

RASPBERRY OS

操作系统的实现项目文档

2053398 马启越

2053399 胡顺

目录

1、项目描述	04
2、项目环境	04
3、实现细则	05
3.1、工作环境搭建	06
3.2、Boot 的制作	06
3.3、建立保护模式	06
3.4、制作 Loader	07
3.5、建立内核雏形	08
3.6、加入中断处理	08
3.7、加入单进程	09
3.8、多进程内核	10
3.9、输入输出系统	10
3.10、多控制台雏形	11
3.11、建立通信机制	12
3.12、建立文件系统	13
3.13、升级控制台	16
3.14、利用函数创建进程	16
3.15、内存分配	17
3.16、函数实现进程终止	18
3.17、编写用户级程序	20
3.18、函数加载进程	21

4、项目界面展示	23
5、项目分工	35
6、源码链接	35

1、 项目描述

在本项目中，我们在书本《ORANGE’ S：一个操作系统的实现》的引导下，一步一步完成了一个简易的操作系统“RASPBERRYOS”。

在原书的基础上，我们自己针对 Orange 系统中的不足做出了改动，首先我们升级了文件系统：我们改写了作者的扁平化 (FLAT) 文件系统，将其较简陋的扁平结构优化为更高效、也更用户友好的文件夹目录结构，使得文件不必都建立在根文件夹下，并进一步实现了文件系统的系统级应用，使用户可以直接通过控制台来控制文件的建立、打开、读取、写入和删除；其次，我们拓展了作者建立的进程管理系统，结合内存管理部分，增添了创建进程、删除进程和显示进程表等用户操作，实现了进程管理的系统级应用，使用户能够完成以上操作。

其次，我们运用书中介绍的“安装”应用程序的方法，在我们的操作系统中安装了一些用户级应用。我们使用将用户级应用程序编译链接，并打包后写入文件系统中，使用户级应用在操作系统中作为普通应用程序出现，减少了内核的冗杂。采用这种方法，我们为操作系统实现了记忆游戏、迷宫游戏和推箱子游戏三个用户级应用。

2、 项目环境

项目语言：C 语言、汇编语言

编译器：GCC（C 语言），NASM（汇编语言）

操作系统运行环境：模拟计算机 Bochs 2.6.11/2.7

Bochs 运行环境：window10 下 VMWare 虚拟机中的 Linux 系统

3、 实现细则

3.1、工作环境的搭建

从原则上来讲，我们实现的操作系统是完全可以直接安装于计算机之上直接运行的，但考虑如果直接安装于计算机之上，我们的调式以及修改工作将会十分复杂繁琐，且一不小心的问题可能引发难以收拾的后果，我们还是退求其次，在虚拟计算机 Bochs 上安装我们的系统来模拟其在计算机上的运作效果。

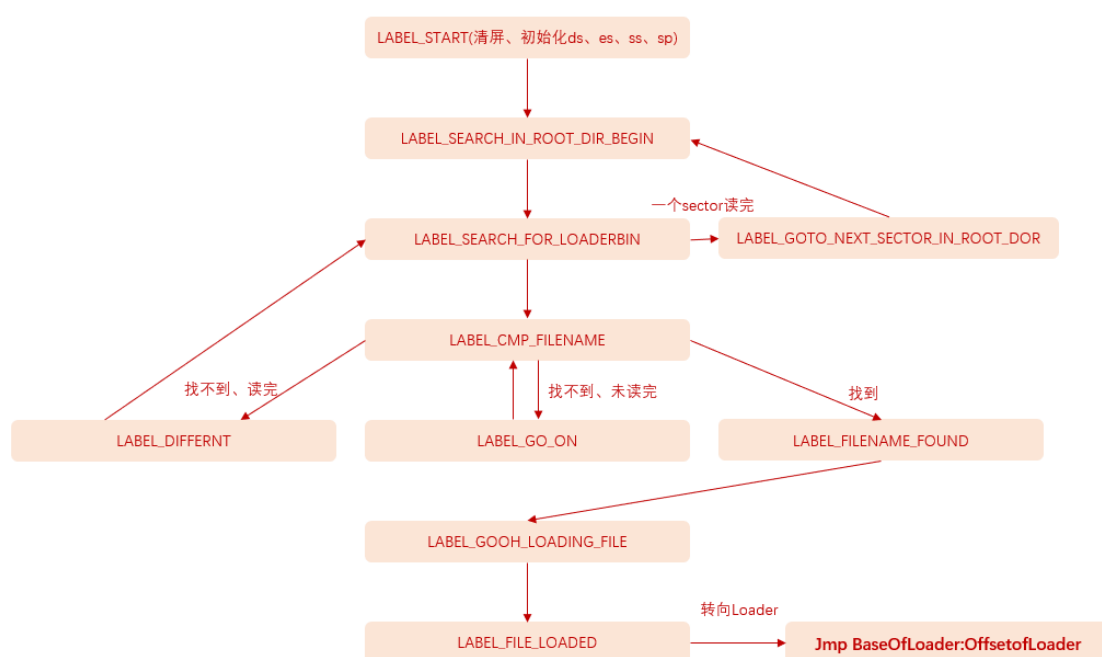
在此之外，考虑相对于 Windows，Linux 提供更加便捷的指令操作，可以让我们快速的编译文件以及方便的对软盘硬盘进行直接操作，选择 Linux 作为我们的大工作环境是个更好的选择。

按照上面的分析，我们首先利用 VMWare 建立了 Linux 虚拟机，又在 Linux 系统中，从 Bochs 官网上下载了 Bochs 进行了安装，至此我们的工作环境搭建完毕，在之后的操作中，我们将用 makefile 指导编译，用 bochsrc 来指导 Bochs 来使用我们建立的虚拟硬盘软盘来运行我们的操作系统。

3.2、Boot 的制作

Boot 的作用是引导 Loader，而 Loader 的作用是引导 kernel，kernel 则是我们操作系统的核心，大部分的操作将围绕着 kernel 展开，由此可见，Boot 的根本作用就是“引导的引导”，是我们操作系统之始，十分重要的。

Boot 虽然重要，但其的结构和任务是简单的，其任务就是在软盘中找到 Loader 且仅有一个文件“boot.asm”，其中的主要结构如图所示：



经过以上操作后，我们得以进入 Loader，以进行下一步操作。

3.3、保护模式的建立

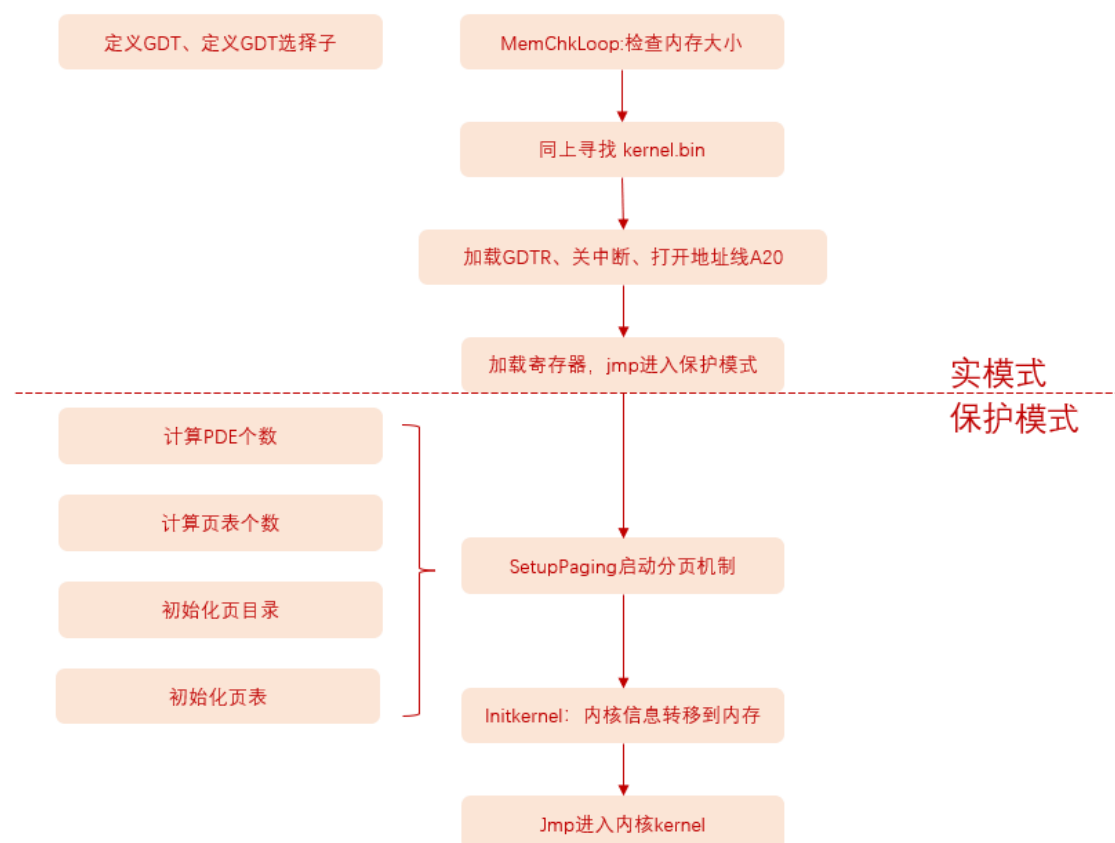
保护模式是一种抽象的说法，其实更加容易理解的说法是 32 位模式，自 Boot 引导我们的操作系统开始，我们的操作系统一直运作在实模式下，也就是 16 位之下，在这种模式下，我们可以管理的内存仅有 1MB，是极为可怜的，为了让我们有更多的操作空间，我们必须通过 Loader 进入 32 位模式进行操作，以获取多达 4G 的内存。

