**DamnOS：一个简单的自制POSIX接口操作系统演示说明**

叶伟国　14301035

数据科学与计算机学院　超算专业

**简要介绍**

这是一个基于i386体系结构的简单的POSIX接口操作系统，实现了内存动态分配，中断与系统调用的 处理，进程调度与通信，文件系统操作，和基本的命令行交互。同时实现了可供用户程序使用的除了 exec外POSIX要求的系统库函数，包括open, seek, stat，read, write, close, fork, dup, wait等。由 于没有实现exec，不能运行文件系统里的程序，因此一些基本的命令都集成在用户shell里实现，包括 cd，ls, pwd, mkdir, rm, echo, cat等，还有包括了一个演示进程通信的命令pingpong。用户shell中还 实现了重定向和管道。但由于没有实现exec，因此管道暂时无法演示。

**实验环境**

操作系统：ubuntu 16.04

虚拟机：qemu-system-i386

**操作说明**

请看Readme.md

**系统结构说明**

**1.基础结构和内核模块：**

bootloader模块：boot.S main.c

完成初始化寄存器和临时全局描述符表、开启保护模式和加载内核的工作。

系统初始化模块：entry.S entrypgdir.c init.c

完成为系统初始化建立临时环境（开启分页，分配初始页目录等），初始化内核模块，最终建立系 统环境。

显示模块：console.c

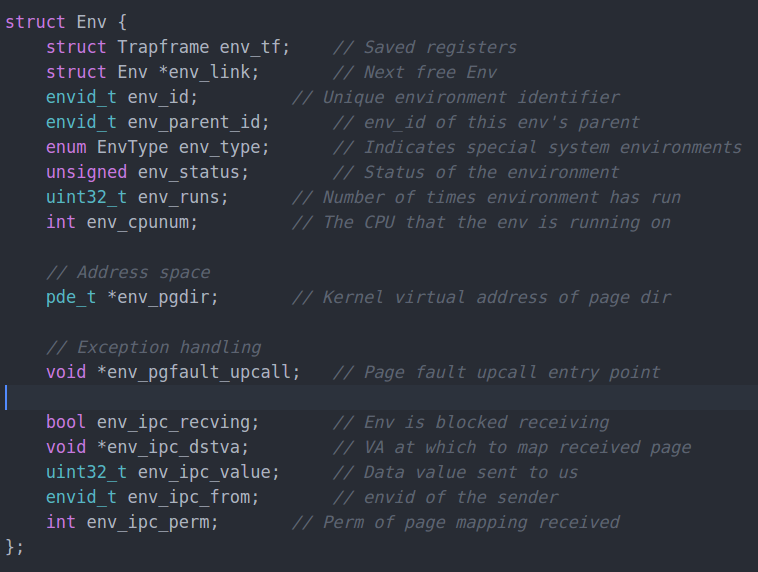
同时建立串口和cga的显示模式，处理屏幕输出，建立缓冲区处理键盘输入。

内存映射模块：pmap.c

分配新页目录和页表，完成内存布局，可完成分配新页，映射新页，解除映射等操作。

进程控制模块：env.c sched.c

完成创建进程，初始化进程，切换进程，删除进程等操作。进程的定义如下，属性描述可看注释。



其中32位的进程号的定义如下。符号位永远是０，用于区别返回的错误信息（都为负数）。而低 10位用来表示该进程在进程向量中的序号，剩下的21用来区别序号一样的不同进程（例如先释放 后又在原序号上创建），因此每次创建进程时都要为Uniquefier加１，也就是加1024。

+1+---------------21-----------------+--------10--------+

|0| Uniqueifier | Environment |

| | | Index |

+--------------------------------------+--------------------+

\--- ENVX(eid) --/

中断与系统调用控制模块：trap.c picirq.c

完成pic的初始化，完成中断和系统调用的处理。

内核锁：spinlock.c

当用户进程通过某种方式进入内核时（中断，系统调用等），锁住内核，保证每次只有一个用户进 程进入内核，不过后来意识到由于只有一个CPU在运行，而内核又是在关中断的情况下运行的， 所以应该也不会同时有两个用户程序进入内核，因此这个模块基本没用了，但对于多核来说还是必 要的。

