

1 总体设计

结构上，系统分为内核部分和用户部分（用户程序、用户库）。内核管理硬件，加载执行用户程序并为其提供服务。内核包括中断/异常管理、I/O 设备驱动、内存管理、进程管理和文件系统。其中，中断/异常管理负责驱动系统运转，以及硬件、内核、用户进程之间的沟通。内存管理为进程提供独立的地址空间并通过硬件实现了一些保护措施。进程管理则提供对 CPU 的多路复用以及进程间的通信机制。文件系统的目的是组织和存储数据、支持用户和程序间的数据共享。I/O 设备驱动管理设备的运作，使系统其他部分免于陷入硬件繁杂的细节中。用户程序和用户库则基于内核，为用户模式下的编程提供便利，并向终端用户直接提供服务。

特权级上，系统分为内核模式和用户模式。当进程进入内核执行时称之为进入内核模式，在内核模式下执行，反之，进程在执行自己的代码时则称之为运行在用户模式下，两者通过中断/异常沟通。内核模式下执行的代码拥有最高特权级（设置为 x86 上的 ring 0 级别），能够使用所有硬件资源；用户模式则受到限制（设置为 x86 上的 ring 3 级别），需要发起系统调用，通过内核来获取资源的使用权^[10]。不同于微内核，整体式内核设计中，将中断/异常管理、I/O 设备驱动、内存管理、进程管理和文件系统均视作内核的一部分，均在内核模式下运行，用户程序和用户库则在用户模式下运行。

这些部分在设计时都以强调理论、概念为主，并体现如何有机的组合成一个整体，成为一个完整的可运行系统。其整体结构见图 1。

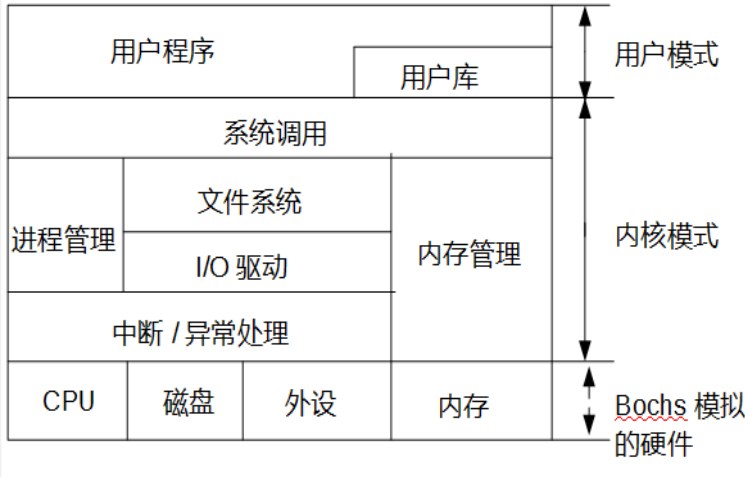


图 1 系统的架构

Fig 1 Architecture of the system

最后，为减小开发难度，提升系统的可移植性，系统选择支持 Multiboot 协议，采用 GRUB 作为 Bootloader，并将 GRUB 和内核安装到一个被格式化为 ext2 文件系统的虚拟软盘上，从软盘启动；文件系统则单独安装在 IDE 硬盘上。同时，基于 Linux，使用 GCC 交叉编译器、GNU ld、nasm、make 等工具进行开发、测试。

2 详细设计与实现

2.1 中断/异常管理以及 I/O 设备驱动

CPU 在运行进程时，一直处于一个取指、更新 EIP、执行，然后再取指的循环中^[11]。但有时用户进程需要进入内核模式，而不是执行下一条指令，比如硬件发起了中断、用户进程请求内核服务（发起系统调用）或者执行了非法指令（引发异常），所有的这些情况，都需要内核正确、高效的处理。

中断/异常处理是和硬件直接相关的。在 x86 架构上，CPU 支持最多 256 个中断号，为实现对这些中断的处理，所有中断/异常处理程序应该在 IDT（Interrupt Descriptor Table，中断描述符表）中注册程序的入口^[12]。被中断/异常打断后，CPU 将在内核栈上压入必要的寄存器，同时根据中断/异常类型以及发起方式的不同还可能压入错误代码，导致进入不同中断/异常处理程序初始时的栈结构不一致；除此之外，所有的中断/异常处理都需要保存进程状态、做相应处理、恢复进程状态然后从处理程序中返回。图 2 显示当特权级变化时，进入中断/异常处理程序时 CPU 构造的栈结构。

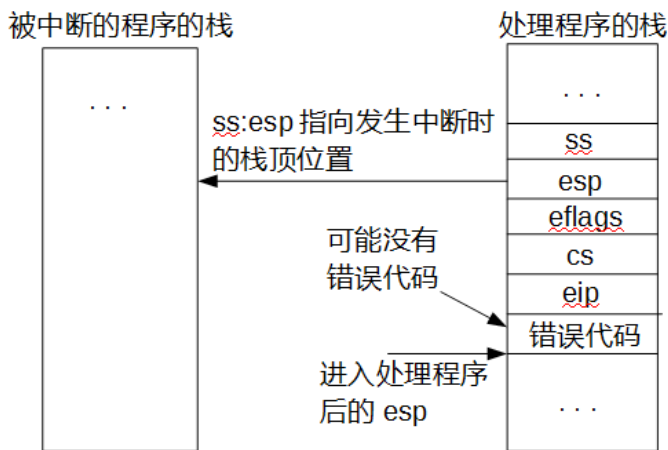


图 2 当特权级变化时，进入中断/异常处理程序时的栈结构

Fig 2 The stack structure when entering the interrupt/exception handler when the privilege level changes

由于上述因素，因此，在设计上，为减少代码冗余，实现中断/异常处理通用化，根据中断/异常处理的特性，将其处理步骤划分为 4 步：入口处理、保存现场、分发处理、恢复现场。在对应中断/异常处理的入口处理中，如果对该中断/异常 CPU 并没有压入错误代码，则会压入一个假的错误代码，以保持初始时所有处理程序的栈结构相同，然后再压入对应的中断号。保存现场和恢复现场部分对于所有中断/异常处理都是相同的，用于在进程内核栈中保存/恢复中断发生时进程的状态（一组特定的寄存器）。分发处理则是根据事先准备的中断号，做不同的处理操作，比如对于时钟中断，会强迫当前进程放弃 CPU 的使用，切换到调度进程，而对于系统调用，则根据系统调用号分发执行。系统并未选择完全使用硬件切换，而是自行定义中断时需要保存的进程状态，这种方式相对来说更加灵活，并避免了硬件切换过程中繁多的特权级检查，因而效率也更高。

中断/异常处理的框架实现如下：

```
;DE 中断入口处理
push 0                ;错误代码已由 CPU 压入，只需压入中断号
jmp isr_common_stub   ;跳到公共的处理部分
;DB 中断入口处理
push 1                ;错误代码已由 CPU 压入，只需压入中断号
jmp isr_common_stub   ;跳到公共的处理部分
...
;TS 中断入口处理
push 0                ;CPU 不压入错误代码，压入一个假的错误代码
push 10               ;压入中断号
jmp isr_common_stub   ;跳到公共的处理部分
;NP 中断入口处理
push 0                ;CPU 不压入错误代码，压入一个假的错误代码
push 11               ;压入中断号
jmp isr_common_stub   ;跳到公共的保存现场部分
...
isr_common_stub:      ;公共的保存现场部分
push ds
push es
```

```

push fs
push gs
pushad                ;现场信息已全部保存，在栈上构成一个特定格式结构
体

mov ax, 0x10          ;切换到内核数据段，栈段由 CPU 决定是否切换
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax

push esp              ;压入当前栈上保存的结构体指针，以此调用分发处理函
数

call interrupt_handler_switcher
add esp, 4

popad                 ;公共的恢复现场部分
pop gs
pop fs
pop es
pop ds
add esp, 8
iretd

