is not equal (up to ECX times).
394             "scasb\n\t" // If equal, jump forward to label 1.
395             "je 1f\n\t" // else set EDI to 1.
396             "movl $1, %0\n\t" // Let EDI point to the char searched. (or NULL).
397             "1:\tdecl %0"
398             : "=D" (__res) : "a" (c), "D" (cs), "c" (count)
399             : "cx");
400 return __res; // Return the char pointer.
401 }

```

```
394      ///< Fill in the memory block with the specified character.
395      ///< Fill in the memory area pointed to by 's' with the char 'c' and fill in 'count' bytes.
396      ///< %0 - eax (char c), %1 - edi (memory address s), %2 - ecx (number of bytes count).
397 extern inline void * memset(void * s, char c, int count)
398 {
399     __asm__( "cld\n\t"
400             "rep\n\t"           // Repeat ECX times,
401             "stosb"           // Fill the char in AL into ES:[EDI++].
402             :: "a" (c), "D" (s), "c" (count)
403             : "cx", "di");
404 return s;           // Returns the memory block pointer.
405 }
406 #endif
```

14.11 `termios.h`

14.11.1 Functionality

The `termios.h` file contains terminal I/O interface definitions, including `termios` data structures and some function prototypes for common terminal interface settings. These functions are used to read or set the properties of the terminal, line control, read or set the baud rate, and read or set the group ID of the terminal front end process. Although this is an early Linux header file, it is fully compliant with the current POSIX standard and has been appropriately extended.

The two terminal data structures `termio` and `termios` defined in this file belong to two types of UNIX series (or clones), `termio` is defined in AT&T system V, and `termios` is specified by POSIX standard. The two structures are basically the same, except that `termio` uses a short integer type to define the mode flag set, while `termios` uses a long integer to define the mode flag set. Since both structures are currently in use, most systems support them for compatibility. In addition, a similar `sgtty` structure has been used before, and it has not been used at present.

14.11.2 Code annotation

Program 14-10 `linux/include/termios.h`

```
1 #ifndef TERMIOS\_H
2 #define TERMIOS\_H
3
4 #include <sys/types.h>
5
6 #define TTY\_BUF\_SIZE 1024           // The buffer size in the tty (in bytes).
7
8 /* 0x54 is just a magic number to make these relatively unique ('T') */
9
10 // The following is the command set used by the tty's ioctl, which encodes it in the low byte.
```



```
// Get the information in the terminal termios structure (see tcgetattr()).
10 #define TCGETS          0x5401      // Terminal Command GET Settings.
    // Set the information in the terminal termios structure (see tcsetattr(), TCSANOW).
11 #define TCSETS          0x5402
    // Before setting the information in termios, you need to wait for all the data in the output
    // queue to be processed (used up). This command is required for modifying the parameters that
    // affect the output (see tcsetattr(), TCSADRAIN option). (TCSETSW - TCSETS Wait)
12 #define TCSETSW         0x5403
    // Before setting the termios information, you need to wait for all the data in the output queue
    // to be processed, and flush (clear) the input queue (See tcsetattr(), TCSAFLUSH option).
13 #define TCSETSF         0x5404
    // Take the attribute information in the termio structure (see tcgetattr()).
14 #define TCGETA          0x5405
    // Set the attribute information in the termio structure (see tcsetattr(), TCSANOW option).
15 #define TCSETA          0x5406
    // Before setting the attribute information of termio, you need to wait for all the data in
    // the output queue to be processed (run out). This type of command is required for modifying
    // the parameters that affect the output (see tcsetattr(), TCSADRAIN option).
16 #define TCSETAW         0x5407
    // Before setting the termio attributes, you need to wait for all data in the output queue to
    // be processed, and flush (used up) the input queue (see tcsetattr(), TCSAFLUSH option).
17 #define TCSETAF         0x5408
    // Wait for the output queue to be processed (empty). If the parameter value is 0, send a break
    // (see tcsendbreak(), tcdrain()).
18 #define TCSBRK          0x5409
    // Start/stop control. If the parameter value is 0, the output is suspended; if it is 1, the
    // pending output is re-opened; if it is 2, the input is suspended; if it is 3, the pending
    // input is re-opened (see tcflow()).
19 #define TCXONC          0x540A
    // Flushes data that has been written but has not been sent, or has been received but not yet
    // read. If the argument is 0, the input queue is flushed (cleared); if it is 1, the output
    // queue is flushed; if it is 2, the input and output queues are flushed (see tcflush()).
20 #define TCFLSH          0x540B
    // Set the terminal serial line exclusive mode.
21 #define TIOCEXCL        0x540C      // Terminal I/O Control Exclude.
    // Reset the terminal serial line exclusive mode.
22 #define TIOCNXCL        0x540D
    // Set tty as the control terminal. (TIOCNOTTY - prohibits the tty as a control terminal).
23 #define TIOCSCTTY        0x540E
    // Get the group ID of the specified terminal device process (see tcgetpgrp()).
24 #define TIOCGPGRP        0x540F      // Terminal IO Control Get PGRP.
    // Set the group ID of the specified terminal device process (see tcsetpgrp()).
25 #define TIOCSPGRP        0x5410
    // Returns the number of characters in the output queue that have not been sent.
26 #define TIOCOUTQ        0x5411
    // Simulate the terminal input. The command takes a pointer to a character as a parameter and
    // pretends that the character is typed on the terminal. The user must have superuser privileges
    // or read permission on the control terminal.
27 #define TIOCSTI          0x5412
    // Get the terminal device window size information (see the winsize structure).
28 #define TIOCGWINSZ        0x5413
    // Set the terminal device window size information (see the winsize structure).
29 #define TIOCSWINSZ        0x5414
```



```

// Returns current status bit flag set for the MODEM status control pin (see lines 185–196).
30 #define TIOCMGET      0x5415
// Set the status of the MODEM state individual control line (true or false).
31 #define TIOCMBS      0x5416
// Clear (reset) the status of the MODEM state individual control line.
32 #define TIOCMBS      0x5417
// Set the status of the MODEM state control line. If a bit is set, the status line corresponding
// to the MODEM will be set to be valid.
33 #define TIOCMSET      0x5418
// Get the software carrier detect flag (1 – on; 0 – off). For locally connected terminals or
// other devices, the software carrier flag is turned on, and it is turned off for terminals
// or devices that use MODEM lines. In order to be able to use these two ioctl calls, the tty
// line should be opened in O_NDELAY mode, so open() will not wait for the carrier.
34 #define TIOCGSOFTCAR  0x5419
// Set the software carrier detect flag (1 – on; 0 – off).
35 #define TIOCSSOFTCAR  0x541A
// Returns the number of characters in the input queue that have not been fetched.
36 #define FIONREAD      0x541B
37 #define TIOCINQ      FIONREAD
38
// Window size attribute structure. It can be used for screen-based applications. The TIOCGWINSZ
// and TIOCSWINSZ commands in ioctls can be used to read or set this information.
39 struct winsize {
40     unsigned short ws_row;        // window character rows.
41     unsigned short ws_col;        // window character columns.
42     unsigned short ws_xpixel;     // window width in pixels.
43     unsigned short ws_ypixel;     // window height in pixels.
44 };
45
// The termio structure of the AT&T system V.
46 #define NCC 8                // the size of the control character array in termio.
47 struct termio {
48     unsigned short c_iflag;        /* input mode flags */
49     unsigned short c_oflag;        /* output mode flags */
50     unsigned short c_cflag;        /* control mode flags */
51     unsigned short c_lflag;        /* local mode flags */
52     unsigned char c_line;          /* line discipline */
53     unsigned char c_cc[NCC];       /* control characters */
54 };
55
// The termios structure defined by POSIX.
56 #define NCCS 17              // the size of the control character array in termios.
57 struct termios {
58     tcflag_t c_iflag;             /* input mode flags */
59     tcflag_t c_oflag;             /* output mode flags */
60     tcflag_t c_cflag;             /* control mode flags */
61     tcflag_t c_lflag;             /* local mode flags */
62     cc_t c_line;                  /* line discipline */
63     cc_t c_cc[NCCS];              /* control characters */
64 };
65
// The following is the index of the item in the control character array c_cc[]. The initial
// value of this array is defined in include/linux/tty.h. The program can change the value in

```

```

// this array. If _POSIX_VDISABLE(\0) is defined, then when the value of an array is equal to
// _POSIX_VDISABLE, it means that the special characters in the array are prohibited.
66 /* c_cc characters */
67 #define VINTR 0 // c_cc[VINTR] = INTR (^C), \003.
68 #define VQUIT 1 // c_cc[VQUIT] = QUIT (^_), \034.
69 #define VERASE 2 // c_cc[VERASE] = ERASE (^H), \177.
70 #define VKILL 3 // c_cc[VKILL] = KILL (^U), \025.
71 #define VEOF 4 // c_cc[VEOF] = EOF (^D), \004.
72 #define VTIME 5 // c_cc[VTIME] = TIME (\0), \0. timer value (see below).
73 #define VMIN 6 // c_cc[VMIN] = MIN (\1), \1. timer value.
74 #define VSWTC 7 // c_cc[VSWTC] = SWTC (\0), \0. switch char.
75 #define VSTART 8 // c_cc[VSTART] = START (^Q), \021.
76 #define VSTOP 9 // c_cc[VSTOP] = STOP (^S), \023.
77 #define VSUSP 10 // c_cc[VSUSP] = SUSP (^Z), \032. suspend char.
78 #define VEOL 11 // c_cc[VEOL] = EOL (\0), \0.
79 #define VREPRINT 12 // c_cc[VREPRINT] = REPRINT (^R), \022.
80 #define VDISCARD 13 // c_cc[VDISCARD] = DISCARD (^O), \017.
81 #define VWERASE 14 // c_cc[VWERASE] = WERASE (^W), \027. word erase char.
82 #define VLNEXT 15 // c_cc[VLNEXT] = LNEXT (^V), \026.
83 #define VEOL2 16 // c_cc[VEOL2] = EOL2 (\0), \0.
84
// The constants of various flags used in the input mode field c_iflag in the termios.
85 /* c_iflag bits */
86 #define IGNBRK 0000001 // Ignore the BREAK condition when input.
87 #define BRKINT 0000002 // Generates a SIGINT signal on BREAK.
88 #define IGNPAR 0000004 // Ignore the character of the parity error.
89 #define PARMRK 0000010 // Mark parity error.
90 #define INPCK 0000020 // Allow input to check parity.
91 #define ISTRIP 0000040 // Mask the 8th bit of the character.
92 #define INLCR 0000100 // Map line feed NL to a carriage return CR.
93 #define IGNCR 0000200 // Ignore the CR.
94 #define ICRNL 0000400 // Map CR to NL.
95 #define IUCLC 0001000 // Convert uppercase chars to lowercase chars.
96 #define IXON 0002000 // Allow start/stop (XON/XOFF) output control.
97 #define IXANY 0004000 // Allow any character to restart the output.
98 #define IXOFF 0010000 // Allow start/stop (XON/XOFF) input control.
99 #define IMAXBEL 0020000 // Rings when the input queue is full.
100
// The constants of various flags used in the output mode field c_oflag in the termios.
101 /* c_oflag bits */
102 #define OPOST 0000001 // Perform output processing.
103 #define OLCUC 0000002 // Convert lowercase chars to uppercase chars.
104 #define ONLCR 0000004 // Map NL to CR-NL.
105 #define OCRNL 0000010 // Map CR to NL.
106 #define ONOCR 0000020 // The CR is not output in the column 0.
107 #define ONLRET 0000040 // The NL performs the function of carriage return.
108 #define OFILL 0000100 // Use fill chars when deferred instead of time delays.
109 #define OFDEL 0000200 // The fill char is DEL. The default is NULL if not set.
110 #define NLDLY 0000400 // Choose a line feed delay.
111 #define NLO 0000000 // NL delay type 0.
112 #define NL1 0000400 // NL delay type 1.
113 #define CRDLY 0003000 // Choose a carriage return delay.
114 #define CRO 0000000 // CR delay type 0.

```

```

115 #define CR1 0001000 // CR delay type 1.
116 #define CR2 0002000 // CR delay type 2.
117 #define CR3 0003000 // CR delay type 2.
118 #define TABDLY 0014000 // Select horizontal TAB delay.
119 #define TAB0 0000000 // TAB delay type 0.
120 #define TAB1 0004000 // TAB delay type 1.
121 #define TAB2 0010000 // TAB delay type 2.
122 #define TAB3 0014000 // TAB delay type 3.
123 #define XTABS 0014000 // Replace TAB with spaces, it is the number of spaces.
124 #define BSDLY 0020000 // Select the backspace (BS) delay.
125 #define BS0 0000000 // BS delay type 0.
126 #define BS1 0020000 // BS delay type 1.
127 #define VTDLY 0040000 // Select vertical tabulation delay.
128 #define VT0 0000000 // VT delay type 0.
129 #define VT1 0040000 // VT delay type 1.
130 #define FFDLY 0040000 // Select Form feed delay.
131 #define FF0 0000000 // FF delay type 0.
132 #define FF1 0040000 // FF delay type 1.
133
// The flags used in the control mode field c_cflag in the termios (octals).
134 /* c_cflag bit meaning */
135 #define CBAUD 0000017 // Transmit baud rate bit mask.
136 #define B0 0000000 /* hang up */
137 #define B50 0000001 // baud rate 50.
138 #define B75 0000002
139 #define B110 0000003
140 #define B134 0000004
141 #define B150 0000005
142 #define B200 0000006
143 #define B300 0000007
144 #define B600 0000010
145 #define B1200 0000011
146 #define B1800 0000012
147 #define B2400 0000013
148 #define B4800 0000014
149 #define B9600 0000015
150 #define B19200 0000016
151 #define B38400 0000017 // baud rate 38400.
152 #define EXTA B19200 // Extended baud rate A.
153 #define EXTB B38400 // Extended baud rate B.

154 #define CSIZE 0000060 // Character bit width mask.
155 #define CS5 0000000 // 5 bits per character.
156 #define CS6 0000020 // 6 bits.
157 #define CS7 0000040 // 7 bits.
158 #define CS8 0000060 // 8 bits.
159 #define CSTOPB 0000100 // Set two stop bits instead of one.
160 #define CREAD 0000200 // Allow to receive.
161 #define PARENB 0000400 // Enable parity operation.
162 #define PARODD 0001000 // The input check is odd check.
163 #define HUPCL 0002000 // Hang up after the last process is closed.
164 #define CLOCAL 0004000 // Ignore the MODEM control lines.
165 #define CIBAUD 03600000 /* input baud rate (not used) */

```

```

166 #define CRTSCTS 020000000000    /* flow control */
167
    // The flags used in the local mode field c_lflag in the termios.
168 /* c_lflag bits */
169 #define ISIG 0000001    // signal generated when receive INTR, QUIT, SUSP or DSUSP .
170 #define ICANON 0000002    // Enable canonical mode (cooked mode).
171 #define XCASE 0000004    // If ICANON is set, terminal displays uppercase chars.
172 #define ECHO 0000010    // Echo the input characters.
173 #define ECHOE 0000020    // If ICANON, ERASE/WERASE erase the previous char/word.
174 #define ECHOK 0000040    // If ICANON, KILL char will erase the current line.
175 #define ECHONL 0000100    // If ICANON, NL is echoed even if ECHO is not enabled.
    // The input/output queues are not flushed when the SIGINT and SIGQUIT signals are generated,
    // and the input queue is flushed when the SIGSUSP signal is generated.
176 #define NOFLSH 0000200
    // Send the SIGTTOU signal to the process group of the background process, which tries to write
    // its own control terminal.
177 #define TOSTOP 0000400
    // If ECHO is set, ASCII control characters other than TAB, NL, START, and STOP will be echoed
    // back to the '^X' pattern, and the X value is the control value +0x40.
178 #define ECHOCTL 0001000
179 #define ECHOPRT 0002000    // If ICANON & IECHO, chars will be displayed when erasing.
180 #define ECHOKE 0004000    // If ICANON, KILL is echoed by erasing each char on the line.
181 #define FLUSHO 0010000    // Output is flushed. it can be toggled by DISCARD char.
182 #define PENDIN 0040000    // All chars in inqueue are reprinted when next char is read.
183 #define TEXTEN 0100000    // Enable implementation-defined input processing.
184
185 /* modem lines */
186 #define TIOCM_LE 0x001    // Line Enable.
187 #define TIOCM_DTR 0x002    // Data Terminal Ready.
188 #define TIOCM_RTS 0x004    // Request to Send.
189 #define TIOCM_ST 0x008    // Serial Transfer.
190 #define TIOCM_SR 0x010    // Serial Receive.
191 #define TIOCM_CTS 0x020    // Clear To Send.
192 #define TIOCM_CAR 0x040    // Carrier Detect.
193 #define TIOCM_RNG 0x080    // Ring indicate.
194 #define TIOCM_DSR 0x100    // Data Set Ready.
195 #define TIOCM_CD TIOCM_CAR
196 #define TIOCM_RI TIOCM_RNG
197
198 /* tcflow() and TCXONC use these */
199 #define TCOOFF 0    // Suspend output ("Terminal Control Output OFF").
200 #define TCOON 1    // Restart suspended output.
201 #define TCIOFF 2    // Send a STOP to stop device transmitting data to system.
202 #define TCION 3    // Send a START to start device transmitting to system.
203
204 /* tcflush() and TCFLSH use these */
205 #define TCIFLUSH 0    // Clear the received data but do not read it.
206 #define TCOFLUSH 1    // Clear the output data but not transmit it.
207 #define TCIOFLUSH 2    // Clear in/out data but not read/transmit it.
208
209 /* tcsetattr uses these */
210 #define TCSANOW 0    // Enable settings now.
211 #define TCSADRAIN 1    // Change occurs after all output has been transmitted.

```

```
212 #define TCSAFLUSH          2          // Change occurs after input/output are all flushed.
213
// The following functions are implemented in the function library libc.a in the build
// environment, not in the kernel. In the library implementation, these functions are implemented
// by calling the system-call ioctl(). For ioctl() see the fs/ioctl.c program.
// Returns the receive baud rate in the termios structure referred to by termios_p.
214 extern speed_t cfgetispeed(struct termios *termios_p);
// Returns the send baud rate in the termios structure.
215 extern speed_t cfsetospeed(struct termios *termios_p);
// Set the receive baud rate in the termios structure to 'speed'.
216 extern int cfsetispeed(struct termios *termios_p, speed_t speed);
// Set the send baud rate in the termios structure.
217 extern int cfsetospeed(struct termios *termios_p, speed_t speed);
// Wait for the written data of the object pointed to by fildes to be transmitted.
218 extern int tcdrain(int fildes);
// Suspend/restart the reception/transmission data of the object pointed to by fildes.
219 extern int tcflow(int fildes, int action);
// Discards all written but not yet transmitted data for the fildes specified object, as well
// as all data that has been received but not yet read.
220 extern int tcflush(int fildes, int queue_selector);
// Get the parameters of the object corresponding to the handle fildes and save it in termios.
221 extern int tcgetattr(int fildes, struct termios *termios_p);
// If the terminal uses asynchronous serial transmission, a series of zero value bits are
// continuously transmitted for a certain period of time.
222 extern int tcsendbreak(int fildes, int duration);
// Use the data of the termios structure to set the parameters related to the terminal.
223 extern int tcsetattr(int fildes, int optional_actions,
224                     struct termios *termios_p);
225
226 #endif
227
```

14.11.3 Information

14.11.3.1 Control Character TIME、MIN

When the canonical mode flag ICANON is disabled, the input is in the non-canonical mode (raw mode). In non-canonical mode, input characters are not processed into lines, and input characters are immediately readable, without the need to enter (user type) carriage-return, line feed, etc. line-defining characters. Therefore, erasing and terminating processing will not occur. The settings of MIN (`c_cc[VMIN]`) and TIME (`c_cc[VTIME]`) in termios structure are used to determine how to handle the received characters, how many characters are read by the associated `read()` system-call, and when to return to the user program.

MIN represents the minimum amount of characters that need to be read when a read operation is satisfied (ie, when the character is returned to the user). TIME is a timing value counted in 1/10 second for timeout values for burst and short-term data transfers. The four combinations of these two control characters and their interactions are described as follows:

◆ MIN > 0, TIME > 0:

In this case, TIME acts as a timer between characters and starts to function after receiving the first character. It will be reset and restarted every time it receives a character. The interaction between MIN and

TIME is as follows: Once a character is received, the inter-character timer begins to work. If MIN characters are received before the timer expires (note that the timer restarts every time a character is received), the read operation is satisfied. If the timer expires before the MIN characters are received, the characters that have been received at this time are returned to the user. Note that if TIME times out, at least one character will be returned because the timer will only start functioning after receiving a character. So in this case ($\text{MIN} > 0$, $\text{TIME} > 0$), the read operation will sleep until the first character is received to activate the MIN and TIME mechanisms. If the number of characters read is less than the number of existing characters, the timer will not be reactivated and subsequent read operations will be satisfied immediately.

◆ $\text{MIN} > 0$, $\text{TIME} = 0$:

In this case, since the value of TIME is 0, the timer does not work and only MIN makes sense. Only when the MIN characters are received, the waiting read operation will be satisfied (the waiting operation will sleep until MIN characters are received). Programs that use this to read the record-based terminal IO will be blocked (arbitrarily) indefinitely in the read operation.

◆ $\text{MIN} = 0$, $\text{TIME} > 0$:

In this case, since $\text{MIN}=0$, TIME no longer functions as a timer between characters, but a read operation timer and functions at the beginning of the read operation. The read operation is satisfied as soon as a character is received or the timer expires. Note that in this case, if the timer expires, no characters will be read. If the timer does not time out, the read operation will only be satisfied after reading one character. Therefore, in this case, the read operation is not blocked indefinitely (indeterminately) to wait for characters. After the start of the read operation, if no characters are received within $\text{TIME} \times 0.10$ seconds, the read operation will return with 0 characters.

◆ $\text{MIN} = 0$, $\text{TIME} = 0$:

In this case, the read operation will return immediately. The minimum number of characters requested to be read or the number of existing characters in the buffer queue will be returned without waiting for more characters to be input into the buffer.

In general, in non-canonical mode, these two values are timeout timing values and character count values. MIN indicates the minimum number of characters that need to be read in order to satisfy the read operation. TIME is a timing value counted by one tenth of a second. When both are set, the read operation will wait until at least one character is read, then return after reading the MIN characters or return the read character due to the TIME timeout. If only MIN is set, the read operation will not return until the MIN characters are read. If only TIME is set, the read will return immediately after reading at least one character or timing out. If neither is set, the read will return immediately, giving only the number of bytes currently read.

14.12 time.h

14.12.1 Functionality

The time.h header file refers to functions that process time and date. There is a very interesting description of time in MINIX: “Time processing is more complicated, such as what is GMT (Greenwich Mean Time, now UTC time), local time or other time. Even though Bishop Ussher (1581-1656) once calculated, according to the Bible, the world begins at 9 am on October 12, 4004 BC. But in the UNIX world, time began at midnight on January 1, 1970, GMT, before it was empty and chaotic”. Where UTC means universal time code.

This file is one of the header files in the standard C library. Since some of the UNIX operating system developers were amateur astronomy enthusiasts at the time, they were particularly strict with the time representation in UNIX systems, so that time and date representations and calculations are particularly complicated in UNIX-like or in standard C-compatible systems. This file defines one constant symbol (macro), four types, and some time and date operation conversion functions. In addition, some of the function declarations given in this file are functions provided by the standard C library, which are not included in the kernel.

In the Linux 0.12 kernel, this file mainly provides the `tm` structure type for the `init/main.c` and `kernel/mktime.c` files, which is used by the kernel to obtain real-time clock information (calendar time) from the system CMOS chip, so that the system boot time can be set. The boot time is the time (in seconds) elapsed since 0:00 on January 1, 1970, and will be stored in the global variable `startup_time` for all code reading by the kernel.

14.12.2 Code annotation

Program 14-11 linux/include/time.h

```

1  #ifndef TIME_H
2  #define TIME_H
3
4  #ifndef TIME_T
5  #define TIME_T
6  typedef long time_t;           // The time from GMT on January 1, 1970 at midnight.
7  #endif
8
9  #ifndef SIZE_T
10 #define SIZE_T
11 typedef unsigned int size_t;
12 #endif
13
14 #ifndef NULL
15 #define NULL ((void *) 0)
16 #endif
17
18 #define CLOCKS_PER_SEC 100      // System clock tick frequency, 100HZ.
19
20 typedef long clock_t;          // Ticks passed by system from the start of a process.
21
22 struct tm {
23     int tm_sec;                  // Seconds [0, 59].
24     int tm_min;                  // Minutes [0, 59].
25     int tm_hour;                 // Hours [0, 59].
26     int tm_mday;                 // The number of days in a month [0, 31].
27     int tm_mon;                  // The number of months in a year [0, 11].
28     int tm_year;                 // The number of years since 1900.
29     int tm_wday;                 // One day of the week [0, 6] (Sunday =0).
30     int tm_yday;                 // One day in a year [0, 365].
31     int tm_isdst;                // Summer time sign. > 0 -in use; = 0 -not used; < 0 -invalid.
32 };
33
34 // A macro that checks to see if it is a leap year.
35 #define isleap(year) \

```

```
35 ((year) % 4 == 0 && ((year) % 100 != 0 || (year) % 1000 == 0))
36
// Here are some function prototypes for time operations.
// Get processor usage time. Returns an appr.processor time (ticks) used by the program.
37 clock_t clock(void);
// Get time. Returns the number of seconds since 1970.1.1:0:0:0 (the calendar time).
38 time_t time(time_t * tp);
// Calculate the time difference. Returns the number of seconds elapsed between time2 and time1.
39 double difftime(time_t time2, time_t time1);
// Convert the time represented by the tm structure to calendar time.
40 time_t mktime(struct tm * tp);
41
// Converts the time in tm structure into a string and returns a pointer to the string.
42 char * asctime(const struct tm * tp);
// Convert the calendar time to a string, such as "Wed Jun 30 21:49:08:1993\n".
43 char * ctime(const time_t * tp);
// Convert the calendar time to UTC time represented by the tm structure.
44 struct tm * gmtime(const time_t *tp);
// Converts the calendar time to the time in specified time zone represented by tm structure.
45 struct tm *localtime(const time_t * tp);
// The time represented by the tm structure is converted to a string of maximum length 'smax'
// using the format string 'fmt' and the result is stored in 's'.
46 size_t strftime(char * s, size_t smax, const char * fmt, const struct tm * tp);
// Initialize the time conversion information and initialize the zname variable using the
// environment variable TZ. This function is called automatically in the time conversion function
// associated with the time zone.
47 void tzset(void);
48
49 #endif
50
```

14.13 unistd.h

14.13.1 Functionality

Standard symbol constants and types are defined in the unistd.h header file. There are many different symbol constants and types defined in this file, as well as some function declarations.如 If the symbol `__LIBRARY__` is defined in the program, it will also include the kernel system-call number and the inline assembly macro `_syscall0()`, `_syscall1()`, and so on.

14.13.2 Code annotation

Program 14-12 linux/include/unistd.h

```
1 #ifndef UNISTD_H
2 #define UNISTD_H
3
4 /* ok, this may be a joke, but I'm working on it */
// The symbol constant below indicates the version of the IEEE Standard 1003.1 implementation
// that the kernel follows, which is an integer value.
```



```

5 #define POSIX_VERSION 198808L
6
7 #define POSIX_CHOWN_RESTRICTED /* only root can do a chown (I think..) */
  // Path names longer than NAME_MAX will generate an error and will not be automatically truncated.
8 #define POSIX_NO_TRUNC /* no pathname truncation (but see in kernel) */
  // _POSIX_VDISABLE is used to control the function of some special characters of the terminal.
  // When a character code value of the c_cc[] array in a terminal termios structure is equal
  // to the value of _POSIX_VDISABLE, it means that the corresponding special character function
  // is prohibited.
9 #define POSIX_VDISABLE '\0' /* character to disable things like ^C */
  // Indicates that the system implementation supports job control.
10 #define _POSIX_JOB_CONTROL
  // Each process has a saved set-user-ID and a saved set-group-ID.
11 #define POSIX_SAVED_IDS /* Implemented, for whatever good it is */
12
13 #define STDIN_FILENO 0 // Standard input file handle (descriptor) number.
14 #define STDOUT_FILENO 1 // Standard output file handle number.
15 #define STDERR_FILENO 2 // Standard error output file handle number.
16
17 #ifndef NULL
18 #define NULL ((void *)0) // Define the value of a null pointer.
19 #endif
20
  // The symbol constants defined below are used for the access() function.
21 /* access */
22 #define F_OK 0 // Check if the file exists.
23 #define X_OK 1 // Check if it is executable (searchable).
24 #define W_OK 2 // Check if it is writable.
25 #define R_OK 4 // Check if it is readable.
26
  // The following symbol constants are used for the lseek() and fcntl() functions.
27 /* lseek */
28 #define SEEK_SET 0 // Set file read/write pointer to the offset.
29 #define SEEK_CUR 1 // Set to the current value plus the offset.
30 #define SEEK_END 2 // Set to the file length plus the offset.
31
  // The following symbol constants are used in the sysconf() function.
32 /* _SC stands for System Configuration. We don't use them much */
33 #define SC_ARG_MAX 1 // The maximum number of arguments.
34 #define SC_CHILD_MAX 2 // The maximum number of child processes.
35 #define SC_CLOCKS_PER_SEC 3 // Ticks per second.
36 #define SC_NGROUPS_MAX 4 // The maximum number of groups.
37 #define SC_OPEN_MAX 5 // The maximum number of opened files.
38 #define SC_JOB_CONTROL 6 // Job control.
39 #define SC_SAVED_IDS 7 // The saved identifier.
40 #define SC_VERSION 8 // Version.
41
  // The following symbol constants are used for the pathconf() function.
42 /* more (possibly) configurable things - now pathnames */
43 #define PC_LINK_MAX 1 // The maximum number of links.
44 #define PC_MAX_CANON 2 // The maximum number of regular files.
45 #define PC_MAX_INPUT 3 // Maximum input length.
46 #define PC_NAME_MAX 4 // The maximum length of the name.

```

```

47 #define PC_PATH_MAX          5    // The maximum length of a path.
48 #define PC_PIPE_BUF          6    // Pipe buffer size.
49 #define PC_NO_TRUNC           7    // The file name is not truncated.
50 #define PC_VDISABLE           8    // The specified control char is disabled.
51 #define PC_CHOWN_RESTRICTED   9    // Change the owner is restricted.
52
// <sys/stat.h> File status header file. Contains file or file system state structures stat{}
// and constants.
// <sys/time.h> The timeval structure and the itimerval structure are defined.
// <sys/times.h> Defines the running time structure tms and the times() function prototype in
// the process.
// <sys/utsname.h> System name structure header file.
// <sys/resource.h> Resource file. Contains information on the limits and utilization of system
// resources used by processes.
// <utime.h> User time header file. The access and modification time structures and the utime()
// prototype are defined.
53 #include <sys/stat.h>
54 #include <sys/time.h>
55 #include <sys/times.h>
56 #include <sys/utsname.h>
57 #include <sys/resource.h>
58 #include <utime.h>
59
60 #ifdef __LIBRARY__
61
// The following is the system-call symbol constants implemented by the kernel, used as an index
// in the system-call function table (see include/linux/sys.h).
62 #define NR_setup              0    /* used only by init, to get system going */
63 #define NR_exit               1
64 #define NR_fork               2
65 #define NR_read               3
66 #define NR_write              4
67 #define NR_open               5
68 #define NR_close              6
69 #define NR_waitpid            7
70 #define NR_creat              8
71 #define NR_link               9
72 #define NR_unlink             10
73 #define NR_execve             11
74 #define NR_chdir              12
75 #define NR_time               13
76 #define NR_mknod              14
77 #define NR_chmod              15
78 #define NR_chown              16
79 #define NR_break              17
80 #define NR_stat               18
81 #define NR_lseek              19
82 #define NR_getpid             20
83 #define NR_mount              21
84 #define NR_umount             22
85 #define NR_setuid             23
86 #define NR_getuid             24
87 #define NR_stime              25

```

88	<code>#define</code>	NR_ptrace	26
89	<code>#define</code>	NR_alarm	27
90	<code>#define</code>	NR_fstat	28
91	<code>#define</code>	NR_pause	29
92	<code>#define</code>	NR_utime	30
93	<code>#define</code>	NR_stty	31
94	<code>#define</code>	NR_gtty	32
95	<code>#define</code>	NR_access	33
96	<code>#define</code>	NR_nice	34
97	<code>#define</code>	NR_ftime	35
98	<code>#define</code>	NR_sync	36
99	<code>#define</code>	NR_kill	37
100	<code>#define</code>	NR_rename	38
101	<code>#define</code>	NR_mkdir	39
102	<code>#define</code>	NR_rmdir	40
103	<code>#define</code>	NR_dup	41
104	<code>#define</code>	NR_pipe	42
105	<code>#define</code>	NR_times	43
106	<code>#define</code>	NR_prof	44
107	<code>#define</code>	NR_brk	45
108	<code>#define</code>	NR_setgid	46
109	<code>#define</code>	NR_getgid	47
110	<code>#define</code>	NR_signal	48
111	<code>#define</code>	NR_geteuid	49
112	<code>#define</code>	NR_getegid	50
113	<code>#define</code>	NR_acct	51
114	<code>#define</code>	NR_phys	52
115	<code>#define</code>	NR_lock	53
116	<code>#define</code>	NR_ioctl	54
117	<code>#define</code>	NR_fcntl	55
118	<code>#define</code>	NR_mpx	56
119	<code>#define</code>	NR_setpgid	57
120	<code>#define</code>	NR_ulimit	58
121	<code>#define</code>	NR_uname	59
122	<code>#define</code>	NR_umask	60
123	<code>#define</code>	NR_chroot	61
124	<code>#define</code>	NR_ustat	62
125	<code>#define</code>	NR_dup2	63
126	<code>#define</code>	NR_getppid	64
127	<code>#define</code>	NR_getpgrp	65
128	<code>#define</code>	NR_setsid	66
129	<code>#define</code>	NR_sigaction	67
130	<code>#define</code>	NR_sgetmask	68
131	<code>#define</code>	NR_ssetmask	69
132	<code>#define</code>	NR_setreuid	70
133	<code>#define</code>	NR_setregid	71
134	<code>#define</code>	NR_sigsuspend	72
135	<code>#define</code>	NR_sigpending	73
136	<code>#define</code>	NR_sethostname	74
137	<code>#define</code>	NR_setrlimit	75
138	<code>#define</code>	NR_getrlimit	76
139	<code>#define</code>	NR_getusage	77
140	<code>#define</code>	NR_gettimeofday	78

```

141 #define NR_settimeofday 79
142 #define NR_getgroups 80
143 #define NR_setgroups 81
144 #define NR_select 82
145 #define NR_symlink 83
146 #define NR_lstat 84
147 #define NR_readlink 85
148 #define NR_uselib 86
149
// The following defines the system-call embedded assembly macro function.
// The system-call macro function with no arguments: type name(void).
// %0 - eax(__res), %1 - eax(__NR_##name). Where name is the name of a system-call, combined
// with __NR_ to form the above system-call symbol constant, which is used to address the
// function pointer in the system call table. In the macro definition, if there are two
// consecutive well signs '##' between the two symbols, it means that the two symbols are
// connected together to form a new symbol when the macro is replaced. For example, '__NR_##name'
// on line 156 below, after replacing the parameter 'name' (for example, 'fork'), the last
// occurrence in the program will be the symbol '__NR_fork'.
// Returns: if the return value is greater than or equal to 0, the value is returned, otherwise
// the error number errno is set and -1 is returned.
150 #define syscall0(type,name) \
151 type name(void) \
152 { \
153 long __res; \
154 __asm__ volatile ("int $0x80" \           // system interrupt 0x80.
155                  : "=a" (__res) \       // The return value is placed in eax(__res).
156                  : "0" (__NR_##name)); \ // The input is system interrupt number __NR_name.
157 if (__res >= 0) \           // If value >=0, the value is returned directly.
158     return (type) __res; \
159 errno = -__res; \          // Otherwise set error number and return -1.
160 return -1; \
161 }
162
// A system-call macro function with 1 parameter: type name(atype a)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a).
163 #define syscall1(type,name,atype,a) \
164 type name(atype a) \
165 { \
166 long __res; \
167 __asm__ volatile ("int $0x80" \
168                  : "=a" (__res) \
169                  : "0" (__NR_##name), "b" ((long) (a))); \
170 if (__res >= 0) \
171     return (type) __res; \
172 errno = -__res; \
173 return -1; \
174 }
175
// A system call macro function with 2 parameters: type name(atype a, btype b)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b).
176 #define syscall2(type,name,atype,a,btype,b) \
177 type name(atype a,btype b) \
178 { \

```

```

179 long __res; \
180 __asm__ volatile ("int $0x80" \
181     : "=a" (__res) \
182     : "0" (__NR_##name), "b" ((long) (a)), "c" ((long) (b))); \
183 if (__res >= 0) \
184     return (type) __res; \
185 errno = -__res; \
186 return -1; \
187 }
188
// A system call macro function with 3 parameters: type name(atype a, btype b, ctype c)
// %0 - eax(__res), %1 - eax(__NR_name), %2 - ebx(a), %3 - ecx(b), %4 - edx(c).
// The system call of this kernel has up to three parameters. If there is more data, you can
// put the data in a buffer and pass the buffer pointer as a parameter to the kernel.
189 #define __syscall3(type, name, atype, a, btype, b, ctype, c) \
190 type name(atype a, btype b, ctype c) \
191 { \
192     long __res; \
193     __asm__ volatile ("int $0x80" \
194         : "=a" (__res) \
195         : "0" (__NR_##name), "b" ((long) (a)), "c" ((long) (b)), "d" ((long) (c))); \
196     if (__res >= 0) \
197         return (type) __res; \
198     errno = -__res; \
199     return -1; \
200 }
201
202 #endif /* __LIBRARY__ */
203
204 extern int errno;                // Error number, a global variable.
205
// The following are the function prototype definitions for each system-call.
// see include/linux/sys.h for details.
206 int access(const char * filename, mode_t mode);
207 int acct(const char * filename);
208 int alarm(int sec);
209 int brk(void * end_data_segment);
210 void * sbrk(ptrdiff_t increment);
211 int chdir(const char * filename);
212 int chmod(const char * filename, mode_t mode);
213 int chown(const char * filename, uid_t owner, gid_t group);
214 int chroot(const char * filename);
215 int close(int fildes);
216 int creat(const char * filename, mode_t mode);
217 int dup(int fildes);
218 int execve(const char * filename, char ** argv, char ** envp);
219 int execl(const char * pathname, char ** argv);
220 int execlp(const char * file, char ** argv);
221 int execl(const char * pathname, char * arg0, ...);
222 int execlp(const char * file, char * arg0, ...);
223 int execl(const char * pathname, char * arg0, ...);
// The keyword 'volatile' before the function name is used to tell the compiler gcc that the
// function will not return. This will allow gcc to produce better code. More importantly, using

```

```

// this keyword avoids generating false warnings for certain (uninitialized variables). This
// is equivalent to gcc's function attribute description:
// void do_exit(int error_code) __attribute__((noreturn));
224 volatile void exit(int status);
225 volatile void \_exit(int status);
226 int fcntl(int fildes, int cmd, ...);
227 int fork(void);
228 int getpid(void);
229 int getuid(void);
230 int geteuid(void);
231 int getgid(void);
232 int getegid(void);
233 int ioctl(int fildes, int cmd, ...);
234 int kill(pid\_t pid, int signal);
235 int link(const char * filename1, const char * filename2);
236 int lseek(int fildes, off\_t offset, int origin);
237 int mknod(const char * filename, mode\_t mode, dev\_t dev);
238 int mount(const char * specialfile, const char * dir, int rwflag);
239 int nice(int val);
240 int open(const char * filename, int flag, ...);
241 int pause(void);
242 int pipe(int * fildes);
243 int read(int fildes, char * buf, off\_t count);
244 int setpgrp(void);
245 int setpgid(pid\_t pid, pid\_t pgid);
246 int setuid(uid\_t uid);
247 int setgid(gid\_t gid);
248 void (*signal(int sig, void (*fn)(int)))(int);
249 int stat(const char * filename, struct stat * stat_buf);
250 int fstat(int fildes, struct stat * stat_buf);
251 int stime(time\_t * tptr);
252 int sync(void);
253 time\_t time(time\_t * tloc);
254 time\_t times(struct tms * tbuf);
255 int ulimit(int cmd, long limit);
256 mode\_t umask(mode\_t mask);
257 int umount(const char * specialfile);
258 int uname(struct utsname * name);
259 int unlink(const char * filename);
260 int ustat(dev\_t dev, struct ustat * ubuf);
261 int utime(const char * filename, struct utimbuf * times);
262 pid\_t waitpid(pid\_t pid, int * wait_stat, int options);
263 pid\_t wait(int * wait_stat);
264 int write(int fildes, const char * buf, off\_t count);
265 int dup2(int oldfd, int newfd);
266 int getppid(void);
267 pid\_t getpgrp(void);
268 pid\_t setsid(void);
269 int sethostname(char *name, int len);
270 int setrlimit(int resource, struct rlimit *rlp);
271 int getrlimit(int resource, struct rlimit *rlp);
272 int getrusage(int who, struct rusage *rusage);
273 int gettimeofday(struct timeval *tv, struct timezone *tz);

```

```
274 int settimeofday(struct timeval *tv, struct timezone *tz);
275 int getgroups(int gidsetlen, gid\_t *gidset);
276 int setgroups(int gidsetlen, gid\_t *gidset);
277 int select(int width, fd\_set *readfds, fd\_set *writefds,
278          fd\_set *exceptfds, struct timeval *timeout);
279
280 #endif
281
```

14.14 utime.h

14.14.1 Functionality

The utime.h file defines the file access and modification time structure `utimbuf{ }`, and the `utime()` function prototype, where time is in seconds.





14.14.2 Code annotation

Program 14-13 linux/include/utime.h

```
1 #ifndef UTIME\_H
2 #define UTIME\_H
3
4 // <sys/types.h> Type header file. The basic system data types and file system parameter
5 // structure are defined.
6 #include <sys/types.h> /* I know - shouldn't do this, but .. */
7
8 struct utimbuf {
9     time\_t actime;           // File access time. seconds from 1970.1.1:0:0:0.
10    time\_t modtime;          // File modified time. seconds from 1970.1.1:0:0:0.
11 };
12
13 // Function to set file access and modify time.
14 extern int utime(const char *filename, struct utimbuf *times);
15
16 #endif
17
```

14.15 Files in the include/asm/ directory

List 14-2 linux/include/asm/目录下的文件

	Filename	Size	Last Modified Time (GMT)	Description
	io.h	477 bytes	1991-08-07 10:17:51	
	memory.h	507 bytes	1991-06-15 20:54:44	
	segment.h	1366 bytes	1991-11-25 18:48:24	
	system.h	1707 bytes	1992-01-13 13:02:10	

14.16 io.h

14.16.1 Functionality

The embedded assembly macro functions that access the hardware IO port are defined in the io.h file: outb(), inb(), and outb_p() and inb_p(). The main difference between the first two functions and the latter two is that the jmp instruction is used in the latter code for time delay.

14.16.2 Code annotation

Program 14-14 linux/include/asm/io.h

```

1  /// Hardware port byte output function.
2  // Parameters: value - the byte of the output; port - the port.
3  1 #define outb(value, port) \
4  2 __asm__ ("outb %%al, %%dx":: "a" (value), "d" (port))
5
6  /// Hardware port byte input function.
7  // Returns the input byte.
8  5 #define inb(port) ({ \
9  6 unsigned char _v; \
10 7 __asm__ volatile ("inb %%dx, %%al":: "=a" (_v): "d" (port)); \
11 8 _v; \
12 9 })
13
14  /// Hardware port byte output function with delay. Use two jump statements to delay for a while.
15  // Parameters: value - the byte to be exported; port - the port.
16  11 #define outb_p(value, port) \
17  12 __asm__ ("outb %%al, %%dx\n" \
18  13          "\tjmp 1f\n" \           // Jump forward to label 1 (the next statement).
19  14          "1: \tjmp 1f\n" \       // Jump forward to label 1.
20  15          "1: :: "a" (value), "d" (port))
21  16

```

```

    /// Hardware port byte input function with delay. Use two jump statements to delay for a while.
    // Returns the input byte.
17 #define inb_p(port) ({ \
18     unsigned char _v; \
19     __asm__ volatile ("inb %%dx, %%al\n" \
20         "\tjmp 1f\n" \           // Jump forward to label 1 (the next statement).
21         "1:\tjmp 1f\n" \       // Jump forward to label 1.
22         "1:": "=a" (_v): "d" (port)); \
23     _v; \
24 })
25

```

14.17 memory.h

14.17.1 Functionality

The memory.h file contains a memory copy embedded assembly macro function `memcpy()`. It is identical to `memcpy()` defined in `string.h`, except that the latter is defined in the form of an embedded assembly C function.

14.17.2 Code annotation

Program 14-15 linux/include/asm/memory.h

```

1  /*
2  *  NOTE!!! memcpy(dest,src,n) assumes ds=es=normal data segment. This
3  *  goes for all kernel functions (ds=es=kernel space, fs=local data,
4  *  gs=null), as well as for all well-behaving user programs (ds=es=
5  *  user data space). This is NOT a bug, as any user program that changes
6  *  es deserves to die if it isn't careful.
7  */
    /// Memory block copy. Copy n bytes from source address src to the destination dest.
    // %0 - edi (address dest), %1 - esi (address src), %2 - ecx (number of bytes n).
8  #define memcpy(dest,src,n) ({ \
9      void * _res = dest; \
10     __asm__ ("cld;rep;movsb" \           // Copy from DS:[ESI++] to ES:[EDI++], total ECX(n) bytes.
11         :: "D" ((long)(_res)), "S" ((long)(src)), "c" ((long)(n)) \
12         : "di", "si", "cx"); \
13     _res; \
14 })
15

```

14.18 segment.h

14.18.1 Functionality

The segment.h file defines some functions that access the Intel CPU segment registers, as well as some memory manipulation functions associated with the segment registers. In a Linux system, when the user program starts executing kernel code through a system-call, the kernel code first loads the kernel data segment descriptor (segment value 0x10) in the GDT into the registers DS and ES, that is, DS and ES are used for access the kernel data segment; and the task data segment descriptor (segment value 0x17) in the LDT is loaded into the FS, that is, the FS is used to access the user data segment. See lines 92--96 in kernel/sys_call.s. Therefore, when executing kernel code, you need to use a special method to access the data in the user program (task). Functions such as `get_fs_byte()` and `put_fs_byte()` in this file are specifically used to access data in the user program.

14.18.2 Code annotation

Program 14-16 linux/include/asm/segment.h

```

1  /// Get a byte at the specified address in the FS segment.
2  // Parameters: addr - the specified memory address.
3  // %0 - (returned byte _v); %1 - (memory address addr).
4  // Returns a byte at memory FS:[addr].
5  extern inline unsigned char get\_fs\_byte(const char * addr)
6  {
7      unsigned register char _v;    // a register variable for efficient access.
8
9      __asm__ ("movb %%fs:%1, %0" : "=r" (_v) : "m" (*addr));
10     return _v;
11 }
12
13 /// Get a word at the specified address in the FS segment.
14 // %0 - (returned word _v); %1 - (memory address addr).
15 // Returns a word at memory FS:[addr].
16 extern inline unsigned short get\_fs\_word(const unsigned short *addr)
17 {
18     unsigned short _v;
19
20     __asm__ ("movw %%fs:%1, %0" : "=r" (_v) : "m" (*addr));
21     return _v;
22 }
23
24 /// Get a long word at the specified address in the FS segment.
25 // %0 - (returned long word _v); %1 - (memory address addr).
26 // Returns a long word at memory FS:[addr].
27 extern inline unsigned long get\_fs\_long(const unsigned long *addr)
28 {
29     unsigned long _v;
30
31     __asm__ ("movl %%fs:%1, %0" : "=r" (_v) : "m" (*addr)); \

```

```

22         return _v;
23     }
24
25     /// Put a byte at the specified memory address in the FS segment.
26     // Parameters: val - the byte value; addr - the memory address.
27     // %0 - register (byte value val); %1 - (memory address addr).
28     extern inline void put\_fs\_byte(char val, char *addr)
29     {
30         __asm__ ("movb %0, %%fs:%1":: "r" (val), "m" (*addr));
31     }
32
33     /// Put a word at th specified memory address in the FS segment.
34     // Parameters: val - the word value; addr - the memory address.
35     // %0 - register (word value val); %1 - (memory address addr).
36     extern inline void put\_fs\_word(short val, short * addr)
37     {
38         __asm__ ("movw %0, %%fs:%1":: "r" (val), "m" (*addr));
39     }
40
41     /// Put a long word at th specified memory address in the FS segment.
42     // Parameters: val - the long word value; addr - the memory address.
43     // %0 - register (long word value val); %1 - (memory address addr).
44     extern inline void put\_fs\_long(unsigned long val, unsigned long * addr)
45     {
46         __asm__ ("movl %0, %%fs:%1":: "r" (val), "m" (*addr));
47     }
48
49     /*
50     * Someone who knows GNU asm better than I should double check the followig.
51     * It seems to work, but I don't know if I'm doing something subtly wrong.
52     * --- TYT, 11/24/91
53     * [ nothing wrong here, Linus ]
54     */
55
56     /// Get FS segment register value (selector).
57     extern inline unsigned long get\_fs()
58     {
59         unsigned short _v;
60         __asm__ ("mov %%fs, %%ax":: "=a" (_v));
61         return _v;
62     }
63
64     /// Get DS segment register value.
65     extern inline unsigned long get\_ds()
66     {
67         unsigned short _v;
68         __asm__ ("mov %%ds, %%ax":: "=a" (_v));
69         return _v;
70     }
71
72     /// Set FS segment register.
73     extern inline void set\_fs(unsigned long val)
74     {

```