保护模式的进入简单来讲主要分为以下几步：

加载 GDTR->关中断->打开地址线 A20->向寄存器写入准备切换到保护模式
->jmp 真正进入保护模式

3.4、Loader 的制作

进入保护模式只是一个过程，在这个过程前后我们都有进入内核前需要准备的事情，而这些事情的准备都要交由 Loader 来执行，Loader 是进入我们操作系统内核之前的最后一步，它承载着包括进入保护模式在内的诸多功能，虽然其功能较多，但我们的 Boot 一样，其核心文件仅有一个即我们的“Loader.asm”，主要工作流程如下所示：



经过以上操作，我们得以进入 kernel 这片更加广阔的天地来进行我们的下一步操作。

3.5、内核雏形建立

在进入 kernel 的领域后，我们不必再被汇编语言局限，我们还可以加入 c 语言的操作以实现更为复杂的工作，但这一切在目前还只是设想，我们需要构建一个基础的内核来让我们的任务走向正轨，目前内核主要的任务为切换堆栈和 GDT 来使我们的操作系统真正进入内核部分，为了实现这个任务，我们主要使用“kernel.asm”、“start.c”这两个文件，具体流程如下：



在此之后，我们正式进入了 kernel，可以进行下一步的工作。

3.6、加入中断处理

在此之前，我们的操作系统仿佛已经是初具规模，可以称为一个所谓的“Baby OS”，但令人遗憾的是，其不会对外界的输入做出任何有意义的响应也没有运行任何一个进程。为了改变这个现状，我们将引入中断处理，来实现 CPU 控制权的转换，以让我们的操作系统可以完成更多任务并对外界的输入做出相应的反应。

中断的建立并不十分复杂，主要有两步：设置 8259A 与和建立 IDT（中断表），具体过程如下。

对于 8259A，其是一个可编程中断控制器，我们使用函数 `init_8259A` 来对其进行初始化，具体过程为依次向端口 20h(主片)或 A0h(从片)写入 ICW1、ICW2、ICW3、ICW4，在进行完这些操作后，设置 8259A 的工作结束（此时所有的中断都被关闭）。

建立 IDT 的工作是稍显复杂的，首先我们类比 GDT 的初始化，在 `start.c` 中建立 IDT 的初始化，紧接着，我们增加对异常的处理，总体的思想为，向栈内压入错误码 0xFFFFFFFF, 再将向量号入栈随后执行处理函数，在此之后我们将建立 IDT 设置函数 `init_prot()`。

以上操作结束后，我们的操作系统就有了自己应付异常的能力，但仍

无法对外界做出相应的响应或者运行任何一个进程，这是因为我们还没有打开任何一个中断，此时我们在 `init_8259A` 中打开键盘的中断，我们的操作系统即可对我们键盘的输入做出相应了，这是令人惊奇的一步，但遗憾的是此时我们仍然无法运行任何一个进程。

3.7、加入进程

为了弥补上述的遗憾，我们开始设计我们的进程部分，进程的很大一部分核心在于时钟中断带来的进程切换，所以我们首先解决进程的切换问题。

对于进程的切换，我们需要做到的核心就是，在一个进程休眠之时其的寄存器是什么样貌，其回来运作之时，寄存器也应该如此，这就要求我们设计一个进程状态保存体系，这个体系即为 PCB，利用 PCB 我们将可以便捷的保存各个进程的状态，以方便的进行进程间的调度，为此我们定义结构体 `s_proc` 来充当进程表的角色，除此之外，考虑到我们的进程存在不同特权级的转换，我们还需要定义一个 TSS（任务状态段），来记录不同特权级下的寄存器状态，以处理不同特权级之间的转换。

在此上述操作以后，我们拥有了记录进程状态的能力，但我们还没有进程切换的动机即——时钟中断，在加入一系列时钟中断的处理以及阻止同类型嵌套中断的中断开关后，我们的进程切换拥有了动机。

在此之后，我们拥有了记录状态的能力，也拥有了动机，此时我们就应该完成我们的第一个进程了，首先我们完成一个小的进程体即进程本身，然后编写函数 `kernel_main` 来初始化我们的进程表，在此之后我们将初始化我们的 TSS，至此第一个进程切换就基本准备妥当了，我们将实现从目前的系统转向我们的进程，在最后，我们在 `kernel.asm` 中加入 `restart`，来完成第一个进程切换的寄存器转换，在此之后，我们就完成了我们第一个进程的切换以及运行。

3.8、多进程内核实现

在我们建立起进程的体系结构后，再加入一个进程成为了较为容易的事情，我们只要做出如下改动即可。

首先，我们新增一个结构体：TASK，以用来记录我们进程的信息，在此之后，我们将我们的进程表扩展为进程表数组以用来存放我们的不同进程的信息。当这些完成时，我们的多进程内核的结构就基本完成了，但我们还需要在我们初始化进程表的地方将初始化过程升级为一个循环的过程以用来初始化我们增多的进程表，在这些操作之后，我们实现了多进程。

3.9、在内核中加入输入输出系统

就算我们实现了多进程，目前来讲，我们的进程也不过之是一些自娱自乐的小玩意罢了，包括我们的操作系统在内，其一经启动，全然是自顾自的运行，为了改变这一现状，我们将加入输入输出系统。

对于我们的操作系统，一次性打开太多的输入中断可能会使我们难以处理，不妨退一步，我们的操作系统仅支持键盘这一种输入方式。为了使用键盘中断，我们建立新的函数仿照上文的方式首先打开键盘中断，再设置一个键盘中断处理程序，我们注意的是，再键盘中断处理程序中我们一定要在每次接收到键盘中断信号后就去读取一次输入缓冲区，否则新的扫描码不会被接收。

在完成这一切后，当我们尝试输出我们所捕获的扫描码时，我们发现我们的每个按键都对应着两个扫描码，而非我们熟悉的一个 ASCII 码，为了应对这个问题，我们建立了 keymap 来对应扫描码和 ASCII 码。

在完场上述过程后，我们有了接受扫描码并将其转化为我们熟知的 ASCII 码的能力，但现在我们的操作系统仅能处理单个字符的键入，对于组合键例如“CTRL + F1”我们无法应对，面对这个问题，我们加入了一个新的结构——键盘输入缓冲区，至此我们的操作系统有了接受组合按键的能力。

有了以上的设计，我们的操作系统仿佛是拥有了与用户交互的能力，但现实是，我们操作系统中并没有任何一个地方可以处理我们获得信息，我们现在能做的只是傻乎乎的输出而已，为此，我们结合上面进程的设计，我们设计一个终端进程用以处理键盘输入以及以后的输出问题。在我们的终端进程中，我们调用一个我们新设计的函数“`keyboard_read`”，在此之中，我们将利用我们设计的 `keymap` 来获取 ASCII 码，并对不同键位做出不同的操作。首先，对于普通字母数字键位，我们对其作输出处理，而另一些功能键，我们可以为其设计相应的处理动作，至此，我们的输出系统初具规模。

对于输出系统，我们可以通过向 VGA 系统的不同寄存器写入不同值来实现不同的功能，再配合我们之前的写入显存的函数，我们姑且算是有了一个输出系统。

3.10、多控制台雏形建立

在此之前我们已经实现了一个控制台的雏形，多控制台对于我们的难点是不同控制台被切换时，要显示不同的内容，而我们的输入输出，也应当对应现在使用的控制台上，为了解决这些问题，我们新建立两个结构体 `TTY` 和 `CONSOLE` 分别用于存储不同控制台输入缓冲区以及显存信息，在以后的使用中，我们就可以以此为指引，实现控制台的切换。

在完成这一切后，我们需要构建一个控制台切换函数，其主要的功能就是更改现存信息以及输入缓冲区信息，完成这一切后，我们就实现了多控制台的建立。

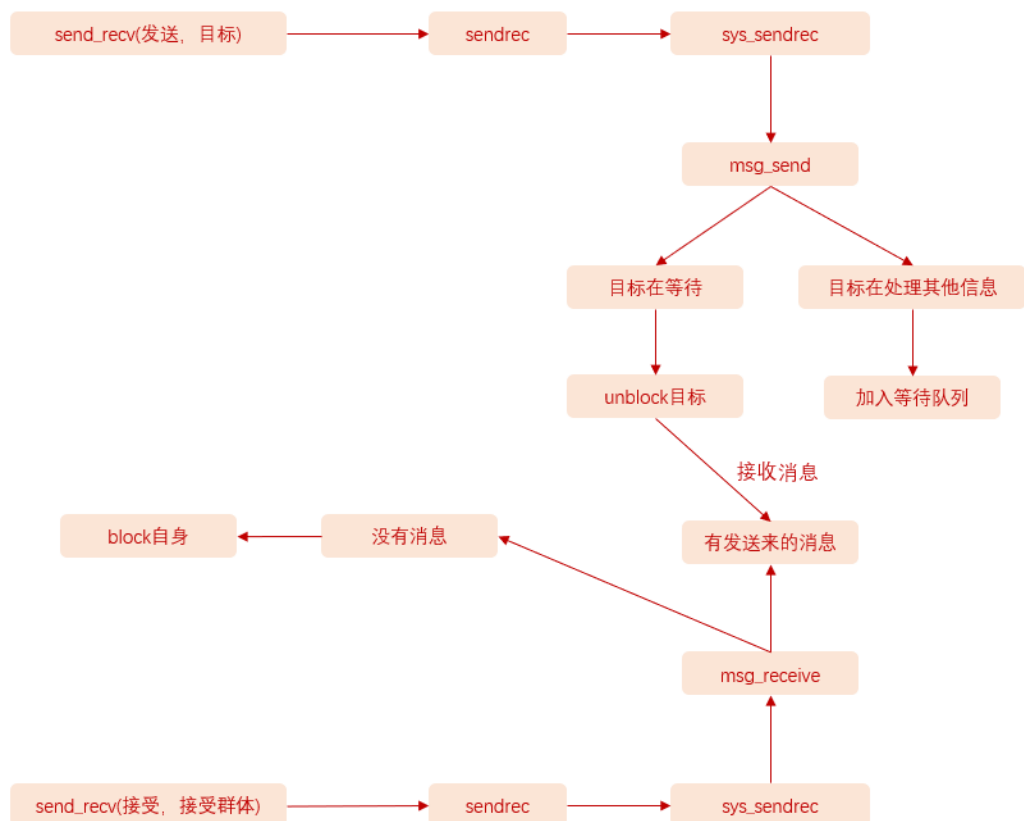
为了是我们能够便捷的切换控制台，我们将增加上述输入系统的功能，我们将组合键“`CTRL + F1/F2/F3`”设定为我们的控制台切换键（原书中使用 `ALT`，但存在与 Linux 键位冲突的问题）。