```

中断的处理和 I/O 设备驱动密不可分。在管理设备时，由于驱动程序和设备同时在并发地执行，所以需要一个有效的协调方式来保证系统能正确、高效地运作。系统采用了典型的中断驱动 I/O 方式，以硬盘驱动程序为例，硬盘访问速度在毫秒级，因此系统应该在等待硬盘工作期间将 CPU 交给其他进程，并使得硬盘完成工作时发起一个中断。设计上，内核维护一个等待中的硬盘请求队列，队首的请求将被交给硬盘处理而其他请求等待前面的请求完成。中断触发时，中断处理程序将控制转交给硬盘驱动程序，随后它将唤醒在等待这个请求完成的进程，并将下一个请求提交给硬盘处理。整个处理过程如图 3。

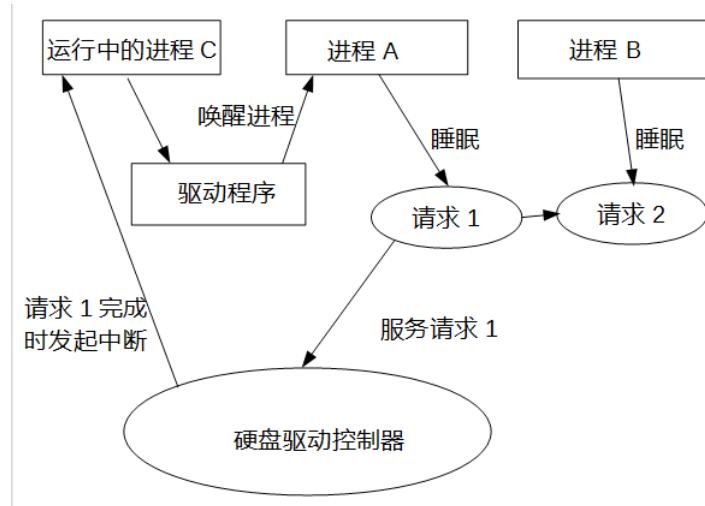


图 3 硬盘请求的处理过程

Fig 3 Process of hard disk requests

系统构建了一个虚拟的终端设备，用于管理键盘和显卡。读取该设备表示从键盘读取数据，写入该设备则意味着写入显存，在屏幕上显示。系统采用行缓冲以及回显模式。按键输出的扫描码被转换后将写入终端设备的缓冲区中，当缓冲区已满或者已经输入一行，则唤醒等待在缓冲区上的进程，且每次按键的效果均在屏幕上显示出来。这种设计将键盘和显卡封装成类似于文件的资源，使之更好的与文件系统结合。

2.2 内存管理

在 x86 保护模式下，可以使用分段式内存管理和分页式内存管理^[13]。分段式内存管理要求通过指定段描述符和段内偏移访问内存，过多地使用段会对编程带来额外的复杂性，而且就所提供的保护机制而言，分页式内存管理机制就已能够满足需求了。但由于 x86 架构的限制，分段是基础的内存管理机制，无法关闭，所以设计上，系统采用平坦内存模型，绕过分段机制，直接使用分页机制^[14]。

在开启分页管理之后，除分页结构中使用的地址外，CPU 使用的所有其他地址均为虚拟地址^[15]。同时，在设计时，将内核映射到了进程地址空间的 0xC0000000（3GB）以上位置。所以进入分页模式之前，预先设置了两类映射，进入分页模式之后，将低端地址的映射撤消，使得低端地址空间可被进程使用，这样便于链接器的处理。图 4 为初始化时系统的虚拟地址空间布局。由于硬件兼容的原因，在物理地址 0x100000（1MB）以下的内存中保存有各种硬件信息，为了简化处理，这段内存保留不作其他用处^[16]。

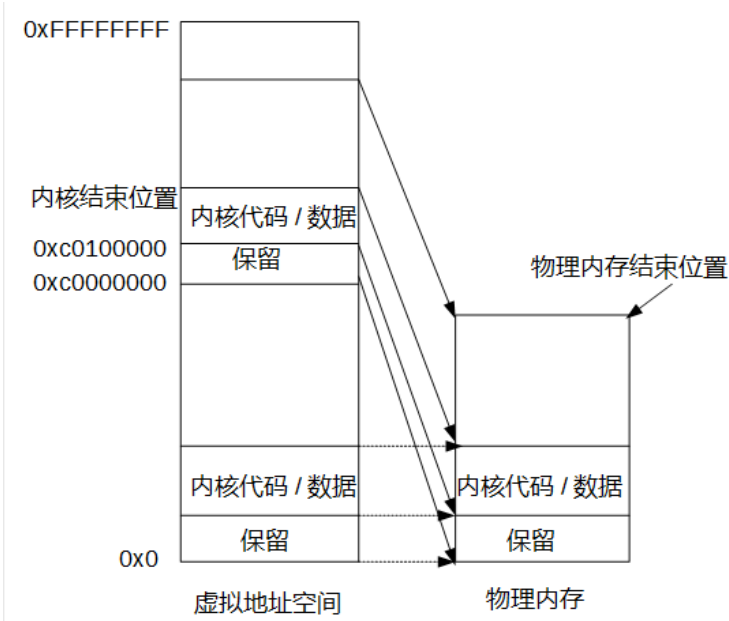


图 4 初始化时系统的虚拟地址空间布局

Fig 4 Virtual address space of the system while it is initialized

系统利用 x86 的分页硬件为进程提供独立的地址空间，并将其地址空间划分为为用户地址空间和内核地址空间，以此提供不同级别的保护。当进程处于用户模式时仅能使用用户地址空间，而当进程因中断/异常进入内核模式时，则全部的地址空间均可用。进程的用户地址空间是从 0x1000（4KB）处开始的，一直延续到 0xC0000000（3GB）处，在此以上则为内核地址空间。在用户地址空间中，从低到高依次是进程的代码段、数据段、可向上扩展的堆以及紧挨内核地址空间的用户栈，栈的下端放置了一个未被映射的页用于检测栈溢出。最低端的页不被映射，便于系统捕捉空指针。图 5 为进程的用户地址空间。