内核终端模块：monitor.c

当没有用户程序运行时（用户shell崩溃）进入内核终端，可查看简单的内核elf的信息。

**2.文件系统结构**

这是一个叫做Damnfs的文件系统，它参考了UNIX文件系统的设计，由superblock记录文件系统的 全部信息，例如块大小，块数量等，每个文件由一个inode代表，其中包含的信息有文件大小，占用块 的数量，每个占用块的块号等。每个目录也是一个文件，包含有该目录下所有条目，一个条目包含有文 件名和文件所指的文件的inode号。简单起见，我们目前不需要储存过大的文件，因此inode并没有设 计间接块，因此每个文件最多可以只拥有１３个块（足够多了）。同时也为了简单起见，inode和 block的数量都不多，因此bitmap可以直接储存在superblock中，又考虑到该文件系统所在磁盘分区 不需要boot（bootblock 和 kernel储存在另一个分区中），因此可以不预留启动快，因此superblock 直接占用所在分区的第一个块（文件系统意义上的数据块），而所有inode信息储存在第二个块。

**ａ．文件系统布局**

｜ superblock ｜ inodes | otherblocks … |

具体的superblock，inode，目录条目的定义可以查看Readme.md

**ｂ．文件系统处理程序**

这里将对文件系统的操作安排给一个用户程序，作为一个文件系统服务端利用进程通信接收来自其 他用户程序的请求，并完成相应的文件操作后返回相应的信息。这个服务端程序是系统第一个创建 的进程，具有读写磁盘的权限，且屏蔽中断，当它完成文件操作重新接收请求时才可被调用。这是 这个程序独有的。当其他文件希望读数据时

文件系统server：serv.c

完成来自其他用户程序的文件请求。

------------------------------------------------------------

文件系统文件与目录处理： damnfs.c：file\_\*, dir\_\*

操作文件与目录，处理路径。

--------------------------------------------------------------

文件系统数据块和inode处理：damnfs.c：\*block\*, \*inode\*

数据块和inode的分配与回收。

--------------------------------------------------------------

文件系统磁盘页缓存：damnfs.c：bc\_\*, \*flush\*

将磁盘空间（逻辑上）映射到相应的内存空间，通过虚拟

地址访问磁盘数据，当发生缺页时，调用相应的缺页处理handler，

将磁盘数据块读到相应的物理页上，并将相应的虚拟地址映射到该物理页。

在适当的时候将物理页上的内容写回磁盘。为了简单起见，考虑到磁盘空间

较小，物理内存较大（几乎为磁盘大小的一半），物理页一般不会用满，因此

这里并没有设计换页机制。

--------------------------------------------------------------

文件系统磁盘块读写：ide.c

实现磁盘的读写。

**ｃ．用户读写文件过程**

当用户要打开文件时，将在自己进程环境中申请一个用户文件描述符空间向文件系统服务端申请打 开某个文件，等待文件系统服务端回复，在收到消息前一直阻塞（状态设为NOT\_RUNNABLE， 不可被调度）。文件系统服务端被调度后，查看请求合法，并在自己进程空间打开该文件，并分配 一个文件描述符保存该被打开文件的信息，回复该文件描述符号，而进程通信过程自动将该文件描 述符号只读地映射到请求进程的相应的消息空间（也就是相应的文件描述符所在的虚拟地址空间）， 这样用户就可以获得该文件描述符号。用户读取文件时则利用该文件描述符号向服务端请求文件内 容，其他操作也遵循类似的过程。具体机制可参考下图：

Regular env FS env

+---------------+ +---------------+

| read | | file\_read |

| (fd.c) | | (damnfs.c) |

......|.......|.......|...|.......^.......|..........