```

63     __asm__ ("mov %0, %%fs"::"a" ((unsigned short) val));
64 }
65
66

```

14.19 system.h

14.19.1 Functionality

The system.h file contains embedded assembly macros that set or modify segment descriptors/interrupt gate descriptors. Among them, the function `move_to_user_mode()` is used for the kernel to manually switch (move) to the initial process (task 0) when the kernel initialization ends, that is, to run from the code of the privilege level 0 code to the privilege level 3. The method used is to simulate the interrupt call return process, that is, use the IRET instruction to implement the privilege level change and the stack switch, thereby moving the CPU execution control flow to the environment of the initial task 0, as shown in Figure 14-3.

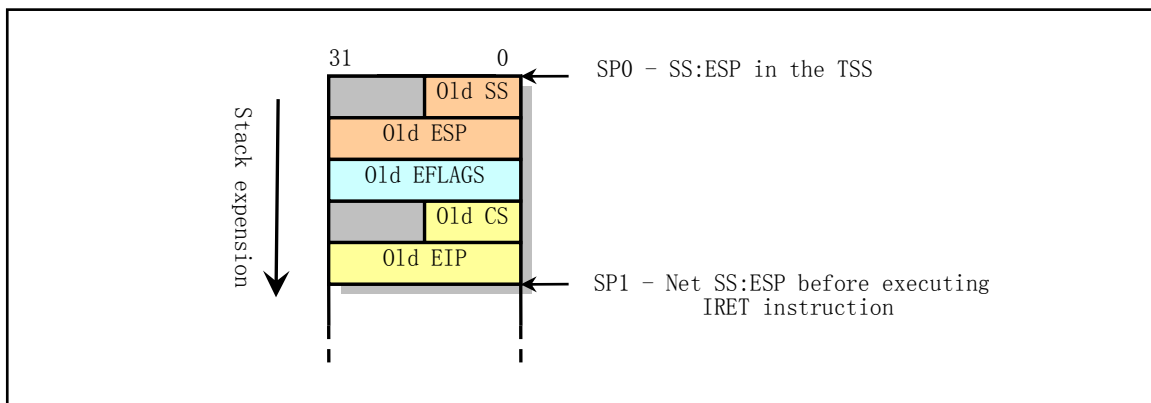


Figure 14-3 Using IRET instruction to implement privilege level change

The use of this method for privilege level changes and control transfer is caused by the CPU protection mechanism. By call gate, interrupt or trap gate, the CPU allows low-level (such as privilege level 3) code to call or transfer control to high-level code, but not vice versa. So the kernel uses the method of simulating IRET instruction returns to low-level code from a high level.

Before executing the task 0 code, the kernel first sets up the stack to simulate the content placement in the stack when the privilege level switch has just entered the interrupt call procedure. The IRET instruction is then executed, causing the system to move to task 0 for execution. When the IRET instruction is executed, the contents of the stack are as shown in Figure 14-3, at which point the stack pointer is ESP1. The stack of task 0 is the stack of the kernel. After the IRET is executed, it is moved to task 0 and executed. Since the task 0 descriptor privilege level is 3, the SS:ESP on the stack will also be popped up. So after IRET, ESP is equal to ESP0. Note that the interrupt return instruction IRET here does not cause the CPU to perform the task switching operation because the flag NT has been reset in `sched_init()` before executing this function. Executing the IRET instruction while NT is in the reset state does not cause the CPU to perform a task switching operation. It can be seen that the start of task 0 is purely manual.

Task 0 is a special process whose data segment and code segment map directly to the kernel code and data

space, that is, 640K memory space starting from physical address 0, and its stack address is the stack used by the kernel code. Therefore, the original SS and the original ESP in the stack in the figure are formed by directly pushing the existing kernel stack pointer onto the stack.

Another part of the system.h file gives macros that set different types of descriptor entries in the interrupt descriptor table IDT. `_set_gate()` is a multi-parameter macro that is a generic macro that is called by the macro `set_intr_gate()` that sets the interrupt gate descriptor, the macro `set_trap_gate()` that sets the trap gate descriptor, and the macro `set_system_gate()` that sets the system gate descriptor. The format of the Interrupt Gate and Trap Gate descriptor entries in the IDT table is shown in Figure 14-4.

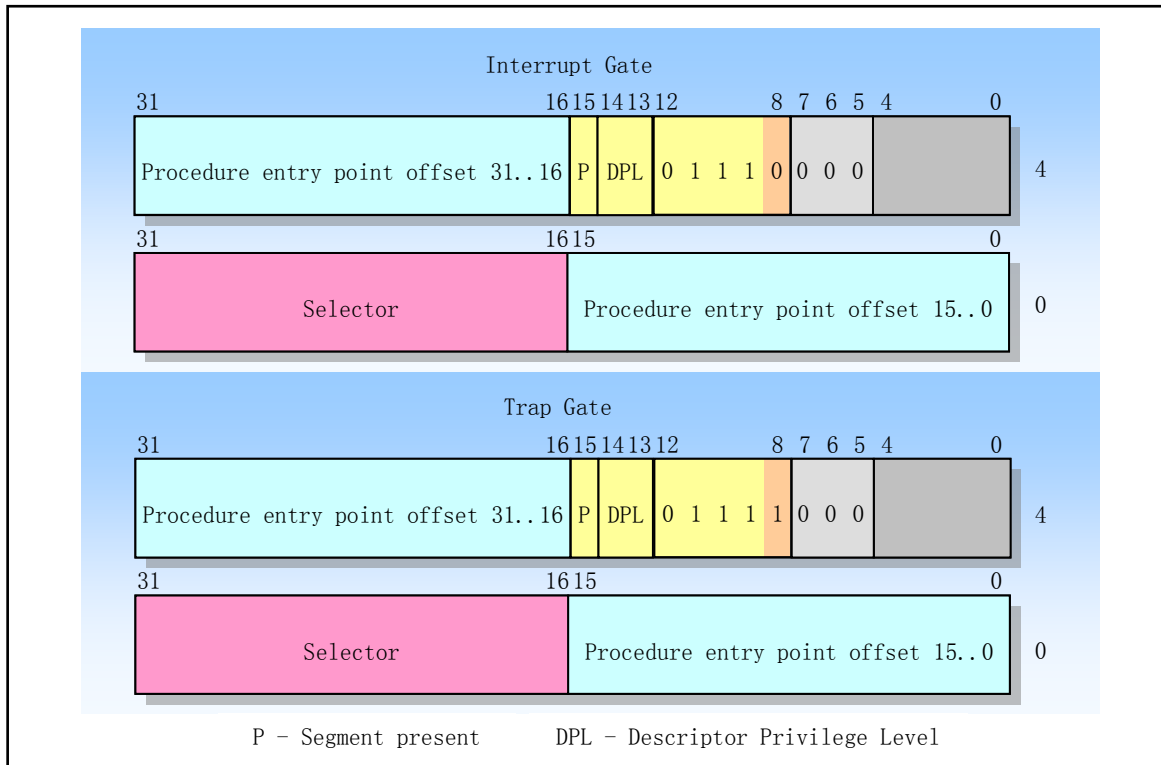


Figure 14-4 Interrupt Gate and Trap Gate Descriptor Format in Table IDT

Where P is the segment presence flag; DPL is the privilege level of the descriptor. The difference between the interrupt gate and the trap gate is the effect of the EFLAGS interrupt enable flag IF. The interrupt executed by the interrupt gate descriptor resets the IF flag, so this way prevents other interrupts from interfering with the current interrupt processing, and the subsequent interrupt end instruction IRET will restore the original value of the IF flag from the stack; The interrupt executed by the trap gate will not affect the IF flag.

In the generic macro `_set_gate (gate_addr, type, dpl, addr)` used to set the descriptor, the parameter `gate_addr` specifies the physical memory address at which the descriptor is located. 'type' indicates the type of descriptor to be set, which corresponds to the lower 4 bits of the 6th byte in the descriptor format in Figure 14-4, so `type=14(0x0E)` indicates the interrupt gate descriptor, `type=15 (0x0F)` indicates the trap gate descriptor. The parameter 'dpl' is the DPL in the corresponding descriptor format, and 'addr' is the 32-bit offset address of the interrupt processing process corresponding to the descriptor. Because the interrupt processing is part of the kernel segment code, their segment selector values are all 0x0008 (specified in the EAX register high word).

The last part of the system.h file is used to set the general segment descriptor content and set the task state segment descriptor and the local table segment descriptor in the global descriptor table GDT. The meanings of

the parameters of these macros is similar to the above.

14.19.2 Code annotation

Program 14-17 linux/include/asm/system.h

```

1  // Move to user mode to run.
2  // This function uses the IRET instruction to move from kernel mode to initial task 0.
3  #define move_to_user_mode() \
4  __asm__ ("movl %%esp, %%eax\n\t" \           // Save stack pointer ESP to EAX register.
5          "pushl $0x17\n\t" \                 // First push the user stack segment SS,
6          "pushl %%eax\n\t" \                 // then push the stack pointer ESP on to stack,
7          "pushfl\n\t" \                     // and push the EFLAGS register too.
8          "pushl $0x0f\n\t" \                 // Then push the code segment CS of task0,
9          "pushl $1f\n\t" \                   // and push the offset (EIP) at label 1.
10         "iret\n\t" \                       // Execute the IRET, causes control ret to label 1.
11         "1:\tmovl $0x17, %%eax\n\t" \       // At this point, kernel begins to execute task 0.
12         "movw %%ax, %%ds\n\t" \            // Initialize segment register points to the data
13         "movw %%ax, %%es\n\t" \            // segment of this local descriptor table.
14         "movw %%ax, %%fs\n\t" \
15         "movw %%ax, %%gs" \
16         ::: "ax")
17 #define sti() __asm__ ("sti::")           // enable interrupt.
18 #define cli() __asm__ ("cli::")           // disable interrupt.
19 #define nop() __asm__ ("nop::")           // no op.
20 #define iret() __asm__ ("iret::")         // interrupt return.
21
22 // Macro for setting the gate descriptor.
23 // The gate descriptor located at the address gate_addr is set according to the interrupt or
24 // exception handling procedure address addr, the gate descriptor type type, and the privilege
25 // level information dpl in the parameter. (Note: The "offset" below is relative to the kernel
26 // code or data segment).
27 // Parameters: gate_addr - gate descriptor address; type - descriptor type field value;
28 // dpl - descriptor privilege level; addr - offset address.
29 // %0 - (type flag word combined by 'dpl', 'type'); %1 - (descriptor low 4 byte address);
30 // %2 - (descriptor high 4 byte address); %3 - EDX (program offset address addr);
31 // %4 - EAX (high word contains segment selector 0x0008).
32 #define _set_gate(gate_addr, type, dpl, addr) \
33 // The offset address low word and selector are combined into a descriptor lower 4 bytes (EAX).
34 // Combine the type flag word with the offset high word into descriptor high 4 bytes (EDX).
35 // Finally, the lower 4 bytes and the upper 4 bytes of the gate descriptor are set separately.
36 __asm__ ("movw %%dx, %%ax\n\t" \
37         "movw %0, %%dx\n\t" \
38         "movl %%eax, %1\n\t" \
39         "movl %%edx, %2" \
40         : \
41         : "i" ((short) (0x8000+(dpl<<13)+(type<<8))), \           // %0
42         "o" (*(char *) (gate_addr)), \                             // %1
43         "o" (*(4+(char *) (gate_addr))), \                         // %2
44         "d" ((char *) (addr)), "a" (0x00080000)) \                 // %3, %4

```

```












    /// Set the interrupt gate (automatically mask subsequent interrupts).
    // Parameters: n - the interrupt number; addr - the interrupt program offset address.
    // Where '&idt[n]' is the offset value of the corresponding entry of the interrupt number n
    // in the interrupt descriptor table IDT; the type of the interrupt descriptor is 14, and the
    // privilege level is 0.
33 #define set_intr_gate(n,addr) \
34     set_gate(&idt[n], 14, 0, addr)
35
    /// Set trap gate.
    // Parameters: n - the interrupt number; addr - the interrupt program offset address.
    // '&idt[n]' is the offset of the corresponding entry of the interrupt number 'n' in the IDT;
    // the type of the interrupt descriptor is 15, and the privilege level is 0.
36 #define set_trap_gate(n,addr) \
37     set_gate(&idt[n], 15, 0, addr)
38
    /// Set system trap gate.
    // The descriptor set by set_trap_gate() above has a privilege level of 0, and here is 3.
    // Therefore, the interrupt processing set by set_system_gate() can be executed by all programs,
    // such as single-step debugging, overflow error, and boundary out of error processing.
    // Parameters: n - the interrupt number; addr - the interrupt program offset address.
    // '&idt[n]' is the offset of the corresponding entry of the interrupt number n in the IDT;
    // the type of the interrupt descriptor is 15, and the privilege level is 3.
39 #define set_system_gate(n,addr) \
40     set_gate(&idt[n], 15, 3, addr)
41
    /// Set the segment descriptor (not used in the kernel).
    // Parameters: gate_addr - descriptor address; type - type field value in the descriptor;
    // dpl - descriptor privilege level; base - segment base address; limit - segment limit.
    // See the format of the segment descriptor. Note that the assignment object here is incorrect
    // (reversed). Line 43 should be '*((gate_addr)+1)', and line 49 is '* (gate_addr)'. However,
    // this macro is not used in the kernel code, so Linux is not aware :-)
42 #define set_seg_desc(gate_addr, type, dpl, base, limit) {\
43     *((gate_addr) = ((base) & 0xff000000) | \           // Descriptor lower 4 bytes
44         (((base) & 0x00ff0000)>>16) | \
45         ((limit) & 0xf0000) | \
46         ((dpl)<<13) | \
47         (0x00408000) | \
48         ((type)<<8); \
49     *((gate_addr)+1) = (((base) & 0x0000ffff)<<16) | \ // Descriptor high 4 bytes.
50         ((limit) & 0xffff); }
51
    /// Set the task status segment/local table descriptor in the global table GDT.
    // The length of the status segment and the local table segment are both set to 104 bytes.
    // Parameters: n - the address corresponding to the descriptor item n in the global table GDT;
    // addr - the base address of the memory where the state segment/local table is located;
    // type - the flag type byte in the descriptor.
    // %0 - eax (address addr); %1 - (address of descriptor item n); %2 - (the offset 2 of the
    // descriptor item n); %3 - (the offset 4 of descriptor item n); %4 - (the offset 5 of the
    // descriptor item n); %5 - (the offset 6 of descriptor item n); %6 - (the offset 7 of the
    // descriptor item n);
52 #define set_tssldt_desc(n, addr, type) \
53     __asm__ ("movw $104, %1\n\t" \           // The TSS length is stored in length field (0-th byte).
54             "movw %%ax, %2\n\t" \           // Put the low word of base into the 2-3rd byte.

```

```
55     "rorl $16, %%eax|n|t" \    // Rotate base high word into AX (low word to high).
56     "movb %%al, %3|n|t" \      // Move low byte of the base high word to the 4th byte.
57     "movb $" type ", %4|n|t" \ // Move flag type byte into the 5th byte.
58     "movb $0x00, %5|n|t" \     // The sixth byte of the descriptor is set to zero.
59     "movb %%ah, %6|n|t" \      // Move high byte of the base high word to the 7th byte.
60     "rorl $16, %%eax" \        // Loop 16 bits to the right, EAX restores the original.
61     :: "a" (addr), "m" (*(n)), "m" (*(n+2)), "m" (*(n+4)), \
62     "m" (*(n+5)), "m" (*(n+6)), "m" (*(n+7)) \
63     )
64
65     /// Set the task status segment (TSS) descriptor in the global table GDT.
66     // n - is a pointer to the descriptor; addr - is the base address of the segment in the descriptor
67     // entry. The type of TSS descriptor is of type 0x89.
68 #define set_tss_desc(n, addr) _set_tssldt_desc(((char *) (n)), addr, "0x89")
69     /// Set the local table (LDT) descriptor in the global table.
70     // n - is a pointer to the descriptor; addr - is the base address of the segment in the descriptor
71     // entry. The type of the local table segment descriptor is 0x82.
72 #define set_ldt_desc(n, addr) _set_tssldt_desc(((char *) (n)), addr, "0x82")
```

14.20 Files in the directory include/linux/

List 14-3 Files in the directory linux/include/linux/

Filename	Size	Last Modified Time(GMT)	Description
 config.h	1545 bytes	1992-01-11 00:13:18	
 fdreg.h	2466 bytes	1991-11-02 10:48:44	
 fs.h	5754 bytes	1992-01-12 07:00:20	
 hdreg.h	1968 bytes	1991-10-13 15:32:15	
 head.h	304 bytes	1991-06-19 19:24:13	
 kernel.h	1036 bytes	1992-01-12 02:17:34	
 math_emu.h	4924 bytes	1992-01-01 17:33:04	
 mm.h	1101 bytes	1992-01-13 15:46:41	
 sched.h	7351 bytes	1992-01-13 22:24:42	
 sys.h	3402 bytes	1992-01-13 21:42:37	
 tty.h	2801 bytes	1992-01-08 22:51:56	

14.21 config.h

14.21.1 Functionality

Config.h is the kernel configuration header file that defines the machine configuration information used by the uname command, as well as the keyboard language type and hard disk type (HD_TYPE) options used.

14.21.2 Code annotation

Program 14-18 linux/include/linux/config.h

```

1 #ifndef CONFIG\_H
2 #define CONFIG\_H
3
4 /*
5  * Defines for what uname() should return
6  */
7 #define UTS\_SYSNAME "Linux"
8 #define UTS\_NODENAME "(none)" /* set by sethostname() */
9 #define UTS\_RELEASE "" /* patchlevel */
10 #define UTS\_VERSION "0.12"
11 #define UTS\_MACHINE "i386" /* hardware type */
12
13 /* Don't touch these, unless you really know what your doing. */

```

```
// The following symbolic constants are used to indicate the memory location during system boot
// and load of the kernel and the default maximum kernel system module size.
14 #define DEF_INITSEG      0x9000      // The segment to which the boot sector be moved.
15 #define DEF_SYSSEG      0x1000      // The segment to which the the system module loaded.
16 #define DEF_SETUPSEG    0x9020      // The segment where the setup program is located.
17 #define DEF_SYSSIZE      0x3000      // The maximum system module size (in units of 16).
18
19 /*
20  * The root-device is no longer hard-coded. You can change the default
21  * root-device by changing the line ROOT_DEV = XXX in boot/bootsect.s
22  */
23
24 /*
25  * The keyboard is now defined in kernel/chr_dev/keyboard.S
26  */
27
28 /*
29  * Normally, Linux can get the drive parameters from the BIOS at
30  * startup, but if this for some unfathomable reason fails, you'd
31  * be left stranded. For this case, you can define HD_TYPE, which
32  * contains all necessary info on your harddisk.
33  *
34  * The HD_TYPE macro should look like this:
35  *
36  * #define HD_TYPE { head, sect, cyl, wpcom, lzone, ctl}
37  *
38  * In case of two harddisks, the info should be sepatated by
39  * commas:
40  *
41  * #define HD_TYPE { h, s, c, wpcom, lz, ctl }, { h, s, c, wpcom, lz, ctl }
42  */
43 /*
44  This is an example, two drives, first is type 2, second is type 3:
45
46 #define HD_TYPE { 4, 17, 615, 300, 615, 8 }, { 6, 17, 615, 300, 615, 0 }
47
48 NOTE: ctl is 0 for all drives with heads<=8, and ctl=8 for drives
49 with more than 8 heads.
50
51 If you want the BIOS to tell what kind of drive you have, just
52 leave HD_TYPE undefined. This is the normal thing to do.
53 */
54
55 #endif
56
```

14.22 fdreg.h

14.22.1 Functionality

The fdreg.h header file is used to describe some of the parameters commonly used in floppy devices and the I/O ports used. Because the control of the floppy disk drive is cumbersome and there are many commands, it is best to refer to the book about the principle of the microcomputer interface before reading the code to understand how the floppy disk controller (FDC) works. Then you will think that the definition here is still reasonable and orderly.

When programming a floppy disk device, you need to access 4 ports, one for each register or multiple registers. For a 1.2M floppy disk controller there will be some of the ports given in Table 14-1.

Table 14-1 Floppy disk controller ports

I/O port	Read/Write	Register name
0x3f2	Write only	Digital output (control) Register
0x3f4	Read only	FDC main status register
0x3f5	Read/Write	FDC data register
0x3f7	Read only	Digital input register
0x3f7	Write only	Floppy disk control register (rate control)

The digital output port (digital control port) is an 8-bit register that controls the drive motor on, drive selection, start/reset FDC, and enable/disable DMA and interrupt requests.

The FDC main status register is also an 8-bit register that reflects the basic state of the floppy disk controller and the floppy disk drive (FDD). Typically, the status bits of the main status register are read before the CPU sends a command to the FDC or before the FDC obtains the result of the operation to determine if the current FDC data register is ready and to determine the direction of data transfer.

The data port of the FDC corresponds to multiple registers (write-only command register and parameter register, read-only result register), but only one register can appear on data port 0x3f5 at any one time. When accessing a write-only register, the DIO direction bit of the main state must be 0 (CPU → FDC), and vice versa when accessing the read-only register. When reading the result, the result is only read after the FDC is not busy. Usually, the result data has a maximum of 7 bytes.

The floppy disk controller can accept a total of 15 commands. Each command goes through three phases: the command phase, the execution phase, and the results phase.

The command phase is that the CPU sends command bytes and parameter bytes to the FDC. The first byte of each command is always the command byte (command code) followed by a parameter of 0-8 bytes. The execution phase is the operation specified by the FDC execution command. In the execution phase, the CPU does not intervene. Generally, the FDC issues an interrupt request to let the CPU know the end of the command execution. If the FDC command sent by the CPU is to transfer data, the FDC can be performed in an interrupt mode or in a DMA manner. The interrupt mode transfers 1 byte at a time. The DMA mode is under the management of the DMA controller, and the FDC and the memory transfer data until all the data is transmitted. At this time, the DMA controller notifies the FDC of the transmission byte count termination signal, and finally the FDC issues an interrupt request signal to inform the CPU that the execution phase is over. The result phase

is that the CPU reads the FDC data register return value to obtain the result of the FDC command execution. The result data returned is 0–7 bytes in length. For commands that do not return result data, the FDC should be sent a status to detect the interrupt status command acquisition operation.

14.22.2 Code annotation

Program 14-19 linux/include/linux/fdreg.h

```

1  /*
2  * This file contains some defines for the floppy disk controller.
3  * Various sources. Mostly "IBM Microcomputers: A Programmers
4  * Handbook", Sanches and Canton.
5  */
6  #ifndef FDREG_H    // This definition is used to exclude duplicate header files in code.
7  #define FDREG_H
8
9  // Prototype declaration of some floppy disk type functions.
10 extern int ticks_to_floppy_on(unsigned int nr);
11 extern void floppy_on(unsigned int nr);
12 extern void floppy_off(unsigned int nr);
13 extern void floppy_select(unsigned int nr);
14 extern void floppy_deselect(unsigned int nr);
15
16 // The definition of some ports and symbols for the floppy disk controller.
17 /* Fd controller regs. S&C, about page 340 */
18 #define FD_STATUS      0x3f4    // Main status register port.
19 #define FD_DATA        0x3f5    // Data port.
20 #define FD_DOR         0x3f2    /* Digital Output Register */
21 #define FD_DIR         0x3f7    /* Digital Input Register (read) */
22 #define FD_DCR         0x3f7    /* Diskette Control Register (write)*/
23
24 /* Bits of main status register */
25 #define STATUS_BUSYMASK 0x0F    /* drive busy mask */ // (one bit per driver).
26 #define STATUS_BUSY    0x10    /* FDC busy */
27 #define STATUS_DMA     0x20    /* 0- DMA mode */
28 #define STATUS_DIR     0x40    /* 0- cpu->fdc */
29 #define STATUS_READY   0x80    /* Data reg ready */
30
31 /* Bits of FD_ST0 */
32 #define ST0_DS          0x03    /* drive select mask */
33 #define ST0_HA          0x04    /* Head (Address) */
34 #define ST0_NR          0x08    /* Not Ready */
35 #define ST0_ECE         0x10    /* Equipment chech error */
36 #define ST0_SE          0x20    /* Seek end */ // or recalitrte end.
37 // Interrupt code bit (interrupt reason), 00 - command ends normally; 01 - command ends
38 // abnormally; 10 - command is invalid; 11 - FDD ready state changes.
39 #define ST0_INTR       0xC0    /* Interrupt code mask */
40
41 /* Bits of FD_ST1 */
42 #define ST1_MAM         0x01    /* Missing Address Mark */
43 #define ST1_WP          0x02    /* Write Protect */
44 #define ST1_ND          0x04    /* No Data - unreadable */ // sector not found.

```

```

41 #define ST1_OR          0x10          /* OverRun */    // Data transfer timeout.
42 #define ST1_CRC         0x20          /* CRC error in data or addr */
43 #define ST1_EOC         0x80          /* End Of Cylinder */
44
45 /* Bits of FD_ST2 */
46 #define ST2_MAM         0x01          /* Missing Address Mark (again) */
47 #define ST2_BC          0x02          /* Bad Cylinder */
48 #define ST2_SNS         0x04          /* Scan Not Satisfied */
49 #define ST2_SEH         0x08          /* Scan Equal Hit */
50 #define ST2_WC          0x10          /* Wrong Cylinder */
51 #define ST2_CRC         0x20          /* CRC error in data field */
52 #define ST2_CM          0x40          /* Control Mark = deleted */
53
54 /* Bits of FD_ST3 */
55 #define ST3_HA          0x04          /* Head (Address) */
56 #define ST3_TZ          0x10          /* Track Zero signal (1=track 0) */
57 #define ST3_WP          0x40          /* Write Protect */
58
59 /* Values for FD_COMMAND */
60 #define FD_RECALIBRATE  0x07          /* move to track 0 */    // recalibrate.
61 #define FD_SEEK         0x0F          /* seek track */
62 #define FD_READ         0xE6          /* read with MT, MFM, SKip deleted */
63 #define FD_WRITE        0xC5          /* write with MT, MFM */
64 #define FD_SENSEI       0x08          /* Sense Interrupt Status */
    // Set the drive parameters (step rate, head unload time, etc.).
65 #define FD_SPECIFY      0x03          /* specify HUT etc */
66
67 /* DMA commands */
68 #define DMA_READ         0x46          // The mode word of DMA read disk (to port 12, 11).
69 #define DMA_WRITE        0x4A          // The mode word of DMA write disk.
70
71 #endif
72

```

14.23 fs.h

14.23.1 Functionality

The fs.h header file mainly defines some constants and structures about the file system, including the data structure of the buffer block in the buffer cache, the super block and i-node structure in the MINIX 1.0 file system, and the file table structure and some pipeline operation macros. .

14.23.2 Code annotation

Program 14-20 linux/include/linux/fs.h

```

1 /*
2  * This file has definitions for some important file table
3  * structures etc.
4  */

```

```

5
6 #ifndef FS\_H
7 #define FS\_H
8
9 #include <sys/types.h>          // type header file. The basic system data types are defined.
10
11 /* devices are as follows: (same as minix, so we can use the minix
12  * file system. These are major numbers.)
13  *
14  * 0 - unused (nodev)
15  * 1 - /dev/mem                // memory device.
16  * 2 - /dev/fd
17  * 3 - /dev/hd
18  * 4 - /dev/ttyx              // tty serial terminal device.
19  * 5 - /dev/tty
20  * 6 - /dev/lp                // printer device.
21  * 7 - unnamed pipes
22  */
23
24 #define IS\_SEEKABLE(x) ((x)>=1 && (x)<=3)    // Determine if a device can find a location.
25
26 #define READ 0
27 #define WRITE 1
28 #define READA 2                /* read-ahead - don't pause */
29 #define WRITEA 3               /* "write-ahead" - silly, but somewhat useful */
30
31 void buffer\_init(long buffer_end);          // buffer cache init function.
32
33 #define MAJOR(a) (((unsigned)(a))>>8)       // get device major number.
34 #define MINOR(a) ((a)&0xff)                 // get device minor number.
35
36 #define NAME\_LEN 14                    // name length is 14.
37 #define ROOT\_INO 1                     // root i-node number.
38
39 #define I\_MAP\_SLOTS 8                   // the number of i-node bitmap slots (blocks).
40 #define Z\_MAP\_SLOTS 8                   // the number of logical block bitmap slots.
41 #define SUPER\_MAGIC 0x137F              // File system magic number.
42
43 #define NR\_OPEN 20                      // The maximum number of files opened by process.
44 #define NR\_INODE 32                     // The maximum number of I-nodes used by system.
45 #define NR\_FILE 64                      // The maximum number of files in the system.
46 #define NR\_SUPER 8                      // The maximum number of superblocks in system.
47 #define NR\_HASH 307                     // Buffer hash-table array items.
48 #define NR\_BUFFERS nr\_buffers          // The number of buffer blocks in the system.
49 #define BLOCK\_SIZE 1024                 // Data block size (in bytes).
50 #define BLOCK\_SIZE\_BITS 10             // The number of bits used by the block size.
51 #ifndef NULL
52 #define NULL ((void *) 0)
53 #endif
54
55 // The number of i-nodes that each block can store (1024/32 = 32).
56 #define INODES\_PER\_BLOCK ((BLOCK\_SIZE)/(sizeof (struct d\_inode)))
57 // The number of directory entries that can be stored in each block (1024/16 = 64).

```

```

56 #define DIR_ENTRIES_PER_BLOCK ((BLOCK_SIZE)/(sizeof (struct dir_entry)))
57 // Get pipe head, tail, size, pipe empty? full?
58 #define PIPE_READ_WAIT(inode) ((inode).i_wait)
59 #define PIPE_WRITE_WAIT(inode) ((inode).i_wait2)
60 #define PIPE_HEAD(inode) ((inode).i_zone[0])
61 #define PIPE_TAIL(inode) ((inode).i_zone[1])
62 #define PIPE_SIZE(inode) ((PIPE_HEAD(inode)-PIPE_TAIL(inode))&(PAGE_SIZE-1))
63 #define PIPE_EMPTY(inode) (PIPE_HEAD(inode)==PIPE_TAIL(inode))
64 #define PIPE_FULL(inode) (PIPE_SIZE(inode)==(PAGE_SIZE-1))
65
66 #define NIL FILP ((struct file *)0) // null file structure pointer.
67 #define SEL_IN 1
68 #define SEL_OUT 2
69 #define SEL_EX 4
70
71 typedef char buffer_block[BLOCK_SIZE]; // buffer block array (1024 items).
72
73 // Buffer block header data structure. (very important!!!)
74 // bh is often used in the code to represent the abbreviation of buffer_head.
75 struct buffer_head {
76     char * b_data; // pointer to data block (1024 bytes)
77     unsigned long b_blocknr; // block number
78     unsigned short b_dev; // device (0 = free)
79     unsigned char b_uptodate;
80     unsigned char b_dirt; // 0-clean, 1-dirty // Modified flag.
81     unsigned char b_count; // users using this block
82     unsigned char b_lock; // 0 - ok, 1 -locked
83     struct task_struct * b_wait; // task wait queue.
84     struct buffer_head * b_prev; // the previous block on the hash queue.
85     struct buffer_head * b_next; // the next block on the hash queue.
86     struct buffer_head * b_prev_free; // the previous block on the free list.
87     struct buffer_head * b_next_free; // the next block on the free list.
88 };
89
90 // I-node data structure (32 bytes) on disk.
91 struct d_inode {
92     unsigned short i_mode; // File type and attribute (rwx bit).
93     unsigned short i_uid; // User id (file owner identifier).
94     unsigned long i_size; // File size (in bytes).
95     unsigned long i_time; // Modified time (from 1970.1.1:0, in seconds).
96     unsigned char i_gid; // Group id (the group in which file owner belong).
97     unsigned char i_nlinks; // Number of links (entries pointed to the i-node).
98     unsigned short i_zone[9]; // logical block number array.
99 }; // direct(0-6), indirect(7) or secondary indirect(8).
100
101 // This is the i-node structure in memory. The first seven items are exactly the same as d_inode.
102 struct m_inode {
103     unsigned short i_mode;
104     unsigned short i_uid;
105     unsigned long i_size;
106     unsigned long i_mtime;
107     unsigned char i_gid;

```

```

104     unsigned char i_nlinks;
105     unsigned short i_zone[9];
106 /* these are in memory also */
107     struct task\_struct * i_wait; // task waiting queue for waiting for the i-node.
108     struct task\_struct * i_wait2;  /* for pipes */
109     unsigned long i_atime;         // i-node access time.
110     unsigned long i_ctime;         // i-node change time.
111     unsigned short i_dev;          // the device where the i-node is located.
112     unsigned short i_num;          // i-node number.
113     unsigned short i_count;        // i-node used count, 0 indicates it's idle (free).
114     unsigned char i_lock;          // lock flag.
115     unsigned char i_dirt;          // modified flag
116     unsigned char i_pipe;          // the i-node is used for pipe.
117     unsigned char i_mount;         // mount flag.
118     unsigned char i_seek;          // used for lseek method of the file.
119     unsigned char i_update;        // updated flag.
120 };
121
122 // File structure (used to establish a relationship between a file handle and the i-node)
123 struct file {
124     unsigned short f_mode;         // file mode (RW bits).
125     unsigned short f_flags;        // file open and control flags.
126     unsigned short f_count;        // file reference count.
127     struct m\_inode * f_inode;      // file's i-node.
128     off\_t f_pos;                  // read and write position in the file.
129 };
130
131 // In-memory disk super block structure.
132 struct super_block {
133     unsigned short s_ninodes;      // number of i-nodes in the file system.
134     unsigned short s_nzones;       // number of zones (logical blocks).
135     unsigned short s_imap_blocks;   // number of data blocks occupied by i-node map.
136     unsigned short s_zmap_blocks;   // number of blocks occupied by logical block map.
137     unsigned short s_firstdatazone; // the first block number in the data zone.
138     unsigned short s_log_zone_size; // Log2(number of data blocks / logical block).
139     unsigned long s_max_size;       // the maximum file size.
140     unsigned short s_magic;         // file system magic number.
141 /* These are only in memory */
142     struct buffer\_head * s_imap[8]; // an array of i-node bitmap buffer blocks.
143     struct buffer\_head * s_zmap[8]; // an array of logical block bitmap buffer blocks.
144     unsigned short s_dev;           // the device number of the super block.
145     struct m\_inode * s_isup;         // The root i-node of the mounted file system.
146     struct m\_inode * s_imount;       // The i-node to which the file system is installed.
147     unsigned long s_time;           // modified time.
148     struct task\_struct * s_wait;     // the wait queue for processes waiting for it.
149     unsigned char s_lock;           // the lock flag.
150     unsigned char s_rd_only;        // read only flag.
151     unsigned char s_dirt;           // dirty flag.
152 };
153
154 // Super block structure on disk. It is exactly the same as lines 131-138 above.
155 struct d_super_block {
156     unsigned short s_ninodes;

```



```

154     unsigned short s_nzones;
155     unsigned short s_imap_blocks;
156     unsigned short s_zmap_blocks;
157     unsigned short s_firstdatazone;
158     unsigned short s_log_zone_size;
159     unsigned long s_max_size;
160     unsigned short s_magic;
161 };
162
163 // File directory entry structure (16 bytes).
164 struct dir_entry {
165     unsigned short inode;           // i-node number.
166     char name[NAME_LEN];           // File name, NAME_LEN = 14.
167 };
168
169 extern struct m_inode inode_table[NR_INODE]; // i-node table (32 entries).
170 extern struct file file_table[NR_FILE];      // file table (64 items).
171 extern struct super_block super_block[NR_SUPER]; // super block array (8 items).
172 extern struct buffer_head * start_buffer;    // starting location of the buffer cache.
173 extern int nr_buffers;                       // the number of buffers.
174
175 // The following are prototypes of disk manipulation function.
176 // Check if the floppy disk in the drive has changed.
177 extern void check_disk_change(int dev);
178 // Check the floppy disk replacement. Returns 1 if floppy is replaced, otherwise returns 0.
179 extern int floppy_change(unsigned int nr);
180 // Set the amount of time to wait to start the drive (set the wait timer).
181 extern int ticks_to_floppy_on(unsigned int dev);
182 // Start the specified drive.
183 extern void floppy_on(unsigned int dev);
184 // Turn off the specified floppy drive.
185 extern void floppy_off(unsigned int dev);
186
187 // The following are function prototypes for file system operation management.
188 // The size of the file specified by the i-node is truncated to 0.
189 extern void truncate(struct m_inode * inode);
190 // Refresh (synchronize) the i-node information.
191 extern void sync_inodes(void);
192 // Waiting for the specified i-node.
193 extern void wait_on(struct m_inode * inode);
194 // block bitmap operation. Get the block number of the data block 'block' on the device.
195 extern int bmap(struct m_inode * inode, int block);
196 // Create a logical block on the device corresponding to the block 'block' and return the logical
197 // block number on the device.
198 extern int create_block(struct m_inode * inode, int block);
199 // Get the i-node number of the specified path name.
200 extern struct m_inode * namei(const char * pathname);
201 // Get the i-node of the specified path name without following the symbolic link.
202 extern struct m_inode * lnamei(const char * pathname);
203 // Prepare to open the file according to the path name.
204 extern int open_namei(const char * pathname, int flag, int mode,
205     struct m_inode ** res_inode);
206 // Release (put back) an i-node (to write device).

```

```
188 extern void input(struct m\_inode * inode);
    // Reads an i-node of from device.
189 extern struct m\_inode * iget(int dev,int nr);
    // Obtain an idle i-node entry from the inode table.
190 extern struct m\_inode * get\_empty\_inode(void);
    // Get (Apply a) pipe i-node. Returns the pointer to the i-node (fail if NULL).
191 extern struct m\_inode * get\_pipe\_inode(void);
    // Find the specified data block in the hash table. Returns the buffer head pointer.
192 extern struct buffer\_head * get\_hash\_table(int dev, int block);
    // Read the specified block from the device (first look in the hash table).
193 extern struct buffer\_head * getblk(int dev, int block);
    // Low-level read/write block function.
194 extern void ll\_rw\_block(int rw, struct buffer\_head * bh);
    // Low-level read/write data pages, that is, 4 data blocks at a time.
195 extern void ll\_rw\_page(int rw, int dev, int nr, char * buffer);
    // Release the specified buffer block.
196 extern void brelse(struct buffer\_head * buf);
    // Read the specified data block.
197 extern struct buffer\_head * bread(int dev,int block);
    // Read a page (4 buffer blocks) to the specified memory address.
198 extern void bread\_page(unsigned long addr,int dev,int b[4]);
    // Reads a specified block of data and marks the block that will be read later.
199 extern struct buffer\_head * breada(int dev,int block,...);
    // Request a disk block from device dev and return the logical block number
200 extern int new\_block(int dev);
    // Frees the logic blocks in the device data area. Reset the logic block bitmap bits.
201 extern void free\_block(int dev, int block);
    // Create a new i-node for device and return the i-node number.
202 extern struct m\_inode * new\_inode(int dev);
    // Release (free) an i-node (when deleting a file).
203 extern void free\_inode(struct m\_inode * inode);
    // Refresh the specified device buffer.
204 extern int sync\_dev(int dev);
    // Get a super block of the specified device.
205 extern struct super\_block * get\_super(int dev);
206 extern int ROOT\_DEV;          // root device number.
207
    // Mount the root file system.
208 extern void mount\_root(void);
209
210 #endif
211
```

14.24 hdreg.h

14.24.1 Functionality

The `hdreg.h` file mainly defines some command constant symbols for programming the hard disk controller. This includes the controller port, the status of each bit of the hard disk status register, controller commands, and

error status constant symbols. The data structure of the hard disk partition table is also given.

14.24.2 Code annotation

Program 14-21 linux/include/linux/hdreg.h

```

1  /*
2  * This file contains some defines for the AT-hd-controller.
3  * Various sources. Check out some definitions (see comments with
4  * a ques).
5  */
6  #ifndef HDREG_H
7  #define HDREG_H
8
9  /* Hd controller regs. Ref: IBM AT Bios-listing */
10 #define HD_DATA          0x1f0  /* _CTL when writing */
11 #define HD_ERROR         0x1f1  /* see err-bits */
12 #define HD_NSECTOR       0x1f2  /* nr of sectors to read/write */
13 #define HD_SECTOR        0x1f3  /* starting sector */
14 #define HD_LCYL          0x1f4  /* starting cylinder */
15 #define HD_HCYL          0x1f5  /* high byte of starting cyl */
16 #define HD_CURRENT       0x1f6  /* 10ldhhh , d=drive, hhhh=head */
17 #define HD_STATUS        0x1f7  /* see status-bits */
18 #define HD_PRECOMP HD_ERROR  /* same io address, read=error, write=precomp */
19 #define HD_COMMAND HD_STATUS /* same io address, read=status, write=cmd */
20
21 #define HD_CMD            0x3f6  // Control register port.
22
23 /* Bits of HD_STATUS */
24 #define ERR_STAT          0x01  // Command execution error.
25 #define INDEX_STAT       0x02  // Received the index.
26 #define ECC_STAT          0x04  /* Corrected error */ // ECC checksum error.
27 #define DRQ_STAT          0x08  // Request service.
28 #define SEEK_STAT         0x10  // End of seek.
29 #define WRERR_STAT        0x20  // Drive error.
30 #define READY_STAT        0x40  // Drive ready.
31 #define BUSY_STAT         0x80  // Controller busy.
32
33 /* Values for HD_COMMAND */
34 #define WIN_RESTORE        0x10  // Drive reset (recalibration).
35 #define WIN_READ           0x20  // Read sector.
36 #define WIN_WRITE          0x30  // Write sector.
37 #define WIN_VERIFY         0x40  // Sector verify.
38 #define WIN_FORMAT         0x50  // Format track.
39 #define WIN_INIT           0x60  // Controller initialize.
40 #define WIN_SEEK           0x70  // Seek track.
41 #define WIN_DIAGNOSE       0x90  // Controller diagnose.
42 #define WIN_SPECIFY        0x91  // Establish drive parameters.
43
44 /* Bits for HD_ERROR */
45 // When execute a diagnostic command, its meaning is different from other commands, as follows:
46 // =====
47 //           Diagnostic command           Other command
48 // -----

```

```

// 0x01      No error                Data mark lost
// 0x02      Controller error        Track 0 error.
// 0x03      Sector buffer error
// 0x04      ECC part error          Command abort
// 0x05      Control process error
// 0x10                      ID not found.
// 0x40                      ECC error.
// 0x80                      Bad sector
//-----
45 #define MARK_ERR      0x01      /* Bad address mark ? */
46 #define TRKO_ERR      0x02      /* couldn't find track 0 */
47 #define ABRT_ERR      0x04      /* ? */
48 #define ID_ERR        0x10      /* ? */
49 #define ECC_ERR        0x40      /* ? */
50 #define BBD_ERR        0x80      /* ? */
51
// Hard disk partition table structure, see the information after the list below.
52 struct partition {
53     unsigned char boot_ind;        /* 0x80 - active (unused) */
54     unsigned char head;           /* ? */
55     unsigned char sector;         /* ? */
56     unsigned char cyl;            /* ? */
57     unsigned char sys_ind;        /* ? */
58     unsigned char end_head;       /* ? */
59     unsigned char end_sector;     /* ? */
60     unsigned char end_cyl;        /* ? */
61     unsigned int start_sect;      /* starting sector counting from 0 */
62     unsigned int nr_sects;        /* nr of sectors in partition */
63 };
64
65 #endif
66

```

14.24.3 Information

14.24.3.1 Hard disk partition table

In order to facilitate management of data or to achieve shared hard disk resources by multiple operating systems, the hard disk can be logically divided into 1--4 partitions. The sector numbers between each partition are contiguous. The partition table consists of four entries, each of which consists of 16 bytes and corresponds to the information of one partition. Each entry has a partition size, a cylinder number, a track number, and a sector number, as shown in Table 14-2. The partition table is stored at the 0x1BE--0x1FD position of the first sector of the 0 cylinder 0 head of the hard disk.

Table 14-2 Hard disk partition table entry structure

Offset	Name	Size	Description
0x00	boot_ind	1 byte	Boot index. Only one partition of the 4 partitions can be booted at a time. 0x00 - Do not boot from this partition; 0x80 - Boot from this partition.
0x01	head	1 byte	Partition start head number. The head number ranges from 0 to 255.
0x02	sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the beginning of the partition.

0x03	cyl	1 byte	The lower 8 bits of the cylinder number at the starting of the partition.
0x04	sys_ind	1 byte	Partition type. 0x0b - DOS; 0x80 - Old Minix; 0x83 - Linux . . .
0x05	end_head	1 byte	The head number at the end of the partition. It ranges from 0 to 255.
0x06	end_sector	1 byte	The sector number (bits 0-5) in the current cylinder and the upper 2 bits (bits 6-7) of the cylinder number at the end of the partition .
0x07	end_cyl	1 byte	The lower 8 bits of the cylinder number at the end of the partition.
0x08-0x0b	start_sect	4 byte	The physical sector number at the beginning of the partition. It counts from 0 in the order of the sector number of the entire hard disk.
0x0c-0x0f	nr_sects	4 byte	The number of sectors occupied by the partition.

14.25 head.h

14.25.1 Functionality

The head.h header file defines the simple structure of the descriptor in the Intel CPU and specifies the item number of the descriptor.

14.25.2 Code annotation

Program 14-22 linux/include/linux/head.h

```

1 #ifndef \_HEAD\_H
2 #define \_HEAD\_H
3
4 // The data structure of the segment descriptor is defined below. This structure only states
5 // that each descriptor is composed of 8 bytes, and each descriptor table has 256 entries.
6 typedef struct desc\_struct {
7     unsigned long a,b;
8 } desc\_table[256];
9
10 // Declare the memory page directory table used by the paging management mechanism. Each
11 // directory entry is 4 bytes. For this kernel, the table starts at physical address 0.
12 extern unsigned long pg\_dir[1024];
13 extern desc\_table idt,gdt; // Interrupt descriptor table, global descriptor table.
14
15 #define GDT\_NUL 0 // The 0th item of the GDT, not used.
16 #define GDT\_CODE 1 // The first item is the kernel code segment descriptor.
17 #define GDT\_DATA 2 // The second item is the kernel data segment descriptor.
18 #define GDT\_TMP 3 // The third item is system segment descriptor, not used.
19
20 #define LDT\_NUL 0 // The 0th item of the LDT, not used.
21 #define LDT\_CODE 1 // The first item is the user code segment descriptor .
22 #define LDT\_DATA 2 // The second item is the user data segment descriptor.
23
24 #endif
25

```

14.26 kernel.h

14.26.1 Functionality

The kernel.h file defines some function prototypes commonly used by the kernel.

14.26.2 Code annotation

Program 14-23 linux/include/linux/kernel.h

```

1  /*
2  * 'kernel.h' contains some often-used function prototypes etc
3  */
4  // Verify that the memory block is overrun. (kernel/fork.c, 24).
5  void verify\_area(void * addr, int count);
6  // Display kernel error messages and then enter an infinite loop. (kernel/panic.c, 16).
7  volatile void panic(const char * str);
8  // The function for the process exit. (kernel/exit.c, 262).
9  volatile void do\_exit(long error_code);
10 // Standard print (display) function. (init/main.c, 179).
11 int printf(const char * fmt, ...);
12 // Kernel-specific print function with the same functionality as printf() (kernel/printk.c).
13 int printk(const char * fmt, ...);
14 // The display function of the console. (kernel/chr_drv/console.c, 995).
15 void console\_print(const char * str);
16 // Write a string of the specified length to the tty. (kernel/chr_drv/tty_io.c, 339).
17 int tty\_write(unsigned ch, char * buf, int count);
18 // Generic kernel memory allocation function. ( lib/malloc.c, 117).
19 void * malloc(unsigned int size);
20 // Frees the memory occupied by the specified object. ( lib/malloc.c, 182).
21 void free\_s(void * obj, int size);
22 // Hard disk processing timed out. (kernel/blk_drv/hd.c, 318).
23 extern void hd\_times\_out(void);
24 // Stop beeping. (kernel/chr_drv/console.c, 944).
25 extern void sysbeepstop(void);
26 // Black screen processing. (kernel/chr_drv/console.c, 981).
27 extern void blank\_screen(void);
28 // Restore the screen that is being blacked out. (kernel/chr_drv/console.c, 988).
29 extern void unblank\_screen(void);
30
31 extern int beepcount; // The beep time tick counts (chr_drv/console.c, 950).
32 extern int hd\_timeout; // Hard disk timeout ticks (kernel/blk_drv/blk.h).
33 extern int blankinterval; // Screen black screen interval (chr_drv/console.c, 138).
34 extern int blankcount; // Black screen time count (chr_drv/console.c, 139).
35
36 #define free(x) free\_s((x), 0)
37
38 /*
39 * This is defined as a macro, but at some point this might become a
40 * real subroutine that sets a flag if it returns true (to do
41 * BSD-style accounting where the process is flagged if it uses root
42 * privs). The implication of this is that you should do normal

```

```

30 * permissions checks first, and check suser() last.
31 */
32 #define suser() (current->euid == 0)           // Check if it is a super user.
33

```

14.27 math_emu.h

14.27.1 Functionality

The `math_emu.h` file contains the constant definitions and structures involved in Chapter 11 (Mathematic Coprocessor), including some of the data structures used by the kernel code to simulate various types of data when simulating mathematical coprocessors.

14.27.2 Code annotation

Program 14-24 `linux/include/linux/math_emu.h`