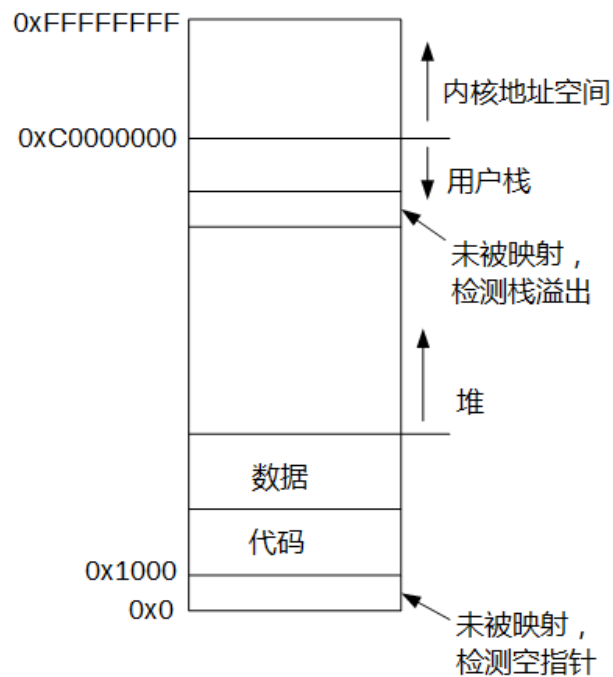


图 5 进程的用户地址空间

Fig 5 User address space of the process

在当前设计中，进程彼此间是隔离的，是一个单独的执行体。使进程的地址空间独立，才能隔离进程。但为避免频繁切换页目录带来性能上的开销，因此设计原则是使每个进程内核地址空间统一而用户地址空间独立。为实现这一点，将不同地址空间的内核地址空间部分固定映射所有可用物理内存，包括内核部分；进程的用户地址空间所需内存、页表结构以及内核栈均从内核地址空间分配，再映射到其用户地址空间中，使得在同一地址空间中多次映射同一段物理内存。这样设计带来的一个额外好处是一个进程可以通过内核空间地址访问另一个进程的地址空间，为分配、回收内存资源带来便利，但也限制了系统只能使用 1GB 以内的物理内存。图 6 为两个进程的地址空间布局。

在内存分配策略上，采用简单的侵入式链表进行按页分配。初始化时，从内核加载结束位置到可用内存的最高处，所有的空闲页均将通过一个链表链接起来，分配和

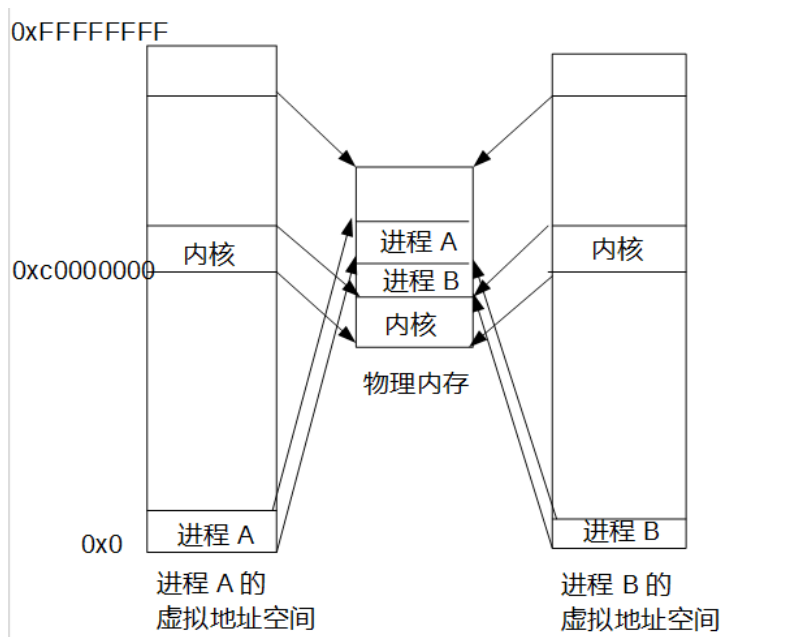


图 6 两个进程的地址空间布局

Fig 6 Address space map of two processes

释放过程就是对这个链表结点的添加和删除工作。由于系统支持 Multiboot 协议，所以使用了 Bootloader 提供的物理内存布局完成分配器的初始化^[17]。图 7 为内存中空闲页构成的链表。

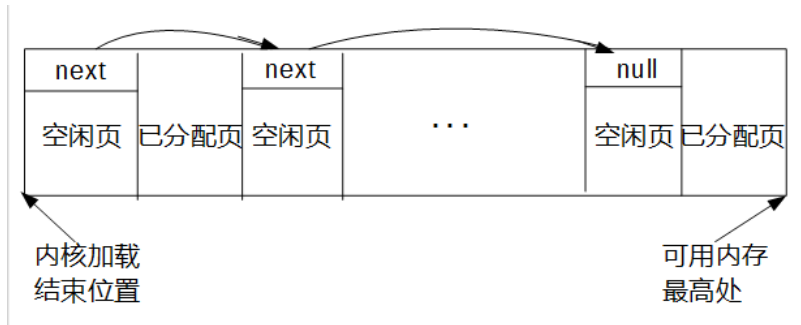


图 7 空闲页链表

Fig 7 Link list of free pages

2.3 进程管理

进程是一个活动的执行体，所以需要维护其运行时信息。内核使用 PCB（Process Control Block，进程控制块）来跟踪一个进程的状态，其中包含 pid（process identifier，进程标识符）、页目录地址、内核栈位置、当前运行状态以及打开文件结构指针等信息。

系统中 PCB 结构定义如下：

```
struct pcb_t{
```



```

uint32_t      pid;
char          name[PROC_NAME_LENGTH + 1];
proc_state_t  state;
pde_t *       pgdir;      //页目录地址
void *        kstack;     //内核栈位置
trapframe_t * tf;         //中断时现场信息
context_t *    context;    //内核上下文
uint32_t      eflags_stack[PROC_PUSHCLI_DEPTH];
uint32_t      eflags_sp;  //eflags 寄存器栈
void *        channel;    //等待队列
mem_inode_t * cwd;        //当前工作目录
file_t *      open_files[PROC_OPEN_FD_NUM]; //打开文件结构指针列

```

表

```

uint32_t      size;        //有效用户地址空间大小
struct pcb_t * parent;     //父进程
int32_t       retval;      //退出状态
};

```

为简化设计，每个进程都是单线程的，其生命周期包括若干状态：当进程刚刚被创建时为 **newborn**，可执行时则为 **runnable**，正在占用 CPU 时为 **running**，等待某些事件发生的过程中为 **sleeping**，进程执行结束后为 **zombie**，等待父进程回收其资源。这些状态的转换如图 8 所示。

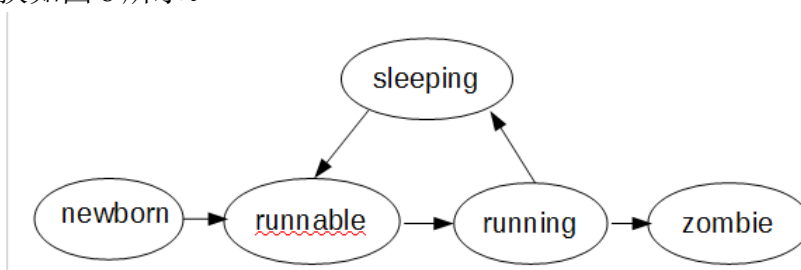


图 8 进程的生命周期

Fig 8 Life cycle of process

一个进程通过 **fork** 系统调用创建新进程。**fork** 创建的新进程被称为子进程，子进程的内存内容从创建它的进程（父进程）中复制而来，与父进程独立，并与父进程共享所打开的文件。**fork** 函数在父进程、子进程中都返回（一次调用两次返回），对于父进程它返回子进程的 **pid**，对于子进程它返回 0。