在建立了多控制台之后，我们就可以着手升级我们的输出系统，使我们的输出系统可以向我们使用的控制台定向输出信息，此时我们构建经典的函数“`printf`”，其通过一系列函数调用确定控制台并输出，具体如下：



3.11、通信机制的建立

为了实现文件系统以及完成更好的进程调度，我们需要加入通信机制的概念，在本书中作者采用了 Minix 系统中 IPC 的思想，构建了一套同步 IPC 系统，大体的运作过程如下：

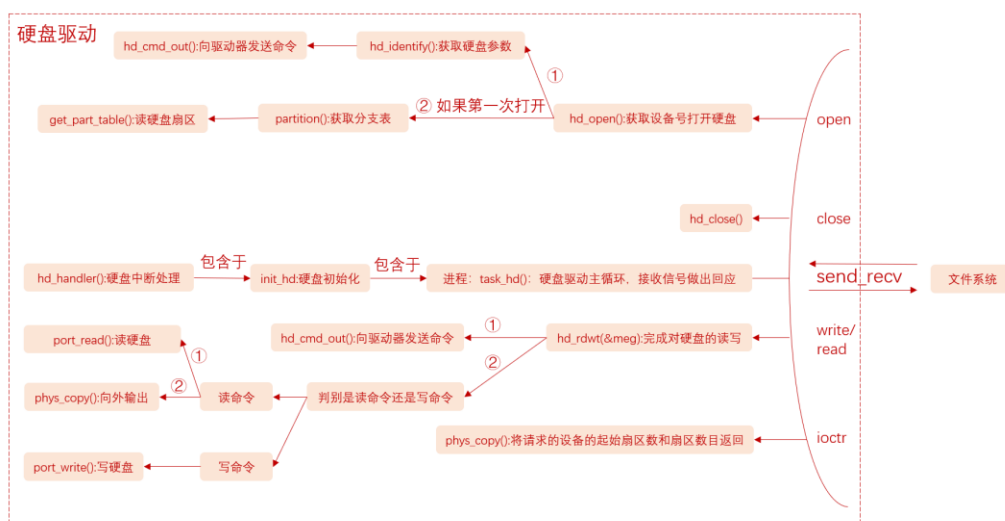


有了这套系统，我们的进程间就有了高效的通讯方式，，除此之外，有了这套系统，我们就可以进行下一步的文件系统的建立。

3.12、文件系统的建立

对于文件系统，由于我们这次是真刀真枪的操作系统设计，我们不可能再在内存或现成硬盘上的一个现成文件里建立文件系统，我们需要从底开始，我们需要从最基础的硬盘开始设计。

与之前的输出系统相似，硬盘也向我们开放了诸多操作 I/O 端口，通过对这些端口的读写，我们得以完成对硬盘信息的获取以及内容读写，以来完成硬盘驱动的搭建，其中具体的过程如图所示：



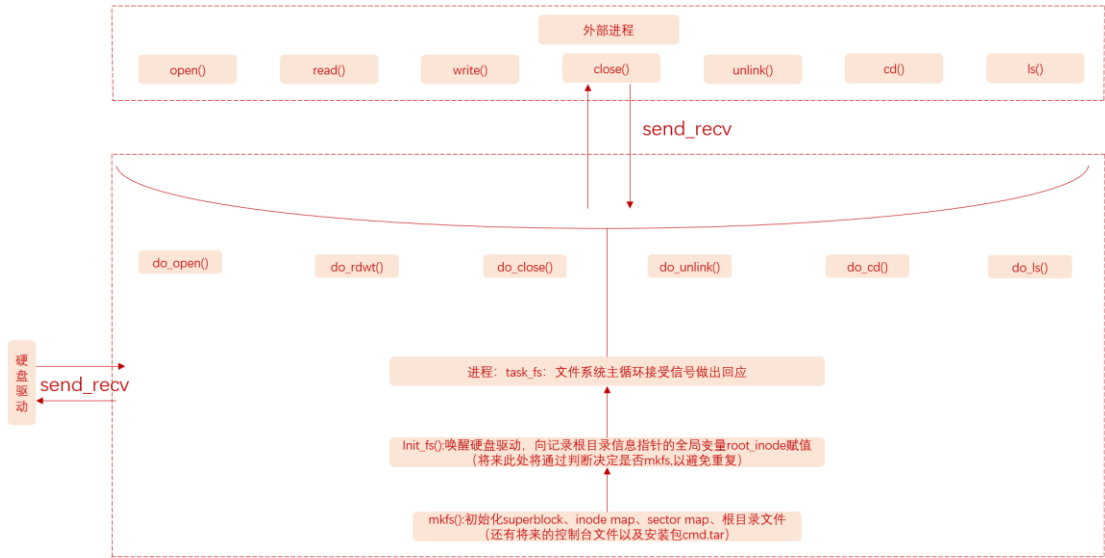
在完成了硬盘驱动以后，我们就有了完成文件系统的基础，但文件系统只有打开、关闭与读写能力是不够的，我们还需要一些记录文件系统的数据结构，为此我们设计了以下四个数据结构。

首先是占整整一个扇区的 superblock，用于记录诸多的最基础的 Metadata，其次我们还设计了一个 sector map 用于记录扇区的使用情况，再其次，我们设计了 inode，用于记录文件的基本信息，另外，在实际使用时，我们会将其串联为一个名为 inode_array 的数组便于管理，除此之外我们还会为其设计一套位图 inode_map 来记录我们 inode_array 的使用情况，最后我们设计了 dir_entry 来存储一个文件的索引和名称，这样我们就可以便捷的在目录文件中存储一个文件的基本信息，以方便我们在目录中快速找到我们需要的文件，更加精妙的是，由于我们在目录中存储了文件的名称，我们的“ls”指令将可以依托于此便捷的完成。

还需要提到的一点是，除了上述描述文件的数据结构，我们还需要，

一个数据结构用以记录打开的文件信息，例如读写到何处以及文件的索引码之类的信息，为此，我们设计了另一个数据结构 `file_desc` 我们称其为进程表，在使用时，我们会将其串为一个数组使用，打开的文件从第 0 位置依次向前排放。在完成这一切后，我们获得了一个意外之喜，我们在文件 `open` 时可以返回此文件在数组中的位置，如此，我们在对文件进行读写时，仅需要以这个位置作为“暗号”，无需输入文件名，这使我们读写函数的调用方便了不少。

在这一切之后，我们将着手于各个文件功能的实现，在我们这个较为简陋的文件中我们将实现的功能为：`open`、`close`、`read`、`write`、`unlink` (删除)、`cd` 与 `ls`，最后的两个功能仿佛看起来不太属于内部的文件系统，且作者也没有实现，但考虑到其需要与文件系统的相应数据互动，故我们还是把他们放在了这里，具体实现如图所示：



对于其中的 `do_open()`，其首先通过函数 `search_file` 来判断文件是否存在，若存在，且 `open` 指令中为进打开不创建，则加入进程表返回在进程表中的位置，若为创建，则返回错误码显示文件已经存在，若文件不存在，且 `open` 指令为不创建，则返回错误码且显示文件不存在，若指令为创建，则使用函数 `create_file()` 创建相应类型文件，并将其加入进程表返回在进程表中的位置。

对于其中的 `do_rdwt()`，其先判断指令的类型为 `read` 还是 `write`，若

为 read, 其直接使用函数 `phys_copy` 从扇区获取信息, 若为 write, 其先使用 `phys_copy` 从信息源获取要写入的内容, 再使用函数 `rw_sector()` 向磁盘写入信息。

对于 `do_close()`, 其任务是简单的, 其主要的任务是由于我们在 `do_open()` 中使用了函数 `search_file` 从而间接调用函数 `get_inode()` 导致这个文件的使用个数被加一, 我们在这里要做的就是使用函数 `put_inode()`, 使使用次数减一。

对于 `do_unlink()`, 我们要做的步骤主要有四项, 分别为释放 `inode map` 中的相应位置, 释放 `sector map` 中的相应位置, 将 `inode_array` 中的相应位置清空, 最后就是在目录中将此项的目录项删除, 至于硬盘上的数据, 我们就不做处理了, 尽管如此有隐私安全隐患, 但我们先不强求我们这个小小的文件系统了。

对于 `do_cd()`, 我们的处理方法是巧妙的, 对于此文件系统来讲, 其原本没有文件夹的概念, 仅有一个根文件夹, 而辨识当前根文件夹的依据就是我们的变量 `root_inode`, 为了改变这种情况, 此时我们建立新的文件标识, 表示此文件为目录文件, 之后, 我们在 `inode` 中新增描述项 `i_father`, 每个文件都会指向其目录文件, 在此之后我们就可以并借着这些基础完成根目录的切换, 当我们建立了一个文件夹文件后, 我们使用 `cd + 文件名`, 我们就会将 `root_inode` 切换为我们新的文件的 `inode`, 当我们使用 `cd + ..` 时, 我们会寻找当前文件夹的 `i_father` 所对应的文件, 然后将 `root_inode` 切换为此文件的 `inode`, 如此, 我们就构建起了文件夹的概念, 也实现了文件夹的进入以及退出。