```

1 /*
2  * linux/include/linux/math_emu.h
3  *
4  * (C) 1991 Linus Torvalds
5  */
6 #ifndef _LINUX_MATH_EMU_H
7 #define _LINUX_MATH_EMU_H
8
9 // The scheduler header file defines the task structure, the initial task 0, and some embedded
10 // assembly function macro statements about the descriptor parameter settings and acquisition.
11 #include <linux/sched.h>
12
13 // The structure of the data on the stack when CPU generates exception INT 7 (device not exist)
14 // is similar to the data distribution in the kernel stack when a system-call is invoked.
15 struct info {
16     long __math_ret;    // The return address of the math_emulate() caller (INT 7).
17     long __orig_eip;    // A place to temporarily save the original EIP.
18     long __edi;        // The registers that the exception handler pushed.
19     long __esi;
20     long __ebp;
21     // When interrupt 7 returns, it will execute the return processing code of the system call.
22     // Below (lines 18--30) are identical to the structure in stack when system-call is invoked.
23     long __sys_call_ret;
24     long __eax;
25     long __ebx;
26     long __ecx;
27     long __edx;
28     long __orig_eax;    // If it's not a sys-call but other interrupts, it is -1.
29     long __fs;
30     long __es;
31     long __ds;

```

```

26     long __eip;           // Lines 26 -- 30 are pushed by the CPU automatically.
27     long __cs;
28     long __eflags;
29     long __esp;
30     long __ss;
31 };
32
33 // Constants defined to facilitate the reference to fields in info structure (data in the stack).
34 #define EAX (info->__eax)
35 #define EBX (info->__ebx)
36 #define ECX (info->__ecx)
37 #define EDX (info->__edx)
38 #define ESI (info->__esi)
39 #define EDI (info->__edi)
40 #define EBP (info->__ebp)
41 #define ESP (info->__esp)
42 #define EIP (info->__eip)
43 #define ORIG_EIP (info->__orig_eip)
44 #define EFLAGS (info->__eflags)
45 #define DS (*(unsigned short *) &(info->__ds))
46 #define ES (*(unsigned short *) &(info->__es))
47 #define FS (*(unsigned short *) &(info->__fs))
48 #define CS (*(unsigned short *) &(info->__cs))
49 #define SS (*(unsigned short *) &(info->__ss))
50
51 // Terminate the math coprocessor emulation operation (in file math_emulation.c, line 488).
52 // The actual effect of the macro definition on lines 52-53 below is to redefine __math_abort
53 // as a function that does not return (that is, add volatile before). The first part of the macro:
54 // '(volatile void (*)(struct info *, unsigned int))' is a function type definition that is used
55 // to re-specify the definition of the __math_abort function. This is followed by its corresponding
56 // parameters. Putting the keyword volatile in front of the function name to decorate the function
57 // is used to tell the gcc compiler that the function will not return, so that gcc can produce
58 // better code.
59 void __math_abort(struct info *, unsigned int);
60
61 #define math_abort(x,y) \
62 ((volatile void (*)(struct info *, unsigned int)) __math_abort)((x),(y))
63
64 /*
65  * Gcc forces this stupid alignment problem: I want to use only two longs
66  * for the temporary real 64-bit mantissa, but then gcc aligns out the
67  * structure to 12 bytes which breaks things in math_emulate.c. Shit. I
68  * want some kind of "no-align" pragma or something.
69  */
70
71 // Temporary real structure.
72 // It has a total of 64 bit mantissas. Where 'a' is the lower 32 bits, 'b' is the upper 32 bits
73 // (including 1 fixed bit), and 'exponent' is the exponent value.
74 typedef struct {
75     long a,b;
76     short exponent;
77 } temp_real;
78

```



```

    // The structure designed to solve the alignment problem mentioned in the original note above
    // and works like the temp_real structure above.
67 typedef struct {
68     short m0,m1,m2,m3;
69     short exponent;
70 } temp_real_unaligned;
71
    // Assign the temp_real type value 'a' to the 80387 stack register 'b' (ST(i)).
72 #define real_to_real(a,b) \
73 ((*(long long *) (b) = *(long long *) (a)), ((b)->exponent = (a)->exponent))
74
    // Long real (double precision) structure.
75 typedef struct {
76     long a,b;                // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
77 } long_real;
78
79 typedef long short_real;    // Define a short real type.
80
    // Temporary integer structure.
81 typedef struct {
82     long a,b;                // 'a' is the lower 32 bits; 'b' is the upper 32 bits.
83     short sign;
84 } temp_int;
85
    // The structure corresponding to the contents of the status word register inside the 80387
    // coprocessor (see Figure 11-6).
86 struct swd {
87     int ie:1;                // Invalid operation exception.
88     int de:1;                // Denormalized exception.
89     int ze:1;                // Divide by zero exception.
90     int oe:1;                // Overflow exception.
91     int ue:1;                // Underflow exception.
92     int pe:1;                // Precision exception.
93     int sf:1;                // Stack error flag, caused by overflow of the accumulator.
94     int ir:1;                // Ir, b: Set if any of the above 6 unmasked exceptions occur.
95     int c0:1;                // c0--c3: Condition code bits.
96     int c1:1;
97     int c2:1;
98     int top:3;               // Indicates the 80-bit register currently at the top of the stack.
99     int c3:1;
100    int b:1;
101 };
102
    // 80387 internal register control mode constants.
103 #define I387 (current->tss.i387)    // 80387 status information of the process.
104 #define SWD (*(struct swd *) &I387.swd)    // Status control word in 80387.
105 #define ROUNDING ((I387.cwd >> 10) & 3)    // Get the rounding mode in the control word.
106 #define PRECISION ((I387.cwd >> 8) & 3)    // Get the precision mode in the control word.
107
    // Constant that define the significant digits of a precision.
108 #define BITS24      0        // Precision Effective bits: 24 bits.
109 #define BITS53      2        // 53 bits.
110 #define BITS64      3        // 64 bits.

```

```

111 // Define rounding mode constants.
112 #define ROUND_NEAREST 0 // Round to the nearest or even.
113 #define ROUND_DOWN 1 // Trend to negative infinite.
114 #define ROUND_UP 2 // Trend to positive infinite.
115 #define ROUND_0 3 // Trend to cut to zero.
116
117 // Constant definitions.
118 #define CONSTZ (temp_real_unaligned) {0x0000, 0x0000, 0x0000, 0x0000, 0x0000} // 0
119 #define CONST1 (temp_real_unaligned) {0x0000, 0x0000, 0x0000, 0x8000, 0x3FFF} // 1.0
120 #define CONSTPI (temp_real_unaligned) {0xC235, 0x2168, 0xDAA2, 0xC90F, 0x4000} // Pi
121 #define CONSTLN2 (temp_real_unaligned) {0x79AC, 0xD1CF, 0x17F7, 0xB172, 0x3FFE} // Loge(2)
122 #define CONSTLG2 (temp_real_unaligned) {0xF799, 0xFBCF, 0x9A84, 0x9A20, 0x3FFD} // Log10(2)
123 #define CONSTL2E (temp_real_unaligned) {0xF0BC, 0x5C17, 0x3B29, 0xB8AA, 0x3FFF} // Log2(e)
124 #define CONSTL2T (temp_real_unaligned) {0x8AFE, 0xCD1B, 0x784B, 0xD49A, 0x4000} // Log2(10)
125
126 // Set 80387 states.
127 #define set_IE() (I387.swd |= 1)
128 #define set_DE() (I387.swd |= 2)
129 #define set_ZE() (I387.swd |= 4)
130 #define set_OE() (I387.swd |= 8)
131 #define set_UE() (I387.swd |= 16)
132 #define set_PE() (I387.swd |= 32)
133
134 // Set 80387 control conditions
135 #define set_C0() (I387.swd |= 0x0100)
136 #define set_C1() (I387.swd |= 0x0200)
137 #define set_C2() (I387.swd |= 0x0400)
138 #define set_C3() (I387.swd |= 0x4000)
139
140 /* ea.c */
141 // Calculates the effective address used by the operand in the emulation instruction, that is,
142 // calculates the effective address according to the addressing mode byte in the instruction.
143 // Params: __info - the contents of the stack at time of interrupt; __code - instruction code.
144 // Returns effective address.
145 char * ea(struct info * __info, unsigned short __code);
146
147 /* convert.c */
148 // Various data type conversion functions implemented in the convert.c file.
149 void short_to_temp(const short_real * __a, temp_real * __b);
150 void long_to_temp(const long_real * __a, temp_real * __b);
151 void temp_to_short(const temp_real * __a, short_real * __b);
152 void temp_to_long(const temp_real * __a, long_real * __b);
153 void real_to_int(const temp_real * __a, temp_int * __b);
154 void int_to_real(const temp_int * __a, temp_real * __b);
155
156 /* get_put.c */
157 // Access functions of various types.
158 void get_short_real(temp_real *, struct info *, unsigned short);
159 void get_long_real(temp_real *, struct info *, unsigned short);
160 void get_temp_real(temp_real *, struct info *, unsigned short);
161 void get_short_int(temp_real *, struct info *, unsigned short);
162 void get_long_int(temp_real *, struct info *, unsigned short);

```

```
157 void get_longlong_int(temp_real *, struct info *, unsigned short);
158 void get_BCD(temp_real *, struct info *, unsigned short);
159 void put_short_real(const temp_real *, struct info *, unsigned short);
160 void put_long_real(const temp_real *, struct info *, unsigned short);
161 void put_temp_real(const temp_real *, struct info *, unsigned short);
162 void put_short_int(const temp_real *, struct info *, unsigned short);
163 void put_long_int(const temp_real *, struct info *, unsigned short);
164 void put_longlong_int(const temp_real *, struct info *, unsigned short);
165 void put_BCD(const temp_real *, struct info *, unsigned short);
166
167 /* add.c */
168 // A function that simulates a floating-point addition instruction.
169 void fadd(const temp_real *, const temp_real *, temp_real *);
170
171 /* mul.c */
172 // Simulate floating point multiply instructions.
173 void fmul(const temp_real *, const temp_real *, temp_real *);
174
175 /* div.c */
176 // Simulate floating point division instructions.
177 void fddiv(const temp_real *, const temp_real *, temp_real *);
178
179 /* compare.c */
180 // Simulate floating point comparison instructions.
181 void fcom(const temp_real *, const temp_real *);    // FCOM, compare two numbers.
182 void fucom(const temp_real *, const temp_real *);    // FUCOM, no order comparison.
183 void ftst(const temp_real *);    // FTST, top stack accumulator compared to 0.
184
185 #endif
186
```

14.28 mm.h

14.28.1 Functionality

The mm.h file is the memory management header file. It mainly defines the size of the memory page and several page release function prototypes.

14.28.2 Code annotation

Program 14-25 linux/include/linux/mm.h

```
1 #ifndef MM_H
2 #define MM_H
3
    // Define the memory page size (in bytes). Note that the cache block size is 1024 bytes.
4 #define PAGE_SIZE 4096
5
    // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
    // used functions of the kernel.
```

```

// <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
// manipulation function prototypes.
6 #include <linux/kernel.h>
7 #include <signal.h>
8
9 extern int SWAP\_DEV;          // Memory page swap device number (mm/memory.c, line 36).
10
11 // The swapped memory page is read in or written to the swap device. The ll_rw_page() function
12 // is defined in the file blk_drv/ll_rw_block.c. The parameter 'nr' is the page number in the
13 // main memory area; 'buffer' is the read/write buffer.
14 #define read\_swap\_page(nr,buffer) ll\_rw\_page(READ,SWAP\_DEV, (nr), (buffer));
15 #define write\_swap\_page(nr,buffer) ll\_rw\_page(WRITE,SWAP\_DEV, (nr), (buffer));
16
17 // Get free physical page in the main memory area. Returns 0 if there is no more memory available.
18 extern unsigned long get\_free\_page(void);
19 // Map a physical memory page whose content has been modified to a specified location in the
20 // linear address space. Almost exactly the same as put_page().
21 extern unsigned long put\_dirty\_page(unsigned long page,unsigned long address);
22 // Release a page of physical memory starting with address 'addr'.
23 extern void free\_page(unsigned long addr);
24 // Free the swapping page specified in the swap device (mm/swap.c, line 58).
25 void swap\_free(int page_nr);
26 // Swap the specified page from the device into memory (mm/swap.c, line 69).
27 void swap\_in(unsigned long *table_ptr);
28
29 // Out of memory (oom) processing function.
30 extern inline volatile void oom(void)
31 {
32 // do_exit() should use the exit code. The error code with the same value of the signal value
33 // SIGSEGV(11) is "Resource is temporarily unavailable", which is synonymous.
34 printk("out of memory\n\r");
35 do_exit(SIGSEGV);
36 }
37
38 // Invalidate the translation look-aside buffer (TLB) on-chip.
39 // In order to improve the efficiency of address translation, the CPU stores the most recently
40 // used page table data in the internal cache of the chip called translation lookaside buffer
41 // (TLB). After modifying the page table information, you need to refresh the buffer. This is
42 // done by reloading the page directory base register (PDBR) CR3. Below, EAX = 0, which is the
43 // base address of the page directory.
44 #define invalidate() \
45 __asm__ ("movl %%eax,%%cr3::\"a\" (0))
46
47 /* these are not to be changed without changing head.s etc */
48 // The maximum memory capacity supported by the Linux 0.12 kernel by default is 16MB, and these
49 // definitions can be modified to accommodate more memory.
50 #define LOW\_MEM 0x100000          // Low end of the physical memory (1MB).
51 extern unsigned long HIGH\_MEMORY; // The highest address of physical memory.
52 #define PAGING\_MEMORY (15*1024*1024) // The size of the paged memory (15MB).
53 #define PAGING\_PAGES (PAGING\_MEMORY>>12) // The number of pages after paging (3840).
54 #define MAP\_NR(addr) (((addr)-LOW\_MEM)>>12) // Map the memory address to the page number.
55 #define USED 100          // Page used flag, see memory.c, line 449.
56

```

```

// Memory mapping byte map, 1 byte represents 1 page. The corresponding byte of each page is
// used to mark the number of times the page is currently referenced (used). It can map up to
// 15Mb of memory space. In the function mem_init() in the memory.c program, the position that
// cannot be used as the main memory area page is set to USED (100) in advance.
37 extern unsigned char mem\_map [ PAGING\_PAGES ];
38
// The symbol constants defined below correspond to some of the flag bits in the page directory
// and page table (secondary page table) entries.
39 #define PAGE\_DIRTY      0x40          // Bit 6, the page is dirty (modified).
40 #define PAGE\_ACCESSED  0x20          // Bit 5, the page was accessed.
41 #define PAGE\_USER      0x04          // Bit 2, the page belongs to: 1-user; 0-superuser.
42 #define PAGE\_RW        0x02          // Bit 1, read/write rights: 1 - write; 0 - read.
43 #define PAGE\_PRESENT   0x01          // Bit 0, page exists: 1 - present; 0 - not exist.
44
45 #endif
46

```

14.29 sched.h

14.29.1 Functionality

Sched.h is the scheduler header file, which defines the task structure `task_struct`, initial task 0 data, and some embedded assembly function macros for memory descriptor parameter settings and acquisition and task context switch macro `switch_to()`. Below we describe in detail the execution process of the task switch macro.

The task switch macro `switch_to(n)` (starting at line 222) first declares a structure `'struct {long a,b;} __tmp'`, which is used to reserve 8 bytes of space on the kernel state stack. This space is used to store the selector for the task status segment TSS of the new task that will be switched to. Then test if we are performing the operation to switch to the current task, and if so, do not need to do anything, just exit. Otherwise we save the selector of the new task TSS to the offset position 4 in the temporary structure `__tmp`, at which point the data in `__tmp` is set to:

```

__tmp+0: Undefined (long)
__tmp+4: New task TSS selector (word)
__tmp+6: Undefined (word)

```

Next, we exchange the new task pointer in the ECX register with the current task pointer in the global variable `'current'`, let `'current'` contain the pointer value of the new task we are going to switch to, and ECX saves the current task. Then execute the instruction `LJMP` that indirectly jumps to `__tmp`. The instruction that jumps to the new task TSS selector will ignore the undefined value part of `__tmp`, and the CPU will automatically jump to the new task specified by the TSS segment to execute, and the task (current task) will be suspended. This is why we don't need to set other undefined parts of the structure variable `__tmp`. See Figure 4-37 in Section 4.7 for a schematic diagram of the task switching operation.

After a period of time, the `LJMP` instruction of a task will jump to the TSS segment selector of the task, causing the CPU to switch back to the task and start execution from the next instruction of `LJMP`. At this point ECX contains a pointer to the current task, so we can use this pointer to check if it is the last (most recently)

task that used the math coprocessor. If the task has not used the coprocessor, it will exit immediately. Otherwise, the CLTS instruction is executed to reset the task switching flag TS in the control register CR0. The CPU sets this flag whenever the task is switched and tests the flag before executing the coprocessor instruction. This method of processing the TS flag in the Linux system allows the kernel to avoid unnecessary saving and recovery operations on the coprocessing state, thereby improving the execution performance of the coprocessor.

14.29.2 Code annotation

Program 14-26 linux/include/linux/sched.h

```

1  #ifndef \_SCHED\_H
2  #define \_SCHED\_H
3
4  #define HZ 100                // Define system clock tick frequency (10ms per tick)
5
6  #define NR\_TASKS          64          // The max number of tasks in the system.
7  #define TASK\_SIZE          0x04000000    // The size of each task (64MB).
8  #define LIBRARY\_SIZE       0x00400000    // The size of the loaded library (4MB).
9
10 #if (TASK\_SIZE & 0x3fffff)
11 #error "TASK_SIZE must be multiple of 4M"
12 #endif
13
14 #if (LIBRARY\_SIZE & 0x3fffff)
15 #error "LIBRARY_SIZE must be a multiple of 4M"
16 #endif
17
18 #if (LIBRARY\_SIZE >= (TASK\_SIZE/2))
19 #error "LIBRARY_SIZE too damn big!"
20 #endif
21
22 #if (((TASK\_SIZE>>16)*NR\_TASKS) != 0x10000)
23 #error "TASK_SIZE*NR_TASKS must be 4GB"
24 #endif
25
26 // The location where the library is loaded in the process logical address space (at 60MB).
27 #define LIBRARY\_OFFSET (TASK\_SIZE - LIBRARY\_SIZE)
28
29 // The following macros CT_TO_SECS and CT_TO_USECS are used to convert the current system ticks
30 // into seconds and microseconds.
31 #define CT\_TO\_SECS(x)    ((x) / HZ)
32 #define CT\_TO\_USECS(x)  ((x) % HZ) * 1000000/HZ)
33
34 #define FIRST\_TASK task[0]          // Task 0 is special, so we define a symbol for it.
35 #define LAST\_TASK task[NR\_TASKS-1] // The last task in the task array.
36
37 // <linux/head.h> Head header file. A simple structure for the segment descriptor is defined,
38 // along with several selector constants.
39 // <linux/fs.h> File system header file. Define the file table structure (file,buffer_head,
40 // m_inode, etc.).
41 // <linux/mm.h> Memory management header file. Contains page size definitions and some page
42 // release function prototypes.
43 // <sys/param.h> Parameter file. Some hardware-related parameter values are given.

```

```

// <sys/time.h> The timeval structure and the itimerval structure are defined.
// <sys/resource.h> Resource file. Contains information on the limits and utilization of system
//     resources used by processes.
// <signal.h> Signal header file. Define signal symbol constants, signal structures, and signal
//     manipulation function prototypes.
34 #include <linux/head.h>
35 #include <linux/fs.h>
36 #include <linux/mm.h>
37 #include <sys/param.h>
38 #include <sys/time.h>
39 #include <sys/resource.h>
40 #include <signal.h>
41
42 #if (NR_OPEN > 32)
43 #error "Currently the close-on-exec-flags and select masks are in one long, max 32 files/proc"
44 #endif
45
// This defines the operating state in which the process may be.
46 #define TASK_RUNNING      0 // process is running or is ready to run.
47 #define TASK_INTERRUPTIBLE 1 // in an interruptible wait state.
48 #define TASK_UNINTERRUPTIBLE 2 // in an uninterruptible wait state (mainly I/O op).
49 #define TASK_ZOMBIE       3 // in a dead state and the father has not yet signaled.
50 #define TASK_STOPPED      4 // process has stopped.
51
52 #ifndef NULL
53 #define NULL ((void *) 0)
54 #endif
55
// Copy the page directory table of a process. ( mm/memory.c, 118 )
// Linux considers this to be one of the most complex functions in the kernel.
56 extern int copy_page_tables(unsigned long from, unsigned long to, long size);
// Frees the memory specified by the page table and the page table itself. ( mm/memory.c, 69 )
57 extern int free_page_tables(unsigned long from, unsigned long size);
58
// The initialization function of the scheduler. (kernel/sched.c, 417 )
59 extern void sched_init(void);
// Process scheduling function. (kernel/sched.c, 119)
60 extern void schedule(void);
// The initialization function for exception (trap) processing. (kernel/traps.c, 185)
61 extern void trap_init(void);
// Display kernel error messages and then enter an infinite loop. (kernel/panic.c, 16)
62 extern void panic(const char * str);
// Write a string of the specified length to the tty. (kernel/chr_drv/tty_io.c, 339)
63 extern int tty_write(unsigned minor, char * buf, int count);
64
65 typedef int (*fn_ptr)(); // Define a function pointer type.
66
// The following is the structure used by the math coprocessor, which is mainly used to save
// the execution status information of i387 when the process is switched out.
67 struct i387_struct {
68     long    cwd; // Control word.
69     long    swd; // Status word.
70     long    twd; // Tag word.

```

```

71     long    fip;           // Coprocessor code ip pointer.
72     long    fcs;           // Coprocessor code segment register.
73     long    foo;           // The offset of the memory operand.
74     long    fos;           // The segment of the memory operand.
75     long    st_space[20];  /* 8*10 bytes for each FP-reg = 80 bytes */
76 };
77
// Task status segment (TSS) data structure.
78 struct tss\_struct {
79     long    back_link;      /* 16 high bits zero */
80     long    esp0;
81     long    ss0;            /* 16 high bits zero */
82     long    esp1;
83     long    ssl;            /* 16 high bits zero */
84     long    esp2;
85     long    ss2;            /* 16 high bits zero */
86     long    cr3;
87     long    eip;
88     long    eflags;
89     long    eax, ecx, edx, ebx;
90     long    esp;
91     long    ebp;
92     long    esi;
93     long    edi;
94     long    es;              /* 16 high bits zero */
95     long    cs;              /* 16 high bits zero */
96     long    ss;              /* 16 high bits zero */
97     long    ds;              /* 16 high bits zero */
98     long    fs;              /* 16 high bits zero */
99     long    gs;              /* 16 high bits zero */
100    long    ldt;             /* 16 high bits zero */
101    long    trace_bitmap;    /* bits: trace 0, bitmap 16-31 */
102    struct i387\_struct i387;
103 };
104
// Below is the task (process) data structure, or process control block, or process descriptor.
// See section 5.7 for detailed descriptions.
//struct task_struct {
//    long state;              // -1 unrunnable, 0 runnable (ready), > 0 stopped.
//    long counter;            // Task run time tick (decrement), run time slice.
//    long priority;           // Priority. When task starts running, counter=priority.
//    long signal;             // Signal bitmap, each bit is a signal( = bit offset + 1).
//    struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
//    long blocked;            // Process signal mask (Bitmap of masked signal).
//    int exit_code;           // Exit code after task stops, its parent will get it.
//    unsigned long start_code; // Code start location in linear address space.
//    unsigned long end_code;   // Code length or size (bytes).
//    unsigned long end_data;   // Code size + data size (bytes).
//    unsigned long brk;        // Total size (number of bytes).
//    unsigned long start_stack; // Stack bottom location.
//    long pid;                // Process identifier.
//    long pgrp;               // Process group number.
//    long session;            // Process session number.

```

```

// long leader;                // Leader session number.
// int groups[NGROUPS];        // Group numbers. A process can belong to more groups.
// task_struct *p_pptr;        // Pointer to parent process.
// task_struct *p_cptr;        // Pointer to youngest child process.
// task_struct *p_ysptr;        // Pointer to younger sibling process created afterwards.
// task_struct *p_osptr;        // Pointer to older sibling process created earlier.
// unsigned short uid;          // User id.
// unsigned short euid;          // Effective user id.
// unsigned short suid;          // Saved user id.
// unsigned short gid;          // Group id.
// unsigned short egid;          // Effective group id.
// unsigned short sgid;          // Saved group id.
// long timeout;                // Kernel timing timeout value.
// long alarm;                  // Alarm timing value (ticks).
// long utime;                  // User state running time (ticks).
// long stime;                  // System state runtime (ticks).
// long ctime;                  // Child process user state runtime.
// long cstime;                  // Child process system state runtime.
// long start_time;              // Time the process started running.
// struct rlimit rlim[RLIM_NLIMITS]; // Resource usage statistics array.
// unsigned int flags;           // per process flags.
// unsigned short used_math;      // Flag: Whether a coprocessor is used.
// int tty;                      // The tty subdevice number used. -1 means no use.
// unsigned short umask;          // The mask bit of the file creation attribute.
// struct m_inode *pwd;           // Current working directory i node structure pointer.
// struct m_inode *root;          // Root i-node structure pointer.
// struct m_inode *executable;    // The pointer to i-node structure of the executable file.
// struct m_inode *library;        // The loaded library i-node structure pointer.
// unsigned long close_on_exec;    // A bitmap flags that close file handles on execution.
// struct file *filp[NR_OPEN];    // File structure pointer table, up to 32 items.
//                               // The index is the value of file descriptor.
// struct desc_struct ldt[3];      // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
// struct tss_struct tss;          // The task status segment structure TSS of the process.
//};

105 struct task_struct {
106     /* these are hardcoded - don't touch */
107     long state;                /* -1 unrunnable, 0 runnable, >0 stopped */
108     long counter;
109     long priority;
110     long signal;
111     struct sigaction sigaction[32];
112     long blocked;              /* bitmap of masked signals */
113     /* various fields */
114     int exit_code;
115     unsigned long start_code, end_code, end_data, brk, start_stack;
116     long pid, pgrp, session, leader;
117     int groups[NGROUPS];
118     /*
119     * pointers to parent process, youngest child, younger sibling,
120     * older sibling, respectively. (p->father can be replaced with
121     * p->p_pptr->pid)
122     */
123     struct task_struct *p_pptr, *p_cptr, *p_ysptr, *p_osptr;

```

```

124     unsigned short uid,euid,suid;
125     unsigned short gid,egid,sgid;
126     unsigned long timeout,alarm;
127     long utime,stime,cutime,cstime,start_time;
128     struct rlimit rlim[RLIM_NLIMITS];
129     unsigned int flags;      /* per process flags, defined below */
130     unsigned short used_math;
131 /* file system info */
132     int tty;                /* -1 if no tty, so it must be signed */
133     unsigned short umask;
134     struct m_inode * pwd;
135     struct m_inode * root;
136     struct m_inode * executable;
137     struct m_inode * library;
138     unsigned long close_on_exec;
139     struct file * filp[NR_OPEN];
140 /* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
141     struct desc_struct ldt[3];
142 /* tss for this task */
143     struct tss_struct tss;
144 };
145
146 /*
147  * Per process flags
148  */
149 #define PF_ALIGNWARN      0x00000001      /* Print alignment warning msgs */
150                                         /* Not implemented yet, only for 486*/
151
152 /*
153  * INIT_TASK is used to set up the first task table, touch at
154  * your own risk!. Base=0, limit=0x9ffff (=640kB)
155  */
156 // Hard-coded information corresponding to the first task of the above task structure.
157 #define INIT_TASK \
158 /* state etc */ { 0,15,15, \           // state, counter, priority
159 /* signals */ 0,{},{},0, \           // signal, sigaction[32], blocked
160 /* ec,brk... */ 0,0,0,0,0,0, \       // exit_code,start_code,end_code,end_data,brk,start_stack
161 /* pid etc.. */ 0,0,0,0, \           // pid, pgrp, session, leader
162 /* suppl grps*/ {NOGROUP}, \        // groups[]
163 /* proc links*/ &init_task.task,0,0,0, \ // p_pptr, p_cptr, p_ysptr, p_osptr
164 /* uid etc */ 0,0,0,0,0,0, \         // uid, euid, suid, gid, egid, sgid
165 /* timeout */ 0,0,0,0,0,0,0, \       // alarm,utime,stime,cutime,cstime,start_time,used_math
166 /* rlimits */ { {0xffffffff, 0xffffffff}, {0xffffffff, 0xffffffff}, \
167                {0xffffffff, 0xffffffff}, {0xffffffff, 0xffffffff}, \
168                {0xffffffff, 0xffffffff}, {0xffffffff, 0xffffffff}}, \
169 /* flags */ 0, \                     // flags
170 /* math */ 0, \                     // used_math, tty, umask, pwd, root, executable, close_on_exec
171 /* fs info */ -1,0022,NULL,NULL,NULL,NULL,0, \
172 /* filp */ {NULL}, \                // filp[20]
173 { \                                  // ldt[3]
174 {0,0}, \
175 {0x9f,0xc0fa00}, \ // code size 640K,base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0xa
176 {0x9f,0xc0f200}, \ // data size 640K,base 0x0,G=1,D=1,DPL=3,P=1 TYPE=0x2

```

```

176     }, \
177 /*tss*/ {0, PAGE_SIZE+(long)&init_task, 0x10, 0, 0, 0, 0, (long)&pg_dir, \      // tss
178          0, 0, 0, 0, 0, 0, 0, \
179          0, 0, 0x17, 0x17, 0x17, 0x17, 0x17, 0x17, \
180          LDT(0), 0x80000000, \
181          {} \
182     }, \
183 }
184
185 extern struct task_struct *task[NR_TASKS];      // An array of task pointers.
186 extern struct task_struct *last_task_used_math;
187 extern struct task_struct *current;             // The current process pointer.
188 extern unsigned long volatile jiffies;          // Ticks (10ms/tick) from start of boot.
189 extern unsigned long startup_time;              // Boot time. seconds since 1970:0:0:0.
190 extern int jiffies_offset;                      // The number of ticks that need to be adjusted.
191
192 #define CURRENT_TIME (startup_time+(jiffies+jiffies_offset)/HZ) // Current time(seconds).
193
194 // Add a timer (ticks, call the function *fn() when timing is up). (kernel/sched.c )
195 extern void add_timer(long jiffies, void (*fn)(void));
196 // Uninterruptible waiting for sleep. (kernel/sched.c)
197 extern void sleep_on(struct task_struct ** p);
198 // Interrupted waiting for sleep. (kernel/sched.c )
199 extern void interruptible_sleep_on(struct task_struct ** p);
200 // Clearly wake up the process of sleep. (kernel/sched.c )
201 extern void wake_up(struct task_struct ** p);
202 // Check if the current process is in the specified user group grp.
203 extern int in_group_p(gid_t grp);
204
205 /*
206  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
207  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
208  */
209 // Look for the entry of the first TSS in the global table. 0-nul (not used), 1-code segment
210 // cs, 2-data segment ds, 3-system segment syscall, 4-task state segment TSS0, 5-local table
211 // LTD0, 6-task state segment TSS1, and so on.
212 // As can be guessed from the original comment, Linus had wanted to put the code of the system
213 // call in the fourth independent segment of the GDT table. But then did not do that, so he
214 // kept the fourth descriptor item (the syscall item) in the GDT table idle.
215 // The following define selector indexes of the first TSS and LDT descriptors in GDT table.
216 #define FIRST_TSS_ENTRY 4
217 #define FIRST_LDT_ENTRY (FIRST_TSS_ENTRY+1)
218
219 // This macro is used to calculate the selector (offset) of the TSS descriptor of the n-th task
220 // in the GDT. Since each descriptor occupies 8 bytes, FIRST_TSS_ENTRY<<3 indicates the starting
221 // offset position of the descriptor in the GDT table. Since each task uses 1 TSS and 1 LDT
222 // descriptor, which occupies a total of 16 bytes, n<<4 is required to indicate the corresponding
223 // TSS start position. The value obtained by this macro is also the index value of the selector
224 // of the TSS. The latter macro defines the selector (offset) of the LDT descriptor in the GDT.
225 #define TSS(n) (((unsigned long) n)<<4)+(FIRST_TSS_ENTRY<<3))
226 #define LDT(n) (((unsigned long) n)<<4)+(FIRST_LDT_ENTRY<<3))
227
228 // The embedded assembly macro is used to load the TSS segment selector of the n-th task into

```

```

// the task register TR. The latter macro is used to load the LDT segment selector of the n-th
// task into the local descriptor table register LDTR.
208 #define ltr(n) __asm__ ("ltr %%ax::"a" (TSS(n)))
209 #define lldt(n) __asm__ ("lldt %%ax::"a" (LDT(n)))

// Get the task number of the current running task (it is the index in the task array, which
// is different from the process number pid), used in program kernel/traps.c, line 78.
// Returns: n - the current task number.
210 #define str(n) \
211 __asm__ ("str %%ax|n|t" \           // Save the TSS selector in the task register to EAX.
212         "subl %2, %%eax|n|t" \     // (EAX - FIRST_TSS_ENTRY * 8) => EAX
213         "shrl $4, %%eax" \         // (EAX / 16) => EAX = current task number.
214         : "=a" (n) \
215         : "a" (0), "i" (FIRST TSS ENTRY<<3))
216 /*
217 *      switch_to(n) should switch tasks to task nr n, first
218 * checking that n isn't the current task, in which case it does nothing.
219 * This also clears the TS-flag if the task we switched to has used
220 * the math co-processor latest.
221 */
// Jumping to a address composed by TSS selector will cause the CPU to switch task.
// Input: %0 - points to __tmp; %1 - points to __tmp.b for the new TSS selector; DX - TSS segment
// selector for new task n; ECX - task structure pointer for new task n task[n].
// The temporary data structure __tmp is used to construct the operand of the far jump instruction
// on line 228. The operand consists of a 4-byte offset address and a 2-byte segment selector.
// Therefore, the value of 'a' in __tmp is a 32-bit offset, and the lower 2 bytes of 'b' are
// selectors for the new TSS segment (high 2 bytes are not used). Jumping to the TSS segment
// selector causes the task to switch to the process corresponding to the TSS. The 'a' value
// is useless for long jumps that cause task switching. The indirect jump instruction on line
// 228 uses a 6-byte operand as the long pointer to the jump destination. The format is:
//      JMP 16-bit segment selector: 32-bit offset.
// After the task is switched back, it is determined by comparing the original task pointer
// with the last used coprocessor task pointer saved in the last_task_used_math variable when
// determining whether the original task was executed last time. See description of the
// math_state_restore() function in kernel/sched.c.
222 #define switch\_to(n) {\
223 struct {long a,b;} __tmp; \
224 __asm__ ("cmpl %%ecx, _current|n|t" \   // Is task n the current task? (current ==task[n]?)
225         "je 1f|n|t" \                 // If so, do nothing and return.
226         "movw %%dx, %1|n|t" \         // The new task TSS selector is stored in __tmp.b
227         "xchgl %%ecx, _current|n|t" \  // Current = task[n]; ECX =the task switched out.
228         "ljmp %0|n|t" \               // Long jump to *&__tmp, causing task switching.
// The following statement will not continue until the task is switched back. First check if
//the original task used the coprocessor last time. If yes, clear the TS flag in CR0.
229         "cmpl %%ecx, _last_task_used_math|n|t" \ // original task used math last time?
230         "jne 1f|n|t" \                 // If not, jump forward and exit.
231         "clts|n" \                     // If yes, then clear the TS flag in CR0.
232         "l:" \                         // exit.
233         ::"m" (&__tmp.a), "m" (&__tmp.b), \
234         "d" (TSS(n)), "c" ((long) task[n])); \
235 }
236
// Page alignment (nowhere in the kernel to reference this!)

```

```

237 #define PAGE_ALIGN(n) (((n)+0xfff)&0xfffff000)
238
// Set each base address fields in the descriptor at address addr.
// %0 - addr offset 2; %1 - addr offset 4; %2 - addr offset 7; EDX - base address.
239 #define set_base(addr, base) \
240 __asm__( "movw %%dx, %0|n|t" \           // Lower 16 bits(15-0) of the base=>[addr+2].
241         "rorl $16, %%edx|n|t" \         // Upper 16 bits(31-16) of the base in EDX => DX.
242         "movb %%dl, %1|n|t" \           // Lower 8 bits(23-16) of the upper 16 bits =>[addr+4]
243         "movb %%dh, %2" \               // Upper 8 bits(31-24) of the upper 16 bits =>[addr+7]
244         :: "m" (*(addr+2)), \
245         "m" (*(addr+4)), \
246         "m" (*(addr+7)), \
247         "d" (base) \
248         : "dx")                        // Tell gcc that the value in EDX has been changed.
249
// Set the segment limit field in the descriptor at address addr.
// %0 - address addr; %1 - addr offset 6; EDX - segment length limit.
250 #define set_limit(addr, limit) \
251 __asm__( "movw %%dx, %0|n|t" \           // Lower 16 bits(15-0) of the segment limit =>[addr].
252         "rorl $16, %%edx|n|t" \         // Upper 4 bits(19-16) of the limit in EDX => DL.
253         "movb %1, %%dh|n|t" \           // Original [addr+6] => DH, the upper 4 bits are flags.
254         "andb $0xf0, %%dh|n|t" \        // Clear lower 4 bits of DH (will be stored at 19-16).
255         "orb %%dh, %%dl|n|t" \          // Combine original high 4-bit flags and upper 4 bits
256         "movb %%dl, %1" \               // (19-16)of the limit into 1 byte, stored at [addr+6].
257         :: "m" (*(addr)), \
258         "m" (*(addr+6)), \
259         "d" (limit) \
260         : "dx")
261
// Set the base address field of the descriptor in the local descriptor table LDT.
// Set the segment limit field of the descriptor in the LDT.
262 #define set_base(ldt, base) set_base( ((char *)&(ldt)) , base )
263 #define set_limit(ldt, limit) set_limit( ((char *)&(ldt)) , (limit-1)>>12 )
264
// Get base from the descriptor at the address addr. It is the opposite of _set_base().
// EDX - store base (__base); %1 - addr offset 2; %2 - addr offset 4; %3 - addr offset 7.
265 #define get_base(addr) ({\
266 unsigned long __base; \
267 __asm__( "movb %3, %%dh|n|t" \           // Upper 8 bits(31-24) of upper 16 bits [addr+7] =>DH.
268         "movb %2, %%dl|n|t" \           // Lower 8 bits(23-16) of upper 16 bits [addr+4] =>DL.
269         "shll $16, %%edx|n|t" \         // Upper 16 bits are moved to the upper 16 bits of EDX.
270         "movw %1, %%dx" \               // Lower 16 (15-0) bits of the base at [addr+2] => DX.
271         : "=d" (__base) \               // Thus EDX contains a 32-bit segment base address.
272         : "m" (*(addr+2)), \
273         "m" (*(addr+4)), \
274         "m" (*(addr+7))) ; \
275 __base;})
276
// Take the base address in the segment descriptor pointed to by 'ldt' in the LDT.
277 #define get_base(ldt) get_base( ((char *)&(ldt)) )
278
// Take the segment limit in the descriptor specified by the segment selector 'segment'.
// The instruction LSL is an abbreviation of Load Segment Limit. It takes the scattered limit

```

```

// length bits from the descriptor of the specified segment and puts the complete segment limit
// value into the specified register. The resulting segment limit value is the actual number
// of bytes minus 1, so you need to add 1 to return.
// %0 - the length of the segment (bytes); %1 - the segment selector 'segment'.
279 #define get\_limit(segment) ({ \
280 unsigned long __limit; \
281 __asm__( "lsl %1, %0\n\tincl %0": "=r" (__limit): "r" (segment)); \
282 __limit;})
283
284 #endif
285

```

14.30 sys.h

14.30.1 Functionality

The sys.h header file lists the prototypes of all system-call functions in the kernel, as well as the system-call function pointer table.

14.30.2 Code annotation

Program 14-27 linux/include/linux/sys.h

```

1  /*
2   * Why isn't this a .c file?  Enquiring minds....
3   */
4
5  extern int sys\_setup();           // 0 - System initializations.      (kernel/blk_drv/hd.c, 74)
6  extern int sys\_exit();           // 1 - Program exit.              (kernel/exit.c, 365)
7  extern int sys\_fork();           // 2 - Create a new process.      (kernel/sys_call.s, 222)
8  extern int sys\_read();           // 3 - Read file.                (fs/read_write.c, 55)
9  extern int sys\_write();          // 4 - Write file.               (fs/read_write.c, 83)
10 extern int sys\_open();            // 5 - Open file.                (fs/open.c, 171)
11 extern int sys\_close();           // 6 - Close file.               (fs/open.c, 219)
12 extern int sys\_waitpid();         // 7 - Wait process to terminate. (kernel/exit.c, 370)
13 extern int sys\_creat();           // 8 - Create file.              (fs/open.c, 214)
14 extern int sys\_link();            // 9 - Ceate hard linke to a file. (fs/namei.c, 837)
15 extern int sys\_unlink();          // 10 - Delete a filename(or file) (fs/namei.c, 709)
16 extern int sys\_execve();          // 11 - Execute a program.        (kernel/sys_call.s, 214)
17 extern int sys\_chdir();           // 12 - Change current directory.  (fs/open.c, 76)
18 extern int sys\_time();           // 13 - Get current time.         (kernel/sys.c, 134)
19 extern int sys\_mknod();           // 14 - Create block/char special file. (fs/namei.c, 464)
20 extern int sys\_chmod();           // 15 - Change file mode.         (fs/open.c, 106)
21 extern int sys\_chown();           // 16 - Chane file owner or group.  (fs/open.c, 122)
22 extern int sys\_break();           // 17 -                           (kernel/sys.c, 33)*
23 extern int sys\_stat();            // 18 - Get file status using path name. (fs/stat.c, 36)
24 extern int sys\_lseek();           // 19 - Reposite r/w file offset.  (fs/read_write.c, 25)
25 extern int sys\_getpid();          // 20 - Get process id.           (kernel/sched.c, 380)
26 extern int sys\_mount();           // 21 - Mount a file-system.       (fs/super.c, 199)
27 extern int sys\_umount();          // 22 - Unmount a file-system.     (fs/super.c, 166)

```

```

28 extern int sys\_setuid(); // 23 - Set process user id. (kernel/sys.c, 196)
29 extern int sys\_getuid(); // 24 - Get process user id. (kernel/sched.c, 390)
30 extern int sys\_stime(); // 25 - Set system time. (kernel/sys.c, 207)
31 extern int sys\_ptrace(); // 26 - Process tracing. (kernel/sys.c, 38)*
32 extern int sys\_alarm(); // 27 - Set alarm time. (kernel/sched.c, 370)
33 extern int sys\_fstat(); // 28 - Get file status by using handle. (fs/stat.c, 58)
34 extern int sys\_pause(); // 29 - Pause the process running. (kernel/sched.c, 164)
35 extern int sys\_utime(); // 30 - Set file access and modified time. (fs/open.c, 25)
36 extern int sys\_stty(); // 31 - Modify terminal settings. (kernel/sys.c, 43)*
37 extern int sys\_gtty(); // 32 - Get terminal settings. (kernel/sys.c, 48)*
38 extern int sys\_access(); // 33 - Check file access permission. (fs/open.c, 48)
39 extern int sys\_nice(); // 34 - Set execution priority. (kernel/sched.c, 410)
40 extern int sys\_ftime(); // 35 - Get date and time. (kernel/sys.c, 28)*
41 extern int sys\_sync(); // 36 - Synchronous data with dev. (fs/buffer.c, 44)
42 extern int sys\_kill(); // 37 - Terminate a process. (kernel/exit.c, 205)
43 extern int sys\_rename(); // 38 - Change filename. (kernel/sys.c, 53)*
44 extern int sys\_mkdir(); // 39 - Make a directory. (fs/namei.c, 515)
45 extern int sys\_rmdir(); // 40 - Remove a directory. (fs/namei.c, 635)
46 extern int sys\_dup(); // 41 - Duplicate a file handle. (fs/fcntl.c, 42)
47 extern int sys\_pipe(); // 42 - Create a pipe. (fs/pipe.c, 76)
48 extern int sys\_times(); // 43 - Get running time. (kernel/sys.c, 216)
49 extern int sys\_prof(); // 44 - Execution time zone. (kernel/sys.c, 58)*
50 extern int sys\_brk(); // 45 - Change data segment len. (kernel/sys.c, 228)
51 extern int sys\_setgid(); // 46 - Set process group id. (kernel/sys.c, 98)
52 extern int sys\_getgid(); // 47 - Set process group id. (kernel/sched.c, 400)
53 extern int sys\_signal(); // 48 - Signal processing. (kernel/signal.c, 85)
54 extern int sys\_geteuid(); // 49 - Get efficient user id. (kernel/sched.c, 395)
55 extern int sys\_getegid(); // 50 - Get efficient group id. (kernel/sched.c, 405)
56 extern int sys\_acct(); // 51 - Process accounting. (kernel/sys.c, 109)*
57 extern int sys\_phys(); // 52 - Map phy mem to process space. (kernel/sys.c, 114)*
58 extern int sys\_lock(); // 53 - (kernel/sys.c, 119)*
59 extern int sys\_ioctl(); // 54 - Device i/o control. (fs/ioctl.c, 31)
60 extern int sys\_fcntl(); // 55 - File operation control. (fs/fcntl.c, 47)
61 extern int sys\_mpx(); // 56 - (kernel/sys.c, 124)*
62 extern int sys\_setpgid(); // 57 - Set process group id. (kernel/sys.c, 245)
63 extern int sys\_ulimit(); // 58 - Statistical resource usage. (kernel/sys.c, 129)
64 extern int sys\_uname(); // 59 - Show system info. (kernel/sys.c, 343)
65 extern int sys\_umask(); // 60 - Get default file creation mode. (kernel/sys.c, 515)
66 extern int sys\_chroot(); // 61 - Change root directory. (fs/open.c, 91)
67 extern int sys\_ustat(); // 62 - Get file system states. (fs/open.c, 20)
68 extern int sys\_dup2(); // 63 - Duplicate file handle. (fs/fcntl.c, 36)
69 extern int sys\_getppid(); // 64 - Get parent process id. (kernel/sched.c, 385)
70 extern int sys\_getpgrp(); // 65 - Get pid (getpgid(0)) (kernel/sys.c, 271)
71 extern int sys\_setsid(); // 66 - Set new session id. (kernel/sys.c, 276)
72 extern int sys\_sigaction(); // 67 - Set signal operation. (kernel/signal.c, 100)
73 extern int sys\_sgetmask(); // 68 - Get signal mask code. (kernel/signal.c, 14)
74 extern int sys\_ssetmask(); // 69 - Set signal mask code. (kernel/signal.c, 19)
75 extern int sys\_setreuid(); // 70 - Set real/efficient uid. (kernel/sys.c, 159)
76 extern int sys\_setregid(); // 71 - Set real/efficient pid. (kernel/sys.c, 74)
77 extern int sys\_sigpending(); // 73 - Check pending signals. (kernel/signal.c, 27)
78 extern int sys\_sigsuspend(); // 72 - Suspending a process. (kernel/signal.c, 48)
79 extern int sys\_sethostname(); // 74 - Set host name. (kernel/sys.c, 357)
80 extern int sys\_setrlimit(); // 75 - Set resource using limit. (kernel/sys.c, 387)

```

```

81 extern int sys_getrlimit(); // 76 - Get resource using limit. (kernel/sys.c, 375)
82 extern int sys_getrusage(); // 77 - Get resource usage. (kernel/sys.c, 412)
83 extern int sys_gettimeofday(); // 78 - Get time of the day. (kernel/sys.c, 440)
84 extern int sys_settimeofday(); // 79 - Set time of the day. (kernel/sys.c, 466)
85 extern int sys_getgroups(); // 80 - Get process all group ids. (kernel/sys.c, 289)
86 extern int sys_setgroups(); // 81 - Set process group array. (kernel/sys.c, 307)
87 extern int sys_select(); // 82 - wait for file change state. (fs/select.c, 216)
88 extern int sys_symlink(); // 83 - Create file sysmbol link. (fs/namei.c, 767)
89 extern int sys_lstat(); // 84 - Get link file state. (fs/stat.c, 47)
90 extern int sys_readlink(); // 85 - Read link file contents. (fs/stat.c, 69)
91 extern int sys_uselib(); // 86 - Select shared lib. (fs/exec.c, 42)
92
// The function pointer table of the system-call, which is used by the system-call interrupt
// handler (int 0x80) as a jump table.
93 fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
94 sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link,
95 sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod,
96 sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount,
97 sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm,
98 sys_fstat, sys_pause, sys_utime, sys_stty, sys_gtty, sys_access,
99 sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir,
100 sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid,
101 sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys,
102 sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit,
103 sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid,
104 sys_getpgrp, sys_setsid, sys_sigaction, sys_sgetmask, sys_ssetmask,
105 sys_setreuid, sys_setregid, sys_sigsuspend, sys_sigpending, sys_sethostname,
106 sys_setrlimit, sys_getrlimit, sys_getrusage, sys_gettimeofday,
107 sys_settimeofday, sys_getgroups, sys_setgroups, sys_select, sys_symlink,
108 sys_lstat, sys_readlink, sys_uselib };
109
110 /* So we don't have to do any more manual updating... */
111 int NR_syscalls = sizeof(sys_call_table)/sizeof(fn_ptr);
112

```

14.31 tty.h

14.31.1 Functionality

The tty.h file defines the terminal data structure and some constants, as well as the macros used by the tty queue buffer operations.

14.31.2 Code annotation

Program 14-28 linux/include/linux/tty.h