进程退出时需要发起 `exit` 系统调用，这将释放诸如内存和打开文件在内的资源，然后进入 `zombie` 状态，但其分页结构、PCB 和内核栈不会被释放。父进程通常会通过 `wait` 系统调用获取它已退出的子进程的信息，并释放其他仍然被它占据的资源，如果没有子进程退出，`wait` 会等候直到有一个子进程退出。如果父进程在子进程 `exit` 之前已经 `exit`，那么其所有子进程都将成为孤儿进程进而被第一个用户进程接管。

系统调用 `exec` 从 ELF（Executable and Linkable Format）可执行文件中读取内存镜像^[18]，以此替换掉调用它的进程的内存空间，然后使该进程以新的参数从指定位置重新开始执行。将创建进程这一动作分成两个过程是向 Unix 学习的结果，这样便于对进程执行程序之前对进程进行修改，使得诸如 I/O 重定向这样的操作能够很方便地被实现^{[19][20]}。

内核初始化时，系统将手动创建第一个用户进程。第一个用户进程初始的内存镜像（汇编编写）被链接在内核的尾部并随内核一起加载进入内存，最终被复制到其用户地址空间中；其内核栈是特殊构造的，使得该进程开始运行时看起来像是从 `fork` 调用中返回一样。随后该进程将被调度执行。这个进程立刻发起 `exec` 调用，从文件系统中加载用 C 编写的可执行文件内存镜像并替换当前进程的内存内容，然后从新的位置开始执行。

第一个用户进程在需要时会创建并打开一个终端设备文件（与键盘和屏幕相关联）用作标准输入、标准输出和标准错误输出，然后通过 `fork` 调用创建子进程并执行 `shell`，最后不断循环，处理没有父进程的僵尸进程直到 `shell` 退出。`shell` 进程读取用户输入的命令，`fork` 出若干子进程来执行用户命令并等待这些子进程的退出，然后再返回继续等待用户输入。图 9 为 `shell` 执行命令 `"ls | grep txt$"` 时系统中的进程关系。

调度进程独立于其他进程，拥有独立的内核栈和地址空间，但其地址空间中仅有内核地址空间部分。调度进程无法被调度，所以也没有其他进程生命周期的概念。当其他进程时间片耗尽（被时钟中断）或者睡眠下去时，将切换到调度进程上，调度进程只是简单地循环检查所有 PCB，找到一个可运行的进程，然后切换过去。

内核中的很多资源都是进程共享的（比如文件、内存），在访问之前必须加锁或者临时关闭中断以保证同时只有一个进程能够使用。对于持有时间较短的资源，内核只是简单地关闭中断，而对于可能需要长时间持有的资源，则使用了一些锁来提高系统的并行性。系统通常使用内核中某个数据结构的地址来区分进程所请求的资源，如果有多个进程在同时等待一个资源，释放该资源时会唤醒所有正在等待的进程。内核没

有进程组、用户、特权级等概念，所有进程以最高特权级运行，能够使用系统所有资源。

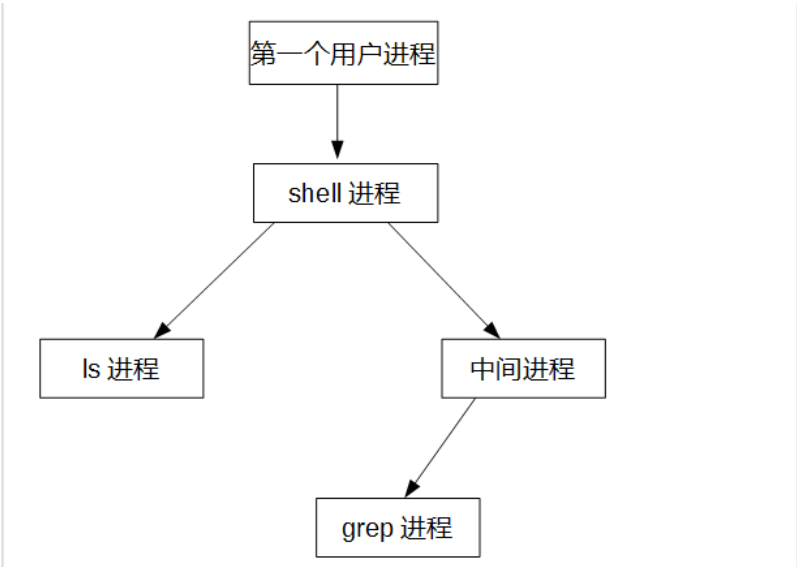


图 9 shell 执行命令"ls | grep txt\$"时系统中的进程关系

Fig 9 Processes relationship while executing “ls | grep txt\$” in shell

内核为进程间通信提供了管道这一机制。管道是一个环形的内核缓冲区，具有两个端口分别用于读写操作，如图 10 所示。管道缓冲区是循环使用的，当管道为空且仍然有进程可能会向管道写入数据时，读取管道的进程会睡眠直到有数据被写入，而当管道已满且仍有进程可能会读取管道数据时，写入管道的进程会睡眠直到管道中有数据被读取出去，使用典型的生产者-消费者模式。管道上的操作是互斥的，同一时间只有一个进程能读取或者写入管道。同时，设计上，将管道作为可被文件系统管理的一种资源，这使得用户程序可以通过文件系统的接口操作管道。

管道结构的实现如下：

```
struct pipe{
uint8_t      buf[PIPE_BUF_SIZE]; //管道缓冲区
uint32_t     ridx;                //读索引
uint32_t     widx;                //写索引
int32_t      ropen;               //是否可读
int32_t      wopen;               //是否可写
};
```

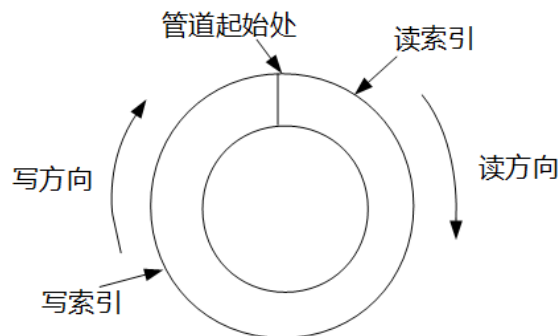


图 10 管道的结构

Fig 10 structure of pipe

2.4 文件系统

文件系统在设计上采用了和传统 Unix 类似的概念（文件，i 结点（inode），数据块（block），文件描述符，目录和路径名等）^[21]，并把数据存储在一块 IDE 硬盘上。硬盘起始扇区保留作为启动扇区；第二个扇区用作超级块，记录文件系统的元数据（位图块数、初始时空闲数据块数和空闲 i 结点数等）；随后若干个扇区用作位图，跟踪空闲数据块/i 结点位置和个数；余下部分则用作存储 i 结点和文件数据块。为简化管理，不支持硬盘分区。在硬盘上的布局如图 11 所示。