对于 `do_ls()`, 其获取文件夹下的文件名并不复杂, 参照函数 `search_file()` 我们可以快速的写出一个获取文件名的函数, 但此函数的难点在于, 我们获取了文件名列表之后, 我们的输出是一个巨大的难题, 在 `do_ls()` 的层面中, 由于不是在控制台环境, 我们的 `printf()` 无法确定输出方向, 所以处于一种不可使用的状态, 所以此时我们的选择是将文件名列表 `list` 直接利用内存拷贝进入发送消息的 `ls()` 中的 `msg.PATHNAME`, `ls()` 再返回此字符串地址, 我们再通过 `printf()` 将其打印出来, 尽管这算不上

很体面的解决方案，但能够使用就也算是成功。

至此，我们的文件系统就建立了起来，我们的文件系统初具样貌但也有一个不太好的问题——我们为了扇区管理方便我们采用了固定文件大小的方式，我们也曾有过在磁盘中建立页式管理机制来实现可变大小文件的想法，但限于自身水平以及对硬盘的理解，我们没能实现这个想法，这个想法可以作为后面继续研究的方向。

3.13、升级控制台

从很多程度上来讲，我们的控制台其实也是一个文件，只不过连接的不是硬盘而是显存，控制台的显示就是文件的 `write()` 即被写入，控制台命令的读入就是文件的 `read`，从这个角度来讲，将控制台纳入我们的文件系统未必不是一个好想法，而 Linux 也正是这么做的。

在书中，作者将文件系统作为了控制台与进程间的传声筒，自此之后，我们只要通过文件的 `open` 即可打开相应的控制台，我们利用文件的 `read` 即可读入用户的输入，我们利用 `write()` 即可向控制台发送显示信息，这无疑是十分便利的。

3.14、使用 `fork()` 函数创建进程

为了进一步完善我们的我们的操作系统的功能，我们希望能够完成一个功能简单的 shell。Shell 的原理并不复杂，其实质其实就是在一个进程中，调用一个子进程在执行一个命令。在此之前我们对控制台已经十分熟悉了，shell 的功能和我们已经实现的控制台有很多相似之处，但是子进程对于我们而言还是一个比较陌生的概念。到目前为止，我们的操作系统运行的所有进程都是在编译时被确定好，并指定其入口地址和堆栈。创建一个新的进程需要以下要素：

自己的代码，数据和堆栈；

在 `proc_table[]` 中占用一个位置；

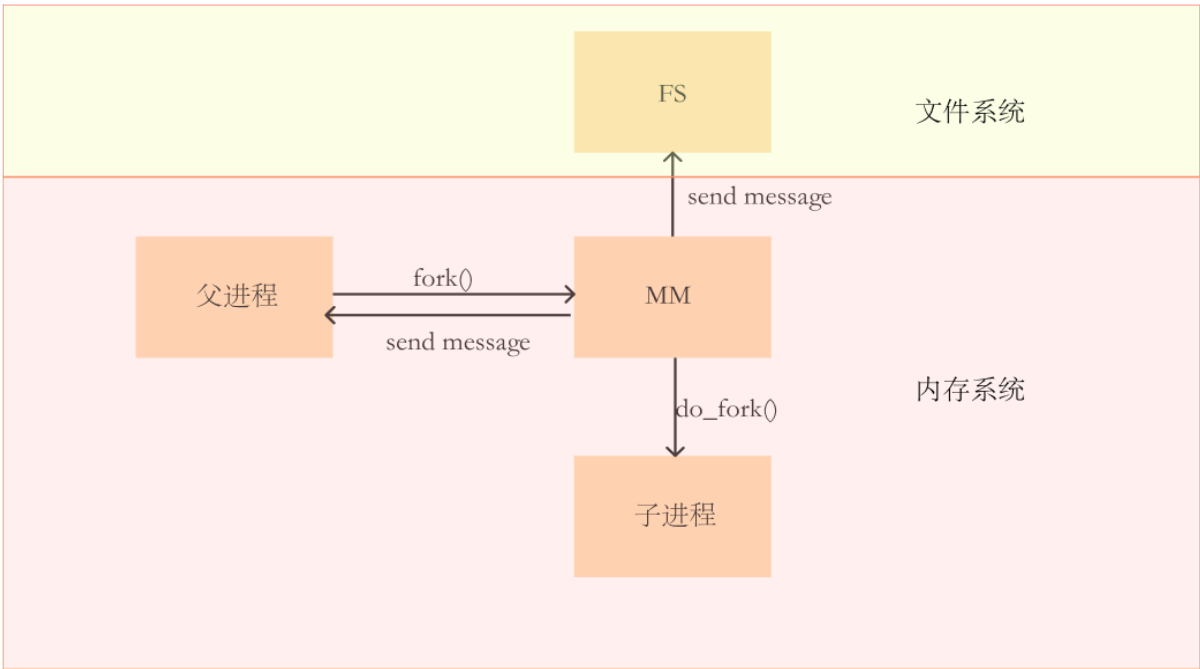
在 GDT 中占用一个位置，用以存放进程对应的 LDT 描述符。

后两个要素我们都能够很容易的实现，至于代码和堆栈的获取，通常来说，我们目前普遍采用的方法是从一个已经存在的进程那里继承或者复

制，以产生一个新进程。假设该已存在的进程为进程 P，产生的新进程为进程 C，则进程 P 被称为父进程，进程 C 被称为子进程，产生子进程的这一系统调用就是 `fork()`。

产生一个新进程的工作不需要全部由 `fork()` 来完成：首先我们在 `proc_table[]` 中预留新进程的空项，这一过程通过改变宏 `NR_PROCS` 来实现；同时我们在 `init()` 中初始化所需的 GDT 项和 `proc_table[]` 中的参数。同 FS 处理过程一样，在我们调用 `fork()` 时，操作系统会向 MM 发送一个 `fork` 消息，MM 接到消息后，调用 `do_fork()` 进行处理，在 `do_fork()` 中首先分配进程表，然后分配内存，之后要向 FS 发送信息进行相关的文件处理，在这些完成之后，为了使父进程和子进程之间能够相互区分（否则进程将无从得知自己是父进程还是子进程），还需要将 0 作为返回值传递给子进程，对于父进程在函数返回后由 MM 中的消息循环将赋值后的 PID 值发送给父进程。由此我们就知道了 `fork()` 函数的目的和实现原理。

实现示意图如图所示：



3. 15、内存分配

一个成熟的内存分配机制通常十分复杂，在此为了降低实现的难度，我们在此使用简单的机制：我们将内存空间划分为若干个大小相当的格子，每个格子的大小相当且固定，当有新的进程产生需要分配内存空间

时，就给他一个格子，这一分配是静态的，即在进程的生命周期内只能使用这个格子的内存。在此，我们把格子的大小定为 1MB，这一数字不是必定的，而是可以在大于内核大小 240KB 的范围内任意选取合适大小。我们的操作系统总共有 32MB 内存大小，这就意味着最多可以存在 32 个进程。

这种分配方案的缺点是十分明显的，对于较小的进程而言，会产生较大的内碎片，即内存空间浪费，较大的进程又无法运行。因此在对操作系统进一步的优化中，内存分配方案无疑是需要重视的。

3.16、exit() 和 wait() 函数实现进程终止

我们已经有了创建新进程的系统调用 `fork()`，但是还不够，因为一个进程有产生就有终止。让进程死亡的系统调用叫做 `exit()`，直译作“退出”，但其实它叫“自杀”更贴切，因为 `exit()` 通常是由要进程自己调用的，而且调用之后这个进程不是“退出”了，而是干脆消失了(把进程表的进程状态设置成 `FREESLOT`，释放其进程表项，清除该进程占用的系统资源，此后不会再参与进程调度了)。

`wait()` 系统调用的作用则在于从子进程向父进程传递信息。当我们执行一个程序之后，有时需要判断其返回值，当返回值被获取时，执行的程序显然已经执行完毕(所以它才有返回值)。容易理解，这个返回值是我们执行的进程返回给 `shell` 的。换言之，是子进程返回给父进程的。父进程得到返回值的方法，就是执行一个 `wait()` 挂起，等待子进程结束并返回。等子进程 `exit()` 时，`wait()` 调用方结束，并且父进程因此得到返回值。