```

1 /*
2  * 'tty.h' defines some structures used by tty_io.c and some defines.
3  *
4  * NOTE! Don't touch this without checking that nothing in rs_io.s or
5  * con_io.s breaks. Some constants are hardwired into the system (mainly

```



```

6  * offsets into 'tty_queue'
7  */
8
9  #ifndef TTY_H
10 #define TTY_H
11
12 #define MAX_CONSOLES 8          // The maximum number of virtual consoles.
13 #define NR_SERIALS 2           // The number of serial terminals.
14 #define NR_PTYS 4              // The number of psesudo terminals.
15
16 extern int NR_CONSOLES;        // The number of virtual consoles.
17
18 // <termios.h> Terminal input and output function header file. It mainly defines the terminal
19 // interface that controls the asynchronous communication port.
20 #include <termios.h>
21
22 #define TTY_BUF_SIZE 1024      // The size of the tty queue buffer.
23
24 // Tty character buffer queue data structure. Used for read, write, and auxiliary (canonical)
25 // buffer queues in the tty_struc structure.
26 // The first field 'data' contains the character line value (not the number of characters) in
27 // the queue buffer. For a serial terminal, the serial port address is stored therein.
28 struct tty_queue {
29     unsigned long data;        // Char lines in the buffer or the serial port.
30     unsigned long head;        // The data header in the buffer.
31     unsigned long tail;        // The data tail in the buffer.
32     struct task_struct * proc_list; // process list waiting for this queue buffer.
33     char buf[TTY_BUF_SIZE];    // The buffer.
34 };
35
36 // These macros are used to check the terminal type.
37 #define IS_A_CONSOLE(min)      (((min) & 0xC0) == 0x00) // console.
38 #define IS_A_SERIAL(min)       (((min) & 0xC0) == 0x40) // serial terminal.
39 #define IS_A_PTY(min)          (((min) & 0x80) == 0x80) // psesudo terminal.
40 #define IS_A_PTY_MASTER(min)   (((min) & 0xC0) == 0x80) // master pty.
41 #define IS_A_PTY_SLAVE(min)    (((min) & 0xC0) == 0xC0) // slate pty.
42 #define PTY_OTHER(min)         ((min) ^ 0x40)          // other type terminal.
43
44 // The buffer operation macros in the tty queue are defined below. (tail is in front, head is
45 // in the back, see figure in tty_io.c).
46 // The buffer pointer 'a' is shifted forward/backward by 1 byte, and if it has exceeded the
47 // right/left side of the buffer, the pointer moves cyclically.
48 #define INC(a) ((a) = ((a)+1) & (TTY_BUF_SIZE-1))
49 #define DEC(a) ((a) = ((a)-1) & (TTY_BUF_SIZE-1))
50 // Empty the buffer. // The size of the remaining free area of the buffer.
51 // The last position in the buffer. // Buffer is full. // The number of chars in the buffer.
52 #define EMPTY(a) ((a)->head == (a)->tail)
53 #define LEFT(a) (((a)->tail-(a)->head-1)&(TTY_BUF_SIZE-1))
54 #define LAST(a) ((a)->buf[(TTY_BUF_SIZE-1)&((a)->head-1)])
55 #define FULL(a) (!LEFT(a))
56 #define CHARS(a) (((a)->head-(a)->tail)&(TTY_BUF_SIZE-1))
57 // Get a character from the 'tail' in the 'queue' buffer, and tail++.
58 // Place a character at the 'head' in the 'queue' queue buffer, and head++.

```

```

44 #define GETCH(queue, c) \
45 (void) ({c=(queue)->buf[(queue)->tail]; INC((queue)->tail);})
46 #define PUTCH(c, queue) \
47 (void) ({(queue)->buf[(queue)->head]=(c); INC((queue)->head);})
48
// Check the type of characters typed on the terminal keyboard.
49 #define INTR\_CHAR(tty) ((tty)->termios.c_cc[VINTR]) // Send signal SIGINT.
50 #define QUIT\_CHAR(tty) ((tty)->termios.c_cc[VQUIT]) // Send signal SIGQUIT.
51 #define ERASE\_CHAR(tty) ((tty)->termios.c_cc[VERASE]) // Erase a char.
52 #define KILL\_CHAR(tty) ((tty)->termios.c_cc[VKILL]) // Kill a line of chars.
53 #define EOF\_CHAR(tty) ((tty)->termios.c_cc[VEOF]) // End of file char.
54 #define START\_CHAR(tty) ((tty)->termios.c_cc[VSTART]) // Start output.
55 #define STOP\_CHAR(tty) ((tty)->termios.c_cc[VSTOP]) // Stop output.
56 #define SUSPEND\_CHAR(tty) ((tty)->termios.c_cc[VSUSP]) // Send signal SIGTSTP.
57
// Terminal data structure.
58 struct tty\_struct {
59     struct termios termios; // Terminal io mode & control structure.
60     int pgrp; // The pgroup the terminal belongs to.
61     int session; // The session.
62     int stopped; // Terminal stopped flag.
63     void (*write)(struct tty\_struct * tty); // tty write function pointer.
64     struct tty\_queue *read_q; // tty read queue.
65     struct tty\_queue *write_q; // tty write queue.
66     struct tty\_queue *secondary; // tty aux or canonical queue.
67 };
68
69 extern struct tty\_struct tty\_table[];
70 extern int fg\_console; // Front console number.
71
// The following macro obtains a pointer to the tty structure corresponding to the terminal
// number 'nr' in tty\_table[] according to the terminal type.
// The second half of line 73 is used to select the corresponding tty structure in the tty\_table[]
// table based on the sub-device number 'dev'. If dev = 0, it means that the foreground terminal
// is being used, so you can use the terminal number 'fg_console' as the tty\_table[] entry index
// to get the tty structure. If dev is greater than 0, then it should be considered in two cases:
// (1) dev is the virtual terminal number; (2) dev is the serial terminal number or pseudo terminal
// number. For virtual terminals, the tty structure in tty\_table[] is indexed by dev-1(0 --
// 63). For other types of terminals, their tty structure index entry is dev.
// For example, if dev = 64, which means it is a serial terminal 1, its tty structure is
// ttb\_table[dev]. If dev = 1, the tty structure of the corresponding terminal is tty\_table[0].
// See lines 70--73 of the tty\_io.c program.
72 #define TTY\_TABLE(nr) \
73 (tty\_table + ((nr) ? ((nr) < 64)? (nr)-1:(nr)) : fg\_console)
74
// Here is the initial value of the special character array c_cc[] that can be changed in the
// terminal termios structure. POSIX.1 defines 11 special characters, but the Linux system
// additionally defines the six special characters used by SVR4. If \_POSIX\_VDISABLE(\0) is
// defined, then when an item value is equal to \_POSIX\_VDISABLE, the corresponding special
// character is prohibited. They are represented by octals in the initial values on line 81.
75 /*      intr=^C      quit=^|      erase=del      kill=^U
76         eof=^D      vtime=\0      vmin=\1      sxtc=\0
77         start=^Q     stop=^S      susp=^Z      eol=\0









```

```
78      reprint=^R      discard=^U      werase=^W      lnext=^V
79      eol2=\0
80  */
81  #define INIT_C_CC "\003\034\177\025\004\0\1\0\021\023\032\0\022\017\027\026\0"
82
83  void rs_init(void);          // serial terminal init. (kernel/chr_drv/serial.c)
84  void con_init(void);        // Control terminal init. (kernel/chr_drv/console.c)
85  void tty_init(void);        // (kernel/chr_drv/tty_io.c)
86
87  int tty_read(unsigned c, char * buf, int n);    // (kernel/chr_drv/tty_io.c)
88  int tty_write(unsigned c, char * buf, int n);   // (kernel/chr_drv/tty_io.c)
89
90  void con_write(struct tty_struct * tty);        // (kernel/chr_drv/console.c)
91  void rs_write(struct tty_struct * tty);         // (kernel/chr_drv/serial.c)
92  void mpty_write(struct tty_struct * tty);       // (kernel/chr_drv/pty.c)
93  void spty_write(struct tty_struct * tty);       // (kernel/chr_drv/pty.c)
94
95  void copy_to_cooked(struct tty_struct * tty);   // (kernel/chr_drv/tty_io.c)
96
97  void update_screen(void);                       // (kernel/chr_drv/console.c)
98
99  #endif
100
```

14.32 Header files in the include/sys/ directory

The include/sys/ directory contains eight header files that are closely related to the system hardware resources and their settings, as shown in Listing 14-4.

List 14-4 Files in the linux/include/sys/ directory

	Filename	Size	Last Modified Time (GMT)	Description
	param.h	196 bytes	1992-01-06 21:10:22	
	resource.h	1809 bytes	1992-01-03 18:52:56	
	stat.h	1376 bytes	1992-01-11 18:42:48	
	time.h	1799 bytes	1992-01-09 03:51:28	
	times.h	200 bytes	1991-09-17 15:03:06	
	types.h	928 bytes	1992-01-14 13:50:35	
	utsname.h	272 bytes	1992-01-04 15:05:42	
	wait.h	593 bytes	1991-12-22 15:08:01	

14.33 param.h

14.33.1 Functionality

The param.h file contains and defines some parameter values related to the system hardware.

14.33.2 Code annotation

Program 14-29 linux/include/sys/param.h

```

1 #ifndef _SYS_PARAM_H
2 #define _SYS_PARAM_H
3
4 #define HZ 100                // The system clock frequency, 100 times per second.
5 #define EXEC_PAGESIZE 4096    // Executable page size.
6
7 #define NGROUPS 32            /* Max number of groups per user */
8 #define NOGROUP -1
9
10 #define MAXHOSTNAMELEN 8      // The maximum length of the host name, 8 bytes.
11
12 #endif
13

```

14.34 resource.h

14.34.1 Functionality

The resource.h header file contains information about the limits and utilization of the system resources used by the process. It defines the rusage structure and symbolic constants RUSAGE_SELF, RUSAGE_CHILDREN used by the system-call (or library function) getrusage(). It also defines the rlimit structure used by system-calls or functions getrlimit() and setrlimit() and the symbol constants used in the arguments.

The information accessed by getrlimit() and setrlimit() is in the rlim[] array of the process task structure. The array has a total of RLIM_NLIMITS items, each of which is an rlimit structure that defines the restrictions on the use of a resource, as shown in Figure 14-5. As shown in the figure, six resource limits are defined for a process in the Linux 0.12 kernel, and they are defined on lines 41-46 in this header file.

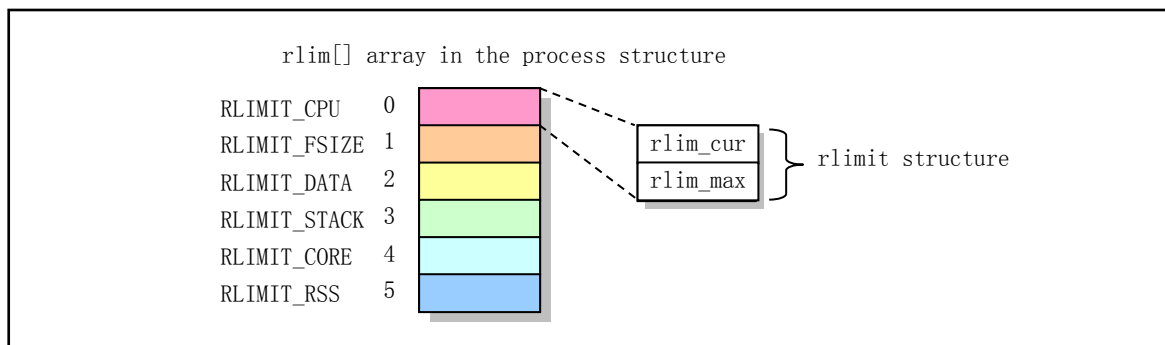


Figure 14-5 The uses of the rlim[] array in the process descriptor

14.34.2 Code annotation

Program 14-30 linux/include/sys/resource.h

```

1  /*
2  * Resource control/accounting header file for linux
3  */
4
5  #ifndef _SYS_RESOURCE_H
6  #define _SYS_RESOURCE_H
7
8  // The following symbol constants and structures are used for getrusage(). See line 412 of the
9  // kernel/sys.c file.
10 /*
11 * Definition of struct rusage taken from BSD 4.3 Reno
12 *
13 * We don't support all of these yet, but we might as well have them...
14 * Otherwise, each time we add new items, programs which depend on this
15 * structure will lose. This reduces the chances of that happening.
16 */
17 // Below is the symbol constants used by the parameter 'who' of the getrusage().
18 // Returns the resource utilization information of the current process.

```

```

// Returns the resource utilization of child processes who has terminated and is waiting.
15 #define RUSAGE_SELF 0 // Returns resource utilization of the current process.
16 #define RUSAGE_CHILDREN -1 // Returns resource utilization of the child processes.
17
// Rusage is the resource utilization statistical structure of the process, and is used by
// getrusage() to return the statistical value of the resource utilization of the specified
// process. The Linux 0.12 kernel uses only the first two fields, which are all timeval structures
// (include/sys/time.h). The ru_utime field is used to store the running time statistics of
// the user in the user state; the ru_stime field is used to store the running time statistics
// of the process in the kernel state.
18 struct rusage {
19     struct timeval ru_utime; /* user time used */
20     struct timeval ru_stime; /* system time used */
21     long ru_maxrss; /* maximum resident set size */
22     long ru_ixrss; /* integral shared memory size */
23     long ru_idrss; /* integral unshared data size */
24     long ru_isrss; /* integral unshared stack size */
25     long ru_minflt; /* page reclaims */
26     long ru_majflt; /* page faults */
27     long ru_nswap; /* swaps */
28     long ru_inblock; /* block input operations */
29     long ru_oublock; /* block output operations */
30     long ru_msgsnd; /* messages sent */
31     long ru_msgrcv; /* messages received */
32     long ru_nsignals; /* signals received */
33     long ru_nvcsw; /* voluntary context switches */
34     long ru_nivcsw; /* involuntary " */
35 };
36
// The following are the symbol constants and structures used by getrlimit() and setrlimit().
37 /*
38 * Resource limits
39 */
// The following are the types of resources defined in the Linux 0.12 kernel. They are the range
// of values for the first parameter resource in getrlimit() and setrlimit(). In fact, these
// symbolic constants are the indexes of the items of the rlim[] array in the process task
// structure. Each item in the rlim[] array is an rlimit structure, see line 58 below.
40
41 #define RLIMIT_CPU 0 /* CPU time in ms */
42 #define RLIMIT_FSIZE 1 /* Maximum filesize */
43 #define RLIMIT_DATA 2 /* max data size */
44 #define RLIMIT_STACK 3 /* max stack size */
45 #define RLIMIT_CORE 4 /* max core file size */
46 #define RLIMIT_RSS 5 /* max resident set size */
47
48 #ifndef notdef
49 #define RLIMIT_MEMLOCK 6 /* max locked-in-memory address space */
50 #define RLIMIT_NPROC 7 /* max number of processes */
51 #define RLIMIT_OFILE 8 /* max number of open files */
52 #endif
53
// This symbolic constant defines the types of resources that are restricted in Linux. The number
// of resource types defined here is 6, so only the first 6 items are valid.

```

```

54 #define RLIM_NLIMITS    6
55
56 #define RLIM_INFINITY    0x7fffffff    // The resource is unlimited or cannot be modified.
57
58 // Resource limit structure.
59 struct rlimit {
60     int    rlim_cur;        // Current resource limit, or soft limit.
61     int    rlim_max;        // Hard limit.
62 };
63 #endif /* _SYS_RESOURCE_H */
64

```

14.35 stat.h

14.35.1 Functionality

The stat.h header file shows the data returned by the file function stat() and its structure type, as well as some property operation test macros and function prototypes.

14.35.2 Code annotation

Program 14-31 linux/include/sys/stat.h

```

1  #ifndef  _SYS_STAT_H
2  #define  _SYS_STAT_H
3
4  #include <sys/types.h>
5
6  // File status data structure. All field values are available from the file's inode structure.
7  struct stat {
8      dev_t    st_dev;        // The device number that contains the file.
9      ino_t    st_ino;        // File i-node number.
10     umode_t   st_mode;       // File type and modes (see below).
11     nlink_t   st_nlink;      // The number of links to the file.
12     uid_t     st_uid;        // The user ID of the file.
13     gid_t     st_gid;        // The group ID of the file.
14     dev_t     st_rdev;       // Device number (if it is a special char or block file).
15     off_t     st_size;       // File size (in bytes).
16     time_t    st_atime;      // Last access time.
17     time_t    st_mtime;      // Last modify time.
18     time_t    st_ctime;      // The time the inode was last changed.
19 };
20
21 //
22 // The following are some of the symbol constants defined for the values used by the st_mode
23 // field. These values are all expressed in octal. For the sake of remembering, these symbolic
24 // names are a combination of the initials or abbreviations of some English words. For example,
25 // each uppercase letter of the name S_IFMT represents the words State, Inode, File, Mask, and
26 // Type; and the name S_IFREG is a combination of the initial letters of State, Inode, File,
27 // and REGular; the name S_IRWXU is State, Inode, Read, Write, eXecute and User. Other names

```

```

// can be deduced by analogy.
// File types:
20 #define S_IFMT 00170000 // File type bit mask (in octal).
21 #define S_IFLNK 0120000 // Symbolic link.
22 #define S_IFREG 0100000 // Regular file.
23 #define S_IFBLK 0060000 // Block special device files, such as harddisk dev/hd0.
24 #define S_IFDIR 0040000 // Directory.
25 #define S_IFCHR 0020000 // Char device file.
26 #define S_IFIFO 0010000 // FIFO special file.
// File mode bits:
// S_ISUID is used to test whether the set-user-ID flag of the file is set. If the flag is set,
// the efficient user ID of the process will be set to the user ID of the file owner when the
// file is executed. S_ISGID performs the same processing for the group ID.
27 #define S_ISUID 0004000 // Set the user ID (set-user-ID) at execution time.
28 #define S_ISGID 0002000 // Set the group ID (set-group-ID) at execution time.
29 #define S_ISVTX 0001000 // For directories, the restricted delete flag.
30
31 #define S_ISLNK(m) ((m) & S_IFMT == S_IFLNK) // Test if it's a symbolic link file.
32 #define S_ISREG(m) ((m) & S_IFMT == S_IFREG) // a regular file.
33 #define S_ISDIR(m) ((m) & S_IFMT == S_IFDIR) // a directory.
34 #define S_ISCHR(m) ((m) & S_IFMT == S_IFCHR) // a char device file.
35 #define S_ISBLK(m) ((m) & S_IFMT == S_IFBLK) // a block device file.
36 #define S_ISFIFO(m) ((m) & S_IFMT == S_IFIFO) // a FIFO special file.
37
// File access permission:
38 #define S_IRWXU 00700 // The owner can read, write, execute/search (U for User).
39 #define S_IRUSR 00400 // The owner can read.
40 #define S_IWUSR 00200 // The owner can write.
41 #define S_IXUSR 00100 // The owner can execute/search.
42
43 #define S_IRWXG 00070 // Group members can read, write, execute/search (G for Group).
44 #define S_IRGRP 00040 // Group members can read.
45 #define S_IWGRP 00020 // Group members can write.
46 #define S_IXGRP 00010 // Group members can execute/search.
47
48 #define S_IRWXO 00007 // Others can read, write, execute/search (O stands for Other).
49 #define S_IROTH 00004 // Others can read (the last 3 letters represent Other).
50 #define S_IWOTH 00002 // Others can write.
51 #define S_IXOTH 00001 // Others can execute/search.
52
53 extern int chmod(const char *_path, mode_t mode); // Change file modes.
54 extern int fstat(int fildes, struct stat *stat_buf); // Get file state info by fhandle.
55 extern int mkdir(const char *_path, mode_t mode); // Make a directory.
56 extern int mkfifo(const char *_path, mode_t mode); // Make a pipe file.
57 extern int stat(const char *filename, struct stat *stat_buf); // Get file state info.
58 extern mode_t umask(mode_t mask); // Set mode mask.
59
60 #endif
61

```

14.36 time.h

14.36.1 Functionality

The time.h header file defines the timeval structure and the internally used itimerval structure, as well as the time zone constants.

14.36.2 Code annotation

Program 14-32 linux/include/sys/time.h

```

1  #ifndef  SYS TIME H
2  #define  SYS TIME H
3
4  /* gettimeofday returns this */
5  struct timeval {
6      long    tv_sec;          /* seconds */
7      long    tv_usec;        /* microseconds */
8  };
9
10 // Time zone structure.
11 // TZ is the abbreviation of Time Zone, and DST is the Daylight Saving Time.
12 struct timezone {
13     int      tz_minuteswest; /* minutes west of Greenwich */
14     int      tz_dsttime;     /* type of dst correction */
15 };
16
17 #define  DST_NONE        0      /* not on dst */
18 #define  DST_USA         1      /* USA style dst */
19 #define  DST_AUST        2      /* Australian style dst */
20 #define  DST_WET         3      /* Western European dst */
21 #define  DST_MET         4      /* Middle European dst */
22 #define  DST_EET         5      /* Eastern European dst */
23 #define  DST_CAN         6      /* Canada */
24 #define  DST_GB          7      /* Great Britain and Eire */
25 #define  DST_RUM         8      /* Rumania */
26 #define  DST_TUR         9      /* Turkey */
27 #define  DST_AUSTALT    10     /* Australian style with shift in 1986 */
28
29 // A setting macros for the file descriptor set, used for the select() function.
30 #define  FD_SET(fd,fdsetp)  (*(fdsetp) |= (1 << (fd)))  // Set the fd in the fd set.
31 #define  FD_CLR(fd,fdsetp)  (*(fdsetp) &= ~(1 << (fd)))  // Clear the fd in the set.
32 #define  FD_ISSET(fd,fdsetp) ((*(fdsetp) >> fd) & 1)      // Is the fd in the set ?
33 #define  FD_ZERO(fdsetp)   (*(fdsetp) = 0)                // Clear all fds in the set.
34
35 /*
36  * Operations on timevals.
37  *
38  * NB: timercmp does not work for >= or <=.
39  */
40 // The operation macro of the timeval time structure.
41 #define  timerisset(tvp)   ((tvp)->tv_sec || (tvp)->tv_usec)

```

```

38 #define timercmp(tvp, uvp, cmp) \
39     ((tvp)->tv_sec cmp (uvp)->tv_sec || \
40      (tvp)->tv_sec == (uvp)->tv_sec && (tvp)->tv_usec cmp (uvp)->tv_usec)
41 #define timerclear(tvp)      ((tvp)->tv_sec = (tvp)->tv_usec = 0)
42
43 /*
44  * Names of the interval timers, and structure
45  * defining a timer setting.
46  */
47 #define ITIMER\_REAL      0          // Decrease in real time.
48 #define ITIMER\_VIRTUAL  1          // Decrease in the virtual time of the process.
49 #define ITIMER\_PROF     2          // Decrease in process virtual time or runtime.
50
51 // Internal time structure.
52 struct itimerval {
53     struct timeval it_interval;    /* timer interval */
54     struct timeval it_value;      /* current value */
55 };
56 #include <time.h>
57 #include <sys/types.h>
58
59 int gettimeofday(struct timeval * tp, struct timezone * tz);
60 int select(int width, fd\_set * readfds, fd\_set * writefds,
61          fd\_set * exceptfds, struct timeval * timeout);
62
63 #endif /* _SYS_TIME_H */
64

```

14.37 times.h

14.37.1 Functionality

The times.h header file mainly defines the file access and modification time structure tms. It will be returned by the times() function. Where time_t is defined in sys/types.h. A function prototype times() is also defined.

14.37.2 Code annotation

Program 14-33 linux/include/sys/times.h

```

1 #ifndef TIMES\_H
2 #define TIMES\_H
3
4 #include <sys/types.h>    // Type header file. The basic system data types are defined.
5
6 struct tms {
7     time\_t tms_ftime;    // CPU time used by the user.
8     time\_t tms_stime;    // CPU time used by the system (kernel).
9

```

```

9      time\_t tms_cutime; // User CPU time used by the terminated child process.
10     time\_t tms_cstime; // The system CPU time used by the terminated child.
11 };
12
13 extern time\_t times(struct tms * tp);
14
15 #endif
16

```

14.38 types.h

14.38.1 Functionality

The types.h header file defines the basic data types. All types are defined as the appropriate mathematical type length. In addition, size_t is an unsigned integer type; off_t is an extended signed integer type; pid_t is a signed integer type.

14.38.2 Code annotation

Program 14-34 linux/include/sys/types.h

```

1  #ifndef _SYS_TYPES_H
2  #define _SYS_TYPES_H
3
4  #ifndef SIZE\_T
5  #define SIZE\_T
6  typedef unsigned int size\_t;           // Used for the size (length) of the object.
7  #endif
8
9  #ifndef TIME\_T
10 #define TIME\_T
11 typedef long time\_t;                   // Used for time (in seconds).
12 #endif
13
14 #ifndef _PTRDIFF_T
15 #define _PTRDIFF_T
16 typedef long ptrdiff\_t;
17 #endif
18
19 #ifndef NULL
20 #define NULL ((void *) 0)
21 #endif
22
23 typedef int pid\_t;                     // Used for process id and process group id.
24 typedef unsigned short uid\_t;          // Used for the user id.
25 typedef unsigned char gid\_t;           // Used for the group id.
26 typedef unsigned short dev\_t;          // Used for the device number.
27 typedef unsigned short ino\_t;          // Used for the inode number.
28 typedef unsigned short mode\_t;         // Used for some file modes.

```

```
29 typedef unsigned short umode\_t;    //
30 typedef unsigned char nlink\_t;      // Used for the file links counting.
31 typedef int daddr\_t;
32 typedef long off\_t;                  // Used for the offset in a file.
33 typedef unsigned char u\_char;        // unsigned char.
34 typedef unsigned short ushort;       // unsigned short.
35
36 typedef unsigned char cc\_t;
37 typedef unsigned int speed\_t;
38 typedef unsigned long tcflag\_t;
39
40 typedef unsigned long fd\_set;        // File descriptor set. Each bit represents 1 descriptor.
41
42 typedef struct { int quot,rem; } div\_t;    // Used for DIV operation.
43 typedef struct { long quot,rem; } ldiv\_t;  // Used for long DIV operation.
44
45 // File system parameter structure for the ustat() function. The last two fields are unused
46 // and always return NULLs.
47 struct ustat {
48     daddr\_t f_tfree;                // Total free blocks in the system.
49     ino\_t f_tinode;                // Total free inodes.
50     char f_fname[6];               // File system name.
51     char f_fpack[6];              // The packed file system name.
52 };
53 #endif
```

14.39 utsname.h

14.39.1 Functionality

utsname.h is the system name structure header file. It defines the utsname structure and the function prototype uname(). This function uses the information in the utsname structure to give information such as the system identifier, version number, and hardware type. In POSIX, the size of the character array should be unspecified, but the data stored in it must be NULL terminated. Therefore, the kernel's utsname structure definition does not meet POSIX requirements (the string array size is defined as 9). In addition, the name utsname is an abbreviation for Unix Timesharing System name.

14.39.2 Code annotation

Program 14-35 linux/include/sys/utsname.h

```
1 #ifndef _SYS_UTSNAME_H
2 #define _SYS_UTSNAME_H
3
4 #include <sys/types.h>    // The basic system data types are defined.
5 #include <sys/param.h>    // Some hardware-related parameter values are given.
6
```

```

6 struct utsname {
7     char sysname[9];           // system name.
8     char nodename[MAXHOSTNAMELEN+1]; // The node name in the network.
9     char release[9];           // release level.
10    char version[9];           // version.
11    char machine[9];           // hardware type.
12 };
13
14 extern int uname(struct utsname * utsbuf);
15
16 #endif
17

```

14.40 wait.h

14.40.1 Functionality

This header file describes the information when the process is waiting, including some symbol constants and wait(), waitpid() function prototype declarations.

14.40.2 Code annotation

Program 14-36 linux/include/sys/wait.h

```

1  #ifndef \_SYS\_WAIT\_H
2  #define \_SYS\_WAIT\_H
3
4  #include <sys/types.h>
5
6  #define LOW(v)          ( (v) & 0377)           // Get the low byte (in octal).
7  #define HIGH(v)         ( ((v) >> 8) & 0377)    // Get the high byte.
8
9  /* options for waitpid, WUNTRACED not supported */
10 // [ Note: In fact, the 0.12 kernel already supports the WUNTRACED option. ]
11 // The following constant symbols are options used in the function waitpid().
12 #define WNOHANG          1                       // Don't hang and return immediately.
13 #define WUNTRACED        2                       // Reports the child status that was stopped.
14
15 // The macros are used to check the meaning of the status word returned by the waitpid().
16 #define WIFEXITED(s)      (!((s)&0xFF))           // True if the child exits normally.
17 #define WIFSTOPPED(s)     (((s)&0xFF)==0x7F)       // True if the child is stopping.
18 #define WEXITSTATUS(s)    (((s)>>8)&0xFF)          // The exit status.
19 #define WTERMSIG(s)        ((s)&0x7F)              // Signal that caused process to terminate.
20 #define WCOREDUMP(s)        ((s)&0x80)              // Check if a core dump has been performed.
21 #define WSTOPSIG(s)         (((s)>>8)&0xFF)          // Signal that caused process to stop.
22 // True if the child process exited due to an uncaptured signal.
23 #define WIFSIGNALED(s)     (((unsigned int)(s)-1 & 0xFFFF) < 0xFF)
24
25 // The wait() and waitpid() functions allow a process to get state information for one of its

```

```
// child processes. The various options of the function allow you to get the status information
// of the child process that has been terminated or stopped. If there are status information
// for two or more child processes, the order of the reports is not specified.
// wait() will suspend the current process until one of its children exits (terminates), or
// receives a signal requesting termination of the process, or needs to call a signal handler.
// Waitpid() suspends the current process until the child specified by pid exits or receives
// a signal requesting termination of the process, or a signal handler needs to be called.
// If pid= -1, options=0, then waitpid() acts the same as the wait() function, otherwise its
// behavior will vary with the pid and options parameters (see kernel/exit.c, 142).
// The parameter 'pid' is the process id; '*stat_loc' is a pointer to the location of the status
// information; 'options' is the wait option, see lines 10, 11 above.
21 pid\_t wait(int *stat_loc);
22 pid\_t waitpid(pid\_t pid, int *stat_loc, int options);
23
24 #endif
25
```

14.41 Summary

This chapter describes all the header files used by the kernel. From the next chapter we will introduce the library file code used by the kernel. The code for these library files will be linked into the kernel code when the kernel is compiled.














15 Library files (lib)

The C language library is a collection of reusable program modules, while the Linux kernel library files are a combination of some commonly used functions that are compiled for use by the kernel. The C files in Listing 15-1 are the programs that makes up the modules in the kernel library file. It mainly includes process execution and exit functions, file access operation functions, memory allocation functions, and string manipulation functions.

Specifically, the functions implemented are: exit function `_exit()`, close file function `close()`, copy file descriptor function `dup()`, file open function `open()`, write file function `write()`, execute program function `execve()`, the memory allocation function `malloc()`, wait for the child process state function `wait()`, create the session system call `setsid()`, and all string manipulation functions implemented in `include/string.h`.

Except for a `malloc.c` program written by Mr. Tytso, the size of the program is very short, and some are only one or two lines of code. They basically invoke the system-calls directly to implement their functions.

List 15-1 Files in the `/linux/lib/` directory

	Filename	Size	Last Modified Time(GMT)	Description
	Makefile	2602 bytes	1991-12-02 03:16:05	
	_exit.c	198 bytes	1991-10-02 14:16:29	
	close.c	131 bytes	1991-10-02 14:16:29	
	ctype.c	1202 bytes	1991-10-02 14:16:29	
	dup.c	127 bytes	1991-10-02 14:16:29	
	errno.c	73 bytes	1991-10-02 14:16:29	
	execve.c	170 bytes	1991-10-02 14:16:29	
	malloc.c	7469 bytes	1991-12-02 03:15:20	
	open.c	389 bytes	1991-10-02 14:16:29	
	setsid.c	128 bytes	1991-10-02 14:16:29	
	string.c	177 bytes	1991-10-02 14:16:29	
	wait.c	253 bytes	1991-10-02 14:16:29	
	write.c	160 bytes	1991-10-02 14:16:29	

In the kernel compilation phase, the relevant instructions in the kernel `Makefile` will compile these programs into `.o` modules, and then build them into a `lib.a` library and linked to the kernel module. Different from the various library files provided by the usual compilation environment (such as `libc.a`, `libufc.a` provided by `gcc`, etc.), the functions in this library are mainly used in the `init/main.c` program of the kernel initialization stage, for its execution of the `init()` function in the user mode. So the included functions are few and very simple. But it is implemented in exactly the same way as a general library.

Creating a function library usually uses the command `ar` (archive abbreviation). For example, to create a function library `libmine.a` with 3 modules `a.o`, `b.o`, and `c.o`, you need to execute the following command:

```
ar -rc libmine.a a.o b.o c.o d.o
```

To add function module dup.o to this library file, execute the following command:

```
ar -rs dup.o
```

15.1 _exit.c

15.1.1 Functionality

The _exit.c file is used by the program to call the kernel's exit system-call function.

15.1.2 Code annotation

Program 15-1 linux/lib/_exit.c

```
1 /*
2  * linux/lib/_exit.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
8 // various functions are declared. If '__LIBRARY__' is defined, it also includes the
9 // system-call number and the inline assembly _syscall0().
10 #define __LIBRARY__
11 #include <unistd.h>
12
13 // The program exits (terminates) function.
14 // The library function directly calls the system interrupt int 0x80, function number __NR_exit.
15 // Parameters: exit_code - exit code.
16 // The keyword 'volatile' before the function name is used to tell the compiler gcc that the
17 // function will not return. This will allow gcc to produce better code and, more importantly,
18 // use this keyword to avoid some false warnings.
19 volatile void _exit(int exit_code)
20 {
21     // %0 - eax(__NR_exit); %1 - ebx(exit_code).
22     __asm__ ("int $0x80"::"a" (__NR_exit), "b" (exit_code));
23 }
24
```

15.1.3 Information

For a description of the system-call interrupt number, see the description in the include/unistd.h file.

15.2 close.c

15.2.1 Functionality

The file close function close() is defined in the close.c file.

15.2.2 Code annotation

Program 15-2 linux/lib/close.c

```
1 /*
2  * linux/lib/close.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
8 // various functions are declared. If '__LIBRARY__' is defined, it also includes the
9 // system-call number and the inline assembly _syscall0().
10 #define LIBRARY
11 #include <unistd.h>
12
13 // Close the file function.
14 // The following macro corresponds to function prototype: int close(int fd). It directly calls
15 // the system int 0x80, with the parameter __NR_close. Where fd is the file descriptor.
16 syscall1(int, close, int, fd)
17
```

15.3 ctype.c

15.3.1 Functionality

The ctype.c program is used to provide auxiliary array structure data for ctype.h for type determination of characters.

15.3.2 Code annotation

Program 15-3 linux/lib/ctype.c

```
1 /*
2  * linux/lib/ctype.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <ctype.h> The character type file. Defines some macros for character type conversion.
8 #include <ctype.h>
9
10 char ctmp; // a tem variable for macros that convert characters in the ctype.h file.
```

```

// The following is an array of character attributes that define the attributes corresponding
// to each character. These attribute types (such as _C, etc.) are defined in ctype.h. It is
// used to check whether the character is a control character (_C), uppercase character (_U),
// lowercase character (_L), etc.
10 unsigned char _ctype[] = {0x00,                                /* EOF */
11  _C, _C, _C, _C, _C, _C, _C, _C,                                /* 0-7 */
12  _C, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S, _C|_S,        /* 8-15 */
13  _C, _C, _C, _C, _C, _C, _C, _C,                                /* 16-23 */
14  _C, _C, _C, _C, _C, _C, _C, _C,                                /* 24-31 */
15  _S|_SP, _P, _P, _P, _P, _P, _P, _P,                            /* 32-39 */
16  _P, _P, _P, _P, _P, _P, _P, _P,                                /* 40-47 */
17  _D, _D, _D, _D, _D, _D, _D, _D,                                /* 48-55 */
18  _D, _D, _P, _P, _P, _P, _P, _P,                                /* 56-63 */
19  _P, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U|_X, _U,            /* 64-71 */
20  _U, _U, _U, _U, _U, _U, _U, _U,                                /* 72-79 */
21  _U, _U, _U, _U, _U, _U, _U, _U,                                /* 80-87 */
22  _U, _U, _U, _P, _P, _P, _P, _P,                                /* 88-95 */
23  _P, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L|_X, _L,            /* 96-103 */
24  _L, _L, _L, _L, _L, _L, _L, _L,                                /* 104-111 */
25  _L, _L, _L, _L, _L, _L, _L, _L,                                /* 112-119 */
26  _L, _L, _L, _P, _P, _P, _P, _C,                                /* 120-127 */
27  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 128-143 */
28  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 144-159 */
29  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 160-175 */
30  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 176-191 */
31  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 192-207 */
32  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 208-223 */
33  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,              /* 224-239 */
34  0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};             /* 240-255 */
35
36

```

15.4 dup.c

15.4.1 Functionality

The dup.c program includes a function dup() that creates a copy of the file descriptor. After a successful return, the new and original descriptors can be used interchangeably. They share locks, file read and write pointers, and file flags. For example, if the file read/write position pointer is modified by one of the descriptors using lseek(), the file read/write pointer is also changed for the other descriptor. This function uses the smallest unused descriptor to create a new descriptor, but the two descriptors do not share the close-on-exec flag.

15.4.2 Code annotation

Program 15-4 linux/lib/dup.c

```

1 /*
2  * linux/lib/dup.c

```

```
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
// <unistd.h> Linux standard header file. Various symbol constants and types are defined and
// various functions are declared. If '__LIBRARY__' is defined, it also includes the
// system-call number and the inline assembly _syscall0().
7 #define LIBRARY
8 #include <unistd.h>
9
//// Duplicate file descriptor (handle) function.
// The macro below corresponds to the function prototype: int dup(int fd). It directly calls
// the system int 0x80, the parameter is __NR_dup. Where fd is the file descriptor.
10 syscall1(int, dup, int, fd)
11
```

15.5 errno.c

15.5.1 Functionality

The program only defines an variable `errno` to store the error number when the function call fails. Please refer to the description in the include/errno.h file.

15.5.2 Code annotation

Program 15-5 linux/lib/errno.c

```
1 /*
2  * linux/lib/errno.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 int errno;
8
```

15.6 execve.c

15.6.1 Functionality

The `execve.c` program contains a system-call function that runs the executables.

15.6.2 Code annotation

Program 15-6 linux/lib/execve.c

```
1 /*
```

```
2  * linux/lib/execve.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7  // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
8  // various functions are declared. If '__LIBRARY__' is defined, it also includes the
9  // system-call number and the inline assembly _syscall0().
10 #define __LIBRARY__
11 #include <unistd.h>
12
13 // Load and execute the child process (other programs) function.
14 // The macro corresponds to function: int execve(const char * file, char ** argv, char ** envp).
15 // Parameters: file - the executable file name; argv - an array of command line argument pointers;
16 // envp - an array of environment variable pointers. It directly calls the system int 0x80,
17 // the parameter is __NR_execve. See include/unistd.h and fs/exec.c.
18 _syscall3(int, execve, const char *, file, char **, argv, char **, envp)
```

15.7 malloc.c

15.7.1 Functionality

The malloc.c program mainly includes the memory allocation function malloc(). In order not to be confused with the malloc() function used by the user program, it is called kmalloc() from the kernel version 0.98, and the free_s() function is renamed to kfree_s().

Note that the memory allocation functions of the same name used by the application are generally implemented in the library file of the development environment, such as the libc.a library in the GCC environment. Since the library functions in the development environment are themselves linked to the user program, they cannot directly use the functions such as get_free_page() in the kernel to implement the memory allocation function. Of course, they also do not need to directly manage the memory page, because the memory allocation function in the library libc.a only needs to dynamically adjust the set value at the end of the process data segment according to the program request, and does not cover the end stack and the environment parameter area. The rest of the specific memory mapping and other operations are done by the kernel. This operation of adjusting the end position of the process data segment is the main purpose of the memory allocation function in the library, and the kernel system-call brk() is called, see line 228 of the kernel/sys.c program. So if you can view the source code of the library function implementation in the development environment, you will find that the memory allocation functions such as malloc() and calloc() only call the kernel system-call brk() in addition to managing the dynamic application memory area. The memory allocation functions in the development environment library are identical to the functions here only in the aspects that they all require dynamic management of allocated memory. The management methods they use are basically the same.

The malloc() function uses the bucket principle to manage the allocated memory. The basic idea is to use the bucket directory (hereinafter referred to as the directory) for the different memory block sizes requested. For example, if the size of the request memory block is 32 bytes or less but more than 16 bytes, the memory block is allocated using the bucket descriptor list corresponding to the second item in the bucket directory. The basic

structure is shown in Figure 15-1. The maximum memory size that the function can allocate at a time is one memory page, which is 4096 bytes.

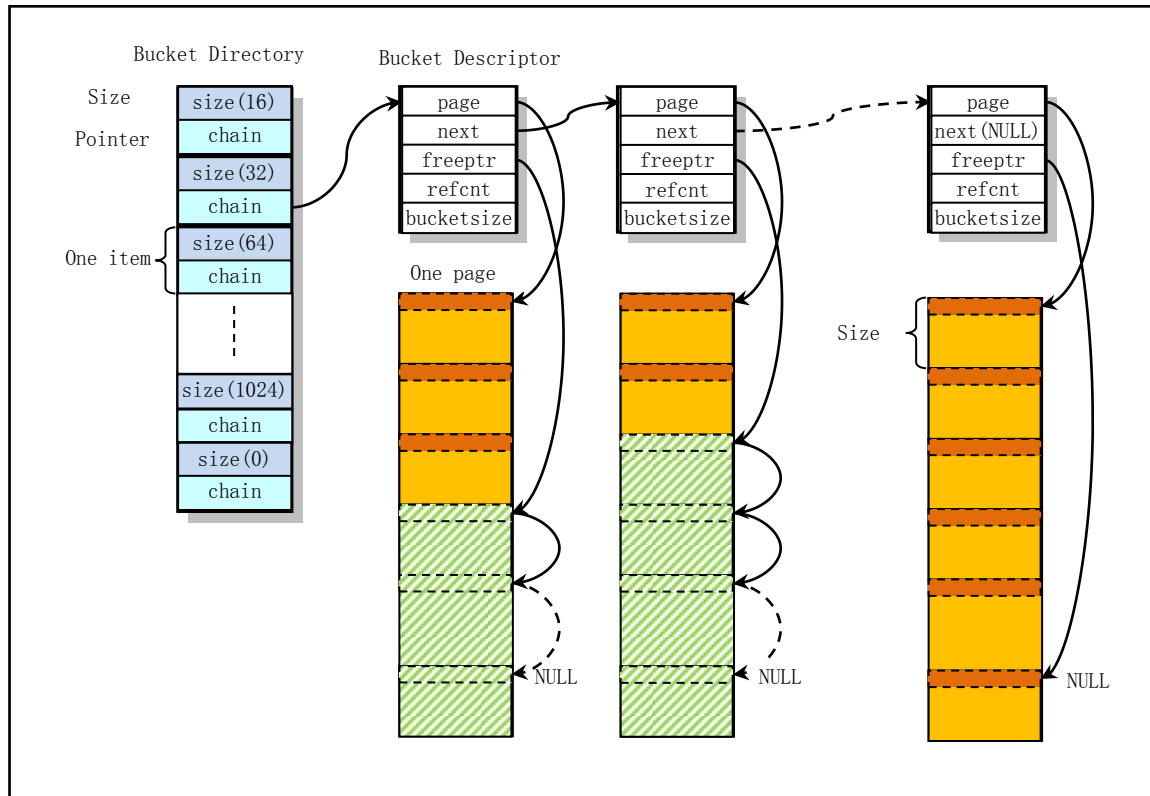


Figure 15-1 Use the bucket principle for memory allocation management

The first time you call the `malloc()` function, you first need to create a free bucket descriptor list for the page, which holds the descriptors that have not been used or have been reclaimed. The structure of the linked list is shown in Figure 15-2, where `free_bucket_desc` is the pointer to the linked list header. Extracting/putting a descriptor from/into the linked list starts from the beginning of the linked list. When a descriptor is fetched, the first descriptor pointed to by the header pointer is fetched; when an idle descriptor is released, it is also placed at the head of the list.

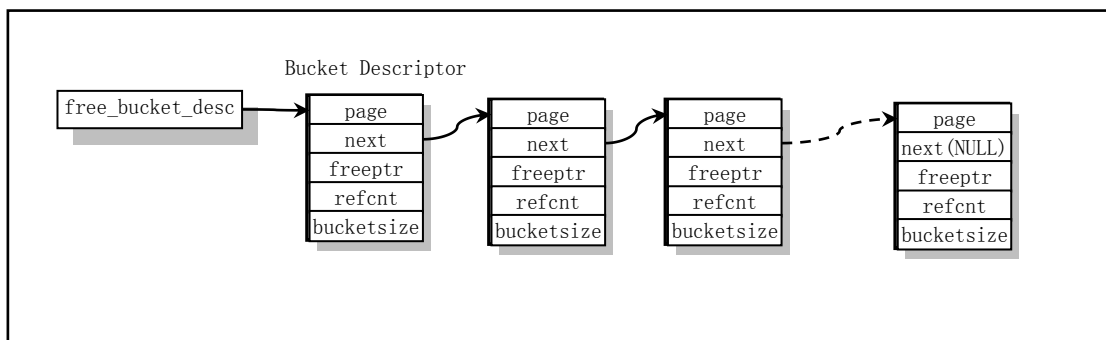


Figure 15-2 Free bucket descriptor chain table structure

During the running of the system, if all bucket descriptors are occupied at a certain time, then `free_bucket_desc` will be NULL. Therefore, without the bucket descriptor being released, the next time you need to use the free bucket descriptor, the program will apply for a page again and create a new free bucket

descriptor list on it as shown in the figure.

The basic steps of the malloc() function are as follows:

1. First search the directory and look for the descriptor list corresponding to the directory entry that matches the size of the requested memory block. When the memory size of the directory entry is larger than the requested byte size, the corresponding directory entry is found. If the search for the entire directory does not find a suitable directory entry, then the memory block requested by the user is too large.
2. Find a descriptor with free space in the descriptor list corresponding to the directory entry. If the free memory pointer freeptr of a descriptor is not NULL, it means that the corresponding descriptor is found. If we don't find a descriptor with free space, then we need to create a new descriptor. The steps to create a new descriptor are as follows:
 - a. If the idle descriptor list header pointer is still NULL, it means that the malloc() function is called for the first time, or all empty bucket descriptors are used up. In this case, you need to use the function init_bucket_desc() to create a list of idle descriptors.
 - b. Then get a descriptor from the header of the idle descriptor chain, initialize the descriptor, make its object reference count 0, the object size is equal that of the directory entry, and apply for a memory page, let the descriptor page pointer points to the memory page, and the free memory pointer of the descriptor also points to the beginning of the page.
 - c. Initialize the page for the memory page according to the size of the object used in this directory entry, and establish a linked list of all objects. That is, each object's head stores a pointer to the next object, and the last object stores a NULL at the beginning.
 - d. Then insert the descriptor into the beginning of the descriptor list for the corresponding directory entry.
3. Copy the free memory pointer freeptr of the descriptor to the memory pointer returned to the user, and then adjust the freeptr to point to the next free object location in the memory page corresponding to the descriptor, and increment the descriptor reference count by one.

The free_s() function is used to reclaim the memory blocks released by the user. The basic method is to first convert the address of the corresponding page according to the address of the memory block, and then search all the descriptors in the directory to find the descriptor corresponding to the page. The released memory block is chained into the free object list pointed to by the freeptr, and the descriptor's object reference count value is decremented by one. If the reference count value is equal to zero at this time, it means that the page corresponding to the descriptor is completely free, and the memory page can be released and the descriptor is retracted into the idle descriptor list.

15.7.2 Code annotation

Program 15-7 linux/lib/malloc.c

```
1 /*  
2  * malloc.c --- a general purpose kernel memory allocator for Linux.  
3  *  
4  * Written by Theodore Ts'o (tytso@mit.edu), 11/29/91  
5  *  
6  * This routine is written to be as fast as possible, so that it
```

```

7  * can be called from the interrupt level.
8  *
9  * Limitations: maximum size of memory we can allocate using this routine
10 *    is 4k, the size of a page in Linux.
11 *
12 * The general game plan is that each page (called a bucket) will only hold
13 * objects of a given size. When all of the object on a page are released,
14 * the page can be returned to the general free pool. When malloc() is
15 * called, it looks for the smallest bucket size which will fulfill its
16 * request, and allocate a piece of memory from that bucket pool.
17 *
18 * Each bucket has as its control block a bucket descriptor which keeps
19 * track of how many objects are in use on that page, and the free list
20 * for that page. Like the buckets themselves, bucket descriptors are
21 * stored on pages requested from get_free_page(). However, unlike buckets,
22 * pages devoted to bucket descriptor pages are never released back to the
23 * system. Fortunately, a system should probably only need 1 or 2 bucket
24 * descriptor pages, since a page can hold 256 bucket descriptors (which
25 * corresponds to 1 megabyte worth of bucket pages.) If the kernel is using
26 * that much allocated memory, it's probably doing something wrong. :-)
27 *
28 * Note: malloc() and free() both call get_free_page() and free_page()
29 *    in sections of code where interrupts are turned off, to allow
30 *    malloc() and free() to be safely called from an interrupt routine.
31 *    (We will probably need this functionality when networking code,
32 *    particularly things like NFS, is added to Linux.) However, this
33 *    presumes that get_free_page() and free_page() are interrupt-level
34 *    safe, which they may not be once paging is added. If this is the
35 *    case, we will need to modify malloc() to keep a few unused pages
36 *    "pre-allocated" so that it can safely draw upon those pages if
37 *    it is called from an interrupt routine.
38 *
39 *    Another concern is that get_free_page() should not sleep; if it
40 *    does, the code is carefully ordered so as to avoid any race
41 *    conditions. The catch is that if malloc() is called re-entrantly,
42 *    there is a chance that unnecessary pages will be grabbed from the
43 *    system. Except for the pages for the bucket descriptor page, the
44 *    extra pages will eventually get released back to the system, though,
45 *    so it isn't all that bad.
46 */
47
48 // <linux/kernel.h> Kernel header file. Contains prototype definitions of some of the commonly
49 //    used functions of the kernel.
50 // <linux/mm.h> Memory management header file. Contains page size definitions and some page
51 //    release function prototypes.
52 // <asm/system.h> System header file. An embedded assembly macro that defines or modifies
53 //    descriptors/interrupt gates, etc. is defined.
54 #include <linux/kernel.h>
55 #include <linux/mm.h>
56 #include <asm/system.h>
57
58 // Bucket descriptor structure.
59 struct bucket_desc {    /* 16 bytes */

```