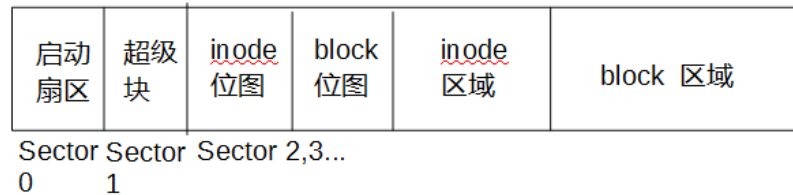


图 11 文件系统在硬盘上的布局

Fig 11 Disk map of the file system

从整体上讲，文件系统分为块分配层、i 结点层、目录层、路径层、打开文件和文件描述符层来实现。块分配层负责分配、回收数据块；i 结点层提供无名文件，每一个这样的文件由一个 i 结点和一系列数据块组成；目录层在 i 结点层的基础上实现目录，即目录只是一种特殊的无名文件，其包含一系列的目录名和对应的 i 结点号；路径层实现路径与 i 结点的映射；目录层和路径层将文件系统中所有结点组织成一个树形的结构，树的根即为文件系统的根目录；最后，打开文件和文件描述符层维护了文件使用过程中的信息，且为文件系统管理的资源提供了一层统一的抽象。

文件系统依赖内核中的块缓冲层对硬盘进行操作。块缓冲层中维护了一个内存缓冲区，用于同步对块设备的访问，使得同一时间设备上每个块最多只有一份内存缓冲区副本，并且只有一个进程使用该副本；同时如果某个块经常被访问，它将被缓冲以加快访问速度。缓冲区大小固定，为保证常用块被缓冲，系统采用 LRU 策略对缓冲区进行置换^[22]。图 12 为文件系统和所关联的块缓冲层组成的整体结构。

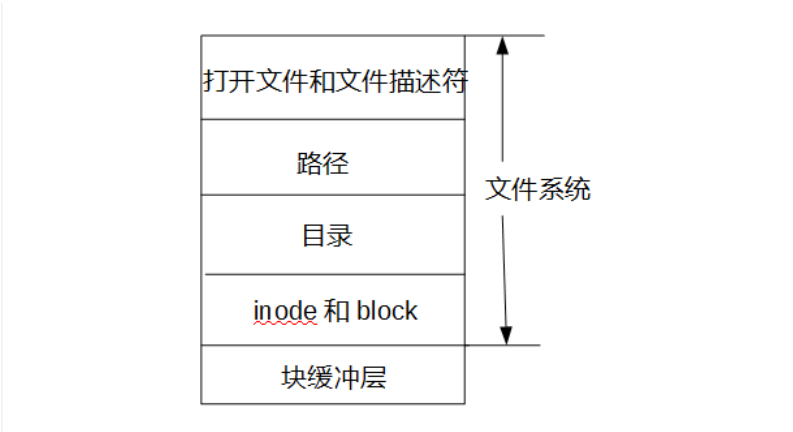


图 12 文件系统和所关联的块缓冲层组成的整体结构

Fig 12 Architecture of the file system and related buffer cache

i 结点用于保存文件的元数据。在 i 结点中，保存有文件的类型（普通文件、目录等）、链接计数（多少个目录项指向了该 i 结点）以及文件大小，如果该 i 结点表示设备文件则还需要保存其所表示设备的主/从设备号，最后，需要记录文件数据块的索引。在记录文件数据块索引时，使用了 12 个直接索引号和 1 个间接索引块。间接索引是使用一个数据块做中间索引，使用时先索引到该数据块，再根据数据块上的索引找到目标数据块。当文件大小超出直接索引所能表示的大小时，则使用间接索引。对于小文件这种索引方式能减少对硬盘的寻址，加快访问速度。i 结点可能有多个目录名与之关联，所以设计上两者是分开的。磁盘上的 i 结点结构如图 13 所示。

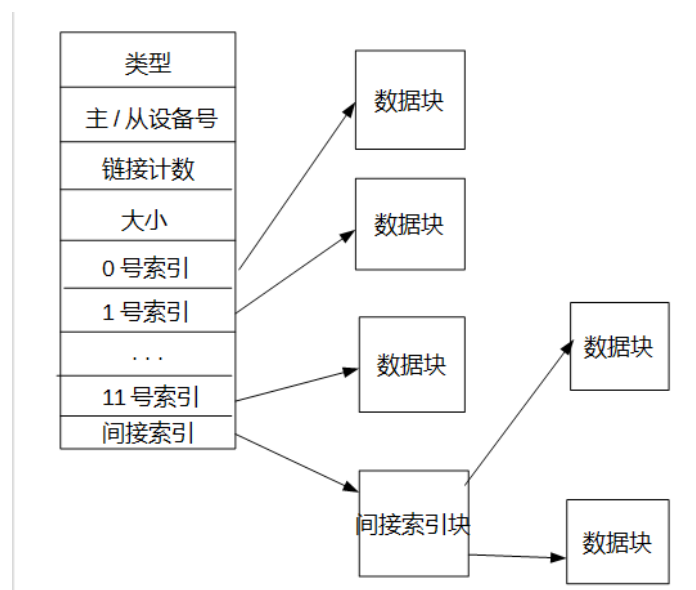


图 13 磁盘上的 i 节点结构

Fig 13 Structure of inode on disk

图 14 为多个 i 结点文件组成的目录结构，其中矩形表示目录文件，椭圆表示非目录文件（普通文件、设备文件）。每一个目录中均含有"."和".."这两个目录项，"."指向自身而".."指向父目录，根目录下的".."则指向自己。文件 b 可通过"/b"和"/c/b"两条路径访问而文件 d 只有"/c/d"这一条路径可访问。

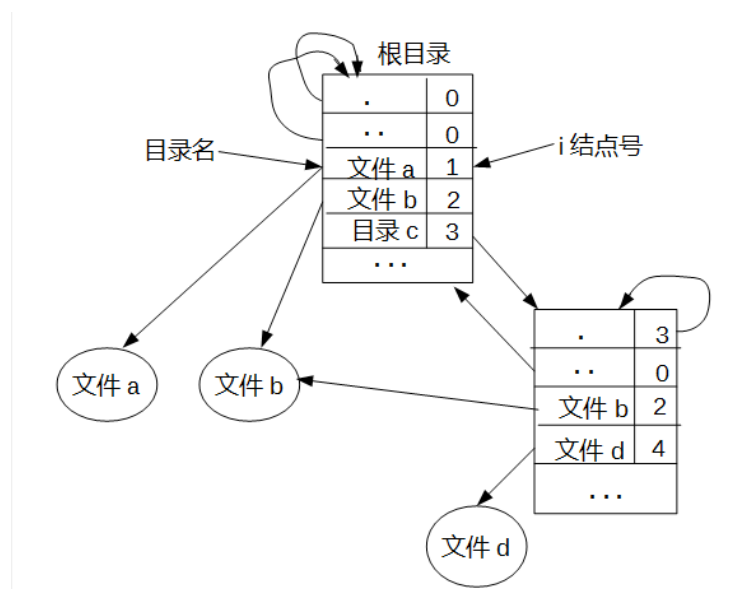


图 14 示例目录结构

Fig 14 An example of directory architecture

文件系统不仅在磁盘保存有 i 结点，同时在内存中还维护有 i 结点的副本并附加了一些信息以同步对 i 结点的访问。其中，设备号和 i 结点号记录该内存 i 结点对应哪

个设备上的 i 结点，引用计数记录该内存 i 结点被引用的次数，标志表明该内存 i 结点中的内容是否可用，以及是否被某个进程锁住。进程使用 i 结点时，首先从内存中查找是否有副本存在，没有时才会选择从磁盘读取 i 结点。内存 i 结点结构如图 15 所示。