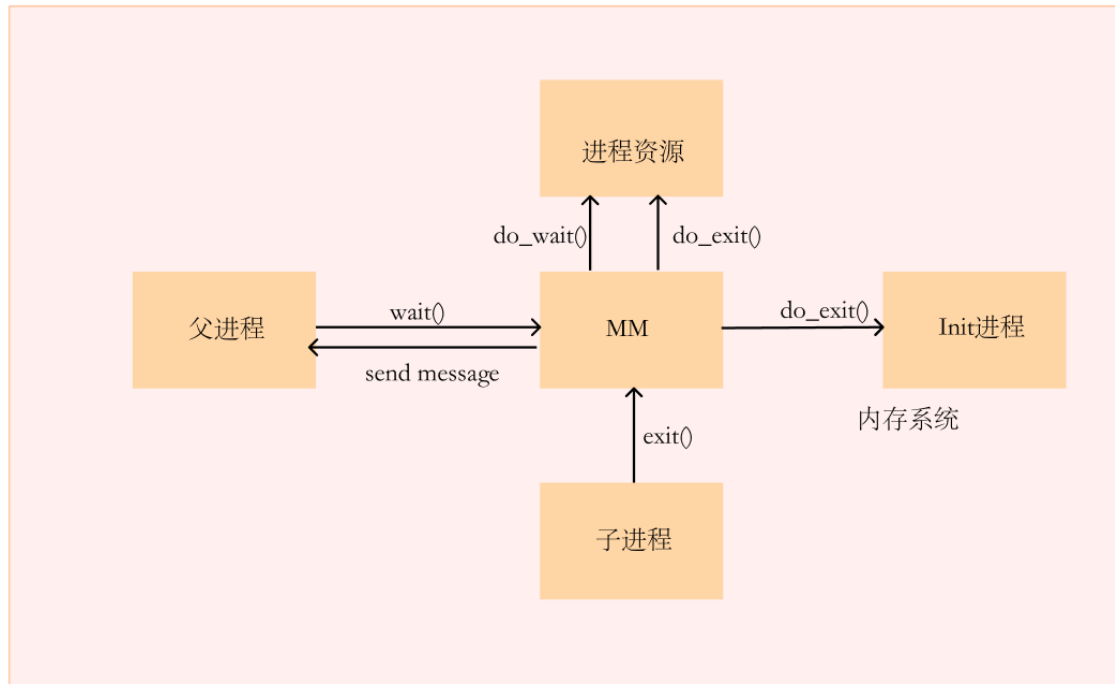
`exit()` 和 `wait()` 就是让子进程传递给父进程一个状态值使用的。子进程必须主动调用 `exit()` 生成返回值，父进程必须主动调用 `wait()` 获取子进程的返回值。为了完成这个功能，需要 MM 中的 `do_exit()` 和 `do_wait()` 函数配合处理相应的逻辑。

`do_exit/do_wait` 跟 `msg_send/msg_receive` 这两对函数是有点类似的，它们最终都是实现一次“握手”。

为配合 `exit()` 和 `wait()`，进程又多了两种状态：`WAITING` 和 `HANGING`。如果一个进程 `X` 被置了 `HANGING` 位，那么 `X` 的所有资源都已被释

放，只剩一个进程表项还占着。之所以 X 进程已经被终止但却保留他的内存表项，是因为这个进程表项里面有个新成员：exit_status，它记录了 X 的返回值。只有当 X 的父进程通过调用 wait() 取走了这个返回值，X 的进程表项才能够被释放。

实现示意图如图所示：



如果一个进程 Y 被置了 WAITING 位，意味着 Y 至少有一个子进程，并且正在等待某个子进程退出。

当如果一个子进程 Z 试图退出，但它的父进程却没有调用 wait() 时，那 Z 的进程表项可能会一直被占用而得不到释放。有个名称专门用来称呼像 Z 这样的进程，叫做“僵尸进程”（zombie）。如果一个进程 Q 有子进程，但它没有 wait() 就自己先 exit() 了，为了不让 Q 的子进程变成 zombie，我们需要使 MM 会把这些子进程过继给 Init，使其变成 Init 的子进程。进一步地，我们再将 Init 的主循环设计成不停地调用 wait()，以便让过继得到的 Q 的子进程们退出并释放进程表项。

因此我们可以看出，对不同状态下的进程执行 exit() 调用，MM 也会进行不同的操作，不仅如此，还会受到 exit()、wait() 调用顺序等诸多因素的影响，因此在此处的处理需要认真留意。

3.17、编写用户级应用程序

无论 Init 进程 fork 出多少进程，它也都只是 Init 进程而已。所以我们还需要一个系统调用，使其能够将当前的进程映像替换成另一个。也就是说，我们可以从硬盘上读取另一个可执行的文件，用它替换掉刚刚被 fork 出来的子进程。因此我们首先至少要有这么一个应用程序以供加载。那么我们要完成的第一个目标就是编写自己的应用程序。

以 shell 中常见的 echo 命令为例。我们输入“echo hello world”，shell 就会 fork 出一个子进程 A，这时 A 跟 shell 一模一样，fork 结束后父进程和子进程分别判断自己的 fork() 返回值，如果是 0 则表明当前进程为子进程 A，这时 A 马上执行一个 exec()，于是进程 A 的内存映像被 echo 替换，它就变成 echo 了。

echo 将以操作系统中普通应用程序的身份出现，它跟操作系统的接口是系统调用。其实本质上，一个应用程序只能调用两种东西：属于自己的函数，以及中断（系统调用其实就是软中断）。可是根据我们写程序的经验，一个应用程序通常都会调用一些现成的函数，很少见写程序时里面满是中断调用的。这是因为编译器偷偷地为我们链接了 C 运行时库（CRT），库里面有已经编译好的库函数代码。这样两者链接起来，应用程序就能正确运行了。

假如我们要写一个 echo，最笨的办法就是将 send_recv()、printf()、write() 等所有用到的系统调用的代码都复制到源文件中，然后编译一下。这肯定是能成功的，但是代码中的重复、冗杂和内存资源的浪费因此出现，这显然是我们所不希望的。更优雅的做法是制作一个类似 C 运行时库的东西。我们把之前已经写就的应用程序可以使用的库函数单独链接成一个文件，每次写应用程序的时候直接链接起来就好了。

借助库，我们便能够更轻松的完成应用程序的编写。值得注意的是，在编译链接我们的应用程序前，还需要做好以下准备：为 main() 函数准备参数 argc 和 argv；调用 main()；将 main() 函数的返回值通过 exit() 传递给父进程。我们通过添加 start.asm 来完成这一任务。

编译成功后的应用程序被打包成 .tar 格式安装到文件系统的特定扇区

中，并在 `init` 进程中将 `.tar` 文件解包，使其中包含的应用程序被存入创建的文件中。在这之前，我们需要在 `mkfs()` 中做好准备，增加一个文件，为 `.tar` 文件预留位置。

完成后编写的应用程序便会出现在我们的文件系统中，供我们执行。接下来我们便要实现 `exec()` 系统调用，将应用程序加载到新创建的进程当中。

3.18、`exec()` 加载进程

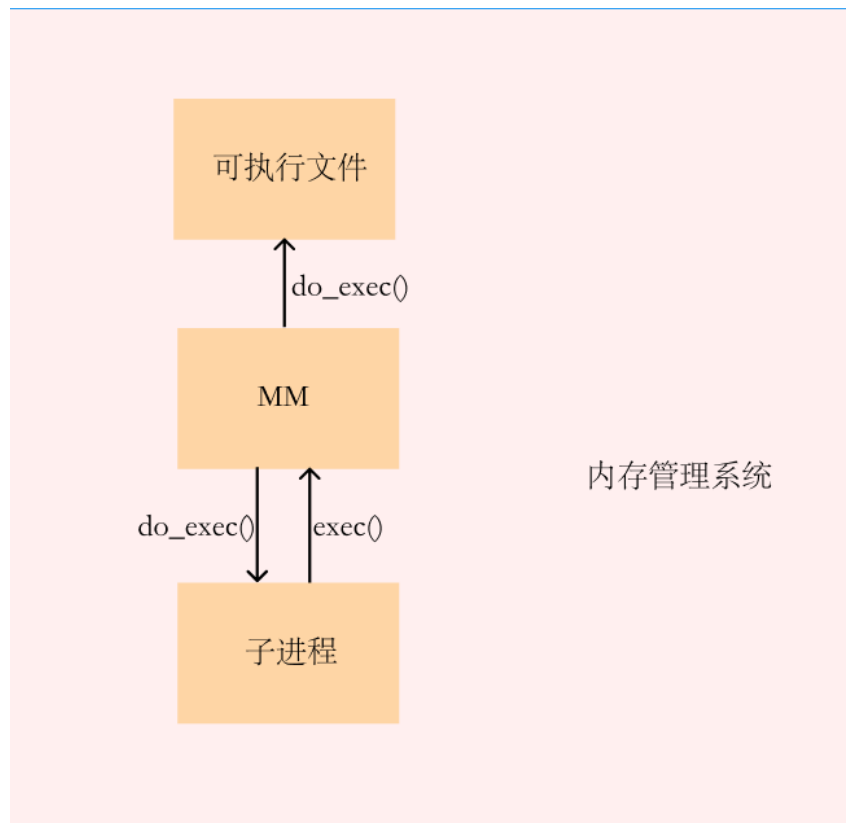
我们在此选择 `execl()` 和 `execv()` 来实现，而 `execl()` 最终调用 `execv()`。究其本质，`execv()` 所作的事只有一件，那就是向 MM 提供最终供调用 `exec()` 的进程使用的堆栈。具体来说，先准备好一块内存 `arg_stack[]`，然后完成以下工作：

遍历其调用者（比如 `execl()`）传递给自己的参数，数一数参数的个数；将指针数组的末尾赋零；遍历所有字符串：先将字符串复制到 `arg_stack[]` 中，再将每个字符串的地址写入指针数组的正确位置。

这项工作做完之后，它将 `arg_stack[r]` 的首地址以及其中有效内容的长度等内容通过消息发送给 MM，使 MM 能够进行实际的 `exec` 操作。

MM 进程在收到消息后调用 `do_exec()` 函数，`do_exec()` 函数的主要操作就是首先从接收到的消息体中获取参数，然后通过一个新的系统调用 `stat()`，获取被执行文件的大小。打开被执行文件，把文件全部读入到 MM 的缓冲区中，并且安装 ELF 文件格式移动文件，根据 ELF 文件的程序头信息，准备好堆栈内容，并且设置好 `eip` 等于 ELF 文件入口地址，这样进程调度程序调度到该进程时，就会从入口地址开始执行，接下来为程序的 `eax`，`ecx`，`esp` 等赋值，最后将进程的名字改成被执行政程序的名字。此时应用程序便通过 `exec()` 系统调用被加载到了新创建的子进程当中。