```

53     void                *page;                // Memory page pointer.
54     struct bucket\_desc *next;                // The next descriptor pointer.
55     void                *freeptr;             // A pointer to the free memory.
56     unsigned short      refcnt;               // Refernce count.
57     unsigned short      bucket_size;          // The size of the bucket.
58 };
59
60 // Bucket descriptor directory structure.
61 struct bucket\_dir { /* 8 bytes */
62     int                size;                // The size (in bytes) of this bucket.
63     struct bucket\_desc *chain;              // Bucket descriptor list pointer.
64 };
65 /*
66  * The following is the where we store a pointer to the first bucket
67  * descriptor for a given size.
68  *
69  * If it turns out that the Linux kernel allocates a lot of objects of a
70  * specific size, then we may want to add that specific size to this list,
71  * since that will allow the memory to be allocated more efficiently.
72  * However, since an entire page must be dedicated to each specific size
73  * on this list, some amount of temperance must be exercised here.
74  *
75  * Note that this list must be kept in order.
76  */
77 // Bucket directory list.
78 struct bucket\_dir bucket\_dir[] = {
79     { 16, (struct bucket\_desc *) 0}, // A 16-byte memory block.
80     { 32, (struct bucket\_desc *) 0}, // A 32-byte memory block.
81     { 64, (struct bucket\_desc *) 0},
82     { 128, (struct bucket\_desc *) 0},
83     { 256, (struct bucket\_desc *) 0},
84     { 512, (struct bucket\_desc *) 0},
85     { 1024, (struct bucket\_desc *) 0},
86     { 2048, (struct bucket\_desc *) 0},
87     { 4096, (struct bucket\_desc *) 0}, // A 4096-byte memory block.
88     { 0, (struct bucket\_desc *) 0}}; /* End of list marker */
89 /*
90  * This contains a linked list of free bucket descriptor blocks
91  */
92 struct bucket\_desc *free\_bucket\_desc = (struct bucket\_desc *) 0;
93
94 /*
95  * This routine initializes a bucket description page.
96  */
97 // Initialize the bucket descriptor.
98 // Create a free bucket descriptor list and let 'free_bucket_desc' point to the first free bucket
99 // descriptor.
100 static inline void init\_bucket\_desc()
101 {
102     struct bucket\_desc *bdesc, *first;
103     int i;

```



```

101 // First apply for a page of memory for storing the bucket descriptor. Then calculate the number
102 // of bucket descriptors that can be stored in a page of memory, and then establish a one-way
103 // link pointer to it.
104 first = bdesc = (struct bucket\_desc *) get\_free\_page();
105 if (!bdesc)
106     panic("Out of memory in init_bucket_desc()");
107 for (i = PAGE\_SIZE/sizeof(struct bucket\_desc); i > 1; i--) {
108     bdesc->next = bdesc+1;
109     bdesc++;
110 }
111 /*
112  * This is done last, to avoid race conditions in case
113  * get_free_page() sleeps and this routine gets called again...
114  */
115 // Add the free bucket descriptor pointer 'first' to the beginning of the list.
116 bdesc->next = free\_bucket\_desc;
117 free\_bucket\_desc = first;
118 }
119
120 /// Memory allocation function.
121 // Parameters: len - the size of the requested memory block.
122 // Returns: a pointer to the allocated memory. Returns NULL if it fails.
123 void *malloc(unsigned int len)
124 {
125     struct bucket\_dir *bdir;
126     struct bucket\_desc *bdesc;
127     void *retval;
128
129     /*
130      * First we search the bucket_dir to find the right bucket change
131      * for this request.
132      */
133     // Search the bucket directory for a bucket descriptor list that is suitable for applying the
134     // size of the memory block. If the bucket size of the directory entry is greater than the
135     // requested number of bytes, the corresponding bucket directory entry is found.
136     for (bdir = bucket\_dir; bdir->size; bdir++)
137         if (bdir->size >= len)
138             break;
139     // If the search for the entire directory does not find a directory entry of the appropriate
140     // size, it indicates that the requested memory block size is too large, exceeding the allocation
141     // limit of the program (up to 1 page). Then the error message is displayed and the crash occurs.
142     if (!bdir->size) {
143         printk("malloc called with impossibly large argument (%d)\n",
144             len);
145         panic("malloc: bad arg");
146     }
147     /*
148      * Now we search for a bucket descriptor which has free space
149      */
150     cli(); /* Avoid race conditions */
151     // Search the descriptor list in the corresponding bucket directory entry to find the bucket
152     // descriptor with free space. If the free memory pointer 'freeptr' of the bucket descriptor

```

```

// is not empty, it indicates that the corresponding bucket descriptor was found.
139     for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next)
140         if (bdesc->freeptr)
141             break;
142     /*
143      * If we didn't find a bucket with free space, then we'll
144      * allocate a new one.
145      */
146     if (!bdesc) {
147         char      *cp;
148         int       i;
149
150         // If free_bucket_desc is still empty, it means that the function is called for the first time,
151         // or all empty bucket descriptors in the linked list are used up. At this point you need to
152         // apply for a page and build and initialize an idle descriptor list on it. free_bucket_desc
153         // will point to the first free bucket descriptor.
154         if (!free_bucket_desc)
155             init_bucket_desc();
156
157         // Take the free bucket descriptor pointed to by free_bucket_desc and let free_bucket_desc point
158         // to the next free bucket descriptor. Then initialize the new bucket descriptor: make the
159         // references count equal to 0; the bucket size is equal to the size of the corresponding bucket
160         // directory; apply a memory page, let the page pointer point to the page; the free memory pointer
161         // also points to the beginning of the page, because it is all idle at this time.
162         bdesc = free_bucket_desc;
163         free_bucket_desc = bdesc->next;
164         bdesc->refcnt = 0;
165         bdesc->bucket_size = bdir->size;
166         bdesc->page = bdesc->freeptr = (void *) cp = get_free_page();
167         // If the application for memory page operation fails, an error occurs and the machine crashes.
168         // Otherwise, the page size is divided by the bucket size specified by the bucket directory
169         // entry, and the first 4 bytes of each object are set to pointers to the next object. The pointer
170         // of the last object is set to 0 (NULL).
171         if (!cp)
172             panic("Out of memory in kernel malloc()");
173         /* Set up the chain of free objects */
174         for (i=PAGE_SIZE/bdir->size; i > 1; i--) {
175             *((char **) cp) = cp + bdir->size;
176             cp += bdir->size;
177         }
178         *((char **) cp) = 0;
179         // Then, the next descriptor pointer field of the bucket descriptor is pointed to the descriptor
180         // pointed to by the bucket directory entry pointer originally, and the bucket directory's chain
181         // points to the bucket descriptor, that is, the descriptor is inserted into the descriptor
182         // chain header.
183         bdesc->next = bdir->chain;    /* OK, link it in! */
184         bdir->chain = bdesc;
185     }
186     // The return pointer is equal to the current free pointer of the page. The free space pointer
187     // is then adjusted to point to the next free object, and the object reference count in the
188     // corresponding page in the descriptor is incremented by one. Finally enable the interrupt
189     // and return the pointer to the free memory object.
190     retval = (void *) bdesc->freeptr;

```

```

169     bdesc->freeptr = *((void **) retval);
170     bdesc->refcnt++;
171     sti();      /* OK, we're safe again */
172     return(retval);
173 }
174
175 /*
176  * Here is the free routine. If you know the size of the object that you
177  * are freeing, then free_s() will use that information to speed up the
178  * search for the bucket descriptor.
179  *
180  * We will #define a macro so that "free(x)" is becomes "free_s(x, 0)"
181  */
182 void free_s(void *obj, int size)
183 {
184     void *page;
185     struct bucket_dir *bdir;
186     struct bucket_desc *bdesc, *prev;
187
188     /* Calculate what page this object lives in */
189     page = (void *) ((unsigned long) obj & 0xfffff000);
190     /* Now search the buckets looking for that page */
191     for (bdir = bucket_dir; bdir->size; bdir++) {
192         prev = 0;
193         /* If size is zero then this conditional is always false */
194         if (bdir->size < size)
195             continue;
196         // Search all the descriptors in the directory entry to find the corresponding page. If a
197         // descriptor page pointer is equal to 'page', it means that the corresponding descriptor is
198         // found, so it jumps to label 'found'. If the descriptor does not contain a corresponding page,
199         // then the descriptor pointer 'prev' is pointed to the descriptor.
200         // If all the descriptors searching for the corresponding directory entry do not find the
201         // specified page, an error message is displayed and the computer crashes.
202         for (bdesc = bdir->chain; bdesc; bdesc = bdesc->next) {
203             if (bdesc->page == page)
204                 goto found;
205             prev = bdesc;
206         }
207         panic("Bad address passed to kernel free_s()");
208     found:
209     // After finding the corresponding bucket descriptor, first turn off the interrupt. The object
210     // memory block is then chained into the free block object list and the object reference count
211     // for that descriptor is decremented by one.
212     cli(); /* To avoid race conditions */
213     *((void **)obj) = bdesc->freeptr;
214     bdesc->freeptr = obj;
215     bdesc->refcnt--;
216
217     // If the reference count is equal to 0, then we can release the corresponding memory page and
218     // the bucket descriptor.

```

```
208     if (bdesc->refcnt == 0) {
209         /*
210          * We need to make sure that prev is still accurate. It
211          * may not be, if someone rudely interrupted us...
212          */
213         // If 'prev' is not the previous descriptor of the searched descriptor, then the previous
214         // descriptor of the current descriptor is re-searched.
215         if ((prev && (prev->next != bdesc)) ||
216             (!prev && (bdir->chain != bdesc)))
217             for (prev = bdir->chain; prev; prev = prev->next)
218                 if (prev->next == bdesc)
219                     break;
220         // If the previous descriptor is found, the current descriptor is removed from the descriptor
221         // chain. If prev==NULL, it means that the current descriptor is the first descriptor of the
222         // directory entry, that is, the chain in the directory entry should directly point to the current
223         // descriptor 'bdesc', otherwise it indicates that there is a problem with the linked list.
224         // Therefore, in order to remove the current descriptor from the linked list, you should have
225         // 'chain' pointing to the next descriptor.
226         if (prev)
227             prev->next = bdesc->next;
228         else {
229             if (bdir->chain != bdesc)
230                 panic("malloc bucket chains corrupted");
231             bdir->chain = bdesc->next;
232         }
233         // Finally, the memory page operated by the current descriptor is released, and the descriptor
234         // is inserted at the beginning of the idle descriptor list.
235         free\_page((unsigned long) bdesc->page);
236         bdesc->next = free\_bucket\_desc;
237         free\_bucket\_desc = bdesc;
238     }
239     sti();          // Enable interrupt and return.
240     return;
241 }
242
243
```

15.8 open.c

15.8.1 Functionality

The `open()` library function in the `open.c` file is used to open a file with the specified file name. When `open()` is called successfully, the file descriptor of the file is returned. This call creates a new open file and is not shared with any other process. When the `exec()` function is executed, the new file descriptor will remain open at all times. The file's read and write pointer is set at the beginning of the file.

The function parameter 'flag' can be one of `O_RDONLY`, `O_WRONLY`, and `O_RDWR`, which means that the file is read-only open, write-only open, and read-write open, respectively, and can be used with some other flags. See also the implementation of the `sys_open()` function in the `fs/open.c` program (line 171).

15.8.2 Code annotation

Program 15-8 linux/lib/open.c

```

1  /*
2   *  linux/lib/open.c
3   *
4   *  (C) 1991  Linus Torvalds
5   */
6
7  // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
8  //      various functions are declared. If '__LIBRARY__' is defined, it also includes the
9  //      system-call number and the inline assembly _syscall0().
10 // <stdarg.h> Standard parameter header file. Define a list of variable parameters in the form
11 //      of macros. It mainly describes one type (va_list) and three macros (va_start, va_arg and
12 //      va_end) for the vsprintf, vprintf, and vfprintf functions.
13 #define  __LIBRARY__
14 #include <unistd.h>
15 #include <stdarg.h>
16
17 // Open file library function.
18 // Open a file or create a file when it does not exist.
19 // Parameters: filename - file name; flag - file open flag; ...
20 // Returns the file descriptor. If an error occurs, the error code is set and -1 is returned.
21 int open(const char * filename, int flag, ...)
22 {
23     register int res;
24     va_list arg;
25
26     // Use the va_start() macro function to get the pointer of the parameter after the flag. Then
27     // call the system interrupt int 0x80 with the function number __NR_open to open the file.
28     // %0 - eax (descriptor returned or error code); %1 - eax (system-call function __NR_open);
29     // %2 - ebx (filename); %3 - ecx (open file flag); % 4 - edx (following the file mode).
30     va_start(arg, flag);
31     __asm__( "int $0x80"
32             : "=a" (res)
33             : "" ( __NR_open), "b" (filename), "c" (flag),
34             "d" (va_arg(arg, int)));
35     // If the system interrupt call returns a value greater than or equal to 0, indicating that
36     // it is a file descriptor, it will return directly. Otherwise, the return value is less than
37     // 0, then it is an error code. So set the error code and return -1.
38     if (res >= 0)
39         return res;
40     errno = -res;
41     return -1;
42 }
43

```

15.9 setsid.c

15.9.1 Functionality

The setsid.c program includes a setsid() system call function. This function is used to create a new session if the calling process is not a leader of a group. The calling process will become the leader of the new session, the group leader of the new process group, and there is no controlling terminal. The group ID and session ID of the calling process are set to the PID of the process. The calling process will become the only process in the new process group and the new session.

15.9.2 Code annotation

Program 15-9 linux/lib/setsid.c

```
1 /*
2  * linux/lib/setsid.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
8 // various functions are declared. If '__LIBRARY__' is defined, it also includes the
9 // system-call number and the inline assembly _syscall0().
10 #define __LIBRARY__
11 #include <unistd.h>
```

15.10 string.c

15.10.1 Functionality

All string manipulation functions already exist in the string.h header file, but appear as inline code. Here, we give the implementation code that contains the string function in string.c by first declaring the 'extern' and 'inline' prefixes to be empty, and then including the string.h header file. See the instructions before the include/string.h header file.

15.10.2 Code annotation

Program 15-10 linux/lib/string.c

```
1 /*
2  * linux/lib/string.c
```

```
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 #ifndef __GNUC__
8 #error I want gcc!
9 #endif
10
11 #define extern
12 #define inline
13 #define LIBRARY
14 #include <string.h>
15
```

15.11 wait.c

15.11.1 Functionality

The wait.c program includes the functions waitpid() and wait(). These two functions allow the process to get state information for one of its child processes. Various options allow you to get child process status information that has been terminated or stopped. If there are status information for two or more child processes, the order of the reports is not specified.

wait() will suspend the current process until one of its child processes exits (terminates), or receives a signal requesting termination of the process, or a signal handler needs to be called.

waitpid() suspends the current process until the child process specified by pid exits (terminates) or receives a signal requesting termination of the process, or a signal handler needs to be called.

If pid= -1, options=0, waitpid() acts the same as the wait() function, otherwise its behavior will vary depending on the pid and options parameters (see kernel/exit.c, 370).

15.11.2 Code annotation

Program 15-11 linux/lib/wait.c

```
1 /*
2  * linux/lib/wait.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
8 // various functions are declared. If '__LIBRARY__' is defined, it also includes the
9 // system-call number and the inline assembly _syscall0().
10 // <sys/wait.h> wait header file. Define the system call wait() core waitpid() and related
11 // constant symbols.
12 #define LIBRARY
13 #include <unistd.h>
14 #include <sys/wait.h>
```

```
10  // Wait for the process to terminate.
    // The macro corresponds to the function: pid_t waitpid(pid_t pid, int *wait_stat, int options)
    // Parameters: pid - the process id of the process waiting to be terminated, or other specific
    // value used to specify a special case; wait_stat - used to store state information;
    // options - WNOHANG or WUNTRACED or 0.
11  _syscall3(pid_t, waitpid, pid_t, pid, int *, wait_stat, int, options)
12
    // wait() system-call. It directly calls the waitpid() function.
13  pid_t wait(int * wait_stat)
14  {
15      return waitpid(-1, wait_stat, 0);
16  }
17
```

15.12 write.c

15.12.1 Functionality

The write.c program includes a write function write() to the file descriptor. This function writes the data of the 'count' byte to the file 'buf' of the file specified by the file descriptor.

15.12.2 Code annotation

Program 15-12 linux/lib/write.c

```
1  /*
2   * linux/lib/write.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
    // <unistd.h> Linux standard header file. Various symbol constants and types are defined and
    // various functions are declared. If '__LIBRARY__' is defined, it also includes the
    // system-call number and the inline assembly _syscall0().
7  #define __LIBRARY__
8  #include <unistd.h>
9
    // Write the file.
    // The macro corresponds to the function: int write(int fd, const char * buf, off_t count)
    // Parameters: fd - file descriptor; buf - write buffer pointer; count - number of write bytes.
    // Returns: the number of bytes written back on success (0 means write 0 bytes); -1 will be
    // returned on error and the error code is set.
10 _syscall3(int, write, int, fd, const char *, buf, off_t, count)
11
```

15.13 Summary

This chapter describes several library function files used by several tasks that the kernel runs in user mode at initialization time. These library functions are implemented in exactly the same way as the generic library functions used in the development environment.

In the next chapter, we explain a tool, `tools/build.c`, contained in the kernel code tree that is used to combine all kernel modules to generate a kernel image file.

16 Building Kernel (tools)

The 'tools' directory in the Linux kernel source code contains a utility program build.c that generates kernel disk image files. This program is compiled separately into an executable file and will be called in the Makefile to be used to connect and merge all kernel compiled modules into a working image file Image. According to the contents of the Makefile, the Make program will first compile boot/bootsect.s and boot/setup.s using the 8086 assembler to generate two object module files 'bootsect' and 'setup' in MINIX format; then compile with GNU. Gcc/gas, compiles and links all other programs in the source code to generate the object module 'system' in a.out format. Finally, use the build tool to remove the extra header data from the three modules and stitch them together into a kernel image file 'Image'. The basic compilation linking/combination structure is shown in Figure 16-1.

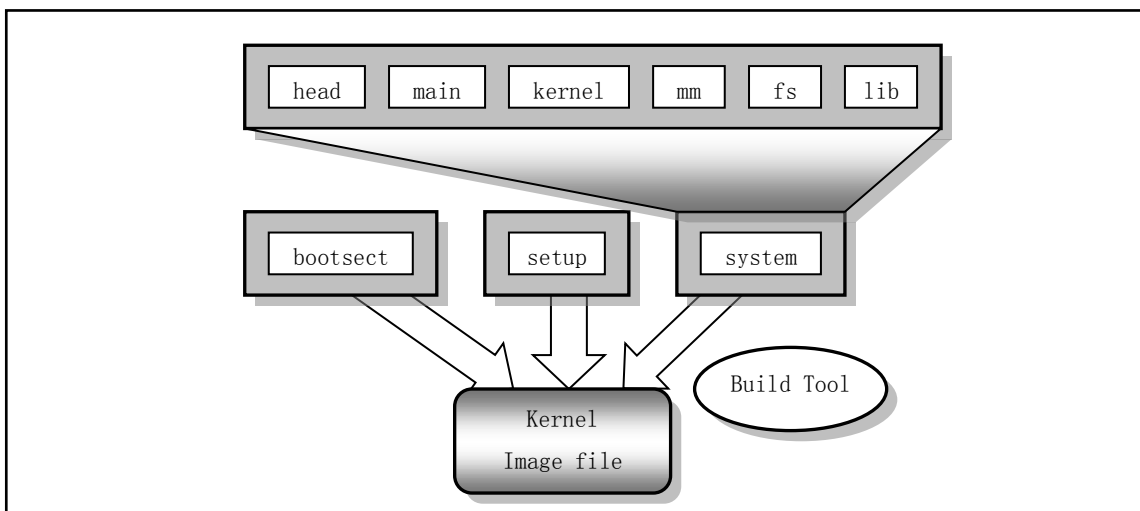


Figure 16-1 Kernel compilation linking/combination structure

16.1 build.c

16.1.1 Functionality

On the linux/Makefile line 42--44, the command line form for executing the build program is as follows:

```
tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) $(SWAP_DEV) > Image
```

The build program uses five parameters, namely the bootsect, setup, system, the optional root fs device name ROOT_DEV, and the optional swap device SWAP_DEV. The bootsect and setup modules are compiled by as86, they have the MINIX executable file format (see the description of the program list), and the system module is linked by modules compiled from other source code, with GNU a.out executable file format. The main job of

the build program is to remove the MINIX execution file header information of bootsect and setup, remove the a.out header information in the system module, retain only their code and data parts, and then combine them in order, and write them sequentially into a file named 'Image'.

The program first checks the last optional parameter on the command line: root device file name. If it exists, the status information structure (stat) of the device file is read, and the device number is extracted. If this parameter is not present on the command line, the default value is used. Then the bootsect file is processed, the minix execution header information of the file is read, the validity is checked, and then the subsequent 512-byte boot code is read to determine whether it has the bootable flag 0xAA55, and then the root device number acquired earlier is written to the 508, 509 offset, and finally the 512-byte code data is written to the stdout standard output, which is redirected to the Image file by the Make file. The next step is to process the setup file in a similar way. If the file is less than 4 sectors in length, it is filled with 0 to the length of 4 sectors and written to the standard output stdout.

Finally process the system file. This file is generated using the GCC compiler, so its execution header format is GCC type, the same as the a.out format defined by linux. After confirming that the execution entry point is 0, the data is written to the standard output stdout. If the code and data size exceeds 128 KB, an error message is displayed. The resulting kernel Image file format is shown in Figure 16-2, where the format is:

- The first sector stores the bootsect code, which is exactly 512 bytes long;
- The 4 sectors (2 - 5 sectors) starting from the 2nd sector store the setup code, and the size is no more than 4 sectors;
- The system module is stored starting from the sixth sector, and its length does not exceed the size (128KB) defined on line 37 of build.c.

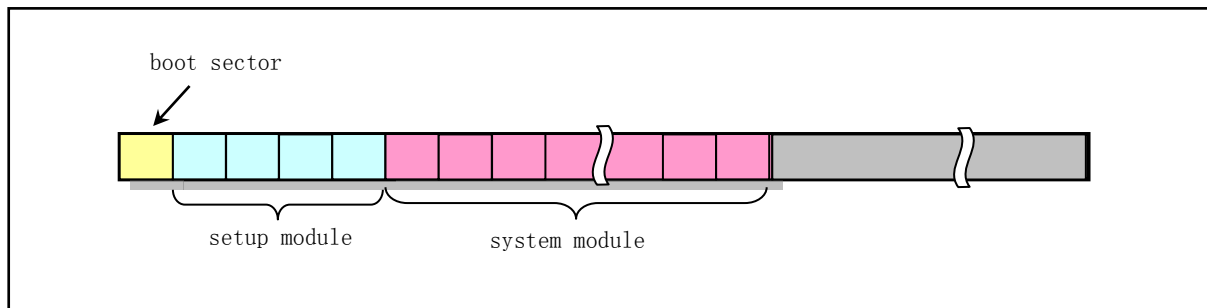


Figure 16-2 The format of Linux kernel on disk

16.1.2 Code annotation

Program 16-1 linux/tools/build.c

```

1  /*
2  *  linux/tools/build.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  This file builds a disk-image from three different files:
9  *
10 *  - bootsect: max 510 bytes of 8086 machine code, loads the rest
11 *  - setup: max 4 sectors of 8086 machine code, sets up system parm

```

```

12  * - system: 80386 code for actual system
13  *
14  * It does some checking that all files are of the correct type, and
15  * just writes the result to stdout, removing headers and padding to
16  * the right amount. It also writes some system data to stderr.
17  */
18
19 /*
20  * Changes by tytso to allow root device specification
21  *
22  * Added swap-device specification: Linux 20.12.91
23  */
24
25 #include <stdio.h>      /* fprintf */
26 #include <string.h>
27 #include <stdlib.h>     /* contains exit */
28 #include <sys/types.h>  /* unistd.h needs this */
29 #include <sys/stat.h>   // file state information structure
30 #include <linux/fs.h>
31 #include <unistd.h>     /* contains read/write */
32 #include <fcntl.h>      // file operation mode constants
33
34 #define MINIX_HEADER 32      // The MINIX object file header size is 32 bytes.
35 #define GCC_HEADER 1024     // The GCC header information size is 1024 bytes.
36
37 #define SYS_SIZE 0x3000      // The max size of the system module (SYS_SIZE*16=128KB).
38
39 // By default, the Linux root fs device is the first partition of the second hard disk (device
40 // number 0x0306). This is because when Mr. Linus developed Linux, he used the first hard disk
41 // as the MINIX system disk and the second hard disk as the Linux root file system disk.
42 #define DEFAULT_MAJOR_ROOT 3 // major device number - 3, hard disk.
43 #define DEFAULT_MINOR_ROOT 6 // minor device number - 6, 1st partition of the 2nd disk.
44
45 #define DEFAULT_MAJOR_SWAP 0 // swap device number.
46 #define DEFAULT_MINOR_SWAP 0
47
48 /* max nr of sectors of setup: don't change unless you also change
49  * bootsect etc */
50 #define SETUP_SECTS 4        // The maximum size of setup is 4 sectors (2KB).
51
52 #define STRINGIFY(x) #x      // Convert x to string type for use in an error display.
53
54 // Display an error message, and terminate the program.
55 void die(char * str)
56 {
57     fprintf(stderr, "%s\n", str);
58     exit(1);
59 }
60
61 // Display the program usage and exit.
62 void usage(void)
63 {
64     die("Usage: build bootsect setup system [rootdev] [> image]");
65 }

```

```

60 }
61
62 // The main program begins...
63 // The program first checks the parameters on the command line for compliance, sets the root
64 // device number and the swap device number, then reads and processes the bootsect, setup, and
65 // system module files, respectively, and writes them to the standard output that has been
66 // redirected to the Image file. .
67 int main(int argc, char ** argv)
68 {
69     int i,c,id;
70     char buf[1024];
71     char major_root, minor_root;
72     char major_swap, minor_swap;
73     struct stat sb;
74
75     // (1) First check the actual command line parameters when the build program is executed, and
76     // set accordingly according to the number of parameters. The build program requires 4 to 6
77     // parameters. If the number of parameters on the command line does not meet the requirements,
78     // the program usage is displayed and exited.
79     // If there are more than 4 parameters on the program command line, if the root device name
80     // is not "FLOPPY", the status information of the device file is taken, and the major and minor
81     // device number are taken as the root device number. If the root device is a FLOPPY device,
82     // set the major and minor device number to 0, indicating that the root device is the current
83     // boot device.
84     if ((argc < 4) || (argc > 6))
85         usage();
86     if (argc > 4) {
87         if (strcmp(argv[4], "FLOPPY")) {
88             if (stat(argv[4], &sb)) {
89                 perror(argv[4]);
90                 die("Couldn't stat root device.");
91             }
92             major_root = MAJOR(sb.st_rdev); // Get the device number.
93             minor_root = MINOR(sb.st_rdev);
94         } else {
95             major_root = 0;
96             minor_root = 0;
97         }
98     }
99     // If there are only 4 parameters, let the major and minor device number be equal to the system
100     // default root device number.
101     } else {
102         major_root = DEFAULT_MAJOR_ROOT;
103         minor_root = DEFAULT_MINOR_ROOT;
104     }
105
106     // If there are 6 parameters on the command line, if the last parameter indicating that the
107     // switching device is not "NONE", the status information of the device file is taken, and the
108     // major and minor device numbers are taken as the swap device number. If the last parameter
109     // is "NONE", the major and the minor device number of the swap device are taken as 0, indicating
110     // that the swap device is the current boot device.
111     if (argc == 6) {
112         if (strcmp(argv[5], "NONE")) {
113             if (stat(argv[5], &sb)) {
114                 perror(argv[5]);

```

```

92             die("Couldn't stat root device.");
93         }
94         major_swap = MAJOR(sb.st_rdev);
95         minor_swap = MINOR(sb.st_rdev);
96     } else {
97         major_swap = 0;
98         minor_swap = 0;
99     }
// If there are no 6 parameters but 5, it means that there is no swap device name on the command
// line. So let the swap device major and minor device numbers equal the system default swap
// device number.
100     } else {
101         major_swap = DEFAULT_MAJOR_SWAP;
102         minor_swap = DEFAULT_MINOR_SWAP;
103     }

// Next, the major and minor device numbers of the root device selected and the major and minor
// device numbers of the swap device are displayed on the standard error terminal. If the major
// device number is not 2 (floppy disk) or 3 (hard disk), nor is it 0 (system default device),
// an error message is displayed and exits. The standard output of the terminal is redirected
// to the file 'Image', so it is used to output the saved kernel code and data to generate a
// kernel image file.
104     fprintf(stderr, "Root device is (%d, %d)\n", major_root, minor_root);
105     fprintf(stderr, "Swap device is (%d, %d)\n", major_swap, minor_swap);
106     if ((major_root != 2) && (major_root != 3) &&
107         (major_root != 0)) {
108         fprintf(stderr, "Illegal root device (major = %d)\n",
109             major_root);
110         die("Bad root device --- major #");
111     }
112     if (major_swap && major_swap != 3) {
113         fprintf(stderr, "Illegal swap device (major = %d)\n",
114             major_swap);
115         die("Bad root device --- major #");
116     }
// (2) The following begins to read the contents of each file and perform the corresponding
// copy processing. First initialize the 1KB buffer, then open the file specified by parameter
// 1 (bootsect) in read-only mode, and read the 32-byte MINIX execution header structure (see
// the list below) into the buffer buf.
117     for (i=0; i<sizeof buf; i++) buf[i]=0;
118     if ((id=open(argv[1], O_RDONLY, 0))<0)
119         die("Unable to open 'boot'");
120     if (read(id, buf, MINIX_HEADER) != MINIX_HEADER)
121         die("Unable to read header of 'boot'");
// Next, check if the bootsect is a valid MINIX executable file based on the MINIX header
// structure. If so, the 512-byte boot sector code and data are read from the file. The value
// "0x04100301" means: 0x0301 - MINIX head magic value a_magic; 0x10 - executable flag a_flag;
// 0x04 - machine type a_cpu, Intel 8086 machine code.
// A series of checks are then performed on the header information. Check if the header size
// field a_hdrlen is correct (32 bytes) (The last three bytes are just useless, it is 0); confirm
// whether the data segment size a_data field (long) content is 0; confirm whether the heap
// a_bss field (long) is 0; confirm whether the execution point a_entry field (long) is 0;
// confirms whether the symbol table size field a_sym is 0.

```

```

122     if (((long *) buf)[0] != 0x04100301)
123         die("Non-Minix header of 'boot'");
124     if (((long *) buf)[1] != MINIX_HEADER)
125         die("Non-Minix header of 'boot'");
126     if (((long *) buf)[3] != 0)
127         die("Illegal data segment in 'boot'");
128     if (((long *) buf)[4] != 0)
129         die("Illegal bss in 'boot'");
130     if (((long *) buf)[5] != 0)
131         die("Non-Minix header of 'boot'");
132     if (((long *) buf)[7] != 0)
133         die("Illegal symbol table in 'boot'");
    // After reading the actual code and data in the file under the condition that the above judgments
    // are correct, the number of read bytes should be 512 bytes. Because the bootsect file contains
    // the boot sector code and data for one sector, and the last 2 bytes should be the bootable
    // flag 0xAA55.
134     i = read(id, buf, sizeof buf);
135     fprintf(stderr, "Boot sector %d bytes. \n", i);
136     if (i != 512)
137         die("Boot block must be exactly 512 bytes");
138     if ((* (unsigned short *) (buf + 510)) != 0xAA55)
139         die("Boot block hasn't got boot flag (0xAA55)");
    // Then modify the contents of the buffer, store the swap device number at the offset of 506,
    // 507, and store the root device number at the offset of 508, 509.
140     buf[506] = (char) minor_swap;
141     buf[507] = (char) major_swap;
142     buf[508] = (char) minor_root;
143     buf[509] = (char) major_root;
    // Next, write the 512 bytes of data to the standard output stdout and close the bootsect file.
    // In linux/Makefile, the build program redirects the standard output to the kernel image
    // filename Image using the ">" indicator, so the boot sector code and data are written to the
    // first 512 bytes of the Image.
144     i = write(1, buf, 512);
145     if (i != 512)
146         die("Write call failed");
147     close(id);
148
    // (3) The file (setup) specified by parameter 2 is opened in read-only mode, and the contents
    // of the 32-byte MINIX execution file header are read to the buffer buf. Next, according to
    // the MINIX header structure, it is checked whether the setup is a valid MINIX execution file.
    // If so, a series of checks are performed on the header information. It is handled in the same
    // way as above.
149     if ((id = open(argv[2], O_RDONLY, 0)) < 0)
150         die("Unable to open 'setup'");
151     if (read(id, buf, MINIX_HEADER) != MINIX_HEADER)
152         die("Unable to read header of 'setup'");
153     if (((long *) buf)[0] != 0x04100301)
154         die("Non-Minix header of 'setup'");
155     if (((long *) buf)[1] != MINIX_HEADER) // header size (32 bytes)
156         die("Non-Minix header of 'setup'");
157     if (((long *) buf)[3] != 0) // data size field a_data
158         die("Illegal data segment in 'setup'");
159     if (((long *) buf)[4] != 0) // a_bss field.

```



```

160         die("Illegal bss in 'setup'");
161     if (((long *) buf)[5] != 0)                // a_entry point.
162         die("Non-Minix header of 'setup'");
163     if (((long *) buf)[7] != 0)
164         die("Illegal symbol table in 'setup'");
    // The subsequent actual code and data in the file are read under the condition that the above
    // checks are correct, and written to the terminal standard output. At the same time, the length
    // of the writing is counted, and the setup file is closed after the operation ends. Then check
    // the code and data size of the write operation, the value can not be greater than (SETUP_SECTS
    // * 512) bytes, otherwise you have to re-edit the number of sectors occupied by the setup set
    // in the build, bootsect and setup programs, and recompile the kernel. If everything is ok,
    // the actual length value of the setup is displayed.
165     for (i=0 ; (c=read(id, buf, sizeof buf))>0 ; i+=c )
166         if (write(1, buf, c)!=c)
167             die("Write call failed");
168     close (id);                                // close setup file.
169     if (i > SETUP_SECTS*512)
170         die("Setup exceeds " STRINGIFY(SETUP_SECTS)
171             " sectors - rewrite build/boot/setup");
172     fprintf(stderr, "Setup is %d bytes. \n", i);
    // After clearing the buffer buf, check the difference between the actual write setup length and
    // (SETUP_SECTS * 512). If the setup size is smaller than the size (4 * 512 bytes), fill the
    // setup with NULL characters to 4 * 512 bytes.
173     for (c=0 ; c<sizeof(buf) ; c++)
174         buf[c] = '\0';
175     while (i<SETUP_SECTS*512) {
176         c = SETUP_SECTS*512-i;
177         if (c > sizeof(buf))
178             c = sizeof(buf);
179         if (write(1, buf, c) != c)
180             die("Write call failed");
181         i += c;
182     }
183
    // (4) Start processing the system module file below. This file is compiled with gas/gcc and
    // therefore has the GNU a.out object file format.
    // First open the system module file in read-only mode, and read the a.out format header structure
    // information (1KB size). After confirming that system is a valid a.out format file, write all
    // subsequent data of the file to the standard output (Image file) and close the file. Then show
    // the size of the system. If the system code and data size exceed the SYS_SIZE section (128KB
    // bytes), an error message is displayed and exits. If there is no error, it returns 0, indicating
    // normal exit.
184     if ((id=open(argv[3], O_RDONLY, 0))<0)
185         die("Unable to open 'system'");
186     if (read(id, buf, GCC_HEADER) != GCC_HEADER)
187         die("Unable to read header of 'system'");
188     if (((long *) buf)[5] != 0)                // entry location should be 0.
189         die("Non-GCC header of 'system'");
190     for (i=0 ; (c=read(id, buf, sizeof buf))>0 ; i+=c )
191         if (write(1, buf, c)!=c)
192             die("Write call failed");
193     close(id);
194     fprintf(stderr, "System is %d bytes. \n", i);

```

```

195     if (i > SYS\_SIZE*16)
196         die("System is too big");
197     return(0);
198 }
199

```

16.1.3 Information

16.1.3.1 MINIX module and executable header data structure

The header structure of the modules and executables generated by MINIX's compiler and linker is as follows:

```

struct exec {
    unsigned char a_magic[2];    // Magic number, should be 0x0301.
    unsigned char a_flags;      // Flags (see below).
    unsigned char a_cpu;        // Machine CPU identifier.
    unsigned char a_hdrlen;     // Reserved header size, 32 or 48 bytes.
    unsigned char a_unused;     // Reserved.
    unsigned short a_version;    // Version information (not used currently).
    long          a_text;        // Code section size (in bytes).
    long          a_data;        // Data section size (in bytes).
    long          a_bss;        // Stack size (in bytes).
    long          a_entry;       // Execute entry point.
    long          a_total;       // Total amount of memory allocated.
    long          a_syms;        // Symbol table size.
    // The structure ends here if the header size is 32 bytes.
    long          a_trsize;      // Code section relocation table size.
    long          a_drsize;      // Data section relocation table size.
    long          a_tbase;       // Code section relocation base address.
    long          a_dbase;       // Data section relocation base address.
};

```

Among them, the flag field `a_flags` in the MINIX execution file header is defined as:

<code>A_UZP</code>	<code>0x01</code>	// Unmapped 0 pages (pages).
<code>A_PAL</code>	<code>0x02</code>	// Adjusted at the page boundary.
<code>A_NSYM</code>	<code>0x04</code>	// New type symbol table.
<code>A_EXEC</code>	<code>0x10</code>	// Executable file.
<code>A_SEP</code>	<code>0x20</code>	// The code and data are separate (I and D are independent).

The CPU identification number field `a_cpu` can be:

<code>A_NONE</code>	<code>0x00</code>	// Unknown.
<code>A_I8086</code>	<code>0x04</code>	// Intel i8086/8088.
<code>A_M68K</code>	<code>0x0B</code>	// Motorola m68000.
<code>A_NS16K</code>	<code>0x0C</code>	// National Semiconductor Co. 16032.
<code>A_I80386</code>	<code>0x10</code>	// Intel i80386.
<code>A_SPARC</code>	<code>0x17</code>	// Sun SPARC.

The above MINIX execution header structure `exec` is similar to the `a.out` format header structure used by the Linux 0.12 system. See the `linux/include/a.out.h` file for the header structure and related information of the Linux `a.out` format executable file.

16.2 Summary

This chapter details the build tool `build.c` given in the kernel source tree. This program is mainly used to modify and combine the kernel modules to generate a bootable kernel boot image file. So far, we have completed a detailed description and comments on all source code files in the kernel.

In order to gain a deeper understanding of the kernel's operating mechanism, in the next chapter we use the Bochs simulation program to detail the test methods in which the Linux 0.12 operating system is set up and running, and give specific test steps for some experiments.

17 Experimental Environment Settings and Usage

In order to learn the working principle of the Linux 0.1x kernel, this chapter introduces the experimental method of using the PC simulation software and running the Linux 0.1x system on the real computer. These include the kernel compilation process, file access and copying in the simulation environment, how to make the boot disk and root file system, and how to use the Linux 0.1x system. Finally, we also introduced how to make a small number of syntax changes to the kernel code, so that it can successfully pass the compilation process under the existing RedHat system (gcc 3.x), and make the corresponding kernel image file.

Before we start the experiment, we first need to prepare some useful tools. If we are going to experiment on the Windows platform, we need to have the following software ready:

- ♦ Bochs 2.6.x open source PC simulation package (<https://sourceforge.net/projects/bochs/>);
- ♦ Notepad++ Editor. Used to edit binary files (<https://sourceforge.net/projects/notepad-plus/>);
- ♦ HxD hex editor. Used to edit binary or disk files, even in-memory data (<https://mh-nexus.de/en/hxd/>);
- ♦ WinImage DOS format floppy image file editing software (<http://www.winimage.com/>).

If you are experimenting with modern Linux systems (such as Redhat, Ubuntu, etc.), then we usually only need to install the Bochs package to perform the simulation. Other operations in the experiment can be done using the common tools of the Linux system.

The best way to run a Linux 0.1x system is to use PC emulation software. There are currently four popular PC simulation software in the world: VMware's VMware Workstation software, Oracle's VirtualBox open source software, Microsoft's Virtual PC and open source software Bochs (the pronunciation is the same as "box"). They can fully virtualize and simulate the operation of a PC. These types of software can virtualize or emulate an Intel x86 hardware environment, allowing us to run a variety of other "customer" operating systems on the platform on which the software is running.

In terms of scope of use and operational performance, the four simulation softwares still have some differences. Bochs uses software to simulate the entire hardware environment (CPU instructions) of the x86 CPU-based PC and its peripherals, so it can be easily ported to many operating systems or platforms with different architectures. Because it mainly uses software simulation technology, its running performance and speed are much slower than other simulation software. The performance of Virtual PC software is between Bochs and VMware Workstation (or VirtualBox). It emulates most of the x86 instructions, while other parts are implemented using virtual technology. VMware Workstation and VirtualBox only simulate some I/O capabilities, while all other parts are executed directly on x86 real-time hardware. That is to say, when the guest operating system is required to execute an instruction, VMware or VirtualBox does not execute this instruction by simulation, but simply "passes" this instruction directly to the hardware of the actual system. So VMware and VirtualBox are the best in terms of speed and performance in these software. Of course, other simulation software, such as Qemu, can also have high simulation performance.

From an application perspective, VMware Workstation and VirtualBox should be a good choice if the simulation environment is primarily used to run applications. But if you need to develop some low-level system software (such as operating system development and debugging, compiler system development, etc.), then

Bochs is a good choice. With Bochs, you can know the specific state and precise timing of the executed program in the simulated hardware environment, rather than the actual hardware system execution. This is why many operating system developers are more inclined to use Bochs. This chapter describes how to run Linux 0.1x using the Bochs simulation environment. Currently, the name of the Bochs website is <http://sourceforge.net/projects/bochs/>. You can download the latest release of the Bochs software from above, or you can download the image files of many ready-to-run systems.

17.1 Bochs Simulation Software

Bochs is a program that fully emulates an Intel 80X86 computer. It can be configured to emulate Intel's 80386, 486, Pentium or above new CPU processors. Throughout the execution phase, Bochs simulates all execution instructions, including emulating all device modules of standard PC peripherals. Since Bochs simulates the entire PC environment, the software executing in it "thinks" that it is running on a real machine. This fully simulated approach allows us to run a large number of software systems without modification in Bochs.

Bochs is a software system developed by Kevin Lawton in 1994 using the C++ language. The system is designed to run in hardware environments such as Intel 80X86, PPC, Alpha, Sun, and MIPS. Regardless of the hardware platform on which the host is running, Bochs can still simulate the Intel hardware platform of the Intel 80X86 CPU. This feature is not available in several other simulation software. In order to perform any activity on the machine being simulated, Bochs needs to interact with the host operating system. When a key is pressed in the Bochs display window, a keystroke event is sent to the keyboard device processing module. When the simulated machine needs to perform a read operation from the simulated hard disk, Bochs will perform a read operation on the hard disk image file on the host.

The installation of Bochs software is very convenient. You can download the Bochs installation package directly from <http://bochs.sourceforge.net>. If the computer operating system you are using is Windows, the installation process is exactly the same as normal software. After the Bochs software is installed, it will generate a directory on the C: 'C:\Program Files\Bochs-2.6' (where the version number varies with different versions). If your system is RedHat or another Linux system, you can download the Bochs RPM package and install it as follows:

```
user$ su
Password:
root# rpm -i bochs-2.6.i386.rpm
root# exit
user$ _
```

You need root privileges when installing Bochs, otherwise you will have to recompile the Bochs system in your own directory. In addition, Bochs needs to run in the X11 environment, so you must have the X Window System installed on your Linux system to use Bochs. After installing Bochs, it is recommended to test and familiarize yourself with the Bochs system using the Linux dlx demo system included with the Bochs package. You can also download some of the other Linux image files from the Bochs website to do some experimentation. We recommend downloading the SLS Linux emulation system package (sls-0.99pl.tar.bz2) on the Bochs website as an auxiliary platform for creating Linux 0.1x emulation systems. When making new hard disk image files, we can use these systems to partition and format the hard disk image files. The image file for this SLS

Linux system can also be downloaded directly from oldlinux.org: <http://oldlinux.org/Linux.old/bochs/sls-1.0.zip>. After extracting the downloaded file, go to its directory and double-click the configuration file name bochsrc.bxrc to let Bochs run the SLS Linux system. If the configuration file name does not have the suffix .bxrc, please modify it yourself. For example, the original name bochsrc was modified to bochsrc.bxrc.

For instructions on recompiling the Bochs system or installing Bochs on other hardware platforms, please refer to the instructions in the Bochs User Manual.

17.1.1 Setting up the Bochs system

In order to run an operating system in Bochs, we need at least some of the following resources or information:

-
- ♦ Bochs and bochsdbg executable files;
 - ♦ BIOS image files (commonly referred to as 'BIOS-bochs-latest');
 - ♦ VGABIOS image files (eg 'VGABIOS-lgpl-latest');
 - ♦ at least one boot image file (floppy, hard disk or CDROM image) file).
-

Bochs.exe is the executable for the Bochs system. If you need to track and debug the program in Bochs, you also need bochsdbg.exe to execute the program. The BIOS and VGABIOS are image files emulating the ROM BIOS of the PC and the BIOS software in the display card, respectively. In addition, we also need the boot image file of the system to be used for emulation. These files need to work together, so we need to set some environment parameters for the simulation before running the Bochs program. These parameters can be passed to the Bochs executable on the command line, but we usually use a textual configuration file (file suffix .bxrc, such as Sample.bxrc) to set the running parameters for a specific application. The following describes how to set up the Bochs configuration file.

17.1.2 *.bxrc Configuration File

Bochs uses the information in the configuration file to find the disk image file used, the configuration of the operating environment peripherals, and other virtual machine setup information. Each simulated system needs to set a corresponding configuration file. If the installed Bochs system is 2.1 or later, the Bochs system will automatically recognize the configuration file with the suffix '.bxrc' and automatically start the Bochs system when you double-click the file icon. For example, we can take the configuration file name as 'bochsrc-0.12.bxrc'. In the Bochs installation home directory (usually C:\Program Files\Bochs-2.6\) there is a template configuration file named 'bochsrc-sample.txt' which lists all available parameters with Detailed explanation. Here are a few of the parameters that are often modified in our experiments.

1. megs

Used to set the memory capacity of the simulated system. The default is 32MB. For example, if you want to set up your simulated machine to have a 128MB system, you need to include the following line in your configuration file:

```
megs: 128
```

2. floppy (floppyb)

Floppya represents the first floppy drive, and floppyb represents the second one. If you need to boot the

system from a floppy disk, then floppya needs to point to a bootable disk. If you want to use a disk image file, then we will write the name of the disk image file after this option. In many operating systems, Bochs can directly read and write the floppy disk drive of the host system. To access the disks in these actual drives, use the device name (Linux system) or drive letter (Windows system). You can also use status to indicate the insertion status of the disk: 'ejected' means not inserted, and 'inserted' means the disk is inserted. Here are a few examples where all the disks are inserted. If there are several rows of parameters with the same name in the configuration file, only the last row of parameters will work.

```
floppya: 1_44=/dev/fd0, status=inserted      # Access to 1.44MB A drive under Linux.
floppya: 1_44=b:, status=inserted           # Access to 1.44MB B drive under Win.
floppya: 1_44=bootimage.img, status=inserted # Use imagefile bootimage.img.
floppyb: 1_44=..\Linux\rootimage.img, status=inserted # Use image ..\Linux\rootimage.img.
```

3. ata0、ata1、ata2、ata3

These four parameter names are used to start up to four ATA channels in the simulated system. For each enabled channel, two IO base addresses and one interrupt request number must be specified. By default only ata0 is enabled and the parameters default to the values shown below:

```
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15
ata2: enabled=1, ioaddr1=0x1e8, ioaddr2=0x3e0, irq=11
ata3: enabled=1, ioaddr1=0x168, ioaddr2=0x360, irq=9
```

4. ata0-master (ata0-slave)

The ata0-master is used to indicate the first ATA device (hard disk or CDROM, etc.) connected to the first ATA channel (0 channel) in the simulated system; the ata0-slave indicates the second ATA device connected to the first channel. An example is shown below, where the options for device configuration are as shown in Table 17-1.