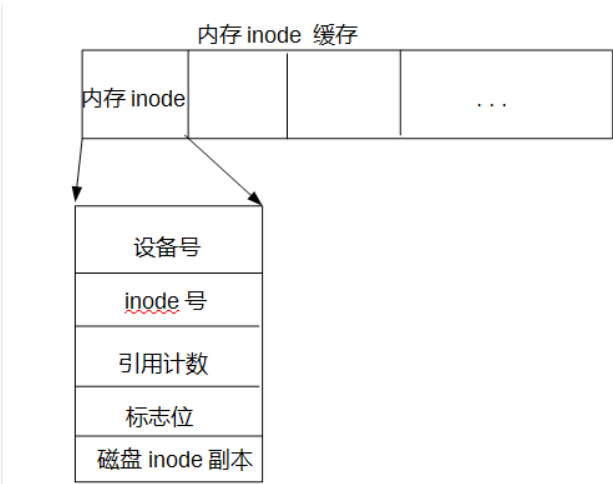


图 15 i 节点缓存

Fig 15 Memory inode cache

内核在内存中只维护活动中的 i 结点，即当存在有指针引用该 i 结点时它才会被缓存在内存中。持有对一个 i 结点的引用相当于一种较弱的锁，能够保证该 i 结点不会被内核从内存中删除，但如果进程需要使用该 i 结点，则需要调用函数锁住它。这个特性使进程能够长期地持有对 i 结点的引用（比如打开文件或者当前工作目录），同时也能避免一些在操作多个 i 结点的函数中可能发生的死锁。只有当 i 结点的引用计数和链接计数均为 0 时该 i 结点文件才会被从磁盘上删除，一个没有任何路径指向的 i 结点文件将一直保留到最后一个使用该文件的进程关闭该文件，这样即使某个进程删除了一个 i 结点文件的最后一个链接，也不会影响其他正在使用该 i 结点文件的进程。

每一个被打开的文件都关联一个打开文件结构，该结构指向一个底层的 i 结点或者管道，并保存其使用过程中的信息（比如文件读写偏移、对这个结构的引用计数），这样如果多个进程相互独立地打开同一个文件，每一个打开实例都将拥有不同的文件读写偏移。同时，每一个进程都拥有一个打开文件结构指针列表，使得一个对

打开文件结构的引用可以出现在这个列表中的多个位置，也可以出现在不同进程的列表中。内核维护一个全局的打开文件结构列表用于管理所有正在打开的文件。进程间互斥的使用同一个打开文件结构，这样多个进程同时写入同一个文件时写入的内容并不会互相覆盖，而是交织在一起。对文件进行操作时，会根据文件所表示资源类型的不同（设备、普通文件、管道）转而执行不同的处理函数，把对不同资源的访问封装为对文件的访问，从而实现“资源即文件”。

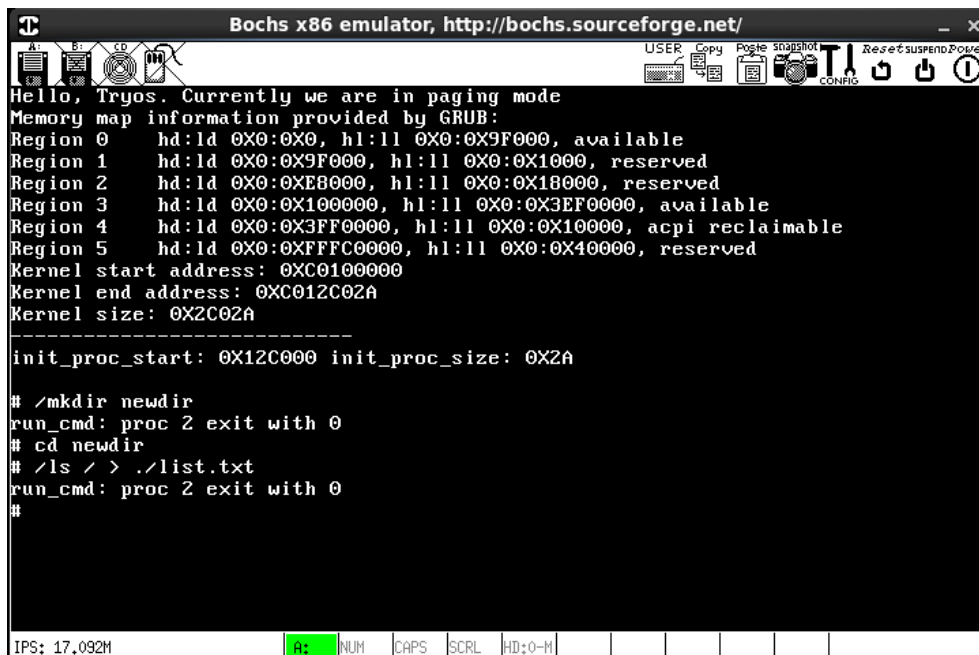
为简化设计，系统并未给文件附加除可读可写以外的其他属性，且可读可写属性被设计为运行时属性而非固有的（并未保存在磁盘上）。

内核提供了若干系统调用用于操作文件。其中，`open` 调用打开或者创建一个普通文件，或者以限定方式打开一个目录/设备文件，并返回一个可用的文件描述符。使用 `dup` 可以复制一个可用文件描述符，该文件描述符和被复制的文件描述符指向同一个打开文件结构。`read/write` 调用则从文件中读取或写入指定数量的字节，不用考虑其底层是设备、管道或者普通文件。可以通过 `fstat` 调用获取文件的信息，包括文件的大小、类型、`i` 结点号等。文件使用完毕后进程应调用 `close` 关闭文件，所有未被进程显式关闭的文件在进程退出时将被内核自动关闭。`mkdir/mknod` 可以用来创建目录或者设备文件。为保护目录结构，只能通过 `link/unlink` 调用修改目录文件。

3 系统测试

基于内核所提供的服务，系统实现了一个简单的 `shell` 用作用户和系统间的操作接口。该 `shell` 支持 I/O 重定向和不同命令间的管道通信。同时，还编制了若干工具程序对系统的测试。这里采用黑盒测试，以下为几个测试用例的演示。

在根目录下创建一个新目录 `"newdir"`，并切换到该目录作为当前 `shell` 的工作目录，然后在该目录下创建一个文件 `"list.txt"`，其内容为根目录的目录内容。该操作能够综合测试键盘驱动、硬盘驱动、命令的加载执行、目录和文件创建、进程的工作目录属性等部分，其结果如图 16 所示。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
Hello, Trygos. Currently we are in paging mode
Memory map information provided by GRUB:
Region 0  hd:1d 0X0:0X0, hl:1l 0X0:0X9F000, available
Region 1  hd:1d 0X0:0X9F000, hl:1l 0X0:0X1000, reserved
Region 2  hd:1d 0X0:0XE8000, hl:1l 0X0:0X18000, reserved
Region 3  hd:1d 0X0:0X100000, hl:1l 0X0:0X3EF0000, available
Region 4  hd:1d 0X0:0X3FF0000, hl:1l 0X0:0X10000, acpi reclaimable
Region 5  hd:1d 0X0:0XFFFC0000, hl:1l 0X0:0X40000, reserved
Kernel start address: 0XC0100000
Kernel end address: 0XC012C02A
Kernel size: 0X2C02A
-----
init_proc_start: 0X12C000 init_proc_size: 0X2A