| v | | | |

| devfile\_read | | serve\_read |

| (file.c) | | (serv.c) |

| | | | ^ |

| v | | | |

| fsipc | | serve |

| (file.c) | | (serv.c) |

| | | | ^ |

| v | | | |

| ipc\_send | | ipc\_recv |

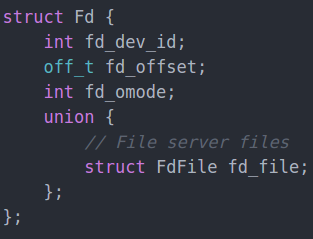
| | | | ^ |

+-------|-------+ +-------|-------+

| |

+-------------------+

对于用户来说文件描述符的定义为：



其中fd\_dev\_id是该文件的设备号，有三种类型的设 备：文件，管道和控制台，不同设备有不同的读写 handler。

fd\_offset是在描述符表中的序号。

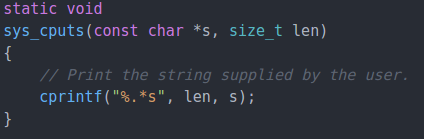
fd\_omode是打开状态，例如只读，只写等。

fd\_file对应着文件系统服务端返回的服务端的文件描 述符号。

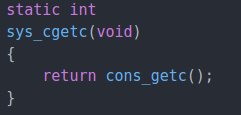
**3.系统库**

系统调用模块：syscall.c

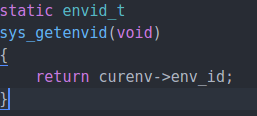
sys\_cputs：请求内核打印一个字符



sys\_cgetc：请求内核返回输入缓冲

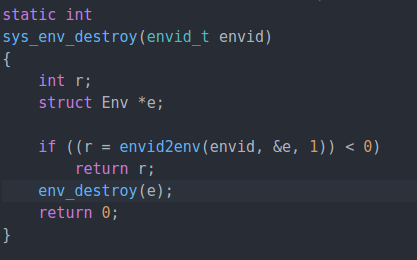


sys\_getenvid: 请求当前进程的进程号

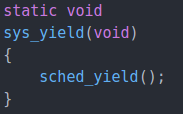


sys\_env\_destroy:

删除提供的进程号对应的进程，然而一个进程只能知道自己的进程号，所以这个系统调用应 该还算安全。。。吧。。。

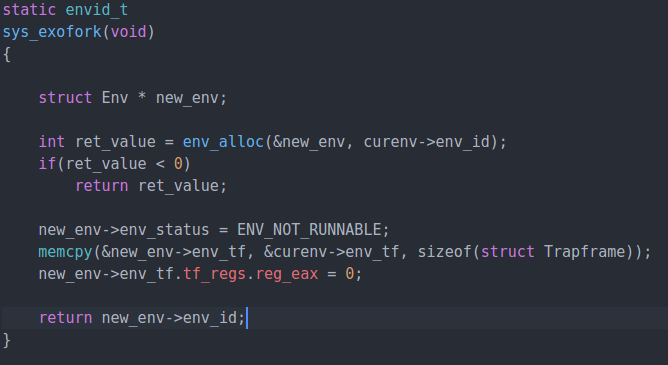


sys\_yield: 放弃目前当前进程被调度的权利，调度其他进程，当一个进程阻塞后调用。

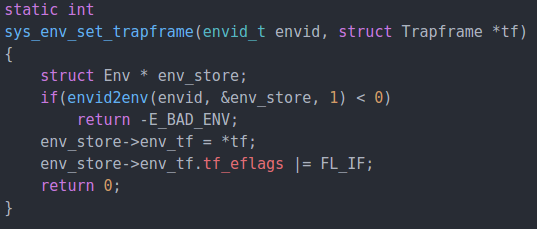


sys\_exefork：

创建一个与当前进程环境一样的子进程，并将子进程eax设为０（标志为子进程），返回进程号。 这里并不处理新进程的页表，这个步骤将在fork.c里考虑。

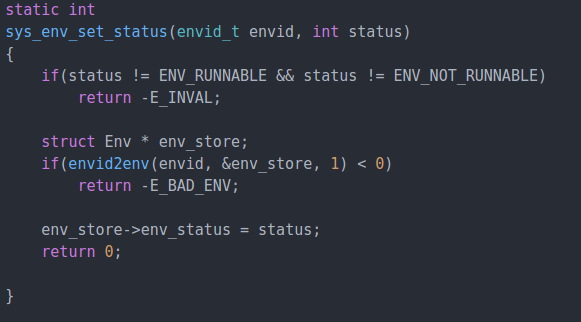


sys\_env\_set\_trapfram：更改程序运行环境（通用寄存器，段寄存器，eip等）



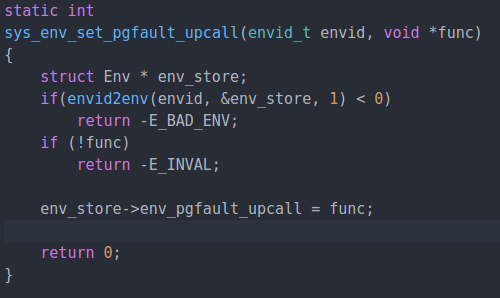
sys\_env\_set\_status:

更改一个进程的运行状态，只能为阻塞或可运行。



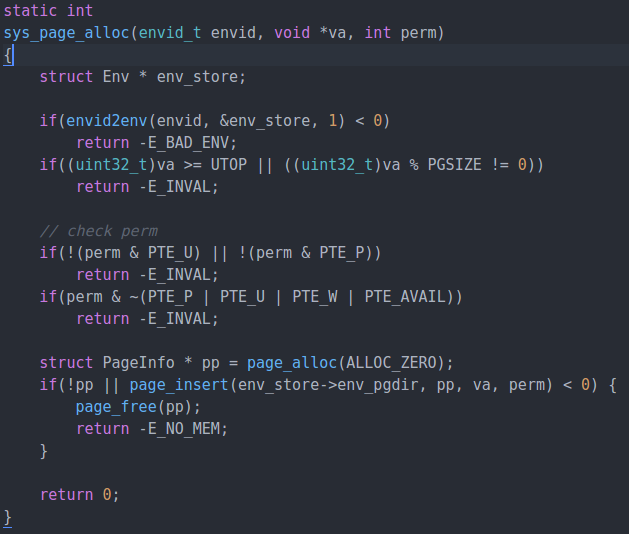
sys\_env\_set\_pgfault\_upcall:

设置进程的缺页处理函数，对于文件系统服务端非常有用。



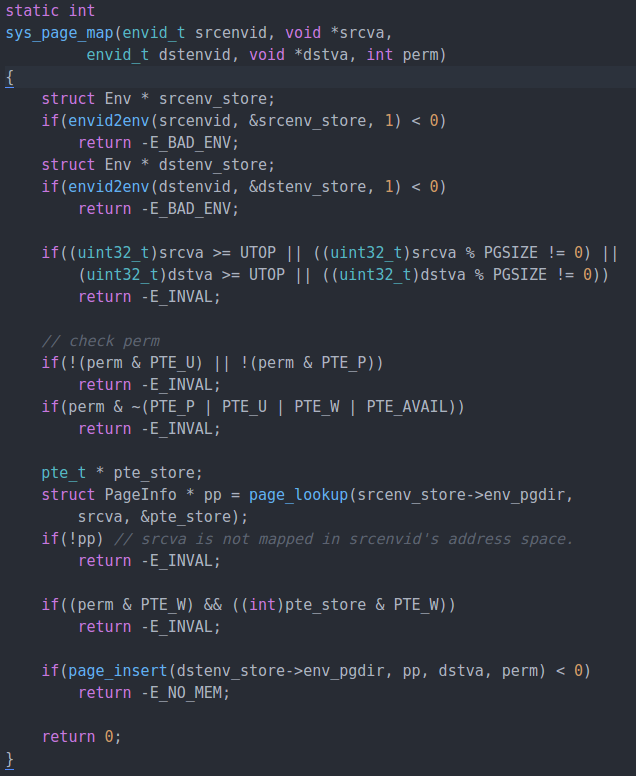
sys\_page\_alloc:

为该进程分配一个物理页，并将给定的虚拟地址映射到该物理地址，设置好给定的页权限。

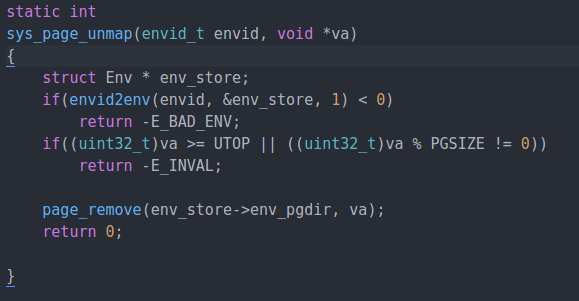


sys\_page\_map:

将某一进程的某一虚拟地址地址对应的物理页映射到另一进程的虚拟地址上，对于进程通信非 常有用。

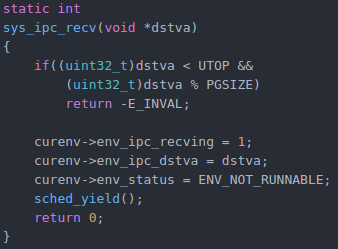


sys\_page\_unmap: 解除进程的虚拟地址与物理页的映射关系



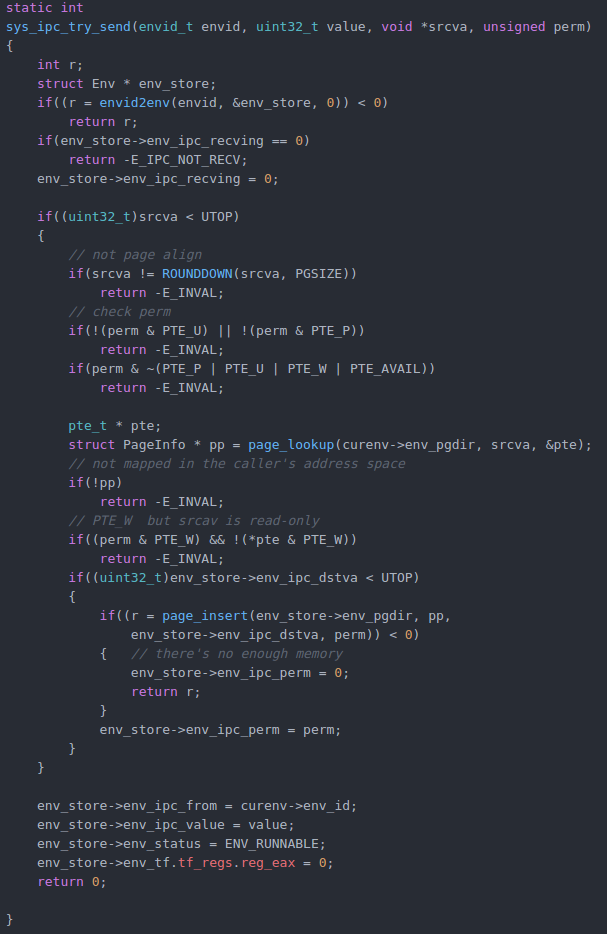
sys\_ipc\_revc:

实现进程接收其他进程的消息。然而实际上真正的传递过程都由发送方实现，请求进程只需要等待 将请求进程的等待标示置１，设置等待的目标进程号，并将状态设置为阻塞，之后运行调度函数来 调度其他进程。



sys\_ipc\_try\_send:

实现消息发送。将发送方所在要发送的消息所在的虚拟地址所映射的物理页只读地映射给接收方相 应的虚拟地址，并取消接收方的阻塞状态，并将接受方的等待位置０，说明发送成功。



------------------------------------------------------------------------------

进程通信模块：ipc.c

包装有关进程通信的系统调用，完成程序间的通信。

用户终端模块：uconsole.c

包装有关屏幕打印的系统调用。

用户缺页handler设置模块：pgfault.c

包装 sys\_env\_set\_pgfault\_upcall

- -------------------------------------------------------------------------------

文件操作模块：file.c fd.c

包装对文件描述符的分配，处理与释放，包装进程对文件系统服务端的访问。

字符串处理模块：string.c

包含标准库中含有的字符串处理函数。

模式化输入输出模块：stdio.c

其实只实现了模式化输出。

pageref模块：pageref.c

只读的让进程查看自己页目录和页表的权限状态。

fork模块：fork.c

包装sys\_exefork，并将父进程的只读页直接映射给子进程，对于可写的页则利用i386体系结构的 分页机制支持的写时拷贝技术，为这些可写页标记写时拷贝的标记，并设置好page fault handler。 当父进程企图写这一页时，分配一个新的物理页来映射原数据所在地址，并把页中原本的数据复制 到新的物理页上。

exit模块：exit.c

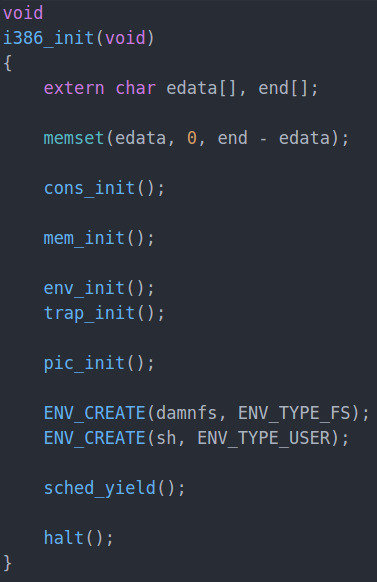
释放当前进程持有的所有文件描述符后删除该进程。

wait模块：wait.c

每次被调度时通过子进程的进程号检查其的状态是否为可运行，若是则利用sys\_yield请求内核调 度其他进程。

用户程序入口：uentry.S libmain

所有用户程序的入口，设置好程序参数和一些重要变量。

**启动运行交互过程演示**

内核被加载后，在entry.S中建立了临时页表和内核栈，并调用init.c中的

i386\_init来实现内核的初始化：

其中edata和end是链接内核的linker脚本中提供bss段的开始和结束，因

此要全部置０。之后cons\_init实现UART串口的初始化，并确认cga显存没有出错， 之后才能进行屏幕打印。然后mem\_init实现内存映射的初始化，主要是从时钟芯片 读取内存大小和设置新的页目录并加载新页目录到cr3寄存器上。