```
ata0-master: type=disk, path=hd.img, mode=flat, cylinders=306, heads=4, spt=17, translation=none
ata1-master: type=disk, path=2G.cow, mode=vmware3, cylinders=5242, heads=16, spt=50, translation=echs
ata1-slave:  type=disk, path=3G.img, mode=sparse, cylinders=6541, heads=16, spt=63, translation=auto
ata2-master: type=disk, path=7G.img, mode=undoable, cylinders=14563, heads=16, spt=63, translation=lba
ata2-slave:  type=cdrom, path=iso.sample, status=inserted
ata0-master: type=disk, path="hdc-large.img", mode=flat, cylinders=487, heads=16, spt=63
ata0-slave:  type=disk, path="..\hdc-large.img", mode=flat, cylinders=121, heads=16, spt=63
```

Table 17-1 Device configuration options

Options	Description	Available value
type	Connected device type	[disk cdrom]
path	Image file path name	
mode	Image file type, valid only for disk	[flat concat external dll sparse vmware3 undoable growing volatile]

cylinders	Valid only for disk	
heads	Valid only for disk	
spt	Valid only for disk	
status	Valid only for disk	[inserted ejected]
biosdetect	Bios detect type	[none auto], Only valid for disk on ata0 [cmos]
translation	The type of bios conversion (int13), valid only for disk	[none lba large rechs auto]
mode	Confirm the string returned by the device ATA command	

When configuring an ATA device, you must specify the type of the connected device, which can be 'disk' or 'cdrom'. You must also specify the pathname of the device. The "pathname" can be a hard disk image file, an iso file of the CDROM, or a CDROM drive pointing directly to the system. In Linux systems, system devices can be used as Bochs hard drives, but for security reasons, direct use of physical hard disks on the system is not recommended under Windows.

For devices of type 'disk', the options 'path', 'cylinders', 'heads' and 'spt' are required. For devices of type 'cdrom', the option 'path' is required.

The disk translation scheme (implemented in the traditional int13 bios function and used for old os like DOS) can be defined as:

- ♦ none: No need to translate, suitable for hard disks with a capacity less than 528MB (1032192 sectors);
- ♦ large: Standard bit shift algorithm for hard disks with a capacity of less than 4.2 GB (8257536 sectors);
- ♦ rechs: The modified shift algorithm uses a pseudo-physical hard disk parameter of 15 heads for hard disks with a capacity less than 7.9 GB (15482880 sectors);
- ♦ lba: Standard lba-assisted algorithm. Suitable for hard drives with a capacity less than 8.4GB (16,450,560 sectors);
- ♦ auto: Automatically select the best conversion scheme (should be changed if system does not start).

The mode option is used to explain how to use the hard disk image file. It can be one of the following modes:

- ♦ flat: a flat sequential file;
- ♦ concat: Multiple files;
- ♦ external: Dedicated by the developer, specified by the C++ class;
- ♦ dll: Dedicated by the developer, used by the DLL;
- ♦ sparse: Stackable, identifiable, retractable;
- ♦ vmware3: Support vmware3 hard disk format;
- ♦ undoable: a flat file with a confirmed redo log;
- ♦ growing: Capacity scalable image file;
- ♦ volatile: A flat file with a variable redo log.

The default values for the above options are:

```
mode=flat, biosdetect=auto, translation=auto, model="Generic 1234"
```

5. boot

'boot' is used to define the drive in the emulated machine for boot. It can be specified as a floppy disk, hard disk or CDROM, or drive letters 'c' and 'a'. Examples are as follows:

```
boot: a
boot: c
boot: floppy
boot: disk
boot: cdrom
```

6. cpu

The 'cpu' is used to define the parameters of the CPU that is simulated in the simulation system. This option can take four parameters: COUNT, QUANTUM, RESET_ON_TRIPLE_FAULT, and IPS.

Where 'COUNT' is used to indicate the number of processors emulated in the system. When the Bochs package is compiled with the SMP support option, Bochs currently supports up to 8 simultaneous threads. However, if the compiled Bochs does not support SMP, COUNT can only be set to 1.

'QUANTUM' is used to specify the maximum number of instructions that can be executed before switching from one processor to another. This option is also only available for Bochs programs that support SMP.

'RESET_ON_TRIPLE_FAULT' is used to specify that the CPU needs to perform a reset operation instead of just a panic when a triple error occurs in the processor.

IPS specifies the number of instructions per second to be simulated. This is the IPS value that Bochs runs on the host system. This value affects many events related to time in the simulation system. For example, changing the IPS value will affect the rate of VGA updates and other simulation system evaluations. It is therefore necessary to set this value depending on the host performance used. Refer to Table 17-2 for settings. For example:

```
cpu: count=1, ips=50000000, reset_on_triple_fault=1
```

Table 17-2 Examples IPS settings

Bochs version	Speed	Machine / Compiler	Typical IPS
2.4.6	3.4Ghz	Intel Core i7 2600 with Win7x64/g++ 4.5.2	85 to 95 MIPS
2.3.7	3.2Ghz	Intel Core 2 Q9770 with WinXP/g++ 3.4	50 to 55 MIPS
2.3.7	2.6Ghz	Intel Core 2 Due with WinXP/g++ 3.4	38 to 43 MIPS
2.2.6	2.6Ghz	Intel Core 2 Due with WinXP/g++ 3.4	21 – 25 MIPS
2.2.6	2.1Ghz	Athlon XP with Linux 2.6/g++ 3.4	12 – 15 MIPS

7. log

Specifying the path name of 'log' allows Bochs to log some log information during execution. If the system running in Bochs does not work properly, you can refer to the information to find out the basic reason. The log is usually set to:

```
log: bochsout.txt
```

17.2 Running Linux 0.1x system in Bochs

To run a Linux operating system, we need a root filesystem (root fs) in addition to the kernel. The root file system is usually an external device that stores the necessary files (such as system configuration files and device files) and data files for Linux system runtime. In modern Linux operating systems, the kernel image file (bootimage) is stored in the root file system. The system boot initiator loads the kernel execution code into memory from this root file system device for execution.

However, the kernel image file and the root file system are not required to be stored on the same device, that is, they are not necessarily required to be placed in a floppy disk or in the same partition of the hard disk. For the case of using only floppy disks, due to the limitation of floppy disk capacity, the kernel image file and the root file system are usually placed on two separate disks. The floppy disk that stores the bootable kernel image file is called the kernel boot disk. (bootimage); The floppy disk that holds the root file system is called the root file system image file (rootimage). Of course, we can also load the kernel image file from the floppy disk, and at the same time use the root file system in the hard disk, or let the system boot the system directly from the hard disk, that is, load the kernel image file from the root file system of the hard disk and use the root file system in the hard disk.

This section describes how to run several Linux 0.1x systems that have been set up in Bochs and explains the settings of several main parameters in the relevant configuration files. First we download the following Linux 0.1x system package from the website to the desktop of the computer:

```
http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip
```

The last 6 digits in the package name are date information. You should usually choose the latest package with the latest download date. After the download is complete, you can use a general decompression program such as unzip, 7-zip or rar to decompress it. Note that you need about 250MB of disk space to unzip this file.

17.2.1 Description of the files in the package

When the linux-0.12-080324.zip file is unzipped, a directory named linux-0.12-080324 is generated. After entering the directory, we can see that there are about 20 files below.

```
[root@www linux-0.12-080324]# ls -o -g
total 256916
-rw-r--r--  1  3078642 Mar 24 10:49 bochs-2.3.6-1.i586.rpm
-rw-r--r--  1  3549736 Mar 24 10:48 Bochs-2.3.6.exe
-rw-r--r--  1    15533 Mar 24 18:04 bochsout.txt
```

```
-rw-r--r-- 1      1774 Mar 24 20:13 bochsrc-0.12-fd.bxrc
-rw-r--r-- 1      5903 Mar 24 17:56 bochsrc-0.12-hd.bxrc
-rw-r--r-- 1     35732 Dec 24 20:15 bochsrc-sample.txt
-rw-r--r-- 1    150016 Mar  6 2004 bootimage-0.12-fd
-rw-r--r-- 1    154624 Aug 27 2006 bootimage-0.12-hd
-rw-r--r-- 1        68 Mar 24 12:21 debug.bat
-rw-r--r-- 1   1474560 Mar 24 15:27 diska.img
-rw-r--r-- 1   1474432 Aug 27 2006 diskb.img
-rw-r--r-- 1       7917 Mar 24 11:32 linux-0.12-README
-rw-r--r-- 1   1474560 Mar 24 17:03 rootimage-0.12-fd
-rw-r--r-- 1 251338752 Mar 24 18:04 rootimage-0.12-hd
-rw-r--r-- 1     21253 Mar 13 2004 SYSTEM.MAP
[root@www linux-0.12-080324]#
```

This package contains two Bochs installers, two Bochs .bxrc configuration files, two bootimage files containing kernel code, a floppy disk and a hard disk root file system (rootimage) file, and other files. 其中 The README file briefly describes the purpose of each file. Here we will explain in more detail the purpose of each file.

- Bochs-2.3.6-1.i586.rpm is the Bochs installer for the Linux operating system. You can re-download the latest program.
- Bochs-2.3.6.exe is the Bochs installer for the Windows operating system platform. The latest version of the Bochs software can be downloaded from <http://sourceforge.net/projects/bochs/>. As Bochs continues to improve, some newer versions may cause compatibility issues. This needs to be resolved by modifying the .bxrc configuration file, and some issues need to be resolved by modifying the Linux 0.1x kernel code.
- bochsout.txt is a log file that is automatically generated when the Bochs system is running. It contains various status information for the Bochs runtime. When running Bochs has problems, you can check the contents of this file to preliminarily determine the cause of the problem.
- bochsrc-0.12-fd.bxrc is the configuration file that allows the system to boot from a floppy disk. This configuration file is used to boot the Linux 0.12 system from the Bochs Virtual A drive (/dev/fd0), ie the kernel image file is set in virtual disk A and the subsequent root file system is required to be inserted into the current virtual boot drive. During the boot process it will ask us to "insert" the root filesystem disk (rootimage-0.12-fd) in the virtual A drive. The kernel image and boot file used by this configuration file is bootimage-0.12-fd. After Bochs is properly installed, double-click this configuration file to run the configured Linux 0.12 system.
- bochsrc-0.12-hd.bxrc is also a configuration file set to boot from drive A, but will use the root file system in the hard drive image file (rootimage-0.12-hd). This configuration file is booted using bootimage-0.12-hd. Similarly, after properly installing Bochs, double-click on this configuration file to run the configured Linux 0.12 system.
- bootimage-0.12-fd is an image file generated by the compiled kernel. It contains the code and data for the entire kernel, including the code for the floppy boot sector. You can run the configured Linux 0.12 system by double-clicking on the relevant configuration file.
- bootimage-0.12-hd is the kernel image file used to use the root file system on the virtual hard disk, that is, the root file system device number of the 509th and 510th bytes of the file has been set to the 1st partition of the C hard disk (/dev /hd1), the device number is 0x0301.
- debug.bat is a batch program that starts the Bochs debugging function on the Windows platform.

Please note that you may need to modify the pathname based on the specific directory where Bochs is installed. In addition, the Bochs system installed and running on Linux systems by default does not include debugging features. You can debug directly using the gdb program on Linux. If you still want to take advantage of Bochs' debugging features, then you need to download the source code of Bochs and customize it yourself.

- `diska.img` and `diskb.img` are two floppy image files in DOS format. It contains some utilities. In Linux 0.12 you can use the command `mcopy` and other commands to access these two image files. Of course, you need to dynamically "insert" the corresponding "floppy disk" before accessing. When you double-click the `bochsrc-0.12-fd.bxrc` or `bochsrc-0.12--hd.bxrc` configuration file to run the Linux 0.12 system, the B drive is configured to be "inserted" with the `diskb.img` disk.
- `rootimage-0.12-hd` is the virtual hard disk image file mentioned above, which contains 3 partitions. The first partition is a MINIX file system type 1.0 root file system, and the other two partitions are also MINIX 1.0 file system types, and some source code files for testing are stored. You can load and use these spaces by using the `mount` command.
- `rootimage-0.12-fd` is the root file system on the floppy disk. This root file system disk is used when running the Linux 0.12 system using the `bochsrc-0.12-fd` configuration file.
- The `SYSTEM.MAP` file is the kernel memory storage location information file generated when the Linux 0.12 kernel is compiled. The contents of this file are very useful when debugging the kernel.

17.2.2 Installing the Bochs

The `bochs-2.3.6-1.i586.rpm` file in the package is the Bochs installer used under Linux. `Bochs-2.3.6.exe` is the Bochs installer on the Windows operating system. The latest version of the Bochs software is always available at the following website location:

<http://sourceforge.net/projects/bochs/>

If we are experimenting with a Linux system, you can install the Bochs software by running the `rpm` command on the command line or by double-clicking on the first file in the above package in the X window:

```
rpm -i bochs-2.3.6-1.i586.rpm
```

If you are on a Windows system, simply double-click the `Bochs-2.3.6.exe` file icon to install the Bochs system. After the installation, please modify the contents of the batch file `debug.bat` according to the specific directory of the installation. In addition, in the following experimental procedures and examples, we mainly introduce the use of Bochs on the Windows platform.

17.2.3 Running the Linux 0.1x System

Running a Linux 0.1x system in Bochs is very simple. Once the Bochs software is properly installed, you can simply double-click on the appropriate Bochs configuration file (`*.bxrc`) to get started. The PC environment for runtime simulation has been set up in each configuration file. You can modify these files with any text editor. To run a Linux 0.12 system, the corresponding configuration files usually only need to include the following lines of information:

```
romimage: file=$BXXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXXSHARE/VGABIOS-lgpl-latest
megs: 16
floppya: 1_44="bootimage-0.12-hd", status=inserted
ata0-master: type=disk, path="rootimage-0.12-hd", mode=flat, cylinders=487, heads=16, spt=63
boot: a
```

The first two lines indicate the ROM BIOS and VGA display card ROM program of the simulated PC, and generally do not need to be modified. Line 3 indicates the physical memory capacity of the PC, which is set to 16MB. Because the default Linux 0.12 kernel only supports up to 16MB of memory, the big settings don't work either. The parameter `floppya` specifies that the floppy disk drive A of the simulated PC uses the 1.44MB disk type, and has been set to use the `bootimage-0.12-hd` floppy image file, and is in the inserted state. The corresponding `floppyb` can be used to indicate the floppy image file used or inserted in the B drive. The parameter `ata-master` is used to specify the virtual hard disk capacity and hard disk parameters attached to the simulated PC. For the specific meaning of these hard disk parameters, please refer to the previous description. In addition, `ata0-slave` can be used to specify the image file and parameters used by the second virtual hard disk. The last 'boot' is used to specify the boot drive that can be set to boot from the A drive or from the C drive (hard drive). Here we set it to boot from the A drive (a).

1. Run the Linux 0.12 system using the `bochsrc-0.12-fd.bxrc` file.

That is, boot the Linux 0.12 system from the floppy disk and use the root file system in the current drive. This way of running a Linux 0.12 system uses only two floppy disks: `bootimage-0.12-hd` and `rootimage-0.12-hd`. The contents of the several lines of configuration files listed above are the basic settings in `bochsrc-0.12-fd.bxrc`, just the boot image file is replaced by `bootimage-0.12-hd`. When you double-click this configuration file to run the Linux 0.12 system, a message will appear in the main window of Bochs display, as shown in Figure 17-1.

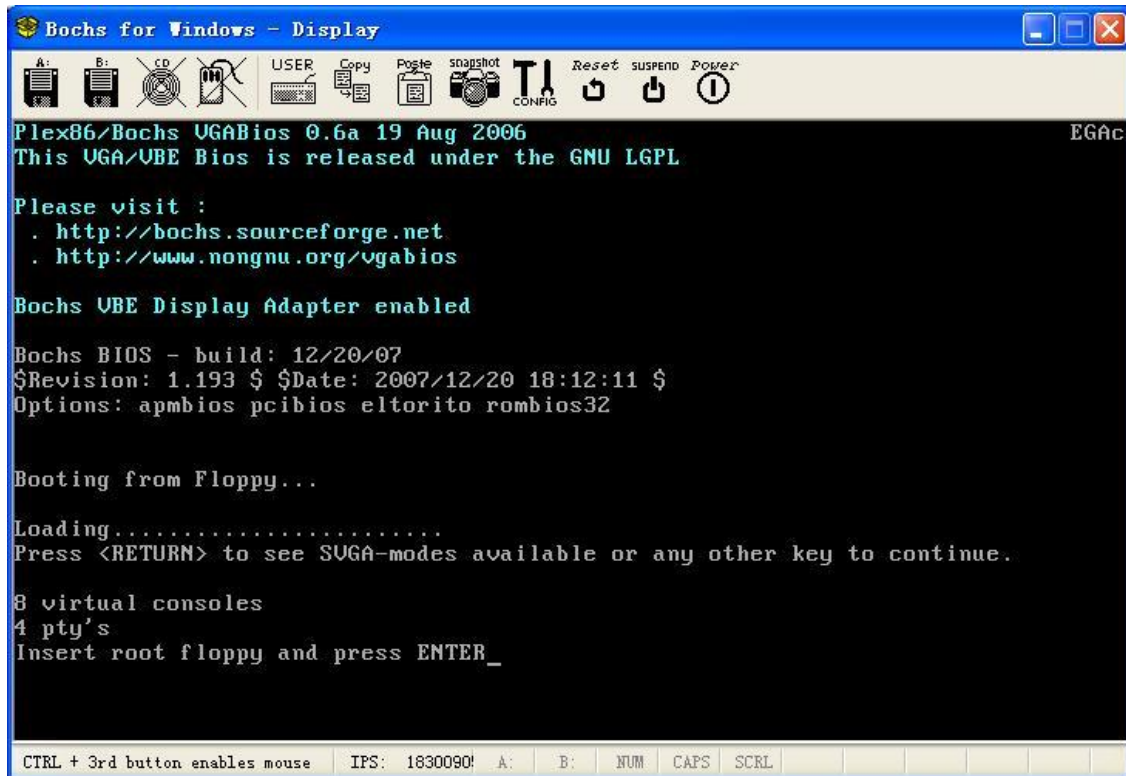


Figure 17-1 Boot from a floppy disk and use the root fs in a floppy disk

Since bochsrc-0.12-fd.bxrc configures the Linux 0.12 runtime to boot from drive A, and the kernel image file bootimage-0.12-fd will require the root file system to be in the drive currently being booted (disk A), so the kernel will display a message asking us to "remove" the kernel boot image file bootimage-0.12-fd and "insert in" the root file system. At this point we can use the A disk icon at the top left of the window to "replace" the A disk. Click this icon and change the original image file name (bootimage-0.12-fd) to rootimage-0.12-fd, so that we have completed the floppy disk replacement operation. After clicking the "OK" button to close the dialog window, press the Enter key to let the kernel load the root file system on the floppy disk, and finally the command prompt line appears, as shown in Figure 17-2.

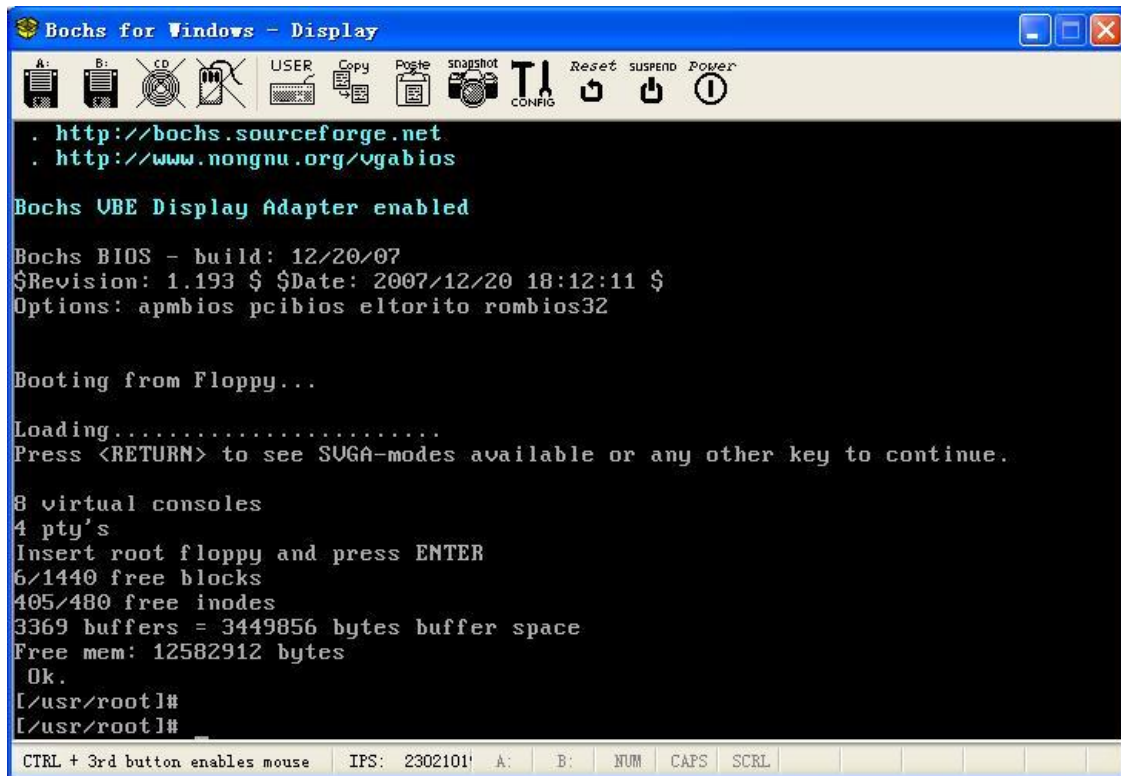


Figure 17-2 "Replace" the floppy disk and press Enter to continue running

2. Run the Linux 0.12 system using the bochsrc-0.12-hd.bxrc file

The configuration file will load the Linux 0.12 kernel image file bootimage-0.12-hd from the boot floppy disk (A disk) and use the root file system in the first partition of the hard disk image file rootimage-0.12-hd. Because the 509th and 510th bytes in the bootimage-0.12-hd file have been set to the device number 0x0301 of the first partition of the C drive (ie 0x01, 0x03), the kernel will automatically start loading the root file system from the first partition of the virtual C drive when the kernel is initialized. At this point, double-click the bochsrc-0.12-hd.bxrc file name to run the Linux 0.12 system directly, and you will get a screen similar to Figure 17-2.

17.3 Access Information in a Disk Image File

Bochs uses disk image files to emulate external storage devices in the simulated system. All files in the simulated operating system are saved in the image file in the format of a floppy disk or hard disk device. This brings about the problem of exchanging information between the host operating system and the simulated system in Bochs. Although the Bochs system can be configured to run directly using physical devices such as a host's floppy disk drive, CDROM drive, etc., it is cumbersome to utilize such an information exchange method. Therefore, it is best to directly read and write the information in the Image file. If you need to add a file to the simulated operating system, save the file in the Image file; if you want to get the file, read it from the Image file. However, since the information stored in the Image file is not only stored in the corresponding floppy disk or hard disk format, but also stored in a certain file system format. Therefore, the program that accesses the Image file must be able to recognize the file system in it to operate. For the purposes of this chapter, we need some tools to identify the MINIX and/or DOS file system formats in the Image file.

In general, if the file size exchanged with the simulated system is small, we can use the floppy Image file

as the exchange medium. If there are large batches of files that need to get from the simulated system or put into the simulated system, then we can use the existing Linux system to mount the image file. Below we discuss several methods that can be used from these two aspects.

- Use the disk image tool to access information (small files or split files) in the floppy image file;
- Use the loop device to access the hard disk image file in Linux. (a large amount exchange);
- Use iso format file for information exchange (a large amount exchange).

17.3.1 Using WinImage Software

By using a floppy Image file, we can exchange a small amount of files with the simulated system. The prerequisite is that the simulated system supports reading and writing DOS format floppy disks, for example, by using the mtools software. mtools is a program for accessing files in the MSDOS file system in a UNIX-like system. The software implements common MSDOS commands such as copy, dir, cd, format, del, md, and rd. Adding the letter m to the name of these commands is the corresponding command in mtools. The specific operation method will be described below by way of example.

Before reading/writing a file, you first need to prepare a 1.44MB Image file (the file name is assumed to be diskb.img) according to the method described above, and modify the bochs.bxrc configuration file of Linux 0.12. Add the following line to the floppy parameter:

```
floppyb: 1_44="diskb.img", status=inserted
```

That is, the second 1.44MB floppy disk device is added to the simulated system, and the image file name used by the device is diskb.img.

If you want to take a file (hello.c) from the Linux 0.12 system, you can now start the Linux 0.12 system by double-clicking the configuration file icon. After entering the Linux 0.12 system, use the DOS floppy disk read and write tool mtools to write the hello.c file to the second floppy image. If the floppy Image was created using Bochs or has not been formatted, you can use the mformat b: command to format it first.

```
[/usr/root]# mcopy hello.c b:
Copying HELLO.C
[/usr/root]# mdir b:
Volume in drive B has no label
Directory for B:/

HELLO    C           74    4-30-104   4:47p
        1 File(s)    1457152 bytes free
[/usr/root]# _
```

Now exit the Bochs system and open the diskb.img file with WinImage. There will be a hello.c file in the WinImage main window. Use the mouse to select the file and drag it to the desktop, which completes the entire operation of fetching the file. If you need to put a file into the simulated system, the steps are exactly the opposite. Also note that WinImage can only access and manipulate disk files with DOS format, it can not access disk files in other formats such as MINIX file system.

17.3.2 Using an Existing Linux System

Existing Linux systems (such as Redhat) have access to a variety of file systems, including the use of loop devices to access file systems stored in image files. For the floppy Image file, we can use the mount command to load the file system in the Image for read/write access. For example, we need to access the file in rootimage-0.12, then just execute the following command.

```
[root@plinux images]# mount -t minix rootimage-0.12 /mnt -o loop
[root@plinux images]# cd /mnt
[root@plinux mnt]# ls
bin dev etc root tmp usr
[root@plinux mnt]# _
```

The "-t minix" option of the mount command indicates that the file system type being read is MINIX, and the "-o loop" option indicates that the file system is loaded by the loop device. If you need to access the DOS format floppy image file, just replace the file type option "minix" in the mount command with "msdos".

If you want to access the hard disk Image file, the operation process is different from the above. Since the floppy Image file generally contains an image of a complete file system, the file system in the floppy disk image can be directly loaded using the mount command, but the hard disk Image file usually contains partition information, and the file system is created in each partition. So we need to load the required partition first, and then we can treat the partition as a complete "big" floppy disk.

Therefore, in order to access the information in a partition of a hard disk Image file, we need to first understand the partition information in the image file to determine the starting sector offset position of the partition to be accessed in the Image file. For the partition information in the hard disk Image file, we can use the fdisk command to view it in the simulation system, or use the method described here. Here, the image file rootimage-0.12-hd.img included in the following package is taken as an example to illustrate the method of accessing the file system in the first partition.

<http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip>

Here you need to use the loop device settings and control commands losetup. This command is mainly used to associate a normal file or a block device with a loop device, or to release a loop device and query the status of a loop device. For a detailed description of this command, please refer to the Linux online manual page.

First execute the following command to associate the rootimage-0.12-hd file with loop1, and use the fdisk command to view the partition information.

```
[root@www linux-0.12-080324]# losetup /dev/loop1 rootimage-0.12-hd
[root@www linux-0.12-080324]# fdisk /dev/loop1
Command (m for help): x
Expert command (m for help): p
Disk /dev/loop1: 16 heads, 63 sectors, 487 cylinders
```

Nr	AF	Hd	Sec	Cyl	Hd	Sec	Cyl	Start	Size	ID
----	----	----	-----	-----	----	-----	-----	-------	------	----

```
1 80 1 1 0 15 63 130 1 132047 81
2 00 0 1 131 15 63 261 132048 132048 81
3 00 0 1 262 15 63 392 264096 132048 81
4 00 0 1 393 15 63 474 396144 82656 82
Expert command (m for help): q
```

```
[root@www linux-0.12-080324]# _
```

As can be seen from the partition information given by fdisk above, the Image file contains 3 MINIX partitions (ID=81) and 1 swap partition (ID=82). If we need to access the contents of the first partition, write down the starting sector number of the partition (that is, the contents of the 'Start' column in the partition table). If you need to access the hard disk Image of another partition, then you just need to remember the starting sector number of the relevant partition.

Next, we first use the "-d" option of losetup to unlink the rootimage-0.12-hd file from loop1 and then re-associate it to the beginning of the first partition of the image file. This requires the use of the "-o" option of losetup, which indicates the associated starting byte offset position. As can be seen from the above partition information, the starting offset position of the first partition here is 1 * 512 bytes. Therefore, after re-associating the first partition with loop1, we can use the mount command to access the files.

```
[root@www linux-0.12-080324]# losetup -d /dev/loop1
[root@www linux-0.12-080324]# losetup -o 512 /dev/loop1 rootimage-0.12-hd
[root@www linux-0.12-080324]# mount -t minix /dev/loop1 /mnt
[root@www linux-0.12-080324]# cd /mnt
[root@www mnt]# ls
bin  etc  home  MCC-0.12  mnt1  root  usr
dev  hdd  image  mnt      README  tmp   vmlinux
[root@www mnt]# _
```

After the access to the file system in the partition is over, finally unmount the file system and disassociate.

```
[root@www mnt]# cd
[root@www ~]# umount /dev/loop1
[root@www ~]# losetup -d /dev/loop1
[root@www ~]# _
```

17.4 Compiling and running the simple kernel

A simple multitasking kernel sample program is given in the previous chapter on the 80386 protection mode and its programming in Chapter 4, which we call the Linux 0.00 system. It contains two tasks running on privilege level 3, which will cycle through the characters A and B on the screen and perform task switching operations under clock timing control. The packages that have been configured to run in the Bochs simulation environment are given on the book's website:

<http://oldlinux.org/Linux.old/bochs/linux-0.00-050613.zip>

<http://oldlinux.org/Linux.old/bochs/linux-0.00-041217.zip>

We can download any of the above files to experiment. The program given in the first package is the same as described here. The program in the second package is slightly different (the kernel head code runs directly at the 0x10000 address), but the principle is exactly the same. Here we will use the program in the first software package as an example to illustrate the experiment. The software of the second package is left to the reader for experimental analysis. After unpacking the linux-0.00-050613.zip package with a decompression software, a linux-0.00 subdirectory will be generated in the current directory. We can see that this package contains the following files:

-
- | | | |
|----|-----------------------|---|
| 1. | linux-0.00.tar.gz | - Compressed source file; |
| 2. | linux-0.00-rh9.tar.gz | - Compressed source file; |
| 3. | Image | - Kernel boot image file; |
| 4. | bochsrc-0.00.bxrc | - Bochs configuration file; |
| 5. | rawrite.exe | - The program to write an Image to a floppy disk under Windows. |
| 6. | README | - Package documentation; |
-

The first file, linux-0.00.tar.gz, is a compressed source file for the kernel example. It can be compiled in the Linux 0.12 system to generate a kernel image file. The second is also a compressed source file for the kernel example, but the source program can be compiled under RedHat 9 Linux. The third file Image is a 1.44MB floppy image file of the runnable code compiled by the source program. The fourth file, bochsrc-0.00.bxrc, is the Bochs configuration file used when running in the Bochs environment. If you have installed the PC emulation software Bochs on your system, you can run the kernel code in the Image by double-clicking on the bochsrc-0.00.bxrc filename. The fifth is a utility program under DOS or Windows for writing image files to a floppy disk. We can run the RAWRITE.EXE program directly and write the kernel image file here to a 1.44MB floppy disk to run.

The source code for the kernel example given above is included in the linux-0.00.tar.gz file. Decompressing this file will generate a subdirectory containing the source files, including a makefile in addition to the boot.s and head.s programs. The beginning part of the 'boot' file generated by as86/ld86 compilation and linking contains 32 bytes of MINIX executable file header information, and the beginning part of the 'head' file compiled and linked by as/ld includes 1024 bytes of a.out header data, so when making the kernel 'Image' file, we need to remove the header information. We can use two 'dd' commands to remove the header information and then combine them into a kernel image Image file.

The Image file is generated by executing the make command directly in the source code directory. If you have already executed the make command, then execute 'make clean' first and then execute the make command.

```
[/usr/root/linux-0.00]# ls -l
total 9
-rw-----  1 root    root      487 Jun 12 19:25 Makefile
-rw-----  1 root    root     1557 Jun 12 18:55 boot.s
-rw-----  1 root    root     5243 Jun 12 19:01 head.s

[/usr/root/linux-0.00]# make
```

```
as86 -O -a -o boot.o boot.s
ld86 -O -s -o boot boot.o
gas -o head.o head.s
gld -s -x -M head.o -o system > System.map

dd bs=32 if=boot of=Image skip=1
16+0 records in
16+0 records out

dd bs=512 if=system of=Image skip=2 seek=1
16+0 records in
16+0 records out
[/usr/root/linux-0.0]#
```

To copy the Image file to the A disk image file, we can execute the command 'make disk' as follows. However, before executing this command, if you are compiling the Linux 0.12 system under Bochs, please copy and save your boot image disk file (for example, bootimage-0.12-hd), so that you can restore the Linux 0.12 system after the test. Image file.

```
[/usr/root/linux-0.0]# ls
Image      System.map  boot.o      head.o      system
Makefile   boot        boot.s      head.s
[/usr/root/linux-0.0]# make disk
dd bs=8192 if=Image of=/dev/fd0
1+1 records in
1+1 records out
sync;sync;sync
[/usr/root/linux-0.0]#
```

To run this simple kernel example, we can use the mouse to directly click the RESET icon on the Bochs window. Its operation is shown in the figure below. After that, if you want to resume running the Linux 0.12 system, please overwrite the startup file with the image file you just saved.

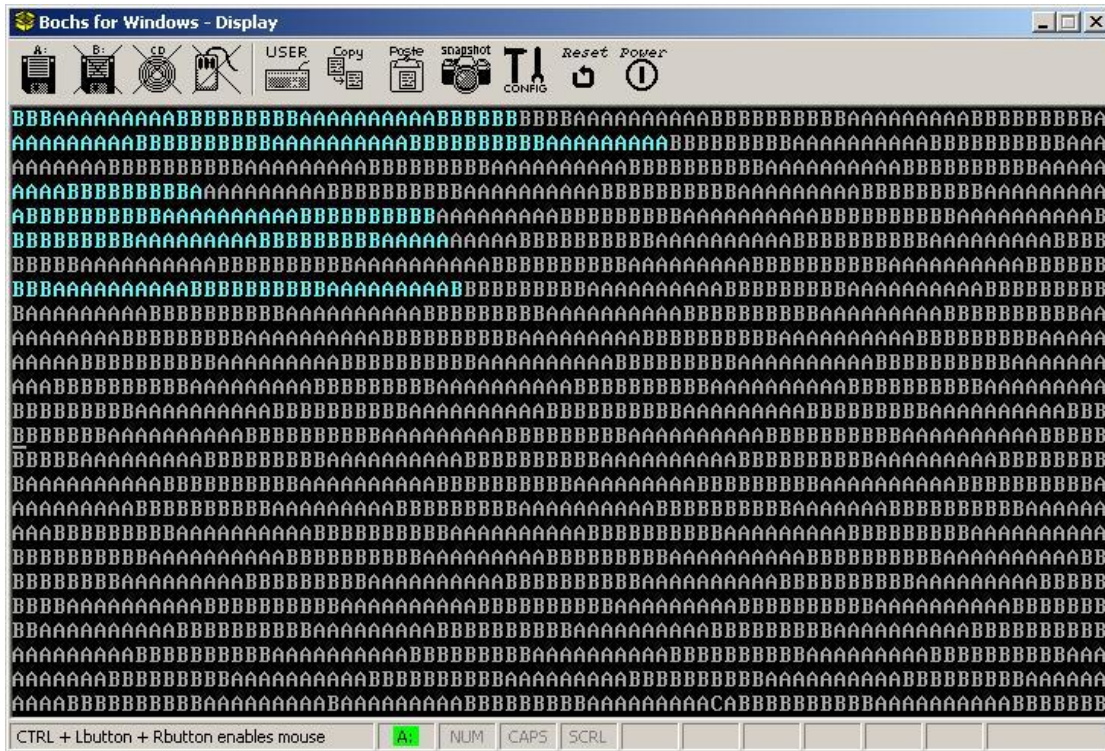


Figure 17-3 Screen display of simple kernel runtime

17.5 Using Bochs to Debug the Kernel

Bochs has very powerful operating system kernel debugging capabilities, which is one of the main reasons why Bochs was chosen as our preferred experimental environment. For a description of Bochs debugging features, see Section 17.2 above, which is based on the Linux 0.12 kernel to illustrate the basic methods of Bochs debugging operations in the Windows environment.

17.5.1 Running the Bochs Debugger

Assume that the Bochs system has been installed in the directory "C:\Program Files\Bochs-2.3.6\", and the Bochs configuration file name for the Linux 0.12 system is "bochsrc-0.12-hd.bxrc". Now we create a simple batch file run.bat in the directory containing the kernel Image file, which reads as follows:

```
"C:\Program Files\Bochs-2.3.6\bochsdbg" -q -f bochsrc-0.12-hd.bxrc
```

Among them, bochsdbg is the debugging execution program of Bochs. The parameter "-q" means quick start (skip configuration interface), and the parameter "-f" means followed by a configuration file name. If the parameter is "-h", the program will display help information for all optional parameters. Run the batch command to enter the debugging environment. At this point, the main display window of Bochs is blank, and the control window will display the following similar content:

```
C:\Linux-0.12>"C:\Program Files\Bochs-2.3.6\bochsdbg" -q -f bochsrc-0.12-hd.bxrc
000000000000i[APIC?] local apic in  initializing
=====
                        Bochs x86 Emulator 2.3.6
                        Build from CVS snapshot on December 24, 2007
=====
000000000000i[      ] reading configuration from bochsrc-hd-new.bxrc
000000000000i[      ] installing win32 module as the Bochs GUI
000000000000i[      ] using log file bochsout.txt
Next at t=0
(0) [0xffffffff] f000:fff0 (unk. ctxt): jmp far f000:e05b          ; ea5be000f0
<bochs:1>
```

At this point the Bochs debug system is ready to start running and the CPU execution pointer has been pointed to the instruction at address 0x000ffff0 in the ROM BIOS. Where "<bochs:1>" is the command line input prompt, where the number indicates the current command serial number. By typing the "help" command after the command prompt, we can list the basic commands for debugging the system. To find out how to use a command in detail, type the 'help' command followed by a specific command enclosed in single quotes, for example: " help 'vbreak' ". See below.

```
<bochs:1> help
help - show list of debugger commands
help 'command' - show short command description
-* Debugger control -*-
    help, q|quit|exit, set, instrument, show, trace-on, trace-off,
    record, playback, load-symbols, slist
-* Execution control -*-
    c|cont, s|step|stepi, p|n|next, modebp
-* Breakpoint management -*-
    v|vbreak, lb|lbreak, pb|pbreak|b|break, sb, sba, blist,
    bpe, bpd, d|del|delete
-* CPU and memory contents -*-
    x, xp, u|disas|disassemble, r|reg|registers, setpmem, crc, info, dump_cpu,
    set_cpu, ptime, print-stack, watch, unwatch, ?|calc
<bochs:2> help 'vbreak'
help vbreak
vbreak seg:off - set a virtual address instruction breakpoint
<bochs:3>
```

Some of the more commonly used commands are listed below. A complete list of all debug commands can be found in Bochs' own help file (internal-debugger.html) or refer to the online help information ("help" command).

1. Execute control commands. Control single or multi-step execution of instructions.

c Continuous execution

stepi [count]	Execute 'count' instructions, the default is 1.
si [count]	lbid.
step [count]	lbid.
s [count]	lbid.
p	Similar to 's', but with interrupt and function instructions as single-step execution, that is, single-step execution of the interrupt or functions.
n (or next)	lbid.
Ctrl-C	Stop execution and go back to the command line prompt.
Ctrl-D	Exit Bochs if you type the command at an empty command line prompt.
quit, q	Exit debugging.

2. Breakpoint setting commands. Where 'seg', 'off' and 'addr' can be hexadecimal numbers starting with '0x', or decimal numbers or octal numbers starting with '0'.

vb	seg:off	Set the instruction breakpoint on the virtual address.
vb	seg:off	lbid.
lb	addr	Set the instruction breakpoint on the linear address.
lb	addr	lbid.
pb	[*] addr	Set the instruction breakpoint on the physical address. Where '*' is an option for compatibility with GDB.
pb	[*] addr	lbid.
break	[*] addr	lbid.
b	[*] addr	lbid.
info break		Displays the status of all current breakpoints.
delete	n	Delete a breakpoint.
del	n	lbid.
d	n	lbid.

3. Memory operation commands

x	/nuf addr	Check the memory contents at the linear address 'addr'. If 'addr' is not specified, the default is the next address.
xp	/nuf addr	Check the memory contents at the physical address 'addr'.
The optional parameters 'n', 'u' and 'f' can be:		
n		The number of memory units to display. The default value is 1.
u		Indicates the unit size, the default selection is the following 'w': b(Bytes), 1 byte; h(Halfwords), 2 bytes; w(Words), 4 bytes; g(Giantwords), 8 bytes. Note: These abbreviations are different from those of Intel, mainly to be consistent with the GDB debugger representation.
f		Display format, the default selection is 'x': x (hex); d (decimal); u (unsigned); o (octal); t (binary); c (char) Display chars. If it cannot be displayed as a char, the code is displayed directly.
crc	addr1 addr2	Displays the CRC checksum of physical memory from addr1 to addr2.
info dirty		Displays the physical memory page that has been modified since the last time this command was executed. Only the first 20 bytes of the page are displayed.

4. Information display and CPU register operation commands

info program	Displays the execution status of the program.
--------------	---

info registers Displays the CPU registers (no floating point registers).
info break Displays the current breakpoint setting status.
set \$reg = val Modify the contents of the CPU register (except segment and flag register).
For example, set \$eax = 0x01234567; set \$edx = 25
dump_cpu Displays all status information of the CPU.
set_cpu Set all status information of the CPU.

The content format displayed by the "dump_cpu" and "set_cpu" commands is:

```
"eax:0x%x\n"  
"ebx:0x%x\n"  
"ecx:0x%x\n"  
"edx:0x%x\n"  
"ebp:0x%x\n"  
"esi:0x%x\n"  
"edi:0x%x\n"  
"esp:0x%x\n"  
"eflags:0x%x\n"  
"eip:0x%x\n"  
"cs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"ss:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"ds:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"es:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"fs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"gs:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"ldtr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"tr:s=0x%x, dl=0x%x, dh=0x%x, valid=%u\n"  
"gdtr:base=0x%x, limit=0x%x\n"  
"idtr:base=0x%x, limit=0x%x\n"  
"dr0:0x%x\n"  
"dr1:0x%x\n"  
"dr2:0x%x\n"  
"dr3:0x%x\n"  
"dr4:0x%x\n"  
"dr5:0x%x\n"  
"dr6:0x%x\n"  
"dr7:0x%x\n"  
"tr3:0x%x\n"  
"tr4:0x%x\n"  
"tr5:0x%x\n"  
"tr6:0x%x\n"  
"tr7:0x%x\n"  
"cr0:0x%x\n"  
"cr1:0x%x\n"  
"cr2:0x%x\n"  
"cr3:0x%x\n"  
"cr4:0x%x\n"  
"inhibit_int:%u\n"  
"done\n"
```

among them:

- 's' means a selector;
- 'dl' is the low 4-byte of the segment descriptor in the selector shadow register;
- 'dh' is the high 4-byte of the segment descriptor in the selector shadow register;
- 'valid' indicates whether a valid shadow descriptor is being stored in the segment register;
- 'inhibit_int' is an instruction delay interrupt flag. If set, it means that the instruction that has just been executed by the previous one is an instruction that delays the CPU from accepting the interrupt (for example, STI, MOV SS);

In addition, when the "set_cpu" command is executed, any error will be reported using the format "Error: ...". These error messages may appear after each input line or after the last "done" is displayed. If the "set_cpu" command is successfully executed, the command will display "OK" to end the command.

5. Disassembly command

```
disassemble start end    Disassemble instructions within a given linear address range.
disas
u
```

The following are some of the new commands from Bochs, but the commands involving file names in the Windows environment may not work properly.

- `record filename` -- Write your input command sequence to the file 'filename' during execution. The file will contain lines of the form "%s %d %x". The first parameter is the event type; the second is the timestamp; the third is the data of the related event.
- `playback filename` -- The execution command is played back using the contents of the file 'filename'. You can also type other commands directly in the control window. Each event in the file will be played back, and the playback time will be counted relative to the time the command was executed.
- `print-stack [num words]` -- Display num 16-bit words at the top of the stack. The default value of num is 16. When the base address of the stack segment is 0, the command can be used normally only in protected mode.
- `load-symbols [global] filename [offset]` -- Load symbol information from the file 'filename'. If the keyword global is given, then all symbols will be visible in the context before the symbol was loaded. The 'offset' (default is 0) is added to each symbol item. The symbol information is loaded into the context of the currently executing code. The format of each line in the symbol file filename is "%x %s". The first value is the address and the second is the symbol name.

In order for Bochs to directly simulate execution to the beginning of the Linux bootloader, we can first set a breakpoint at 0x7c00 using the breakpoint command and then let the system continue to run to 0x7c00 to stop. The sequence of commands executed is as follows:

```
<bochs:3> vbreak 0x0000:0x7c00
<bochs:4> c
(0) Breakpoint 1, 0x7c00 (0x0:0x7c00)
Next at t=4409138
(0) [0x00007c00] 0000:7c00 (unk. ctxt): mov ax, 0x7c0          ; b8c007
<bochs:5>
```

At this point, the CPU executes the first instruction at the beginning of the boot.s program, and the Bochs main window will display some information such as "Boot From floppy...". Now we can follow the debugger by stepping through the command 's' or 'n' (not tracking into the subroutine). Bochs' breakpoint setting commands, disassembly commands, information display commands, etc. can be used to assist in our debugging operations. Here are some examples of common commands:

```
<bochs:8> u /10                                # Disassemble 10 instructions.
00007c00: (                ): mov ax, 0x7c0                ; b8c007
00007c03: (                ): mov ds, ax                  ; 8ed8
00007c05: (                ): mov ax, 0x9000              ; b80090
00007c08: (                ): mov es, ax                  ; 8ec0
00007c0a: (                ): mov cx, 0x100               ; b90001
00007c0d: (                ): sub si, si                  ; 29f6
00007c0f: (                ): sub di, di                  ; 29ff
00007c11: (                ): rep movs word ptr [di], word ptr [si] ; f3a5
00007c13: (                ): jmp 9000:0018               ; ea18000090
00007c18: (                ): mov ax, cs                  ; 8cc8
<bochs:9> info r                                # View the contents of the current register
eax          0xaa55          43605
ecx          0x110001        1114113
edx          0x0             0
ebx          0x0             0
esp          0xffffe         0xffffe
ebp          0x0             0x0
esi          0x0             0
edi          0xffe4          65508
eip          0x7c00          0x7c00
eflags       0x282           642
cs           0x0             0
ss           0x0             0
ds           0x0             0
es           0x0             0
fs           0x0             0
gs           0x0             0
<bochs:10> print-stack                          # Display the current stack
  0000ffff [0000ffff] 0000
  00010000 [00010000] 0000
  00010002 [00010002] 0000
  00010004 [00010004] 0000
  00010006 [00010006] 0000
  00010008 [00010008] 0000
  0001000a [0001000a] 0000
...
<bochs:11> dump_cpu                             # Displays all registers in the CPU.
eax:0xaa55
ebx:0x0
ecx:0x110001
edx:0x0
ebp:0x0
esi:0x0
```

```
edi:0xffe4
esp:0xffff
eflags:0x282
eip:0x7c00
cs:s=0x0, dl=0xffff, dh=0x9b00, valid=1      # s-selector;dl,dh - low & high 4-byte of desc.
ss:s=0x0, dl=0xffff, dh=0x9300, valid=7
ds:s=0x0, dl=0xffff, dh=0x9300, valid=1
es:s=0x0, dl=0xffff, dh=0x9300, valid=1
fs:s=0x0, dl=0xffff, dh=0x9300, valid=1
gs:s=0x0, dl=0xffff, dh=0x9300, valid=1
ldtr:s=0x0, dl=0x0, dh=0x0, valid=0
tr:s=0x0, dl=0x0, dh=0x0, valid=0
gdtr:base=0x0, limit=0x0
idtr:base=0x0, limit=0x3ff
dr0:0x0
dr1:0x0
dr2:0x0
dr3:0x0
dr6:0xffff0ff0
dr7:0x400
tr3:0x0
tr4:0x0
tr5:0x0
tr6:0x0
tr7:0x0
cr0:0x60000010
cr1:0x0
cr2:0x0
cr3:0x0
cr4:0x0
inhibit_mask:0
done
<bochs:12>
```

Since the 32-bit code of the Linux 0.1X kernel is stored from the absolute physical address 0, if you want to execute directly to the beginning of the 32-bit code (that is, at the beginning of the `head.s` program), we can set a breakpoint at linear address 0x0000, and run the command 'c' to execute to that location.

In addition, when you press the Enter key directly at the command prompt, the previous command will be executed repeatedly; pressing the up arrow will display the previous command. Please refer to the 'help' command for other commands.

17.5.2 Locating Variables or Data Structures in the Kernel

A 'system.map' file is generated when the kernel is compiled. This file lists the global variables in the kernel Image (bootimage) file and the offset address locations of the local variables in each module. After the kernel is compiled, you can use the file export method described above to extract the 'system.map' file to the host environment (windows). See Chapter 3 for the detailed purpose and role of the 'system.map' file. Some of the contents of the 'system.map' sample file are shown below. Using this file, we can quickly locate a variable or jump to the specified function code in the Bochs debug system.