# /mkdir newdir
run_cmd: proc 2 exit with 0
# cd newdir
# /ls / > ./list.txt
run_cmd: proc 2 exit with 0
#

IPS: 17.092M  a: NUM CAPS SCRL HD:0-H
```

图 16 创建新目录并切换进去

Fig 16 Create new directory and working within it

通过 `ls` 命令列出根目录和当前工作目录下的内容。该操作能够检查目录的结构是否被正确地维护，此时根目录下应该有 "newdir" 子目录且 `"/newdir"` 下应该有 "list.txt" 文件，执行后其结果如图 17 所示。

将 `cat` 命令的标准输入重定向到文件 "list.txt"，并将其输出写到文件 "list2.txt" 而不是屏幕上。该操作能够综合测试 shell 的重定向、文件操作（创建、写入、读取等），此时在 `"/newdir"` 下应该有 "list2.txt" 文件且其内容和 "list.txt" 相同，执行后其结果如图 18 所示。

通过 `ls` 命令列出当前工作目录下的内容并将输出通过管道传递给 `grep` 命令，从中找出结尾为 "txt" 的行，然后再写入文件 "result.txt" 中。该操作能够综合测试系统的管道机制、多进程执行等。这次执行的结果取决于在 `ls` 进程遍历当前工作目录之前文件 "result.txt" 是否已被创建好，若已经创建则会存在三个以 "txt" 结尾的行（"list.txt"、"list2.txt" 以及 "result.txt"），否则仅能找到 "list.txt" 和 "list2.txt" 这两行。执行后其结果如图 19 所示。

```
Bochs x86 emulator, http://bochs.sourceforge.net/
run_cmd: proc 2 exit with 0
# /ls / .
/:
0 0 dir (0,0) 3 448 .
0 0 dir (0,0) 3 448 ..
0 1 fil (0,0) 1 13989 uinit
0 2 fil (0,0) 1 18077 sh
0 3 fil (0,0) 1 13931 echo
0 4 fil (0,0) 1 14006 ls
0 5 fil (0,0) 1 13979 cat
0 6 fil (0,0) 1 14046 grep
0 7 fil (0,0) 1 13932 mkdir
0 8 fil (0,0) 1 13931 link
0 9 fil (0,0) 1 13933 unlink
0 10 fil (0,0) 1 13968 wc
0 11 chr (0,0) 1 0 tty
0 12 dir (0,0) 2 96 newdir
.:
0 12 dir (0,0) 2 96 .
0 0 dir (0,0) 3 448 ..
0 13 fil (0,0) 1 367 list.txt
run_cmd: proc 2 exit with 0
#
IPS: 15,829M
```

图 17 根目录和当前工作目录内容

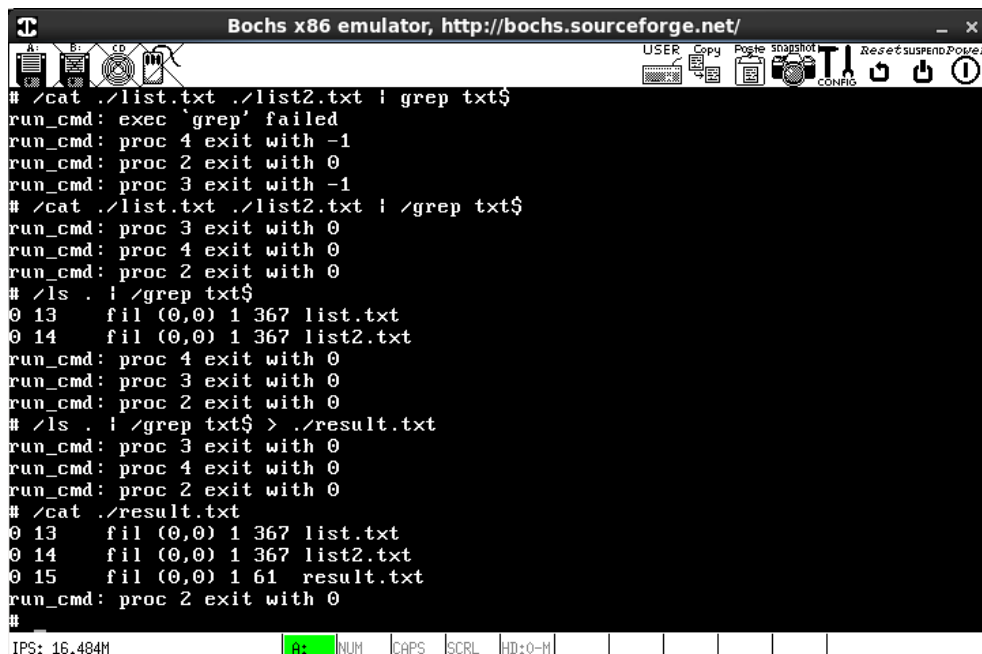
Fig 17 Contents of root directory and currently working directory

```
Bochs x86 emulator, http://bochs.sourceforge.net/
0 0 dir (0,0) 3 448 ..
0 13 fil (0,0) 1 367 list.txt
run_cmd: proc 2 exit with 0
# /cat < ./list.txt > ./list2.txt
run_cmd: proc 2 exit with 0
# /cat ./list2.txt
/:
0 0 dir (0,0) 3 448 .
0 0 dir (0,0) 3 448 ..
0 1 fil (0,0) 1 13989 uinit
0 2 fil (0,0) 1 18077 sh
0 3 fil (0,0) 1 13931 echo
0 4 fil (0,0) 1 14006 ls
0 5 fil (0,0) 1 13979 cat
0 6 fil (0,0) 1 14046 grep
0 7 fil (0,0) 1 13932 mkdir
0 8 fil (0,0) 1 13931 link
0 9 fil (0,0) 1 13933 unlink
0 10 fil (0,0) 1 13968 wc
0 11 chr (0,0) 1 0 tty
0 12 dir (0,0) 2 96 newdir
run_cmd: proc 2 exit with 0
#
IPS: 19,400M
```

图 18 输入/输出重定向

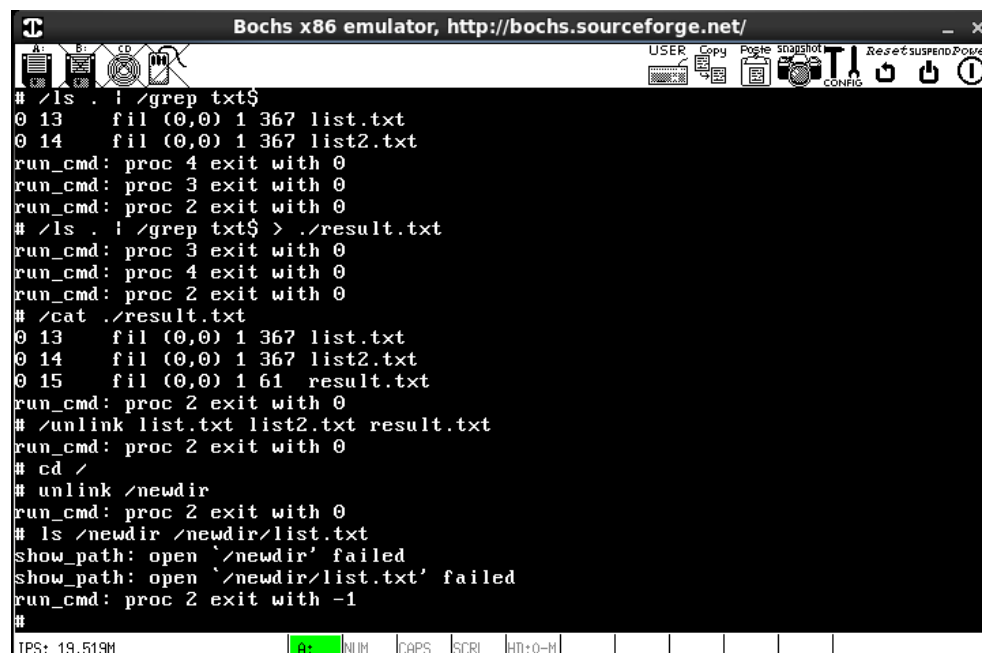
Fig 18 Redirection of input/output

最后，切换回根目录，并将"/newdir"删除（包括目录下的所有文件）。该操作能够测试目录、文件的删除操作，此时若再通过 ls 命令查询"/newdir"和"/newdir/list.txt"的信息，则会报错。执行后其结果如图 20 所示。