实现示意图如图所示：



4.1.2、help 界面

该界面显示指令及其对应详细功能的说明。介绍我们实现的应用及其运行方法，对用户访问用户级应用进行引导。

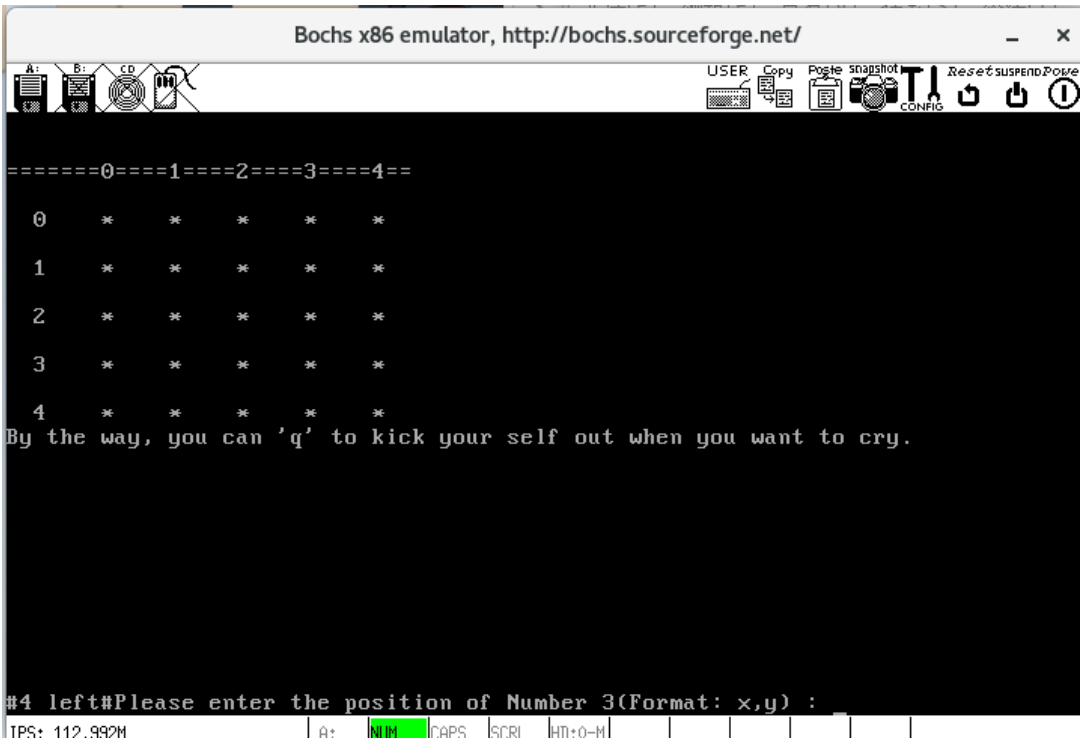


4.1.3、应用 maze 迷宫游戏

该界面是迷宫游戏界面，通过 W, A, S, D 方向键进行人物控制，走出迷宫，最多支持 0 - 4 物种难度选择。到达底部终点即为游戏胜利，你也可以使用 q 键退出游戏。



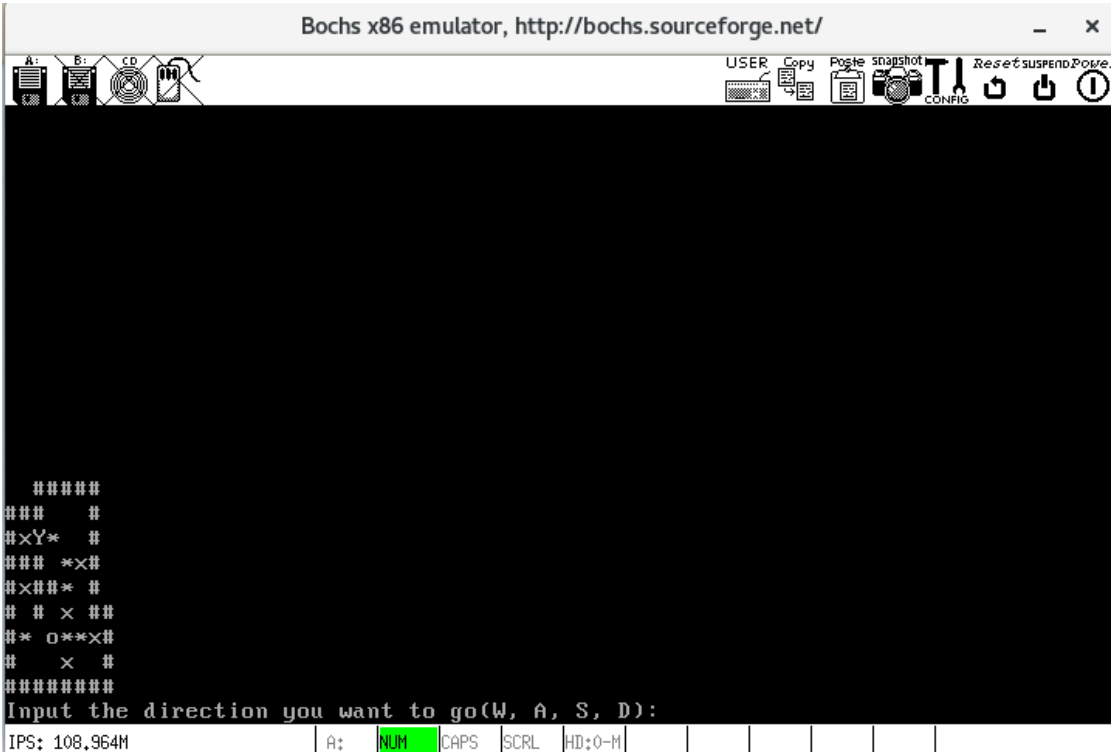
4.1.4、应用 memory 记忆游戏



该界面是记忆游戏界面，游戏开始时向玩家展示一个包含数字的随机生成的矩阵，在给予玩家一定时间记忆后遮挡并给出某一数字，玩家需要输入数字出现的位置，输入全部位置后即可获得游戏胜利。你也可以使用 q 键退出游戏。游戏提供 0 - 2 三种难度。

4.1.5、应用 sokoban 推箱子

该界面是推箱子游戏界面，Y 为玩家位置，*表示箱子，#则为无法通过的墙壁，玩家需要推动*箱子到标志为 x 的终点处，通过 W, A, S, D 方向键进行人物控制。当所有箱子都到达终点时，玩家获得游戏胜利。你同样也可以 q 键退出游戏。



4.2.2.3、进程管理创建进程

展示操作系统创建新进程的功能，由于创建进程后没有对应程序执行，仅作为功能展示，因此设置进程在被创建后完成输出后 `exit()`。

```
Bochs x86 emulator, http://bochs.sourceforge.net/

# $ clear i clear the cmd #
# You can use [CTRL + F1/F2/F3] to switch consoles #
=====
@RASPBERRYOS ~ process-manager: $ ps
=====
ProcessID * ProcessName * ProcessPriority * If Running
=====
0 TTY 15 yes
1 SYS 15 yes
2 HD 15 yes
3 FS 15 yes
4 MM 15 no
5 INIT 5 yes
6 TestA 5 no
7 TestB 5 no
8 TestC 5 no
9 INIT_9 5 yes
10 INIT_10 5 yes
=====
@RASPBERRYOS ~ process-manager: $ create
[parent is running, child pid:11]
[child is running, pid:11]
[Child 11 exited with status: 666.]
@RASPBERRYOS ~ process-manager: $
```

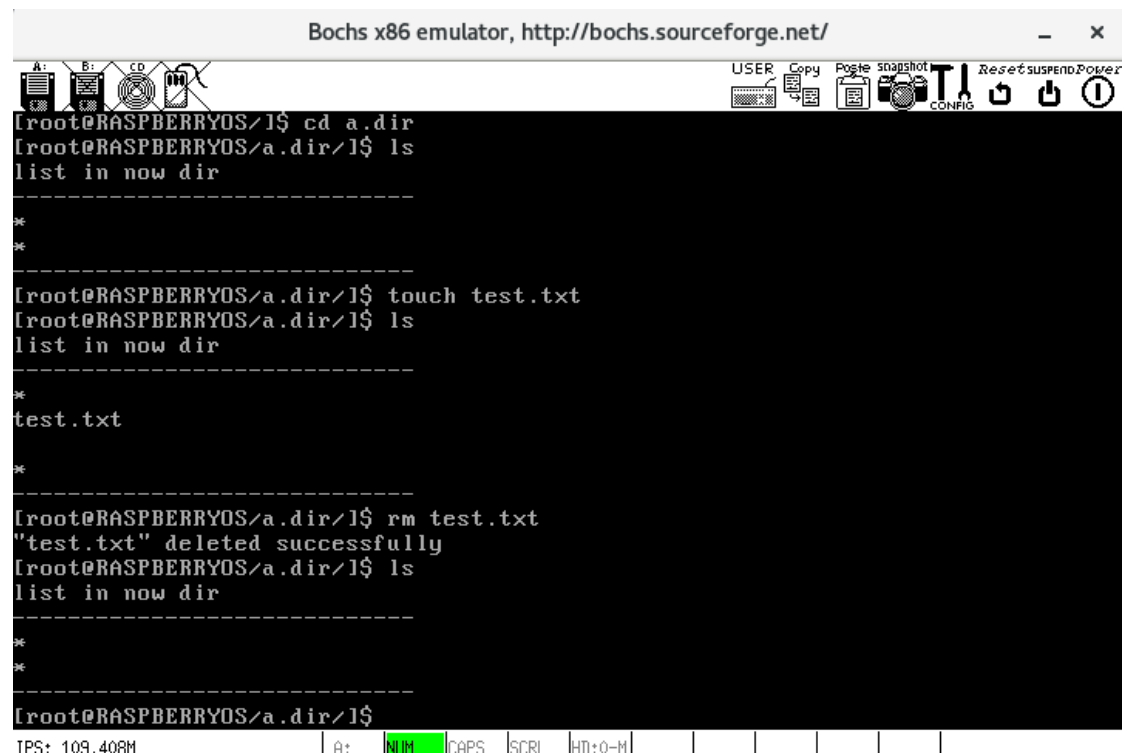
4.2.2.4、进程管理结束进程

使用 kill pid 指令可以使进程终止

[illegible]