```
...
Global symbols:

_dup: 0x16e2c
_nmi: 0x8e08
_bmap: 0xc364
_iput: 0xc3b4
_blk_dev_init: 0x10ed0
_open: 0x16dbc
_do_execve: 0xe3d4
_con_init: 0x15ccc
_put_super: 0xd394
_sys_setgid: 0x9b54
_sys_umask: 0x9f54
_con_write: 0x14f64
_show_task: 0x6a54
_buffer_init: 0xd1ec
_sys_settimeofday: 0x9f4c
_sys_getgroups: 0x9edc
...
```

Similarly, since the 32-bit code of the Linux 0.1X kernel is stored from the absolute physical address 0, the offset position of the global variable in 'system.map' is the linear address position in the CPU. So we can set breakpoints directly at the variable or function name location of interest and let the program execute continuously to the specified location. For example, if we want to debug the function `buffer_init()`, we can see that it is located at `0xd1ec` from the 'system.map' file. At this point we can set a linear address breakpoint there and execute the command 'c' to let the CPU execute to the beginning of the specified function, as shown below.

```
<bochs:12> lb 0xd1ec                                # Set a linear address breakpoint.
<bochs:13> c                                          # Continuous execution.
(0) Breakpoint 2, 0xd1ec in ?? ()
Next at t=16689666
(0) [0x0000d1ec] 0008:0000d1ec (unk. ctxt): push ebx ; 53
<bochs:14> n                                          # Next instruction.
Next at t=16689667
(0) [0x0000d1ed] 0008:0000d1ed (unk. ctxt): mov eax, dword ptr ss:[esp+0x8] ; 8b442408
<bochs:15> n                                          # Next instruction.
Next at t=16689668
(0) [0x0000d1f1] 0008:0000d1f1 (unk. ctxt): mov edx, dword ptr [ds:0x19958] ; 8b1558990100
<bochs:16>
```

Program debugging is a skill that requires more practice to make it happen. Some of the basic commands described above need to be combined to give you the flexibility to observe the overall environment of kernel code execution.

17.6 Creating a Disk Image File

A disk image file is a complete image of the data on a floppy disk or hard disk and is saved as a file. The format of the information stored in the disk image file is exactly the same as the format of the information stored on the real disk. An empty disk image file is a file with the same capacity as the disk we created but with a content of all 0s. These empty image files are like new floppy disks or hard disks that have just been purchased, and they need to be partitioned or / and formatted before they can be used.

Before making a disk image file, we first need to determine the capacity of the image file we created. For floppy image files, the capacity of various specifications (1.2MB or 1.44MB) is fixed. So here is how to determine the capacity of the hard disk image file you need. The structure of a conventional hard disk consists of stacked metal discs. The upper and lower sides of each disc are used to store data, and the entire surface is divided into tracks by concentric circles, or Cylinders. A head is required on each side of each disc to read and write data on the disc. As the disc rotates, the head only needs to be moved radially to move over any track, thereby providing access to all valid positions on the surface of the disc. Each track is divided into sectors, and the sector size is generally composed of 256 - 1024 bytes. For most systems, the sector size is typically 512 bytes. A typical hard disk structure is shown in Figure 17-4.

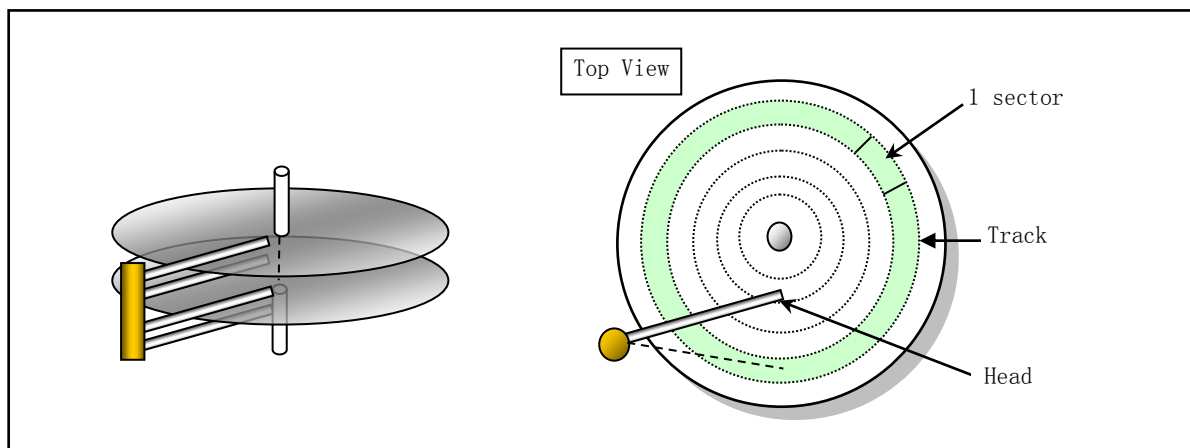


Figure 17-4 Typical hard disk internal structure

The figure shows a hard disk structure with two metal disks with four physical heads. The maximum number of cylinders contained is determined at the time of production. When the hard disk is partitioned and formatted, the magnetic media on the surface of the disk is initialized to data in a specified format such that each track (or cylinder) is divided into a specified number of sectors. Therefore the total number of sectors of this hard disk is:

$$\text{Total sectors} = \text{number of physical tracks} * \text{number of physical heads} * \text{number of sectors per track}$$

The parameters such as the track and head used in an operating system are different from those of the actual physical parameters in a hard disk, which are called logical parameters. However, the total number of sectors calculated by these parameters is definitely the same as that calculated by the physical parameters of the hard disk. Since the performance and capacity of the hardware device are not developed so rapidly when designing the PC system, some of the ROM BIOS's representation of the hard disk parameters are too small to meet the physical parameters of the actual hard disk. Therefore, the current remedy commonly used in operating

systems or machine BIOS is to properly adjust the number of tracks, the number of heads, and the number of sectors per track to ensure compatibility and parameter representation constraints while ensuring that the total number of sectors of the hard disk is constant. The "translation" option in the Bochs configuration file for hard disk device parameters is also set for this purpose.

When we made the hard disk Image file for the Linux 0.1X system, considering the small amount of its own code, and the maximum capacity of the MINIX 1.5 file system used is 64MB, the maximum size of each hard disk partition can only be 64MB. In addition, the Linux 0.1X system does not yet support extended partitions, so for a hard disk Image file, there are up to 4 partitions. Therefore, the maximum capacity of a hard disk Image file that can be used by a Linux 0.1X system is $64 \times 4 = 256\text{MB}$. In the following description, we will illustrate the creation of a hard disk Image file with 4 partitions and 60MB per partition.

For a USB flash disk, we can treat it as a hard disk. For a floppy disk, we can think of it as a non-partitioned ultra-small hard disk with a fixed number of tracks (number of cylinders), number of heads, and number of sectors per track. For example, a floppy disk with a capacity of 1.44 MB is 80 tracks, 2 heads and 18 sectors per track, and 512 bytes per sector. The total number of sectors is 2880 and the total capacity is $80 \times 2 \times 18 \times 512 = 1474560$ bytes. Therefore, all the methods described below for making hard disk image files can be used to create USB disk and floppy image files. For the convenience of the description, we refer to all disk image files as Image files unless otherwise specified.

17.6.1 Using Bochs' own Image Creation Tool

The Bochs system comes with an disk Image creation tool 'bximage.exe', which can be used to create empty Image files for floppy disks and hard disks. When you run it and the Image creation interface appears, the program will first prompt you to select the type of Image you want to create (hard disk hd or floppy disk fd). If you create a hard disk, you will also be prompted to enter the mode type of the hard disk image, usually only need to select its default value 'flat'. Then enter the image size you need to create. The program will display the corresponding hard disk parameter values: number of cylinders (number of tracks), number of heads, and number of sectors per track, and ask for the name of the image file. After the Image file is generated, the program displays a configuration message for setting the hard disk parameters in the Bochs configuration file. You can write down this information and edit it into the configuration file. Below is the process of creating a 256MB hard drive Image file.

```
=====
                                bximage
          Disk Image Creation Tool for Bochs
    $Id: bximage.c,v 1.19 2006/06/16 07:29:33 vruppert Exp $
=====

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd]

What kind of image should I create?
Please type flat, sparse or growing. [flat]

Enter the hard disk size in megabytes, between 1 and 32255
[10] 256

I will create a 'flat' hard disk image with
    cyl=520
    heads=16
```

```
sectors per track=63
total sectors=524160
total size=255.94 megabytes
```

What should I name the image?
[c.img] **hdc.img**
Writing: [] Done.
I wrote 268369920 bytes to (null).

The following line should appear in your bochsrc:
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63

Press any key to continue

If you already have a hard disk Image file with the required capacity, you can also copy the file directly to generate another Image file, and then you can process the file according to your own requirements. The process for creating a floppy Image file is similar to the above, except that you will be prompted to select a floppy disk type. Similarly, if you already have other floppy Image files, you can use the direct copy method.

17.6.2 Creating an Image File Using the dd Command on a Linux

The dd command is a command line tool on Linux systems, mainly used to copy files and convert file data formats. As explained above, the Image file just created is an empty file with all contents 0, but its capacity is consistent with the requirements. So we can first calculate the number of sectors of the Image file that requires the capacity, and then use the 'dd' command to generate the corresponding Image file.

For example, we want to create a hard disk Image file with a cylinder number of 520, a head count of 16, and a sector number of 63 per track. The total number of sectors is: $520 * 16 * 63 = 524160$, then the command is:

```
dd if=/dev/zero of=hdc.img bs=512 count=524160
```

The parameter 'if' is the name of the copied input file, '/dev/zero' is the device file that can generate 0 value bytes; 'of' is the output file name generated; 'bs' specifies the size of the copied data block; 'count' is the number of data blocks copied. For a 1.44MB floppy Image file, the number of sectors is 2880, so the command is:

```
dd if=/dev/zero of=diska.img bs=512 count=2880
```

17.6.3 Creating a floppy disk image file in DOS format using WinImage

WinImage is a DOS format Image file access and creation tool. After associating with the software, double-click the icon of the DOS floppy Image file and you can browse, delete or add files to it. In addition, it can also be used to browse the contents of the CDROM iso file. When you use WinImage to create a floppy disk Image, you can generate an Image file with DOS format. Methods as below:

- a) Run WinImage. Select the "Options->Settings" menu and select the Image Settings page. Set Compression to "None" (that is, pull the indicator to the far left);
- b) Create an Image file. Select the menu "File->New" and a floppy disk selection box will pop up.

Please choose a format with a capacity of 1.44MB;

- c) Select the boot sector property menu item "Image->Boot Sector properties" and click the MS-DOS button in the dialog box;
- d) Save the file.

Note that you must select "All files (*.*)" in the "Save Type" dialog box. Otherwise, the created Image file will contain some WinImage information, which will cause the Image file to not work properly under Bochs. We can determine if the newly created Image meets the requirements by looking at the file size. The standard 1.44MB floppy disk should have a capacity of 1474560 bytes. If the new Image file size is larger than this value, please re-create or use the binary editor such as Notepad++ (requires Hex-Editor plugin) to delete the extra bytes. The method of deleting the operation is as follows:

- Open the Image file with Notepad++ and run the plugin Hex-Editor. According to the 511, 512 bytes of the disk image file are 55, AA two hexadecimal numbers, we push back 512 bytes, delete all the previous bytes. At this point, for a disk using MSDOS 5.0 as the boot, the first few bytes of the file should be similar to "EB 3C 90 4D ...".
- Then pull down the right scroll bar and move to the end of the img file. Delete all data after "...F6 F6 F6". Usually it is to delete all data starting from 0x168000. The last line when the operation is completed should be a complete line "F6 F6 F6...". Save and exit to use the Image file.

17.7 Making a Root File System

The goal of this section is to create a root file system on the hard disk. Although the floppy disk and the hard disk root file system Image file can be downloaded from the oldlinux.org website, the creation process is described in detail here for your reference. In the process of establishing, you can also refer to the installation article written by Mr. Linus: INSTALL-0.11. Before making the root file system disk, we first download the rootimage-0.12 and bootimage-0.12 image files (please download the latest files):

<http://oldlinux.org/Linux.old/images/bootimage-0.12-20040306>

<http://oldlinux.org/Linux.old/images/rootimage-0.12-20040306>

Modify these two file names into easy-to-remember names bootimage-0.12 and rootimage-0.12, and create a subdirectory named Linux-0.12 for them. During the making process, we need to copy some of the executables in the rootimage-0.12 floppy disk and boot the simulation system using the bootimage-0.12 boot disk. So before you start working on the root file system, you first need to confirm that you have been able to run the minimum Linux system consisting of these two floppy Image files.

17.7.1 Root File System and Root File Device

When the Linux boot starts, the default file system containing the root directory is the root file system. The following subdirectories and files are generally included in the root directory:

-
- | | |
|--------|---|
| ♦ etc/ | This directory mainly contains some configuration files, such as 'rc' file; |
| ♦ dev/ | Contains device special files for operating the device with files; |

- ♦ bin/ Store system execution programs. Such as sh, mkfs, fdisk, etc. ;
- ♦ usr/ Store library functions, manuals, and other files;
- ♦ usr/bin/ Store commands commonly used by users;
- ♦ var/ Used to store data when the system is running or for information such as logs.

The device that holds the file system is the file system device. For example, for the Windows operating system, the hard disk C drive is the file system device, and the files stored on the hard disk according to certain rules constitute the file system. Windows can have file systems in formats such as NTFS or FAT32, while the file system supported by the Linux 0.1X kernel is the MINIX 1.0 file system.

When the Linux boot disk loads the root file system, the root file system is loaded from the specified device according to the root file system device number in a word (ROOT_DEV) at the 509th and 510th bytes of the boot sector on the boot disk. If the device number is 0, it means that the root file system needs to be loaded from the current drive where the boot disk is located. If the device number is a hard disk partition device number, the root file system is loaded from the specified hard disk partition. The hard disk device numbers supported by the Linux 0.1X kernel are shown in Table 17-3.

Table 17-3 Hard disk logical device number

Device nr	Device file	Description
0x0300	/dev/hd0	Represents the entire first hard driv
0x0301	/dev/hd1	The first partition of the first disk
0x0302	/dev/hd2	The second partition of the first disk
0x0303	/dev/hd3	The third partition of the first disk
0x0304	/dev/hd4	The fourth partition of the first disk
0x0305	/dev/hd5	Represents the entire second hard driv
0x0306	/dev/hd6	The first partition of the second disk
0x0307	/dev/hd7	The second partition of the second disk
0x0308	/dev/hd8	The third partition of the second disk
0x0309	/dev/hd9	The fourth partition of the second disk

If the device number is a floppy device number, the kernel will load the root file system from the floppy drive specified by the device number. The floppy device numbers used in the Linux 0.1X kernel are shown in Table 17-4. For the calculation method of the floppy disk drive device number, please refer to the description after the floppy.c program in Chapter 9.

Table 17-4 Floppy drive logic device number

Device Nr	Device file	Description
0x0208	/dev/at0	1.2MB A drive
0x0209	/dev/at1	1.2MB B drive
0x021c	/dev/fd0	1.44MB A drive
0x021d	/dev/fd1	1.44MB B drive

17.7.2 Creating a File System

For the hard disk Image file created above, we must also partition and create a file system on it before it can be used. The usual practice is to attach the hard disk Image file to be processed to the existing simulation system (such as SLS Linux mentioned above) under Bochs , and then use the commands in the simulation system to process the new Image file. The following assumes that you have installed the SLS Linux emulation

system and it is stored in a subdirectory named SLS-Linux. We use it to partition the 256MB hard disk image file hdc.img created above and create a MINIX file system on it. We will create a partition in this Image file and create a MINIX file system. The steps we performed are as follows:

1. Create a subdirectory named Linux-0.12 in the SLS-Linux directory and move the hdc.img file to it.
2. Go to the SLS-Linux directory and edit the Bochs configuration file 'bochsrc.bxrc' for the SLS Linux system. Add the configuration parameter line of our hard disk Image file under the option 'ata0-master':

```
ata0-slave:type=disk, path=..\Linux-0.12\hdc.img, cylinders=520, heads=16, spt=63
```

3. Exit the editor. Double-click the icon for the 'bochsrc.bxrc' file to run the SLS Linux emulation system. Type 'root' at the Login prompt and press Enter. If Bochs does not work properly at this time, generally because the configuration file information is incorrect, please re-edit the configuration file.
4. Use fdisk to create 1 partition in the hdc.img file. Below is the sequence of commands to create the first partition. The process of creating another three partitions is similar. Since the partition type established by SLS Linux by default is 81 type (Linux/MINIX) that supports the MINIX2.0 file system, you need to use the fdisk t command to change the type to 80 (Old MINIX) type. Please note here that we have hooked hdc.img to the second hard drive under the SLS Linux system. According to the Linux 0.1X hard disk naming rules, the overall device name of the hard disk should be /dev/hd5. However, since the Linux kernel version 0.95, the naming rules for the hard disk have been changed to the currently used rules, so the device name of the second hard disk under SLS Linux is /dev/hdb.

```
[/)# fdisk /dev/hdb
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-520): 1
Last cylinder or +size or +sizeM or +sizeK (1-520): +63M

Command (m for help): t
Partition number (1-4): 1
Hex code (type L to list codes): L
 0 Empty                8 AIX                  75 PC/IX                b8 BSDI swap
 1 DOS 12-bit FAT        9 AIX bootable         80 Old MINIX             c7 Syrix
 2 XENIX root            a OPUS                 81 Linux/MINIX           db CP/M
 3 XENIX user            40 Venix               82 Linux swap            e1 DOS access
 4 DOS 16-bit <32M      51 Novell?             83 Linux extfs           e3 DOS R/0
 5 Extended              52 Microport           93 Amoeba                f2 DOS secondary
 6 DOS 16-bit >=32      63 GNU HURD            94 Amoeba BBT            ff BBT
 7 OS/2 HPFS             64 Novell              b7 BSDI fs

Hex code (type L to list codes): 80

Command (m for help): p
Disk /dev/hdb: 16 heads, 63 sectors, 520 cylinders
Units = cylinders of 1008 * 512 bytes
   Device Boot   Begin    Start    End  Blocks   Id  System
```

```
/dev/hdb1      1      1    129   65015+  80  Old MINIX
```

```
Command (m for help):w
```

```
The partition table has been altered.
```

```
Please reboot before doing anything else.
```

```
[/]#
```

5. Remember the number of data blocks in this partition (here is 65015), which is used when creating the file system. When the partition is set up, it is necessary to restart the system as usual, so that the SLS Linux kernel can correctly identify the newly added partition.
6. After entering the SLS Linux emulation system again, we use the `mkfs` command to create the MINIX file system on the first partition we just created. The commands and information are as follows. This creates a partition with 64,000 data blocks (one block of data is 1 KB).

```
[/]# mkfs /dev/hdb1 64000
21333 inodes
64000 blocks
Firstdatazone=680 (680)
Zonesize=1024
Maxsize=268966912
[/]#
```

At this point, we have completed the work of creating a file system in the first partition of the `hdc.img` file. Of course, creating a file system can also be established when running the root file system on a Linux 0.12 floppy disk. Now we can build this partition into a root file system.

17.7.3Bochs Configuration File for Linux-0.12

When running a Linux 0.12 system in Bochs, the following configuration is usually required in the configuration file `bochsrc.bxrc`.

```
romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
megs: 16
floppya: 1_44="bootimage-0.12", status=inserted
ata0-master: type=disk, path="hdc.img", mode=flat, cylinders=520, heads=16, spt=63
boot: a
log: bochsout.txt
panic: action=ask
#error: action=report
#info: action=report
#debug: action=ignore
ips: 1000000
mouse: enabled=0
```

We can copy the Bochs configuration file `bochsrc.bxrc` from SLS Linux into the `Linux-0.12` directory and

modify it to the same content as above. Special attention should be paid to 'floppya', 'ata0-master' and 'boot'. These three parameters must be consistent with the above. Now we double click on this configuration file with the mouse. First, the screen in Figure 17-5 should appear in the Bochs display window.

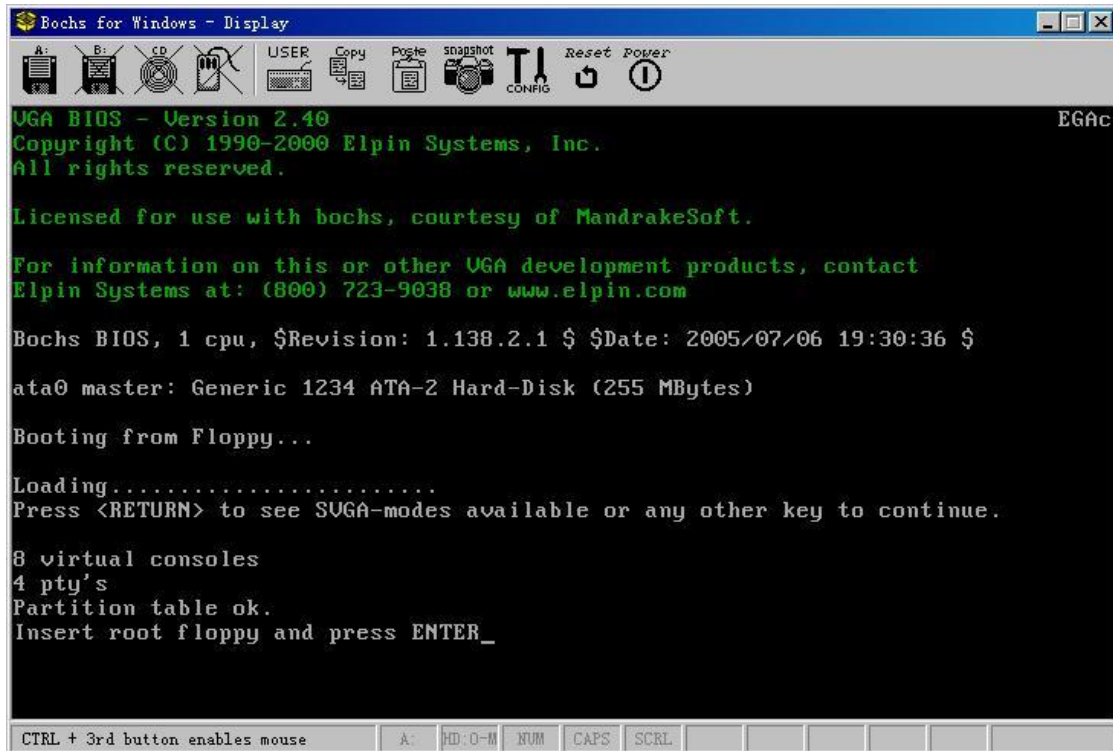


Figure 17-5 Bochs system running window

At this point, you should click the A: floppy disk icon on the window menu bar, and configure the A disk as the rootimage-0.12 file in the dialog box. Or use the Bochs configuration window to set it up by clicking the 'CONFIG' icon on the menu bar to enter the Bochs settings window (you need to click the mouse to bring the window to the front). The contents of the setting window display are shown in Figure 17-6.

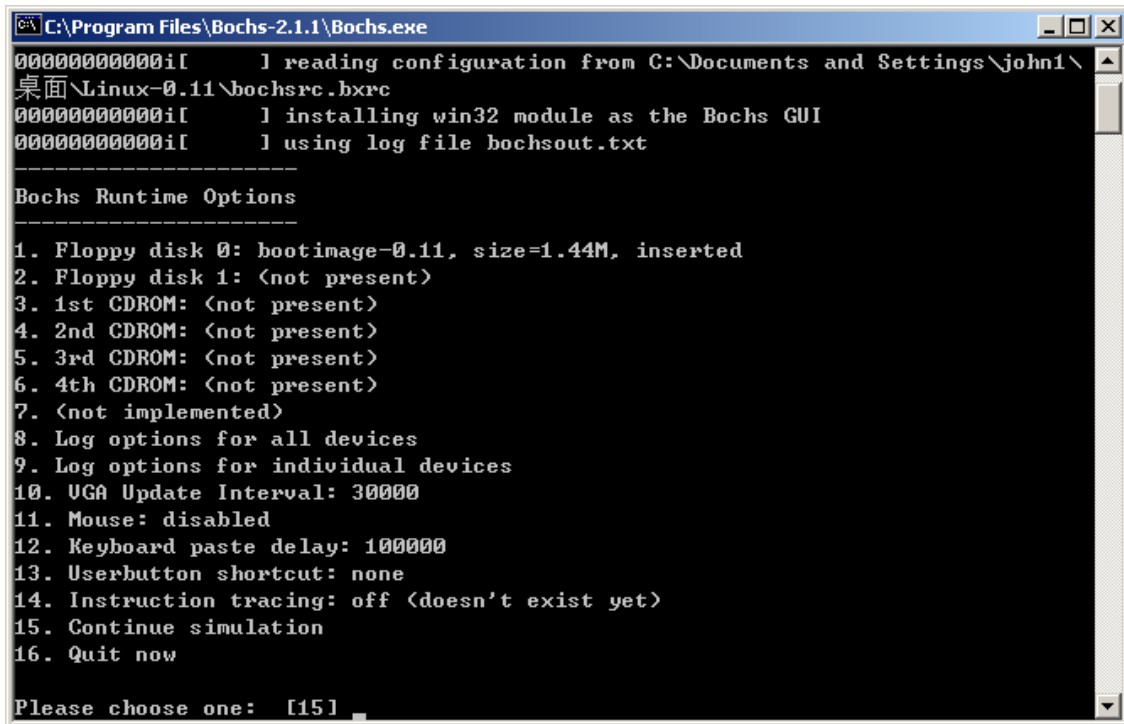


Figure 17-6 Bochs system configuration window

Modify the floppy disk setting of item 1 to point to the rootimage-0.12 disk. Then press the Enter key continuously until the last line of the setup window displays 'Continuing simulation'. At this point, switch to the Bochs Run window and click Enter to formally enter the Linux 0.12 system, as shown in Figure 17-7.

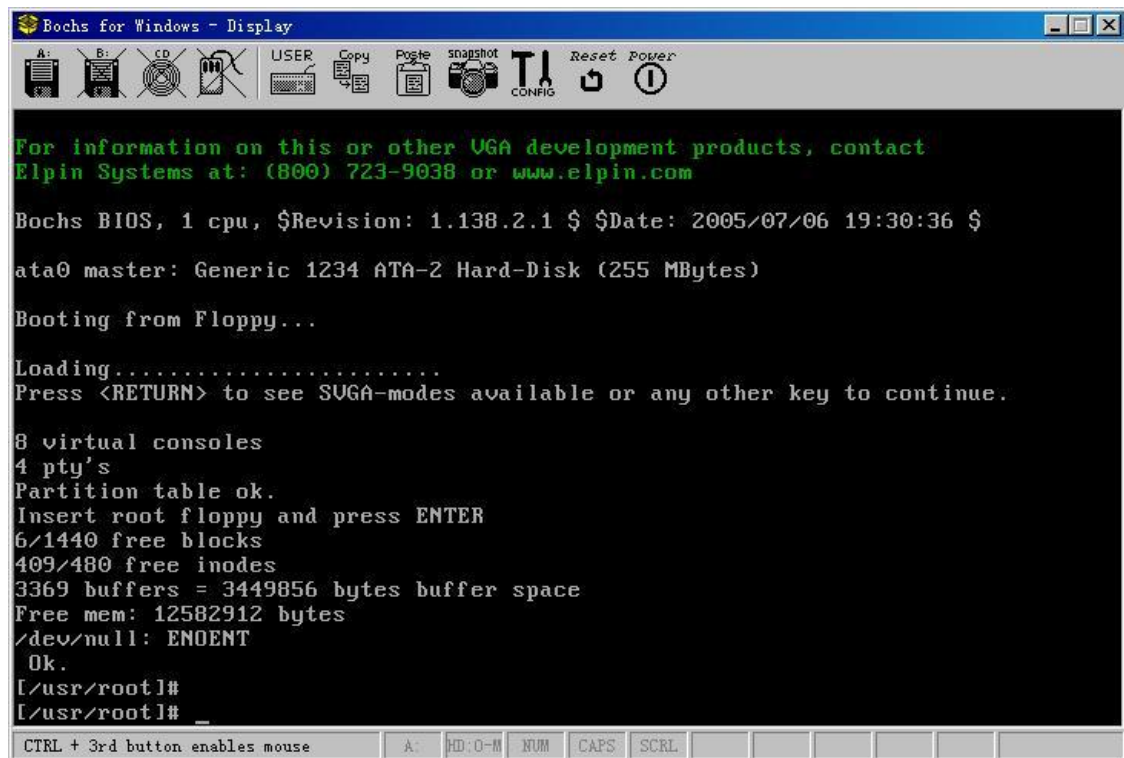


Figure 17-7 Linux 0.12 system running in Bochs

17.7.4 Establishing a Root File System in hdc.img

Since the floppy disk capacity is too small, if you want the Linux 0.12 system to really do something, you need to create a root file system on the hard disk (here, the hard disk Image file). In the previous section, we have created a 256MB hard disk image file `hdc.img`, and it is already connected to the running Bochs environment, so a message about the hard disk appears in the figure above:

```
"ata0 master: Generic 1234 ATA-2 Hard-Disk (255 Mbytes)"
```

If you do not see this message, your Linux 0.12 configuration file is not set correctly. Please re-edit the `bochsrc.bxrc` file and re-run the Bochs system until the same screen as above appears. Now, we have previously created the MINIX filesystem on the first partition of `hdc.img`. If you haven't built it yet or want to try it again, then type the command to create a 64MB file system:

```
[/usr/root]# mkfs /dev/hd1 64000
```

Now we can start loading the file system on the hard disk. Execute the following command to load the new file system into the `/mnt` directory.

```
[/usr/root]# cd /  
[/]# mount /dev/hd1 /mnt  
[/]#
```

After loading the file system on the hard disk partition, we can copy the root file system on the floppy disk to the hard disk. Please execute the following command:

```
[/]# cd /mnt  
[/mnt]# for i in bin dev etc usr tmp  
> do  
> cp +recursive +verbose /$i $i  
> done
```

At this point, all the files on the floppy root file system will be copied to the file system on the hard disk. A lot of information similar to the following will appear during the copy process:

```
/usr/bin/mv -> usr/bin/mv  
/usr/bin/rm -> usr/bin/rm  
/usr/bin/rmdir -> usr/bin/rmdir  
/usr/bin/tail -> usr/bin/tail
```

```
/usr/bin/more -> usr/bin/more
/usr/local -> usr/local
/usr/root -> usr/root
/usr/root/.bash_history -> usr/root/.bash_history
/usr/root/a.out -> usr/root/a.out
/usr/root/hello.c -> usr/root/hello.c
/tmp -> tmp
[/mnt]# _
```

Now that you have built a basic root filesystem on your hard drive. You can view it anywhere in the new file system, then unmount the hard disk file system and type 'logout' or 'exit' to exit the Linux 0.12 system. The following messages will be displayed:

```
[/mnt]# cd /
[/]# umount /dev/hd1
[/]# logout
```

```
child 4 died with code 0000
[/usr/root]# _
```

17.7.5 Using the Root File System on the Hard Disk Image

Once you have created a filesystem on your hard disk image file, you can have Linux 0.12 launch it as the root filesystem. This can be done by modifying the contents of the 509th, 510th byte (0x1fc, 0x1fd) of the bootimage-0.12 file. Please follow the steps below:

1. First copy the two files bootimage-0.12 and bochssrc.bxrc to generate the bootimage-0.12-hd and bochssrc-hd.bxrc files.
2. Edit the bochssrc-hd.bxrc configuration file, change the file name on the 'floppya:' option to 'bootimage-0.12-hd', and save it;
3. Edit the bootimage-0.12-hd binary with Notepad++ or any other binary editor and modify the 509th and 510th bytes (ie 0x1fc, 0x1fd). The original value should be 00, 00, modified to 01, 03, indicating that the root file system device is on the first partition of the hard disk Image, and then save and exit. If you have the file system installed on another partition, you will need to modify the first byte to correspond to your partition.

```
000001f0h: 00 00 00 00 00 00 00 00 00 00 00 00 00 01 03 55 AA ; .....U?
```

Now you can double-click on the icon of the bochssrc-hd.bxrc file. The Bochs system should quickly enter the Linux 0.12 system and display the graphics in Figure 17-8.

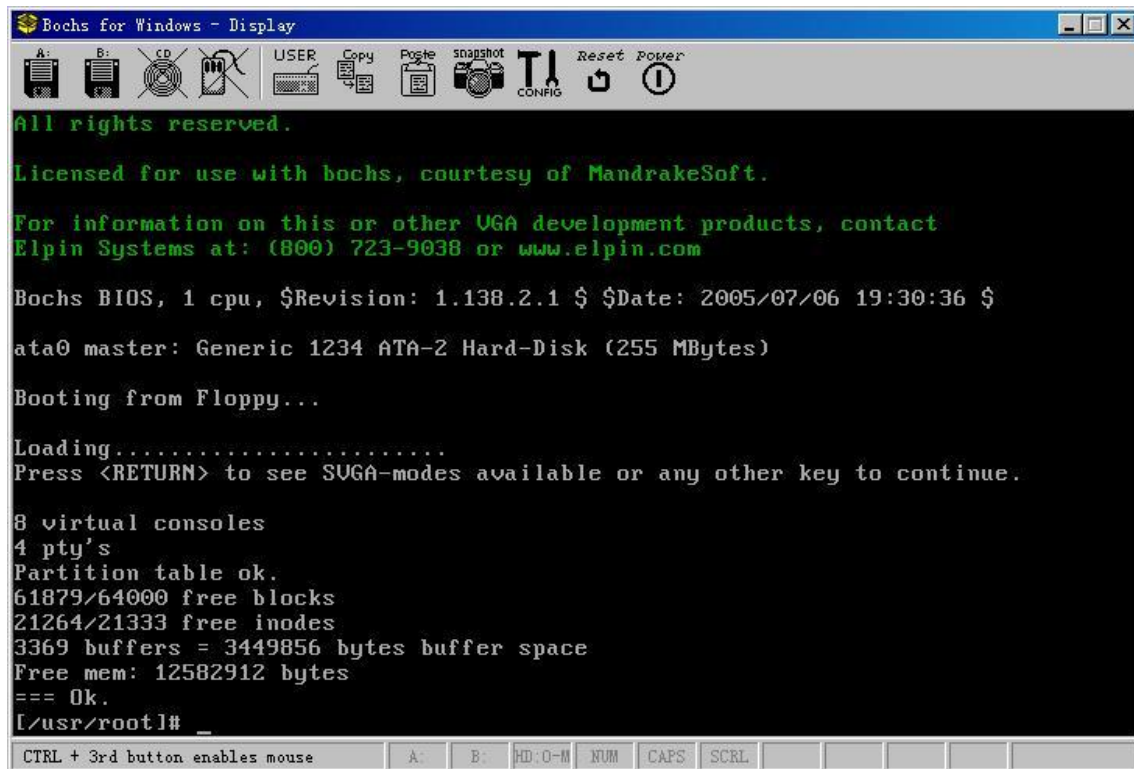


Figure 17-8 Use the file system in the hard disk image file

17.8 Compile Kernel on Linux 0.12 System

The author has reorganized a Linux 0.12 system package with the gcc 1.40 build environment. The system is set up to run under the Bochs simulation system and the corresponding bochs configuration file has been configured. This package is available from the following address:

<http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip>

The package contains a README file that explains the purpose and use of all the files in the package. If you have a bochs system installed on your system, you can run this Linux 0.12 by simply double-clicking the icon of the configuration file bochsrc-0.12-hd.bxrc. It uses the hard disk Image file as the root file system. After running the system, type the 'make' command in the /usr/src/linux directory to compile the Linux 0.12 kernel source code and generate the boot image file 'Image'. If you need to output this Image file, you can first backup the bootimage-0.12-hd file, and then use the following command to replace bootimage-0.12-hd with the new boot file. Now restart Bochs directly, you can use the bootimage-0.12-hd generated by the new compiler to boot the system.

```
[usr/src/linux]# make
[usr/src/linux]# dd bs=8192 if=Image of=/dev/fd0
[usr/src/linux]# _
```

You can also use the `mttools` command to write the newly generated Image file to the second floppy image file 'diskb.img', and then use the tool software WinImage to extract the 'Image' file in 'diskb.img'.

```
[/usr/src/linux]# mdir a:
Probable non-MSDOS disk
mdir: Cannot initialize 'A:'
[/usr/src/linux]# mcopy Image b:
Copying IMAGE
[/usr/src/linux]# mcopy System.map b:
Copying SYSTEM.MAP
[/usr/src/linux]# mdir b:
Volume in drive B is B.
Directory for B:/
GCCLIB-1 TAZ      934577      3-29-104      7:49p
IMAGE            121344      4-29-104     11:46p
SYSTEM  MAP      17162      4-29-104     11:47p
README           764      3-29-104      8:03p
      4 File(s)      382976 bytes free
[/usr/src/linux]# _
```

If you want to use the new boot image file with the root file system `rootimage-0.12` on the floppy disk, first edit the Makefile before compiling, and comment out the '`ROOT_DEV=`' line with '#'.

It is usually done very smoothly when compiling the kernel. A possible problem is that the gcc compiler does not recognize the option '`-mstring-ins`'. This option is an extended experimental parameter implemented by Linus for the gcc 1.40 compiler compiled by itself. It is used to optimize the gcc when generating string instructions. In order to solve this problem, you can directly delete this option in all Makefiles and recompile the kernel. Another possible problem is that the '`gar`' command cannot be found. In this case, you can directly link or copy/rename '`ar`' under `/usr/local/bin/` to '`gar`'.

17.9 Compile kernel under Redhat system

The original Linux operating system kernel was cross-compiled and developed on the Minix-i386, an extended version of the Minix 1.5.10 operating system. The Minix 1.5.10 operating system was released by Prentice Hall Publishing Company along with the first edition of A.S. Tanenbaum's "Design and Implementation of Minix". Although this version of Minix can run on the 80386 and its compatible microcomputers, it does not take advantage of the 80386 32-bit protection mechanism. In order to develop 32-bit operating systems on the system, Linus used Bruce Evans' patch to upgrade it to MINIX-386, and ported GNU series development tools gcc, gld, emacs, bash, etc. on to the Minix-386. On this platform, Linus cross-compile and develop kernels of versions 0.01, 0.03, 0.11, and 0.12. According to some articles in the Linux mailing list, the author has established a development platform similar to Mr. Linus's then, and successfully compiled the early version of Linux kernel.

However, since Minix 1.5.10 is outdated and the development platform is very cumbersome, here is a brief introduction to how to modify the Linux 0.12 kernel source code so that it can be compiled in the current RedHat system compilation environment, and generate a bootimage file bootimage that can be run. Readers can run it on a regular PC or in virtual machine software such as Bochs. Only the main modification aspects are given here. All the modifications can use the tool diff to compare the modified and unmodified code to find out

the difference. If the unmodified code is in the `linux/` directory and the modified code is in `linux-mdf/`, you need to execute the following command:

```
diff -r linux linux-mdf > dif.out
```

The file 'dif.out' contains all the modified places in the source code. The Linux 0.1X kernel source code that has been modified and can be compiled under RedHat 9 can be downloaded from the following address:

```
http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.tar.gz
http://oldlinux.org/Linux.old/kernel/linux-0.11-040327-rh9.diff.gz
http://oldlinux.org/Linux.old/kernel/linux-0.11-060617-gcc4-diff.gz
http://oldlinux.org/Linux.old/kernel/linux-0.11-060618-gcc4.tar.gz
http://oldlinux.org/Linux.old/kernel/linux-0.12-080328-gcc4-diff.gz
http://oldlinux.org/Linux.old/kernel/linux-0.12-080328-gcc4.tar.gz
```

When booting with the compiled boot image file, the following information should appear on the screen:

```
Booting from Floppy...
Loading system ...
```

```
Insert root floppy and press ENTER
```

Note that if there is no response after displaying "Loading system...", this means that the kernel does not recognize the hard disk controller subsystem in the computer. At this point, you can use VirtualBox, VMware, bochs and other virtual machine software to test. When you are asked to insert the root file system disk, if you press the Enter key directly, the following information that cannot be loaded the root file system will be displayed and it will crash. To run the Linux 0.1X operating system completely, you need a matching root file system, which can be downloaded from the oldlinux.org website.

17.9.1 Modifying the Makefile

In the Linux 0.1X kernel source directory, almost every subdirectory includes a Makefile, which needs to be modified as follows:

- a. Rename 'gas' to 'as', 'gld' to 'ld'. Because now 'gas' and 'gld' have been directly renamed to 'as' and 'ld'.
- b. 'as' (original gas) has no need to use the '-c' option, so the -c compilation option needs to be removed from the Makefile in the kernel home directory Linux.
- c. Remove gcc's compile flag options: '-fcombine-regs', '-mstring-insns', and these two options in the Makefile in all subdirectories. The '-fcombine-regs' option was not found in the 1994 gcc manual, and '-mstring-insns' is an option that Linus added to gcc modifications, so this optimization option is definitely not included in your gcc.
- d. In the gcc compile option, add the '-m386' option. In this way, the kernel image file compiled under RedHat 9 will not contain the instructions of the CPU of 80486 and above, so the kernel can run on the 80386 machine.

17.9.2 Modifying Comments in the Assembly Language Programs

The as86 compiler does not recognize the `c` comment statement, so you need to use the `!` to comment out the `C` comment in the `boot/bootsect.s` file.

17.9.3 Modifying the align value of the memory alignment statement

The use of the `'align'` statement has changed in the three assembly language programs in the `boot` directory. The value after the original `'align'` refers to the power value of the memory location, but now it needs to directly give the value of the integer address. Therefore, the original statement:

```
.align 3
```

Need to be modified to ($2^3=8$):

```
.align 8
```

17.9.4 Modifying Inline Macro Assembly Language Programs

Due to the continuous improvement of the `as` assembler, it is now more and more automated, so there is no need to manually specify the CPU register to be used for a variable. Therefore all `'__asm__("ax")'` in the kernel code needs to be removed. For example, on lines 20 and 26 of the `fs/bitmap.c` file, on line 68 of the `fs/namei.c` file.

In the inline assembly code, it is also necessary to remove all declarations that are invalid for the contents of the register (the registers that will be modified). For example, line 84 in `include/string.h`:

```
: "si", "di", "ax", "cx");
```

All registers need to be removed, leaving only the colon and the right parenthesis: `");`.

Sometimes there are some problems with this modification. Since `gcc` sometimes optimizes the program according to the above statement, in some places, deleting the contents of the register that will be modified will cause a `gcc` optimization error. Therefore, some places in the program code need to retain some of these declarations depending on the situation, such as line 342 in the `include/string.h` file `memcpy()` definition.

17.9.5 Reference representation of C variables in assembly statements

The assembler used in the development of the Linux 0.1X kernel needs to add the underscore character `'_'` to the variable name when referring to the `C` variables. The current `gcc` compiler can directly recognize the `C` variables referenced in these assemblies, so Remove the underscore before all `C` variables in the assembler (including embedded assembly statements). For example, the statement at line 15 in the `boot/head.s` program:

```
.globl _idt, _gdt, _pg_dir, _tmp_floppy_area
```

Need to be changed to:

```
.globl idt, gdt, pg_dir, tmp_floppy_area
```

The variable name "_stack_start" on the 31st line statement needs to be modified to "stack_start".

17.9.6 Debug Display Function in Protected Mode

Before entering protected mode, you can use the int 0x10 call in the ROM BIOS to display information on the screen. However, after entering the protection mode, these interrupt calls cannot be used. In order to understand the internal data structure and state of the kernel in a protected mode environment, we can use the following function check_data32() to display the kernel data (previously provided by a friend 'notrump' on the oldlinux.org forum). Although there is a printk() display function in the kernel, it needs to call tty_write(), which is not available when the kernel is not fully functional.

After entering protected mode, this check_data32() function can print what you are interested in on the screen. Whether the page function is enabled or not does not affect the use. Because the virtual memory in 4M just uses the first page table directory entry, and the page table directory starts from physical address 0, plus the kernel data segment base address is 0, so in the 4M range, the addresses of virtual memory, linear memory and physical memory are the same. Mr. Linus may have considered this in the first place, and feels that this setting is more convenient to use.

```
/*
 * Purpose: Display a 32-bit integer in hexadecimal on the screen.
 * Params:  value -- the integer to display.
 *          pos  -- The screen position, in units of 16 chars wide, for example, 2, which means
 *                  that the display starts at the width of 32 chars from the upper left corner.
 * Return: None.
 * If you want to use it in an assembly language program, make sure that the function is compiled
 * and linked into the kernel. The usage in gcc assembly is as follows:
 * pushl pos          // 'pos' should be replaced with your actual data, such as pushl $4
 * pushl value        // 'pos' and 'value' can be any legal addressing method.
 * call  check_data32
 */
inline void check_data32(int value, int pos)
{
    __asm__ __volatile__(
        "shl    $4, %%ebx\n\t"          // Multiply the pos by 16, plus VGA memory start address,
        "addl   $0xb8000, %%ebx\n\t"    // get the position from top left of the screen in EBX.
        "movl   $0xf0000000, %%eax\n\t" // Set a 4-bit mask.
        "movb   $28, %%cl\n\t"         // Set the initial right shift bit value.
        "1:\n\t"
        "movl   %0, %%edx\n\t"          // Put the displayed value to EDX.
        "andl   %%eax, %%edx\n\t"       // Take 4 bits specified by EAX in EDX.
        "shr    %%cl, %%edx\n\t"       // Shift 28 bits to right, EDX is the value of 4 bits taken.
        "add    $0x30, %%dx\n\t"       // Convert this value to ASCII code.
        "cmp    $0x3a, %%dx\n\t"       // If less than 10, jumps forward to label 2.
        "jb2f\n\t"
        "add    $0x07, %%dx\n\t"       // Otherwise add 7 and convert the value to A-F.
    );
}
```

```
"2:\n\t"
"add    $0x0c00, %%dx\n\t"      // Set the display attributes.
"movw   %%dx, (%ebx)\n\t"      // Put this value in the display memory.
"sub    $0x04, %%cl\n\t"      // Prepare the next hex number, the number of shifts minus 4.
"shr    $0x04, %%eax\n\t"      // The mask is shifted to the right by 4 bits.
"add    $0x02, %%ebx\n\t"      // Update the display memory position.
"cml    $0x0, %%eax\n\t"      // The mask has moved out of the right (8 hexs displayed)?
"jnz1b\n\t"                    // No, there still numbers to be displayed, jump to lable 1.
: "m" (value), "b" (pos));
}
```

17.10 Integrated Boot Disk and Root FS

This section explains how to make an integrated disk image file that is a combination of a kernel boot image file and a root file system. Its main purpose is to understand the working principle of the Linux 0.1X kernel memory virtual disk, and further understand the concept of the boot disk and the root file system disk, and deepen the understanding of the kernel/blk_drv/ramdisk.c program running method. In fact, the boot module, kernel module, and file system module image structure stored in Flash in a typical embedded system is similar to the integrated disk here.

Below we take the process of making an integrated disk using the Linux 0.11 kernel as an example to illustrate the making process. As an exercise, readers use the 0.12 kernel to implement a similar integrated disk. Before making this integrated disk, we need to first download or prepare the following experimental software (the latter two are used for the building of the 0.12 kernel integrated disk):

```
http://oldlinux.org/Linux.old/bochs/linux-0.11-devel-040923.zip
http://oldlinux.org/Linux.old/images/rootimage-0.11-for-orig
http://oldlinux.org/Linux.old/bochs/linux-0.12-080324.zip
http://oldlinux.org/Linux.old/images/rootimage-0.12-20040306
```

'linux-0.11-devel' is a Linux 0.11 system with a development environment running under Bochs. The 'rootimage-0.11' is a Linux 0.11 root file system in a 1.44MB floppy image file. The suffix 'for-orig' refers to the kernel boot image file that is compiled for the unmodified Linux 0.11 kernel source code. Of course, the "unmodified" mentioned here means that there has not been any major changes to the kernel, because we still need to modify the compiled configuration file Makefile to compile the kernel code containing the memory virtual disk.

17.10.1 Integrated disk building principle

Usually we need two disks when booting Linux 0.1X system using floppy disk (here, "disk" refers to the image file corresponding to the floppy disk): one is the kernel boot disk and the other is the root file system disk. This requires two disks to boot the system to run a basic Linux system, and the root file system disk must remain in the floppy disk drive during runtime. The integrated disk we describe here is a combination of the contents of the kernel boot disk and a basic root file system disk on one disk. This way we can boot a Linux

0.1X system to a command prompt using only one integrated disk. The integrated disk is actually a kernel boot disk with a root file system.

In order to run the integrated disk system, the function of the memory virtual disk (RAMDISK) needs to be turned on in the kernel code on the disk. The root file system on the integrated disk can be loaded into the virtual disk in memory so that the two floppy drives on the system can be freed for mounting other file system disks or for other purposes. Below we will introduce in detail the principles and steps of making an integrated disk on a 1.44MB disk.

17.10.1.1 Principle of the boot process

The Linux 0.1X kernel will determine whether the virtual disk area should be reserved in the system physical memory according to the RAMDISK option set in the compile-time Makefile. If RAMDISK is not set (ie its size is 0), the kernel will load the root file system from the floppy disk or hard disk according to the device number of the root file system set by ROOT_DEV, and perform the general startup process when there is no virtual disk.

If the size of the RAMDISK is defined in its linux/Makefile when compiling the Linux 0.1X kernel source code, the kernel code will first try to detect the boot disk from the 256th disk block after booting and initializing the RAMDISK memory area. Is there a root file system? The detection method is to check if there is a valid file system super block in the 257th disk block. If so, the file system is loaded into the RAMDISK area and used as the root file system. So we can use a boot disk that integrates the root file system to boot the system to the shell command prompt. If a valid root file system is not stored on the boot disk at the specified disk block location (starting with the 256th disk block), the kernel will prompt to insert the root file system disk. After the user presses the Enter key to confirm, the kernel reads the root file system on the independent disk and reads it into the virtual disk area of the memory. This detection and loading process can be seen in Figure 9-7.

17.10.1.2 Structure of the integrated disk

For the Linux 0.1X kernel, the size of the code plus data segment is very small, and the size is about 120KB to 160KB. In the initial stage of developing Linux system, even considering the kernel extension, Mr. Linus still thinks that the size of the kernel will not exceed 256KB, so you can store a basic root file system at the beginning of the 256th disk block of the 1.44MB boot disk. Thereby combined to form an integrated disk. A schematic diagram of a boot disk (ie, an integrated disk) with a basic root file system added is shown in Figure 17-9. For the detailed structure of the file system, please refer to the description in the file system chapter.

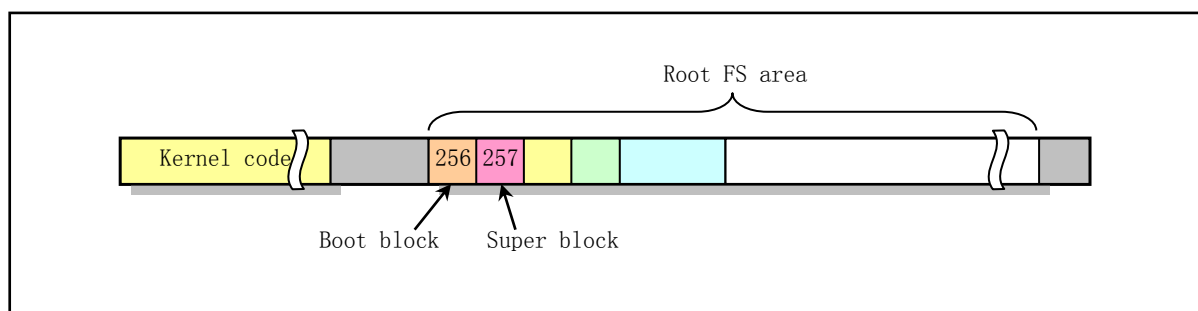


Figure 17-9 Integrated disk code structure

As mentioned above, the location and size of the root file system placement on the integrated disk is primarily related to the length of the kernel and the size of the defined RAMDISK area. Mr. Linus defines in the ramdisk.c program that the root file system's starting placement position is the beginning of the 256th disk

block. For the Linux 0.1X kernel, the compiled kernel image file (ie the boot disk Image file) is about 120KB to 160KB. Therefore, putting the root file system at the beginning of the 256th disk block of the disk is certainly no problem, only a little wasted disk space. There are still a total of $1440 - 256 = 1184$ KB space available for the root file system. Of course, we can also adjust the starting disk block location of the root file system according to the specific compiled kernel size. For example, we can modify the value of the 'block' of line 75 of ramdisk.c to 130, so that the starting position of the root file system is moved backwards to free up more disk space for the root file system on the disk.

17.10.2 Integrated disk construction process

Without changing the default disk block location in the kernel program ramdisk.c, we assume that the root file system on the integrated disk needs to be 1024KB (maximum of 1184KB). The main idea of making an integrated disk is to first create a 1.44MB empty Image disk file, and then copy the newly compiled kernel image file with RAMDISK function to the beginning of the disk. Then copy the customized file system with a size of 1024KB or less to the beginning of the 256th disk block of the disk. The specific building steps are as follows.

17.10.2.1 Recompiling the kernel

Recompile the kernel Image file with the RAMDISK definition, assuming the RAMDISK area is set to 2048KB. The method is to run the linux-0.1X system in Bochs. Edit the /usr/src/linux/Makefile file and modify the following settings line:

```
RAMDISK = -DRAMDISK = 2048  
ROOT_DEV = FLOPPY
```

Then recompile the kernel source code to generate a new kernel image file:

```
make clean; make
```

17.10.2.2 Making a Temporary Root File System

Make a root file system Image file with a size of 1024KB, and now assume its file name is 'rootram.img'. The Bochs system is run during the building process using a configuration file (bochsrc-hd.bxrc) with a hard disk Image. The building method is as follows:

- (1) Make an empty Image file of 1024KB in size using the method described earlier in this chapter. The name of the file is specified as 'rootram.img'. You can use the following command to generate under the current Linux system:

```
dd bs=1024 if=/dev/zero of=rootram.img count=1024
```

- (2) Run the linux-0.1X system in Bochs. Then configure the driver disks in the main Bochs window: disk A is rootimage-0.1X (0.11 kernel is rootimage-0.11-orign); disk B is rootram.img.
- (3) Use the following command to create an empty file system of size 1024KB on the rootram.img disk. Then mount the A and B disks to the /mnt and /mnt1 directories respectively. If the directory /mnt1 does not exist, you can create one.