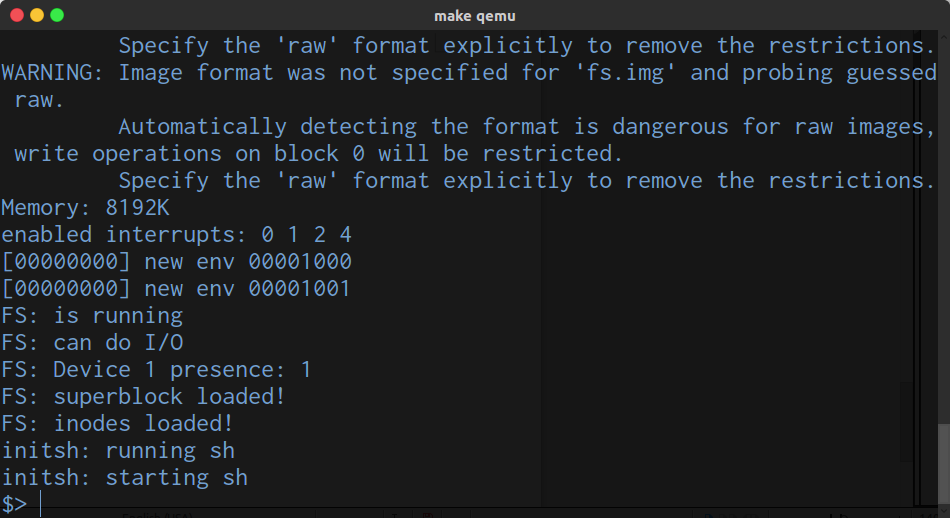
之后env\_init初始化进程向量，其实就是将向量置０。然后pic\_init初始化 pic，设置pic中断号的偏移，这里设为32，并设置irm，表示要屏蔽哪些中端，这里 触发了时钟， 串口和键盘中断，中断号分别为０，１，４，同时触发pic的从片引