4.2.3、文件系统

4.2.3.1、文件系统——touch&rm



```
Bochs x86 emulator, http://bochs.sourceforge.net/

[ root@RASPBERRYOS / ]$ cd a.dir
[ root@RASPBERRYOS/a.dir / ]$ ls
list in now dir
-----
*
*
-----

[ root@RASPBERRYOS/a.dir / ]$ touch test.txt
[ root@RASPBERRYOS/a.dir / ]$ ls
list in now dir
-----
*
test.txt
*
-----

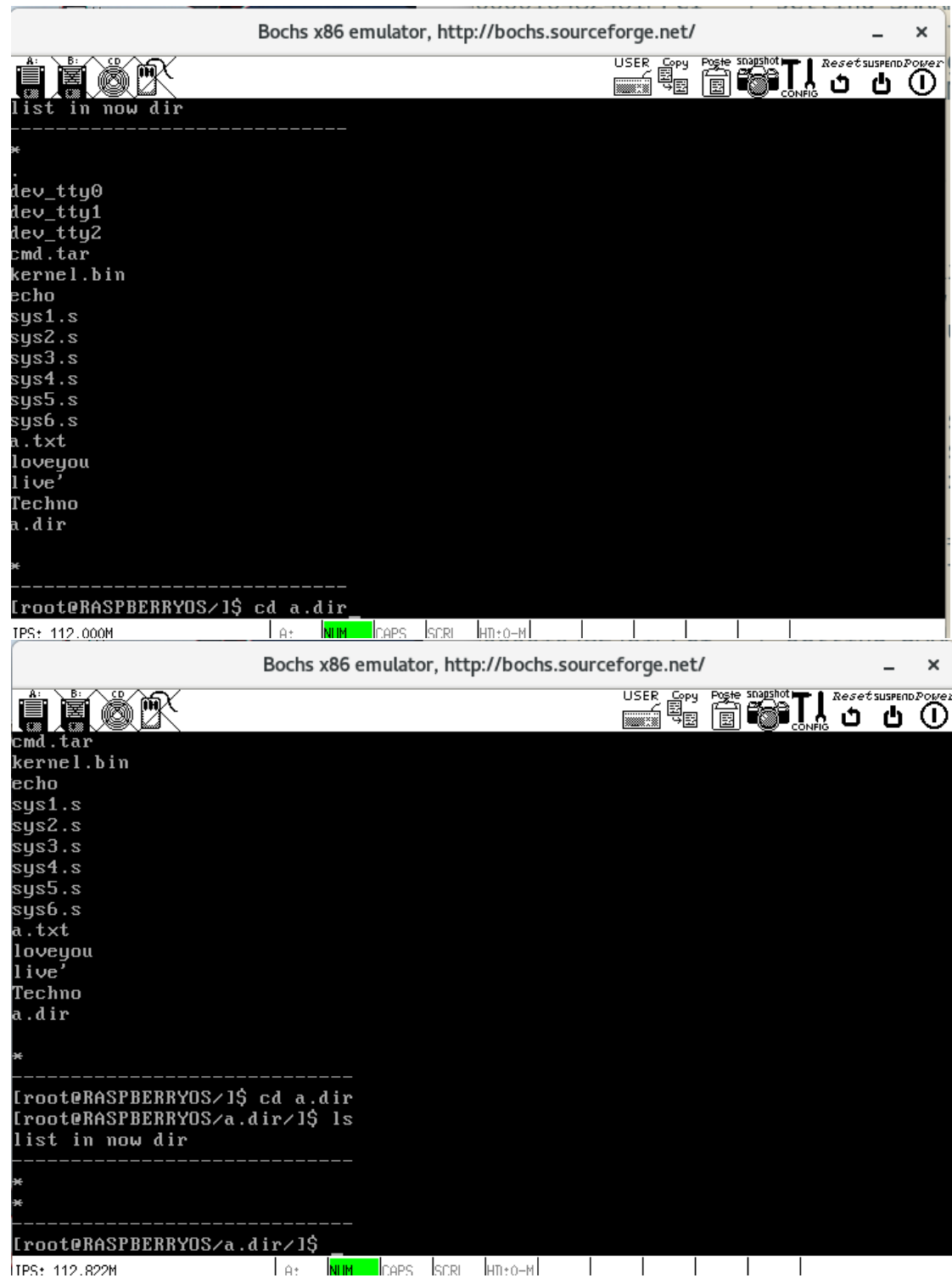
[ root@RASPBERRYOS/a.dir / ]$ rm test.txt
"test.txt" deleted successfully
[ root@RASPBERRYOS/a.dir / ]$ ls
list in now dir
-----
*
*
-----

[ root@RASPBERRYOS/a.dir / ]$
```

输入 touch XXX.XXX 即可创建文件名为 XXX，对应的使用 rm XXX.XXX 即可将该文件删除。

4.2.3.2、文件系统——cd

输入 `cd XXX` 打开文件夹，转移当前目录进入下一级的 XXX 文件。通过 `cd ..` 指令能够回退到上一级目录。

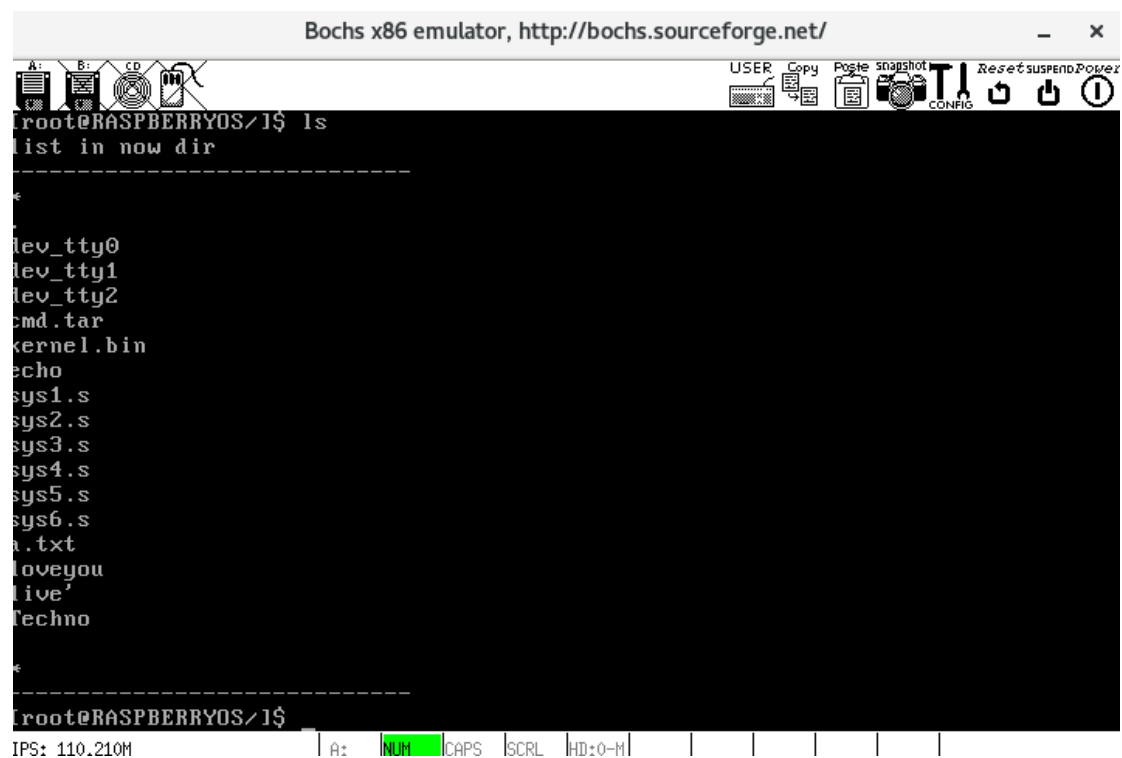


```
Bochs x86 emulator, http://bochs.sourceforge.net/
A: B: CD
[root@RASPBERRYOS/1$ cd a.dir
[root@RASPBERRYOS/a.dir/1$ ls
list in now dir
-----
*
*
-----
[root@RASPBERRYOS/a.dir/1$ touch test.txt
[root@RASPBERRYOS/a.dir/1$ ls
list in now dir
-----
*
test.txt
*
-----
[root@RASPBERRYOS/a.dir/1$ rm test.txt
"test.txt" deleted successfully
[root@RASPBERRYOS/a.dir/1$ ls
list in now dir
-----
*
*
-----
[root@RASPBERRYOS/a.dir/1$ cd ..
IPS: 111.363M | A: NUM | CAPS | SCRL | HD:0-M |
```

```
Bochs x86 emulator, http://bochs.sourceforge.net/
A: B: CD
list in now dir
-----
*
.
dev_tty0
dev_tty1
dev_tty2
cmd.tar
kernel.bin
echo
sys1.s
sys2.s
sys3.s
sys4.s
sys5.s
sys6.s
a.txt
loveyou
live'
Techno
a.dir
*
-----
[root@RASPBERRYOS/1$
IPS: 111.646M | A: NUM | CAPS | SCRL | HD:0-M |
```