```
Bochs x86 emulator, http://bochs.sourceforge.net/
# /cat ./list.txt ./list2.txt | grep txt$
run_cmd: exec 'grep' failed
run_cmd: proc 4 exit with -1
run_cmd: proc 2 exit with 0
run_cmd: proc 3 exit with -1
# /cat ./list.txt ./list2.txt | /grep txt$
run_cmd: proc 3 exit with 0
run_cmd: proc 4 exit with 0
run_cmd: proc 2 exit with 0
# /ls . | /grep txt$
0 13   fil (0,0) 1 367 list.txt
0 14   fil (0,0) 1 367 list2.txt
run_cmd: proc 4 exit with 0
run_cmd: proc 3 exit with 0
run_cmd: proc 2 exit with 0
# /ls . | /grep txt$ > ./result.txt
run_cmd: proc 3 exit with 0
run_cmd: proc 4 exit with 0
run_cmd: proc 2 exit with 0
# /cat ./result.txt
0 13   fil (0,0) 1 367 list.txt
0 14   fil (0,0) 1 367 list2.txt
0 15   fil (0,0) 1 61 result.txt
run_cmd: proc 2 exit with 0
#
IPS: 16,484M
```

图 19 管道通信



```
Bochs x86 emulator, http://bochs.sourceforge.net/
# /ls . | /grep txt$
0 13   fil (0,0) 1 367 list.txt
0 14   fil (0,0) 1 367 list2.txt
run_cmd: proc 4 exit with 0
run_cmd: proc 3 exit with 0
run_cmd: proc 2 exit with 0
# /ls . | /grep txt$ > ./result.txt
run_cmd: proc 3 exit with 0
run_cmd: proc 4 exit with 0
run_cmd: proc 2 exit with 0
# /cat ./result.txt
0 13   fil (0,0) 1 367 list.txt
0 14   fil (0,0) 1 367 list2.txt
0 15   fil (0,0) 1 61 result.txt
run_cmd: proc 2 exit with 0
# /unlink list.txt list2.txt result.txt
run_cmd: proc 2 exit with 0
# cd /
# unlink /newdir
run_cmd: proc 2 exit with 0
# ls /newdir /newdir/list.txt
show_path: open '/newdir' failed
show_path: open '/newdir/list.txt' failed
run_cmd: proc 2 exit with -1
#
IPS: 19,519M
```

图 20 恢复原先的状态

Fig 20 Restore original state

上述测试用例的成功执行验证了系统运作正确且符合预期。

参考文献

- [1] Tanenbaum A S, Bos H. Modern operating systems[M]. Pearson, 2015: 1.
- [2] 吴晓诗. 《计算机操作系统》课程教学改革初探[J]. 考试周刊, 2011, 15: 8-9.
- [3] 杨晓敏, 吴炜, 刘志芳, 孔贵琴. 《计算机操作系统》课程教学改革之探索与实践[J]. 教育教学论坛, 2015, 7: 110-111.
- [4] 张铭, 马殿富, 等. 计算机科学课程体系规范 2013[OL]. 北京: 高等教育出版社, 2015: 120-126. https://www.acm.org/binaries/content/assets/education/cs2013_chinese.pdf.
- [5] OSDEV wiki. Required_Knowledge[OL]. US: OSDEV wiki, 2017. https://wiki.osdev.org/Required_Knowledge.
- [6] Wikipedia. Monolithic_kernel[OL]. US: Wikipeida, 2018. https://en.wikipedia.org/wiki/Monolithic_kernel.
- [7] Roch B. Monolithic kernel vs. Microkernel[J]. TU Wien, 2004.
- [8] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture[M]. US: Intel, 2017: 33.
- [9] Kevin Lawton, Bryce Denney, etc. Bochs User Manual[OL]. Bochs, 2017. <http://bochs.sourceforge.net/doc/docbook/user/index.html>.
- [10] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide[M]. US: Intel, 2017: 208.
- [11] 唐朔飞, 刘旭东, 王诚. 计算机组成原理第二版[M]. 高等教育出版社, 2000: 17.
- [12] Intel. Intel 80386 Programmer's Reference Manual[OL]. US: Intel, 1987: 152. <https://pdos.csail.mit.edu/6.828/2018/readings/i386.pdf>.
- [13] Jerry Breecher. Intel's View of Memory Management[OL]. EN: Worcester Polytechnic Institute, 2007. <https://web.cs.wpi.edu/~cs3013/c07/lectures/Section09.1-Intel.pdf>.
- [14] James Mickens, Nickolai Zeldovich. Memory Organization and Segmentation[OL]. US: MIT, 2014. https://css.csail.mit.edu/6.858/2014/readings/i386/s02_01.htm.
- [15] 李忠. x86 汇编语言从实模式到保护模式[M]. 电子工业出版社, 2013: 310.
- [16] OSDEV wiki. Memory Map (x86)[OL]. US: OSDEV, 2019, [https://wiki.osdev.org/Memory_Map_\(x86\)](https://wiki.osdev.org/Memory_Map_(x86)).
- [17] Bryan Ford, Erich Stefan Boleyn. Multiboot Specification version 0.6.96[OL]. US: Free Software Foundation, 2010.

- [18] Tool Interface Standard. Executable and Linkable Format Specification v1.2[OL]. US: TIS Commitee, 1995. <http://refspecs.linuxbase.org/elf/elf.pdf>.
- [19] Tanenbaum A S, Woodhull A S. Operating systems: design and implementation 3rd[M]. Englewood Cliffs: Prentice Hall, 2006: 69.
- [20] Bach M J. The design of the UNIX operating system[M]. Englewood Cliffs, NJ: Prentice-Hall, 1986: 226-227.
- [21] Ritchie D M, Thompson K. The UNIX time-sharing system[J]. Bell System Technical Journal, 1978, 57(6): 1905-1929.
- [22] Wikipedia. Cache replacement policies[OL]. US: Wikipedia, 2018. https://en.wikipedia.org/wiki/Cache_replacement_policies.
- [23] Russ Cox, Frans Kaashoek, Robert Morris. xv6 a simple, Unix-like teaching operating system[OL]. MIT, 2014. <https://pdos.csail.mit.edu/6.828/2014/xv6/book-rev8.pdf>.