脚，为２。

之后创建了两个进程，分别是文件系统服务端和用户shell。这两个elf二进制文 件和内核链接在一起，利用linker自动生成的全局变量找到每个文件的起始位置。

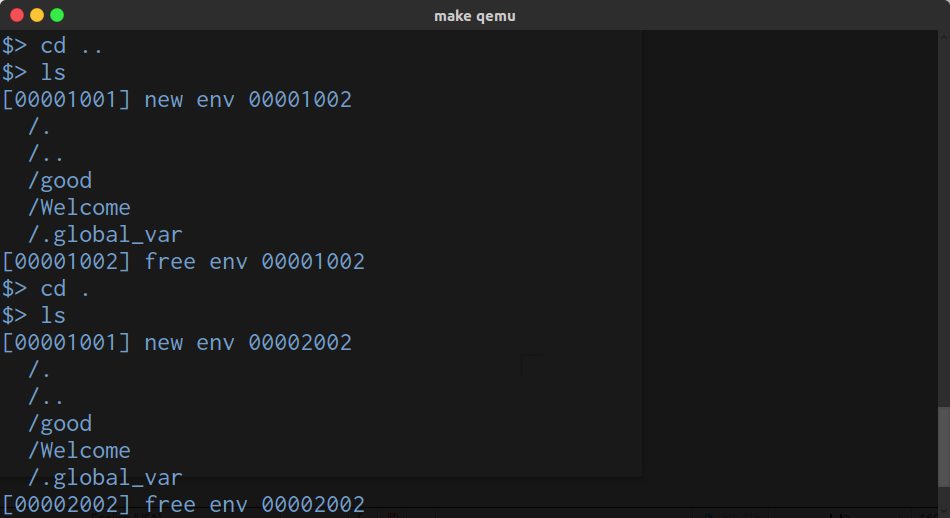
之后就可以运行调度函数开始进程调度了。

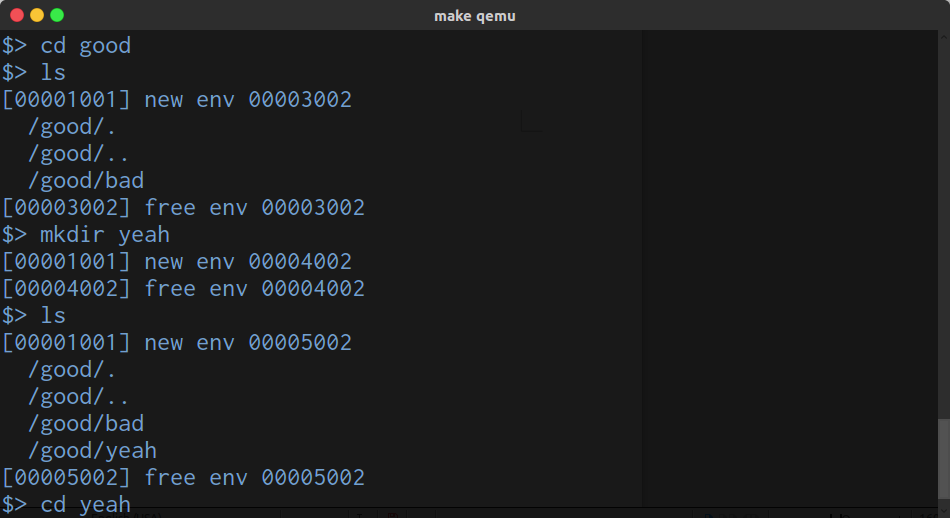
启动后界面：

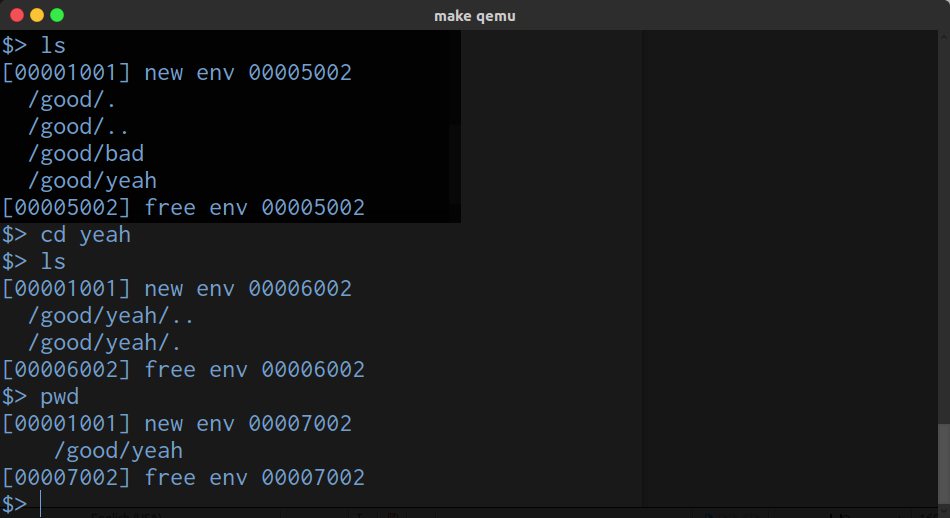


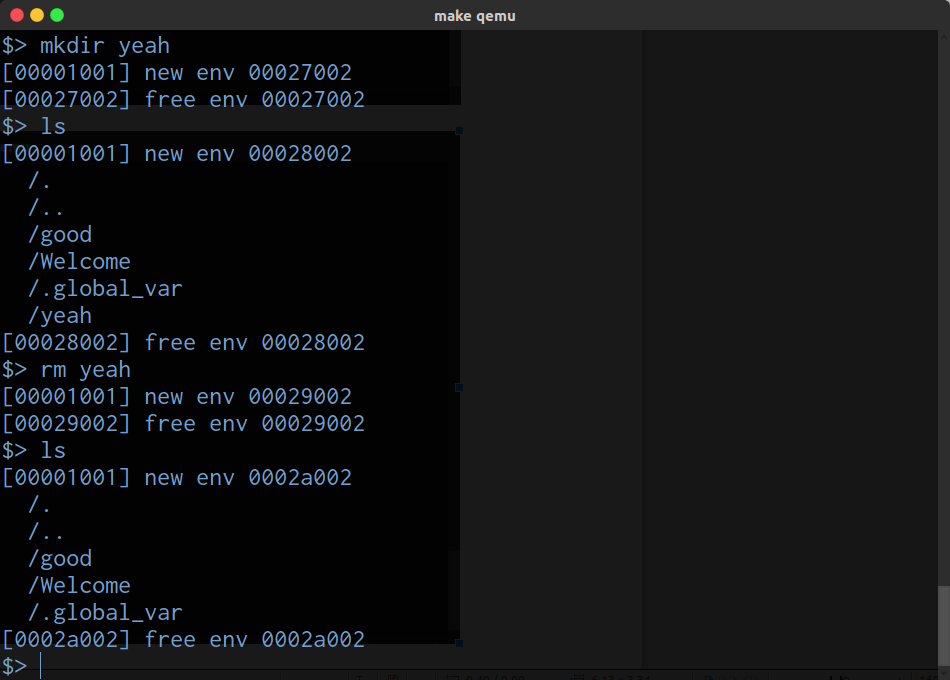
mem\_init中检测内存为8192KB，也就是8MB，pic\_init中开启的中断为０，１，２，４，分别是时钟，键盘，从片， 和串口。之后创建了两个进程，进程号分别是00001000和00001001。这里为了方便跟踪每个进程的创建和释放，将 在shell中打印出信息。之后测试文件系统服务端被调度，测试磁盘读写，是否存在第二个磁盘分区和文件系统是否正 常，之后进入等待消息模式，之后用户shell被调度，shell先申请一个类型为控制台的文件描述符，进入了交互界面。