4.2.3.3、文件系统 ls

输入 ls 显示当前目录下的文件信息。

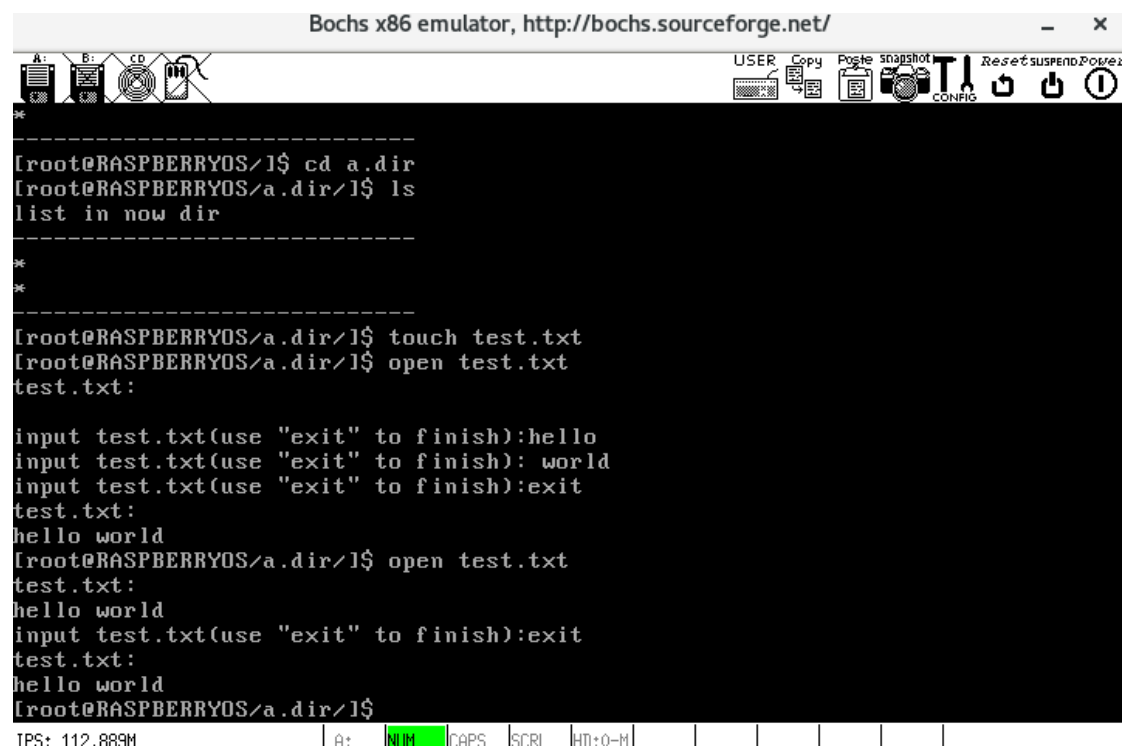


The screenshot shows the Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The menu bar includes "USER", "Copy", "Paste", "Snapshot", "CONFIG", "Reset", "SUSPEND", and "Power". The main window displays a terminal session with the following text:

```
[root@RASPBERRYOS/1]$ ls
list in now dir
-----
*
dev_tty0
dev_tty1
dev_tty2
cmd.tar
kernel.bin
echo
sys1.s
sys2.s
sys3.s
sys4.s
sys5.s
sys6.s
a.txt
loveyou
live'
Techno
*
-----
[root@RASPBERRYOS/1$
```

At the bottom of the window, the status bar shows "IPS: 110.210M" and a row of icons for "A:", "NUM", "CAPS", "SCRL", and "HD:0-M".

4.2.3.4、文件系统 open



The screenshot shows the Bochs x86 emulator window. The title bar reads "Bochs x86 emulator, http://bochs.sourceforge.net/". The menu bar includes "USER", "Copy", "Paste", "Snapshot", "CONFIG", "Reset", "SUSPEND", and "Power". The main window displays a terminal session with the following text:

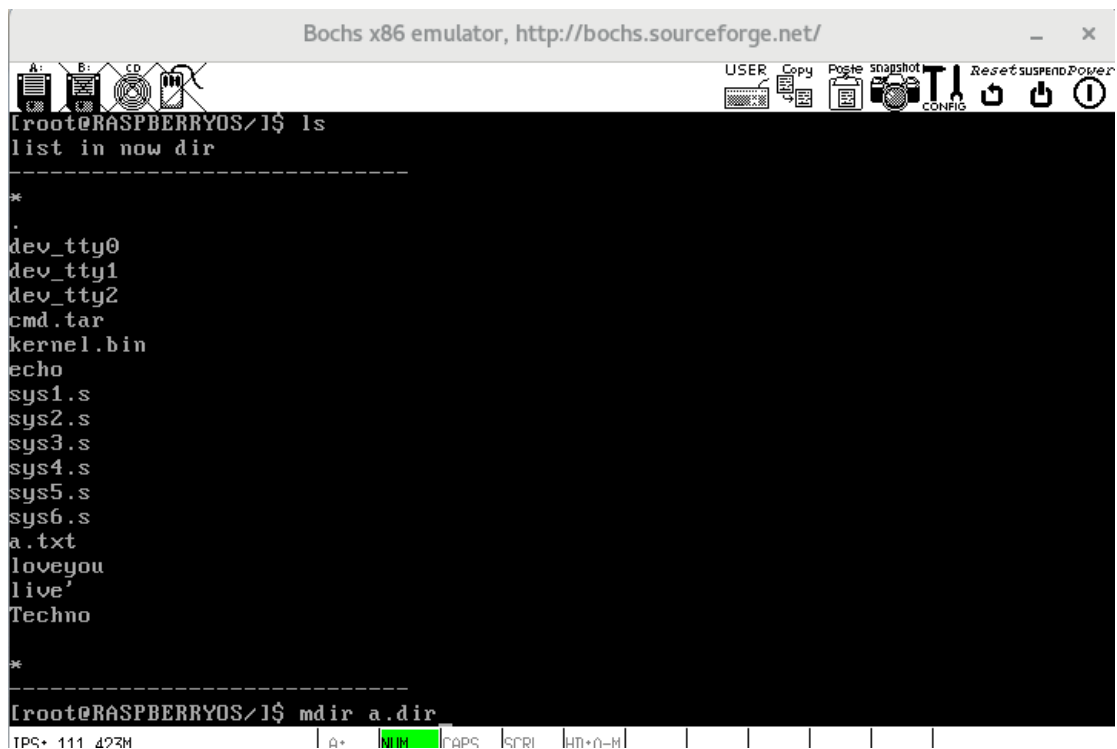
```
[root@RASPBERRYOS/1$ cd a.dir
[root@RASPBERRYOS/a.dir/1$ ls
list in now dir
-----
*
*
-----
[root@RASPBERRYOS/a.dir/1$ touch test.txt
[root@RASPBERRYOS/a.dir/1$ open test.txt
test.txt:
input test.txt(use "exit" to finish):hello
input test.txt(use "exit" to finish): world
input test.txt(use "exit" to finish):exit
test.txt:
hello world
[root@RASPBERRYOS/a.dir/1$ open test.txt
test.txt:
hello world
input test.txt(use "exit" to finish):exit
test.txt:
hello world
[root@RASPBERRYOS/a.dir/1$
```

At the bottom of the window, the status bar shows "IPS: 112.889M" and a row of icons for "A:", "NUM", "CAPS", "SCRL", and "HD:0-M".

open 指令完成对某文件的编辑和访问等操作。

4.2.3.5、文件系统——mdir

输入 `mkdir XXX` 创建名为 XXX 的文件夹。



The screenshot shows a Bochs x86 emulator window titled "Bochs x86 emulator, http://bochs.sourceforge.net/". The terminal window displays the command `ls` and its output, listing files in the current directory. The files listed are: `dev_tty0`, `dev_tty1`, `dev_tty2`, `cmd.tar`, `kernel.bin`, `echo`, `sys1.s`, `sys2.s`, `sys3.s`, `sys4.s`, `sys5.s`, `sys6.s`, `a.txt`, `loveyou`, `live'`, and `Techno`. The status bar at the bottom shows "IPS: 111,423M" and various keyboard status indicators.

```
Bochs x86 emulator, http://bochs.sourceforge.net/
[root@RASPBERRYOS/1$ ls
list in now dir
-----
*
.
dev_tty0
dev_tty1
dev_tty2
cmd.tar
kernel.bin
echo
sys1.s
sys2.s
sys3.s
sys4.s
sys5.s
sys6.s
a.txt
loveyou
live'
Techno
*
-----
[root@RASPBERRYOS/1$ mkdir a.dir
IPS: 111,423M
```



The screenshot shows the same Bochs x86 emulator window after the `mkdir a.dir` command has been executed. The terminal window displays the command `ls` and its output, which now includes the newly created directory `a.dir` at the bottom of the list. The status bar at the bottom shows "IPS: 112,243M" and various keyboard status indicators.

```
Bochs x86 emulator, http://bochs.sourceforge.net/
list in now dir
-----
*
.
dev_tty0
dev_tty1
dev_tty2
cmd.tar
kernel.bin
echo
sys1.s
sys2.s
sys3.s
sys4.s
sys5.s
sys6.s
a.txt
loveyou
live'
Techno
a.dir
*
-----
[root@RASPBERRYOS/1$
IPS: 112,243M
```

5、 项目分工

在项目完成过程中，小组两位成员都积极有效地参与学习和讨论，并认真负责地完成自己的分工任务。本项目也因此得以顺利完成。具体分工内容如下：

内容学习、架构搭建、文档编写：共同

文件系统的优化改写和应用实现：马启越

进程管理、界面设计、用户级应用：胡顺

6、 源码链接

<https://github.com/Pepsi-berry/Raspberry-OS>