```
mkfs /dev/fd1 1024
mkdir /mnt1
mount /dev/fd0 /mnt
mount /dev/fd1 /mnt1
```

- (4) Use the 'cp' command to selectively copy /mnt files from rootimage-0.1X to the /mnt1 directory and create a root filesystem in /mnt1. If you encounter any error message, then the content is usually more than 1024KB. First reduce the files in /mnt/ to meet the capacity requirements of no more than 1024KB. We can remove some files under /bin and /usr/bin to achieve this. Regarding capacity, we can use the 'df' command to view it. For example, the files we can choose to keep are the following:

```
[/mnt/bin]# ll
total 495
-rwx--x--x 1 root root 29700 Apr 29 20:15 mkfs
-rwx--x--x 1 root root 21508 Apr 29 20:15 mknod
-rwx--x--x 1 root root 25564 Apr 29 20:07 mount
-rwxr-xr-x 1 root root 283652 Sep 28 10:11 sh
-rwx--x--x 1 root root 25646 Apr 29 20:08 umount
-rwxr-xr-x 1 root 4096 116479 Mar 3 2004 vi
[/mnt/bin]# cd /mnt/usr/bin
[/mnt/usr/bin]# ll
total 364
-rwxr-xr-x 1 root root 29700 Jan 15 1992 cat
-rwxr-xr-x 1 root root 29700 Mar 4 2004 chmod
-rwxr-xr-x 1 root root 33796 Mar 4 2004 chown
-rwxr-xr-x 1 root root 37892 Mar 4 2004 cp
-rwxr-xr-x 1 root root 29700 Mar 4 2004 dd
-rwx--x--x 1 root 4096 36125 Mar 4 2004 df
-rwx--x--x 1 root root 46084 Sep 28 10:39 ls
-rwxr-xr-x 1 root root 29700 Jan 15 1992 mkdir
-rwxr-xr-x 1 root root 33796 Jan 15 1992 mv
-rwxr-xr-x 1 root root 29700 Jan 15 1992 rm
-rwxr-xr-x 1 root root 25604 Jan 15 1992 rmdir
[/mnt/usr/bin]#
```

- (5) Then use the following command to copy the file. In addition, you can modify the contents of /mnt/etc/fstab and /mnt/etc/rc as needed. At this point, we have created a file system with a size of 1024KB or less in fd1(/mnt1).

```
cd /mnt1
for i in bin dev etc usr tmp
do
cp +recursive +verbose /mnt/$i $i
done
sync
```

- (6) Use the 'umount' command to unmount the filesystems on /dev/fd0 and /dev/fd1, then use the 'dd' command to copy the filesystem from /dev/fd1 to the Linux-0.1X system and create a name called rootram-0.1X root file system Image file:

```
dd bs=1024 if=/dev/fd1 of=rootram-0.1X count=1024
```

At this time, in the Linux-0.1X system under Bochs, we have a newly compiled kernel image file /usr/src/linux/Image and a simple root file system image file rootram-0.1X with a capacity of 1024KB or less.

17.10.2.3 Creating an Integrated Disk

Now combine the above two image files to create an integrated disk. Modify the A disk configuration in the main Bochs window and set it to the previously prepared 1.44MB image file named bootroot-0.1X. Then execute the following commands:

```
dd bs=8192 if=/usr/src/linux/Image of=/dev/fd0
dd bs=1024 if=rootram-0.1X of=/dev/fd0 seek=256
sync;sync;sync;
```

The option 'bs=1024' means that the size of the definition buffer is 1KB; 'seek=256' means that the first 256 disk blocks are skipped when the output file is written. Then exit the Bochs system. At this point, we get a running integrated disk image file bootroot-0.1X in the current directory of the host.

17.10.3 Running the Integrated Disk System

First, let's make a simple Bochs configuration file, bootroot-0.1X.bxrc, for the integrated disk. The main settings are:

```
floppya: 1_44=bootroot-0.1X
```

Then double-click the configuration file with the mouse to run the Bochs system. At this point, the results should be as shown in Figure 17-10.

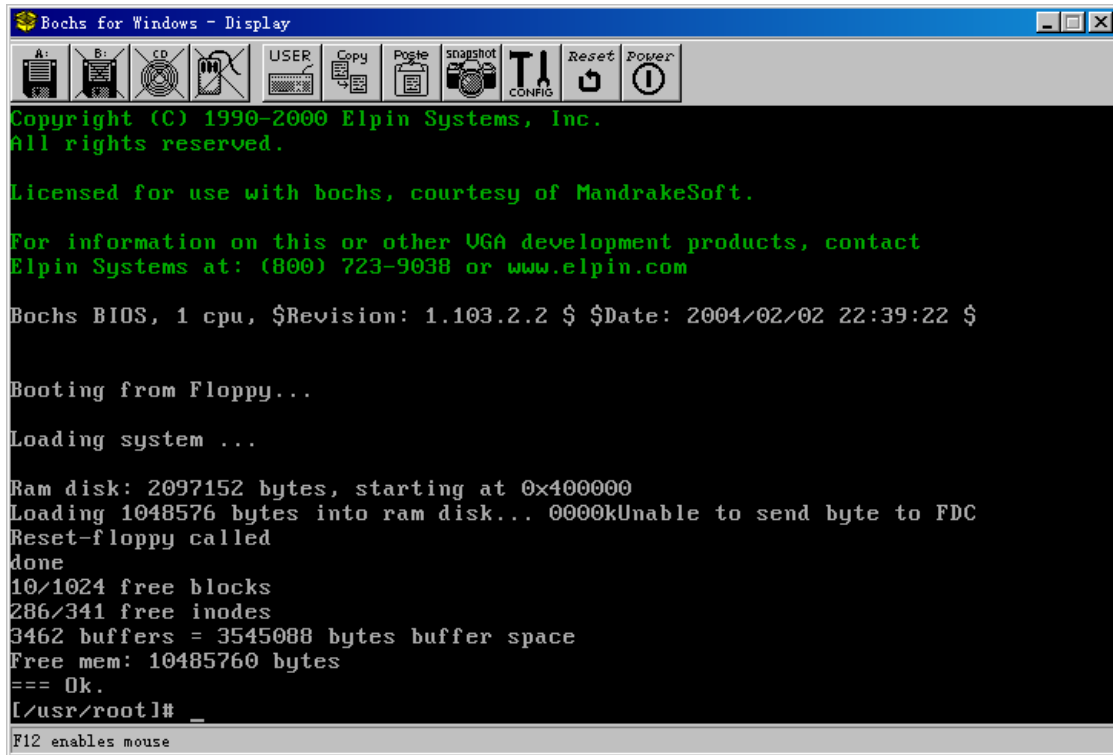


Figure 17-10 Integrated disk running interface

In order to facilitate the experiment, you can also download the integrated disk software of 0.11 kernel that is ready and can run immediately from the following website:

<http://oldlinux.org/Linux.old/bochs/bootroot-0.11-040928.zip>

17.11 Debugging Kernel Code with GDB and Bochs

This section explains how to use the Bochs emulation environment and gdb tools to debug Linux 0.1X kernel source code on existing Linux systems such as RedHat or Fedora. Before using this method, the X window system should already be installed on the existing Linux system. Since the Bochs executable in the RPM installation package provided by the Bochs website does not have the 'gdbstub' module that communicates with the gdb debugger, we need to download the Bochs source code to compile the running program with this module.

The 'gdbstub' module allows the Bochs program to listen for commands from gdb on the local 1234 network port and send command execution results to gdb. So we can use gdb to debug the C language level of the Linux 0.1X kernel. Of course, the Linux0.1X kernel also needs to be recompiled with the '-g' option to have the generated kernel code with debugging information.

17.11.1 Compiling a Bochs System with gdbstub

The Bochs User Manual describes how to compile the Bochs system yourself. Here we also give the methods and steps to compile the Bochs system with gdbstub. First download the latest Bochs system source

code from the following website (for example: bochs-2.6.tar.gz):

<http://sourceforge.net/projects/bochs/>

Decompressing the package with 'tar' will generate a bochs-2.6 subdirectory in the current directory. After entering this subdirectory, run the configuration program 'configure' with the option "--enable-gdb-stub", then run 'make' and 'make install' as shown below:

```
[root@plinux bochs-2.2]# ./configure --enable-gdb-stub
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
...
[root@plinux bochs-2.2]# make
[root@plinux bochs-2.2]# make install
```

If we encounter some problems when running './configure' and cannot generate the Makefile used for compilation, this is usually caused by not installing the X window development environment software or related library files. At this point we must first install the necessary software and then recompile Bochs.

17.11.2 Compiling the Linux 0.1X Kernel with Debug Information

By linking Bochs' simulation runtime environment to the gdb symbolic debugging tool, we can either use the kernel module with debugging information compiled under Linux 0.1X system to debug, or use the 0.1X kernel module compiled in RedHat environment to debug. In both environments, all Makefiles in the 0.1X kernel source directory need to be modified by adding the '-g' option to the compile flag line and removing the '-s' option on the link flag line:

<code>LDFLAGS = -M -x</code>	<code>// Remove '-s' flag.</code>
<code>CFLAGS =-Wall -O -g -fomit-frame-pointer \</code>	<code>// Add '-g' flag.</code>

After entering the kernel source directory, we can use the 'find' command to find all the following Makefiles that need to be modified:

```
[root@plinux linux-0.1X]# find ./ -name Makefile
./fs/Makefile
./kernel/Makefile
./kernel/chr_drv/Makefile
./kernel/math/Makefile
./kernel/blk_drv/Makefile
./lib/Makefile
./Makefile
./mm/Makefile
[root@plinux linux-0.1X]#
```

In addition, since the kernel code module compiled at this time contains debugging information, the system module size may exceed the default maximum value of the write kernel code image file `SYSSIZE = 0x3000` (defined in line 7 of the `boot/bootsect.s` file). At this point, we can modify the rules of the Image file generated in the Makefile in the root directory of the source code by removing the symbol information in the kernel module 'system' and then writing it to the Image file. The original 'system' module with the symbol information is reserved for use by the gdb debugger. Note that the implementation command for the target in the Makefile needs to start with a tab.

```
Image: boot/bootsect boot/setup tools/system tools/build
      cp -f tools/system system.tmp
      strip system.tmp
      tools/build boot/bootsect boot/setup system.tmp $(ROOT_DEV) $(SWAP_DEV) > Image
      rm -f system.tmp
      sync
```

Of course, we can also modify the `SYSSIZE` value in `boot/bootsect.s` and `tools/build.c` to `0x8000` to handle this situation.

17.11.3 Debugging methods and steps

Below we explain the debugging methods and steps according to the kernel code compiled on a modern Linux system (such as RedHat or Fedora) and compiled on a Linux 0.1X system running in Bochs. The debugging methods and steps of the Linux 0.11 kernel code are given below. The debugging method and steps of the 0.12 kernel are exactly the same.

17.11.3.1 Debugging the Linux 0.11 kernel compiled on modern Linux

Assuming that our Linux 0.11 kernel source root directory is `linux-rh9-gdb/`, we first modify all Makefiles in this directory according to the above method, then create a Bochs configuration file in it and download a root file system image file that supporting the kernel. We can also download the following packages that have been set up directly from the website to do the experiment:

<http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-rh9-050619.tar.gz>

After unpacking this package with the command "`tar zxvf linux-gdb-rh9-050619.tar.gz`", you can see that it contains the following files and directories:

```
[root@plinux linux-gdb-rh9]# ls -l
total 1600
-rw-r--r--    1 root    root      18055 Jun 18 15:07 bochsrc-fdl-gdb.bxrc
drwxr-xr-x   10 root    root        4096 Jun 18 22:55 linux
-rw-r--r--    1 root    root  1474560 Jun 18 20:21 rootimage-0.11-for-orig
-rwxr-xr-x    1 root    root        35 Jun 18 16:54 run
[root@plinux linux--gdb-rh9]#
```

The first file 'bochsrc-fdl-gdb.bxrc' is the Bochs configuration file, in which the file system image file 'rootimage-0.11-for-orig' has been set to be inserted in the second "floppy drive". The main difference between this Bochs configuration file and other Linux 0.1X configuration files is that the following line is added to the front of the file, indicating that when Bochs runs with this configuration file, it will listen for commands from the gdb debugger on the local network port 1234:

```
gdbstub: enabled=1, port=1234, text_base=0, data_base=0, bss_base=0
```

The second item of linux/ is the Linux 0.11 source code directory, which contains the kernel source code files that have been modified for all Makefiles. The third file 'rootimage-0.11-for-orig' is the root file system image file that is associated with this kernel code. The fourth file 'run' is a simple script that contains a line of Bochs startup command. The basic steps to run this experiment are as follows:

1. Open two terminal windows under the X window system;
2. In one of the terminal windows, switch the working directory to the linux-gdb-rh9/ directory and run the program './run'. At this point, a message waiting for gdb to connect is displayed: "Wait for gdb connection On localhost:1234", and the system will create a Bochs main window (no content at this time);
3. In another terminal window, we switch the working directory to the kernel source directory linux-gdb-rh9/linux/ and run the command: "gdb tools/system";
4. Type the command "break main" and "target remote localhost:1234" in the window where gdb is run. At this time, gdb will display the information that has been connected to Bochs.
5. Execute the command "cont" in the gdb environment. After a while, gdb will show that the program stops at the main() function of init/main.c.

After that we can use the gdb command to observe the source code and debug the kernel. For example, we can use the 'list' command to observe the source code, use the 'help' command to get online help information, use 'break' to set other breakpoints, use 'print/set' to display/set some variable values, use 'Next/step' to perform single-step debugging, use the 'quit' command to exit gdb, and so on. Please refer to the gdb manual for the specific usage of gdb. Below are some examples of commands that run gdb and execute in it.

```
[root@plinux linux]# gdb tools/system // Start gdb to execute the system module.
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) break main // Set a breakpoint at the main() function.
Breakpoint 1 at 0x6621: file init/main.c, line 110.
(gdb) target remote localhost:1234 // Connecting to Bochs.
Remote debugging using localhost:1234
0x0000fff0 in sys_mkdir (pathname=0x0, mode=0) at namei.c:481
481 namei.c: No such file or directory.
    in namei.c
(gdb) cont // Continue execution until the breakpoint.
```

```

Continuing.
Breakpoint 1, main () at init/main.c:110          // Stop running at the breakpoint.
110          ROOT_DEV = ORIG_ROOT_DEV;
(gdb) list                                       // View the source code.
105  {                                           /* The startup routine assumes (well, ...) this */
106  /*
107   * Interrupts are still disabled. Do necessary setups, then
108   * enable them
109   */
110      ROOT_DEV = ORIG_ROOT_DEV;
111      drive_info = DRIVE_INFO;
112      memory_end = (1<<20) + (EXT_MEM_K<<10);
113      memory_end &= 0xffff000;
114      if (memory_end > 16*1024*1024)
(gdb) next                                     // Single step execution.
111      drive_info = DRIVE_INFO;
(gdb) next                                     // Single step.
112      memory_end = (1<<20) + (EXT_MEM_K<<10);
(gdb) print /x ROOT_DEV                       // Print the variable ROOT_DEV.
$3 = 0x21d                                     // The second floppy device number.
(gdb) quit                                    // Exit gdb.
The program is running.  Exit anyway? (y or n) y
[root@plinux linux]#

```

When debugging kernel source code in gdb, sometimes the problem that the source program does not find is displayed. For example, gdb sometimes displays "memory.c: No such file or directory". This is because when compiling mm/memory.c and other files, the Makefile indicates that the ld linker has linked the file module under mm/ to generate a relocatable module 'mm.o', and in the source code root directory linux / Down, it is again used as the input module for ld. Therefore, we can copy these files to the linux/ directory and re-execute the kernel debugging operation.

17.11.3.2 Debug the 0.1X kernel compiled on Linux 0.1X system

In order to debug a kernel compiled on a 0.1X system in a modern Linux operating system such as RedHat, we need to copy the entire 0.1X kernel source directory to the Redhat system after modifying and compiling the kernel image file Image. Then follow the similar steps above. We can use the linux-0.1X environment described above to compile the kernel, then compress the kernel source tree containing the Image file, then use the mcopy command to write to the second floppy image file of Bochs, and finally use WinImage software or the mount command to extract the compressed file. The basic steps of the process of compiling and extracting files are given below.

1. Run Linux-0.1X under Bochs, enter the directory /usr/src/, and create the directory 'linux-gdb ';
2. Use the command to copy the entire 0.1X kernel source tree first: "cp -a linux linux-gdb/". Then enter the linux-gdb/linux/ directory, modify all Makefiles as described above, and compile the kernel;
3. Go back to the /usr/src/ directory and use the 'tar' command to compress the linux-gdb/ directory to get the 'linux-gdb.tgz' file.
4. Copy the compressed file to the second floppy (b drive) image file: "mcopy linux-gdb.tgz b:". If the b disk space is not enough, please use the delete file command "mdel b: file name" to make some space on the b disk.
5. If the host environment is a Windows operating system, then use WinImage to extract the compressed

file in the b disk image file and put it into the Redhat system through the FTP or other methods; if the host environment is originally Redhat or other modern Linux system, then Use the 'mount' command to load the b disk image file and copy the compressed kernel file from it.

6. Decompressing the copied compressed file on the modern Linux system will generate a linux-gdb/ directory containing the 0.1X kernel source tree. Go to the linux-gdb/ directory and create the bochs configuration file 'bochsrc-fd1-gdb.bxrc'. You can also take the contents of the 'bochsrc-fdb.bxrc' configuration file from the 'linux-0.11-devel' package and add the 'gdbstub' parameter line yourself. Then download the 'rootimage-0.11' root file system floppy image file from the oldlinux.org website, which is also saved in the linux-gdb/ directory.

After that we can continue to perform the source code debugging experiment according to the steps in the previous section. Below is an example of the above steps. Let's assume that the host environment is a Redhat system and run the Linux 0.11 system in Bochs.

```

[/usr/root]# cd /usr/src                                // Enter the source code directory.
[/usr/src]# mkdir linux-gdb                             // Ceate directory linux-gdb/
[/usr/src]# cp -a linux linux-gdb/                     // Copy source code to linux-ddb/
[/usr/src]# cd linux-gdb/linux
[/usr/src/linux-gdb/linux]# vi Makefile                // Modify the Makefiles.
...
[/usr/src/linux-gdb/linux]# make clean; make            // Compling the kernel.
...
[/usr/src/linux-gdb/linux]# cd ../../
[/usr/src]# tar zcvf linux-gdb.tgz linux-gdb           // Create compressed file.
...
[/usr/src]# mdir b:                                     // Check the contents of b disk.
Volume in drive B is Bt
Directory for B:/
LINUX-GD TGZ      827000    6-18-105  10:28p
TPUT      TAR      184320    3-09-132   3:16p
LILLO      TAR      235520    3-09-132   6:00p
SHOELA~1 Z       101767    9-19-104   1:24p
SYSTEM     MAP       17771   10-05-104  11:22p
      5 File(s)      90624 bytes free
[/usr/src]# mdel b:linux-gd.tgz                        // Space not enough, delete some file.
[/usr/src]# mcopy linux-gdb.tgz b:                    // Copy the linux-gdb.tgz to b disk.
Copying LINUX-GD. TGZ
[/usr/src]#

```

After closing the Bochs system, we get a compressed file named 'LINUX-GD.TGZ' in the b disk image file. The debug experiment directory can be established by using the following command sequence in the Redhat Linux host environment.

```

[root@plinux 0.11]# mount -t msdos diskb.img /mnt/d4 -o loop,r // Mount b disk image file.
[root@plinux 0.11]# ls -l /mnt/d4                             // Check contents.
total 1234
-rwxr-xr-x  1 root    root      235520 Mar  9  2032 lilo.tar
-rwxr-xr-x  1 root    root      723438 Jun 19  2005 linux-gd.tgz
-rwxr-xr-x  1 root    root      101767 Sep 19  2004 shoela~1.z

```

```
-rwxr-xr-x  1 root    root      17771 Oct  5  2004 system.map
-rwxr-xr-x  1 root    root      184320 Mar  9  2032 tput.tar
[root@plinux 0.11]# cp /mnt/d4/linux-gd.tgz .           // Copy the file.
[root@plinux 0.11]# umount /mnt/d4                   // Unmount the b disk.
[root@plinux 0.11]# tar zxvf linux-gd.tgz             // Untar the file.
...
[root@plinux 0.11]# cd linux-gdb
[root@plinux linux-gdb]# ls -l
total 4
drwx--x--x  10 15806   root      4096 Jun 19  2005 linux
[root@plinux linux-gdb]#
```

After that, we also need to create the Bochs configuration file 'bochsrc-fd1-gdb.bxrc' in the linux-gdb/ directory and download the floppy root file system image file 'rootimage-0.11'. For convenience, we can also create a script file 'run' containing only one line of "bochs -q -f bochsrc-fd1-gdb.bxrc" and set the file attribute to executable. In addition, a package for direct debugging experiments has been created for everyone on oldlinux.org, which contains the same content as the package compiled directly under Redhat:

<http://oldlinux.org/Linux.old/bochs/linux-0.11-gdb-050619.tar.gz>

17.12 Summary

This is the last chapter of the book. This chapter describes the experimental operation of Linux 0.1X using the Bochs simulation environment. The basic usage of the Bochs system is given. The method of transferring files between the simulation system and the host system is described in detail. It also gives specific methods and steps for compiling and debugging the Linux 0.1X kernel.

Although the content of this book has ended so far, I hope that readers will regard this as the starting point of the new journey, and begin to further learn and study the new technologies and new functions used in the kernel code of today's Linux system. Thanks again to all the friends who have strong will to accompany the author's poor writings here! I wish you all a happy new journey. Thank you!

References

- [1] Intel Co. INTEL 80386 Programmer's Reference Manual 1986, INTEL CORPORATION,1987.
- [2] Intel Co. IA-32 Intel Architecture Software Developer's Manual Volume.3: System Programming Guide. <http://www.intel.com/>, 2005.
- [3] James L. Turley. Advanced 80386 Programming Techniques. Osborne McGraw-Hill,1988.
- [4] Brian W. Kernighan, Dennis M. Ritchie. The C programming Language. Prentice-Hall 1988.
- [5] Leland L. Beck. System Software: An Introduction to Systems Programming,3nd. Addison-Wesley,1997.
- [6] Richard Stallman, Using and Porting the GNU Compiler Collection,the Free Software Foundation, 1998.
- [7] The Open Group Base Specifications Issue 6 IEEE Std 1003.1-2001, The IEEE and The Open Group.
- [8] David A Rusling, The Linux Kernel, 1999. <http://www.tldp.org/>
- [9] Linux Kernel Source Code, <http://www.kernel.org/>
- [10] Digital co.ltd. VT100 User Guide, <http://www.vt100.net/>
- [11] Clark L. Coleman. Using Inline Assembly with gcc. <http://oldlinux.org/Linux.old/>
- [12] John H. Crawford, Patrick P. Gelsinger. Programming the 80386. Sybex, 1988.
- [13] FreeBSD Online Manual, <http://www.freebsd.org/cgi/man.cgi>
- [14] Andrew S.Tanenbaum.Operating Systems: Design and Implementation. Prentice-Hall-International Editions. 1990.4
- [15] Maurice J. Bach. The Design of the UNIX Operating System. Prentice Hall. 1990
- [16] John Lions. Lions' Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications, Inc. 1996
- [17] Andrew S. Tanenbaum, Albert S. Woodhull. Operating Systems:Design and Implementation (Second Edition). Prentice Hall. 1997.
- [18] Alessandro Rubini, Jonathan. Linux Device Drivers. O'Reilly & Associates. Inc. 2001
- [19] Daniel P. Bovet, Marco Cesati. Understanding The Linux Kernel. China Electric Power Press. 2001.
- [20] 张载鸿. 微型机(PC 系列)接口控制教程, 清华大学出版社, 1992.
- [21] 李凤华, 周利华, 赵丽松. MS-DOS 5.0 内核剖析. 西安电子科技大学出版社, 1992.
- [22] RedHat 9.0 Online manual. <http://www.plinux.org/cgi-bin/man.cgi>
- [23] W.Richard Stevens. Advanced Programming in the UNIX Environment. China Machine Press. 2000.2
- [24] Linux Weekly Edition News. <http://lwn.net/>
- [25] P.J. Plauger. The Standard C Library. Prentice Hall, 1992
- [26] Free Software Foundation. The GNU C Library. <http://www.gnu.org/> 2001
- [27] Chuck Allison. The Standard C Library. C/C++ Users Journal CD-ROM, Release 6. 2003
- [28] Bochs simulation system. <http://bochs.sourceforge.net/>
- [29] Brennan "Bas" Underwood. Brennan's Guide to Inline Assembly. <http://www.rt66.com/~brennan/>
- [30] John R. Levine. Linkers & Loaders. <http://www.iecc.com/linker/>
- [31] Randal E. Bryant, David R. O'Hallaron. Computer Systems A programmer's Perspective. Publishing House of Electronics Industry. 2004.3
- [32] Intel. Data Sheet: 8254 Programmable Interval Timer. 1993.9
- [33] Intel. Data Sheet: 8259A Programmable Interrupt Controller. 1988.12
- [34] Intel. Data Sheet: 82077A CMOS Single-chip Floppy Disk Controller. 1994.5
- [35] Robert Love. Linux Kernel Development. China Machine Press. 2004
- [36] Adam Chapweske. The PS/2 Keyboard Interface. <http://www.computer-engineering.org/>

- [37] Dean Elsner, Jay Fenlason & friends. Using as: The GNU Assembler. <http://www.gnu.org/> 1998
- [38] Steve Chamberlain. Using ld: The GNU linker. <http://www.gnu.org/> 1998
- [39] Michael K. Johnson. The Linux Kernel Hackers' Guide. <http://www.tldp.org/> 1995
- [40] Richard F. Ferraro. Programmer's Guide to the EGA, VGA, and Super VGA Cards. 3rd ed. Addison-Wesley, 1995.

Appendix

A1 ASCII Code Table

Decimal	Hex	Character	Decimal	Hex	Character	Decimal	Hex	Character
0	00	NUL	43	2B	+	86	56	V
1	01	SOH	44	2C	,	87	57	W
2	02	STX	45	2D	-	88	58	X
3	03	ETX	46	2E	.	89	59	Y
4	04	EOT	47	2F	/	90	5A	Z
5	05	ENQ	48	30	0	91	5B	[
6	06	ACK	49	31	1	92	5C	\
7	07	BEL	50	32	2	93	5D]
8	08	BS	51	33	3	94	5E	^
9	09	TAB	52	34	4	95	5F	_
10	0A	LF	53	35	5	96	60	`
11	0B	VT	54	36	6	97	61	a
12	0C	FF	55	37	7	98	62	b
13	0D	CR	56	38	8	99	63	c
14	0E	SO	57	39	9	100	64	d
15	0F	SI	58	3A	:	101	65	e
16	10	DLE	59	3B	;	102	66	f
17	11	DC1	60	3C	<	103	67	g
18	12	DC2	61	3D	=	104	68	h
19	13	DC3	62	3E	>	105	69	i
20	14	DC4	63	3F	?	106	6A	j
21	15	NAK	64	40	@	107	6B	k
22	16	SYN	65	41	A	108	6C	l
23	17	ETB	66	42	B	109	6D	m
24	18	CAN	67	43	C	110	6E	n
25	19	EM	68	44	D	111	6F	o
26	1A	SUB	69	45	E	112	70	p
27	1B	ESC	70	46	F	113	71	q
28	1C	FS	71	47	G	114	72	r
29	1D	GS	72	48	H	115	73	s
30	1E	RS	73	49	I	116	74	t
31	1F	US	74	4A	J	117	75	u
32	20	(space)	75	4B	K	118	76	v
33	21	!	76	4C	L	119	77	w
34	22	"	77	4D	M	120	78	x
35	23	#	78	4E	N	121	79	y
36	24	\$	79	4F	O	122	7A	z
37	25	%	80	50	P	123	7B	{
38	26	&	81	51	Q	124	7C	
39	27	'	82	52	R	125	7D	}
40	28	(83	53	S	126	7E	~
41	29)	84	54	T	127	7F	DEL
42	2A	*	85	55	U			

A2 Common C0, C1 Control Characters

Common C0 control characters table

Mnemonic	Code	Actions taken
NUL	0x00	Null -- Ignored when received (not saved in the input buffer).
ENQ	0x05	Enquiry -- Sends a reply message.
BEL	0x07	Bell -- makes a sound.
BS	0x08	Backspace -- Moves the cursor one character position to the left. If the cursor is already on the left edge, there is no action.
HT	0x09	Horizontal Tabulation -- Moves the cursor to the next tab stop. If there is no tab stop on the right side, move to the right edge.
LF	0x0a	Linefeed -- This code causes a carriage return or line feed operation (see linefeed mode).
VT	0x0b	Vertical Tabulation -- acts like LF.
FF	0x0c	Form Feed -- acts like LF.
CR	0x0d	Carriage Return -- Moves the cursor to the left edge of the current line.
SO	0x0e	Shift Out -- Uses the G1 character set selected by the SCS control sequence. G1 can specify one of five character sets.
SI	0x0f	Shift In -- Uses the G0 character set selected by the SCS control sequence. G0 can specify one of five character sets.
DC1	0x11	Device Control 1 -- XON. Let the terminal resume transmission.
DC3	0x13	Device Control 3 -- XOFF. Stop sending all other codes except sending XOFF and XON.
CAN	0x18	Cancel -- If sent during a control sequence, the sequence will not execute and will terminate immediately. The error character is also displayed.
SUB	0x1a	Substitute -- works the same as CAN.
ESC	0x1b	Escape -- Generates an Escape Control Sequence.
DEL	0x7f	Delete -- Ignore when typing (not saved in the input buffer).

Common C1 control characters table

Mnemonic	Code	7B seq.	Actions taken
IND	0x84	ESC D	Index -- The cursor moves down one row in the same column. If the cursor is already on the bottom line, a scrolling operation is performed.
NEL	0x85	ESC H	Next Line -- The cursor moves to the first column of the next line. If the cursor is already on the bottom line, a scrolling operation is performed.
HTS	0x88	ESC E	Horizontal Tab Set -- Sets a horizontal tab stop at the cursor.
RI	0x8d	ESC M	Reverse Index -- The cursor moves one line up the same column. If the cursor is already on the top line, a scrolling operation is performed.
SS2	0x8e	ESC N	Single Shift G2 -- Temporarily uses the G2 character set in GL for the display of the next character. G2 is specified by the Selective Character Set (SCS) control sequence (see the escape sequence and control sequence table in Appendix 3).
SS3	0x8f	ESC O	Single Shift G3 -- Temporarily calls the G3 character set in GL for the display of the next character. G3 is specified by the Selective Character Set (SCS) control sequence (see the

			escape sequence and control sequence table in Appendix 3).
DCS	0x90	ESC P	Device Control String -- Used as the starting qualifier of the device control string.
CSI	0x9b	ESC [Control Sequence Introducer -- Used as a control sequence leader code.
ST	0x9c	ESC \	String Terminator -- Used as the ending qualifier of the DCS string.

A3 Escape and Control Sequences

Sequence and Name	Description
<p>ESC (Ps or ESC) Ps Select Character Set</p>	<p>Select Character Set (SCS) -- The G0 and G1 character sets can each specify one of five character sets. 'ESC (Ps' specifies the character set used by G0, 'ESC) Ps' specifies the character set used by G1. Parameters Ps: A - UK character set; B - US character set; 0 - graphic character set; 1 - alternative ROM character set; 2 - optional ROM special character set.</p> <p>A terminal can display up to 254 different characters, however the terminal only stores 127 display characters in its ROM. You must install additional character set ROM for the other 127 display characters. At some point, the terminal is able to select 94 characters (one character set). Therefore, the terminal can use one of five character sets, some of which appear in multiple character sets. At any one time, the terminal can use two active character sets. The computer can use the SCS sequence to specify any two character sets as G0 and G1. You can then switch between these two character sets using a single control character. The Shift In - SI (14) control character is used to select the G0 character set, and the Shift Out - SO (15) control character can be used to select the G1 character set. The specified character set will be used as the current character set until the terminal receives another SCS sequence.</p>
<p>ESC [Pn A Cursor up (Terminal <--> Host)</p>	<p>Cursor Up (CUU) -- The CUU control sequence moves the cursor up but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 'Pn', the cursor moves up the 'Pn' line. The cursor is moved up to the top row at most. Note that 'Pn' is an ASCII numeric variable. If you do not select a parameter or the parameter value is 0, the terminal will assume a parameter value of 1.</p>
<p>ESC [Pn B or ESC [Pn e Cursor Down (Terminal <--> Host)</p>	<p>Cursor Down (CUD) -- The CUD control sequence moves the cursor down but the column position is unchanged. The number of moving character positions is determined by parameters. If the parameter is 1 or 0, the cursor moves down 1 line. If the parameter is 'Pn', the cursor moves down the 'Pn' line. The cursor moves down to the bottom line at most.</p>
<p>ESC [Pn C or ESC [Pn a Cursor Forward (Terminal <--> Host)</p>	<p>Cursor Forward (CUF) -- The CUF control sequence moves the current cursor to the right. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the right border at most.</p>
<p>ESC [Pn D Cursor Backward (Terminal <--> Host)</p>	<p>Cursor Backward (CUB) -- The CUB control sequence moves the current cursor to the left. The number of moving positions is determined by parameters. If the argument is 1 or 0, move 1 character position. If the parameter value is 'Pn', the cursor moves by 'Pn' character positions. The cursor moves up to the left border at most.</p>
<p>ESC [Pn E Cursor moves down</p>	<p>Cursor Next Line (CNL) -- This control sequence moves the cursor to the first character of the 'Pn' line below.</p>
<p>ESC [Pn F Cursor moves up</p>	<p>Cursor Last Line (CLL) -- This control sequence moves the cursor up to the first character of the 'Pn' line.</p>
<p>ESC [Pn G or ESC [Pn ` Cursor moves in line</p>	<p>Cursor Horizon Absolute (CHA) -- This control sequence moves the cursor to the 'Pn' character position of the current line.</p>
<p>ESC [Pn ; Pn H or</p>	<p>Cursor Position (CUP), Horizontal And Vertical Position (HVP) -- The CUP control sequence moves</p>

ESC [Pn;Pn f Cursor positioning	the current cursor to the position specified by the parameter. The two parameters specify the row and column values, respectively. If the value is 0, it is the same as 1, indicating that one position is moved. In the default condition without parameters, it is equivalent to moving the cursor to the home position (ie ESC [H).
ESC [Pn d Set the row position	Vertical Line Position Absolute -- Moves the cursor to the 'Pn' line of the current column. If you try to move below the last line, the cursor will stay on the last line.
ESC [s Save cursor position	Save Current Cursor Position -- This control sequence has the same effect as DECSC except that the page number displayed on the cursor is not saved.
ESC [u Restore cursor position	Restore Saved Cursor Position -- This control sequence has the same effect as DECRC except that the cursor is still on the same display page and not moved to the display page where the cursor is saved.
ESC D Index	Index (IND) -- This control sequence moves the cursor down one line, but the column number does not change. If the cursor is on the bottom line, it will cause the screen to scroll up one line.
ESC M Reverse index	Reverse Index (RI) -- This control sequence moves the cursor up one line, but the column number does not change. If the cursor is on the top line, it will cause the screen to scroll down one line.
ESC E Move down one line	Next Line (NEL) -- This control sequence will move the cursor to the beginning of the left side of the next line. If the cursor is on the bottom line, it will cause the screen to scroll up one line.
ESC 7 Save cursor	Save Cursor (DECSC) -- This control sequence will cause the cursor position, graphics to be reproduced, and the character set to be saved.
ESC 8 Restore cursor	Restore Cursor (DECRC) -- This control sequence will cause the previously saved cursor position, graphics to be reproduced, and the character set to be restored.
ESC [Ps; Ps; ... ; Ps m Set character attributes	Select Graphic Rendition (SGR) - Character re-display and attribute are characteristics that affect the display of a character without changing the character code. The control sequence sets the character display attributes according to the parameters. All characters sent to the terminal in the future will use the attributes specified here until the control sequence reset the attributes of the character again. Parameter 'Ps': 0 - no attribute (default attribute); 1 - bold and bright; 4 - underline; 5 - flashing; 7 - reverse; 22 - non-bold; 24 - no underline; 25 - no flicker; ;30--38 Set foreground color; 39 - Default foreground color (White); 40--48 - Set background color; 49 - Default background color (Black). 30--37 and 40-47 correspond to colors: Black, Red, Green, Yellow, Blue, Magenta, Cyan, White.
ESC [Pn L Insert line	Insert Line (IL) -- This control sequence inserts one or more blank lines at the cursor. The cursor position does not change after the operation is completed. When a blank line is inserted, the line in the scroll area below the cursor moves down. The line scrolling out of the display page is lost.
ESC [Pn M Delete line	Delete Line (DL) -- This control sequence deletes one or more lines from the line where the cursor is located in the scroll area. When the line is deleted, the line below the deleted line in the scroll area moves up, and 1 blank line is added to the bottom line. If 'Pn' is greater than the number of lines remaining on the display page, then this sequence only deletes these remaining lines and does not work outside the scroll area.
ESC [Pn @ Insert character	Insert Character (ICH) -- This control sequence inserts one or more space characters at the current cursor using the normal character attribute. 'Pn' is the number of characters inserted. The default is 1. The cursor will still be at the first inserted space character. The character at the cursor and right border will shift to the right. Characters that exceed the right border will be lost.
ESC [Pn P Delete character	Delete Character (DCH) -- This control sequence deletes 'Pn' characters from the cursor. When a character is deleted, all characters to the right of the cursor are shifted to the left. This will produce a

	null character at the right border. Its properties are the same as the last left-shift character.																		
<p>ESC [Ps J</p> <p>Erase character</p>	<p>Erase In Display (ED) -- This control sequence erases some or all of the displayed characters, depending on the parameters. The erase operation removes characters from the screen without affecting other characters. The erased characters are discarded. The cursor position does not change when erasing characters or lines. While erasing characters, the attributes of the characters are also discarded. Any entire line erased by this control sequence will return the line to a single character width mode. Parameter 'Ps':</p> <p>0 - Erase the cursor to all characters at the bottom of the screen; 1 - Erase the top of the screen to the cursor except all characters; 2 - Erase the entire screen.</p>																		
<p>ESC [Ps K</p> <p>Erase in line</p>	<p>Erase In Line (EL) -- Erases some or all of the characters in the line of the cursor according to the parameters. The erase operation removes characters from the screen without affecting other characters. The erased characters are discarded. The cursor position does not change when erasing characters or lines. While erasing characters, the attributes of the characters are also discarded. Parameter 'Ps':</p> <p>0 - Erases the cursor to all characters at the end of the line; 1 - Erases the left border to all characters at the cursor; 2 - Erases an entire line.</p>																		
<p>ESC [Pn ; Pn r</p> <p>Set top &bottom margins</p>	<p>Set Top and Bottom Margins (DECSTBM) -- This control sequence sets the upper and lower areas of the scroll screen. The scrolling margin is an area on the screen where we can receive new characters by taking away the original characters from the screen. This area is defined by the top and bottom borders of the screen. The first parameter is the first line of the start of the scrolling area, and the second parameter is the last line of the scrolling area. By default it is the entire screen. The smallest scrolling area is 2 lines, ie the top border line must be smaller than the bottom border line. The cursor will be placed in the home position.</p>																		
<p>ESC [Pn c or ESC Z</p> <p>Device attributes (Terminal <--> Host)</p>	<p>In response to a host request, the terminal can send a report message. These messages provide the identity (terminal type), cursor position, and terminal operational status. There are two types of reports: device attributes and device status reports. Device Attributes (DA) -- The host sends a device attribute (DA) control sequence (the same as ESC Z) with no parameters or parameter 0. The terminal sends one of the following sequences in response to the host's sequence.</p> <table> <tr> <th>Terminal optional attribute</th><th>Send sequence</th></tr> <tr> <td>None, VT101</td><td>ESC [?1;0c</td></tr> <tr> <td>Processor option (STP)</td><td>ESC [?1;1c</td></tr> <tr> <td>Advance Video (AV0) VT100</td><td>ESC [?1;2c</td></tr> <tr> <td>AV0 and STP</td><td>ESC [?1;3c</td></tr> <tr> <td>Graphic property option (GP0)</td><td>ESC [?1;4c</td></tr> <tr> <td>GP0 and STP</td><td>ESC [?1;5c</td></tr> <tr> <td>GP0 and AV0, VT102</td><td>ESC [?1;6c</td></tr> <tr> <td>GP0, STP and AV0</td><td>ESC [?1;7c</td></tr> </table>	Terminal optional attribute	Send sequence	None, VT101	ESC [?1;0c	Processor option (STP)	ESC [?1;1c	Advance Video (AV0) VT100	ESC [?1;2c	AV0 and STP	ESC [?1;3c	Graphic property option (GP0)	ESC [?1;4c	GP0 and STP	ESC [?1;5c	GP0 and AV0, VT102	ESC [?1;6c	GP0, STP and AV0	ESC [?1;7c
Terminal optional attribute	Send sequence																		
None, VT101	ESC [?1;0c																		
Processor option (STP)	ESC [?1;1c																		
Advance Video (AV0) VT100	ESC [?1;2c																		
AV0 and STP	ESC [?1;3c																		
Graphic property option (GP0)	ESC [?1;4c																		
GP0 and STP	ESC [?1;5c																		
GP0 and AV0, VT102	ESC [?1;6c																		
GP0, STP and AV0	ESC [?1;7c																		
<p>ESC c</p> <p>Reset to initial state</p>	<p>Reset To Initial State (RIS) -- Lets the terminal reset to its initial state, that is, just turned on. All characters received during the reset phase will be lost. There are two ways to avoid this: 1. (Auto XON/XOFF) After the transmission, the host assumes that the terminal has sent XOFF. The host stops sending characters until it receives XON. 2. Delay at least 10 seconds and wait for the terminal reset operation to complete.</p>																		

A4 The First Set of Keyboard Scan Code

KEY	MAKE	BREAK	KEY	MAKE	BREAK	KEY	MAKE	BREAK
A	1E	9E	9	0A	8A	[1A	9A
B	30	B0	`	29	89	INSERT	E0, 52	E0, D2
C	2E	AE	-	0C	8C	HOME	E0, 47	E0, 97
D	20	A0	=	0D	8D	PG UP	E0, 49	E0, C9
E	12	92	\	2B	AB	DELETE	E0, 53	E0, D3
F	21	A1	BKSP	0E	8E	END	E0, 4F	E0, CF
G	22	A2	SPACE	39	B9	PG DN	E0, 51	E0, D1
H	23	A3	TAB	0F	8F	Up Arrow	E0, 48	E0, C8
I	17	97	CAPS	3A	BA	Left Arrow	E0, 4B	E0, CB
J	24	A4	Left SHFT	2A	AA	Down Arrow	E0, 50	E0, D0
K	25	A5	Left CTRL	1D	9D	Right Arrow	E0, 4D	E0, CD
L	26	A6	Left GUI	E0, 5B	E0, DB	NUM LOCK	45	C5
M	32	B2	Left ALT	38	B8	KP /	E0, 35	E0, B5
N	31	B1	Right SHFT	36	B6	KP *	37	B7
O	18	98	Right CTRL	E0, 1D	E0, 9D	KP -	4A	CA
P	19	99	Right GUI	E0, 5C	E0, DC	KP +	4E	CE
Q	10	90	Right ALT	E0, 38	E0, B8	KP ENTER	E0, 1C	E0, 9C
R	13	93	APPS	E0, 5D	E0, DD	KP .	53	D3
S	1F	9F	ENTER	1C	9C	KP 0	52	D2
T	14	94	ESC	01	81	KP 1	4F	CF
U	16	96	F1	3B	BB	KP 2	50	D0
V	2F	AF	F2	3C	BC	KP 3	51	D1
W	11	91	F3	3D	BD	KP 4	4B	CB
X	2D	AD	F4	3E	BE	KP 5	4C	CC
Y	15	95	F5	3F	BF	KP 6	4D	CD
Z	2C	AC	F6	40	C0	KP 7	47	C7
0	0B	8B	F7	41	C1	KP 8	48	C8
1	02	82	F8	42	C2	KP 9	49	C9
2	03	83	F9	43	C3]	1B	9B
3	04	84	F10	44	C4	;	27	A7
4	05	85	F11	57	D7	'	28	A8
5	06	86	F12	58	D8	,	33	B3
6	07	87	PRNT SCRN	E0, 2A, E0, 37	E0, B7, E0, AA	.	34	B4
7	08	88	SCROLL	46	C6	/	35	B5
8	09	89	PAUSE	E1, 1D, 45 E1, 9D, C5	无			

Note 1: All values in the table are in hexadecimal.

Note 2: In the table, KP - KeyPad, represents the key on the numeric keypad.

Note 3: The colored parts in the table are all extended keys.