可以测试一下一些内置命令。值得注意的是这个shell是第一个申请文件描述符的，因此描述符号应该为０。除了cd命 令以外的其他命令都是fork一个子进程来完成的，避免命令崩溃造成整个shell的崩溃。可以从命令的创建和释放提示 看出这一点。

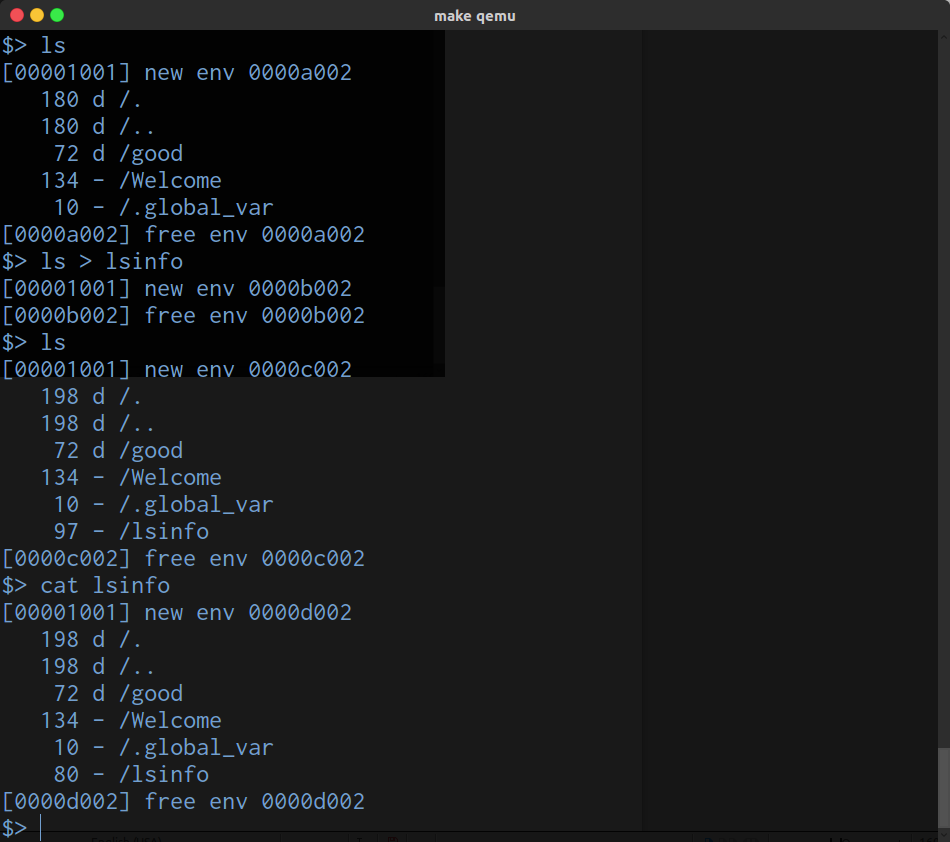


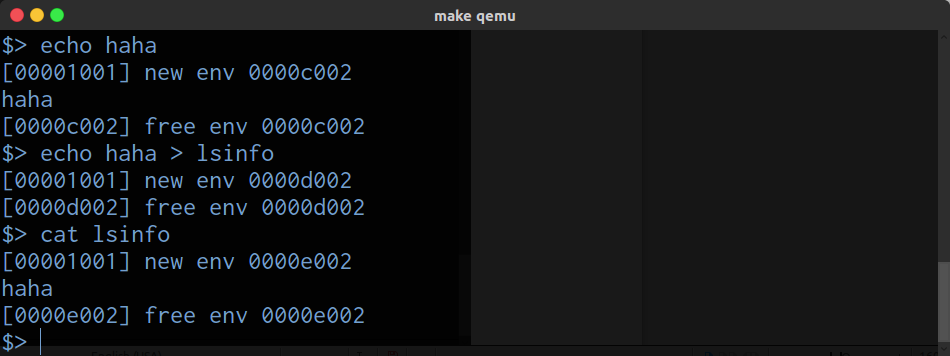






以下是重定向的演示：





进程通信pingpong的演示：

