OSLab实验报告

丁保荣 171860509

1770048119@qq.com

L0:

L0中只是很简单得实现了一个滑块的移动，通过上下左右键来控制滑块，复用了框架代码的一部分。

代码的架构很简单：就是是获取屏幕大小，并将初始滑块的坐标置于屏幕正中央，然后进入一个死循环，读取键盘输入，如果输入了上下左右，就进行滑块的移动，在滑块的移动中加入边界检测。然后只有当滑块移动后，才进行屏幕的重新绘制。

在实现滑块移动的时候，一开始是每隔一秒将屏幕整个清除，重新绘制。但运行后发现屏幕的输出很诡异，完全不是想要的效果，猜测是不是重新绘制屏幕代价太大了。因此，最后修改为只有当检测到滑块移动的时候更新屏幕。而且更新的时候不是清除整个屏幕，而是只清除滑块的上一个位置，并且只绘制滑块的下一个位置，这样写屏幕的需求就降低了很多。

L1:

L1实现kalloc和free。采用链表的思想，但因为没有办法申请内存，所以直接讲链表嵌入堆区。一开始是只有一个节点，空间是所有可用空间。

kalloc调用的时候，从链表头开始寻找空闲空间，当寻找到一个空闲空间大于所需空间的时候，就停下来。如果空闲空间特别大，会将当前的节点分成两个节点，并将第一个节点的分配内存的地址返回。

free调用的时候，根据需要free的地址找到对应节点的地址，然后将这个节点管理的空间声明为可用。如果当前节点能和周围的空闲节点合并，那就将他们合并，直到不能合并为止。

在所有分配的地址中都是按照8字节对齐的。

关于锁的问题，我只实现了spin-lock,但没有关中断。基于这么几个原因。1. 如果采用内联汇编，用户程序(kalloc)是根本没有关中断的权限，我试了一下，会直接segmentation fault。这样的话，在native上根本没用办法执行。2. 网上的资料说malloc这种函数都是不可重入的，我认为在linux中断处理程序中很可能不会调用malloc,而且即使要分配内存，也是调用操作系统内部提供的接口，如brk之类的。

L2:

L2主要实现内核的线程管理以及一些锁的实现。

os->on_irq是用来实现注册中断处理程序。我主要用一个链表是存储handler，整个链表是按照seq的大小排序的，这样在调用中断处理程序的时候只要从前往后扫一遍就好了。实现的时候也没有什么bug。

os->trap是用来调用对应的handler的，所以只要把整个handler的链表从头到尾扫一遍就好了。

kmt->spinlock相关的代码是仿照xv6的实现。其中对于中断的处理，用一个栈来保存，这样就可以实现中断的嵌套。

kmt->context_save很简单，直接把传进来的context保存在current里就好了。

kmt->context_switch用来实现线程切换。线程的状态只有两个READY和SLEEP。每次切换的时候，都选择READY的切换，如果没有READY的，就继续执行当前线程(这里的实现导致我的sem_wait的实现也和标准实现有点区别)。关于进程切换的公平性。 一开始我是每次都从头到尾扫，扫到READY的，就调度它，并把它放到链表的最后，这样就能保持公平性。但后来发现这样的开销有点大，因为每次时钟中断的时候，都要拆链表。因此后面我修改了，每次扫当前线程后面的，当扫到链表尾时，再回到链表头。这样就不需要对链表进行修改了。

kmt>create实现的是线程的创建，主要内容包括绑定到某一cpu，赋予name，把状态设为READY，分配栈空间，创建上下文，加入线程链表。

kmt->teardown 把线程的栈空间释放，并把它从线程链表中移除。

kmt->sem_wait实现的过程中主要遇到的bug是，把if改成了while，这样会导致改线程不断入队列。

kmt->sem_signal把count加一，如果队列不为空，就唤醒队列里的第一个线程。

L3:

L3主要实现文件系统包括blkfs, devfs和procfs。

主要的文件是三个:blkfs.c, devfs.c, procfs.c。其他的一些结构体声明，函数声明，宏定义分散在/framework/kernel.h, /include/common.h中。

最主要的模块是vfs，它提供面向应用程序的函数接口，统一不同的文件系统，管理挂载点，它为文件系统抽象出了一套API。并将这些API对应到不同文件系统的实现。blkfs, devfs, procfs 都有自己的API实现。

首先是blkfs设计。blkfs的设计仿照ext2，并去除了很多不需要的信息。磁盘的最开始是inode bitmap, 其次是block bitmap, 然后是inode, 最后是block. 一个inode(blkinode_t)占128字节,储存着文件大小，储存文件的块编号还有指向该inode的文件的数量(用于link和unlink)和文件的类型(file或者directory)。一个block占1024字节。对于目录的存储，是把目录名作为一个文件，在它的数据区存储这其目录下的文件和子目录。每一个表项(blkdire_t)占32字节，里面存储inode编号、文件大小、文件名和文件类型。 lookup实现对文件路径的解析，从根目录一层一层地解析寻找，最后返回文件或目录的inode。open的话，首先寻找文件，如果不存在就创建文件，然后将file_t里的offset设为0。 read的话，根据file_t里的off判断开始的block，根据off+size判断结束的block，把这些block里需要的内容拷贝到buf里。 write的话，跟read差不多，只不过如果当前文件的block不够时，要申请新的block，并对block bitmap进行修改。lseek的话，有三种flag: SEEK_SET, SEEK_CUR, SEEK_END。这些和标准库函数的定义一样。在最后改offset之前，要确认不小于0，也不超过文件大小。mkdir就在上层目录的文件中增加一个新目录的表项，并且申请一个新的inode，来存放新的目录。rmdir把当前目录的inode删除，分配的block删除，并把上层目录里的表项删除。link的话在上层目录的block里创建一个新的表项，并把表项里的inode指向oldpath对应的inode。unlink的话首先在上层目录的block中删除这个文件的表项，如果对应inode的refcnt变为0，则把该inode也回收。readdir的话是把该目录下的文件和子目录的文件名都写入到buf里，以\n分隔。create_blkfs创建blkfs，主要是对fs的赋值。

我实现的blkfs的参数：一个inode大小为128byte, 一个block大小为1024byte, inode数量和block数量均为1024个。目录文件里的每个表项占32byte，除去.和..,所以每个目录下最多有30个文件或子目录。因为一个inode里可以存20个block_id,所以每个文件的大小最大为20KB.

然后是devfs的设计，主要实现了ramdisk0, ramdisk1, tty1 tty2, tty3, tty4这几个设备，可以读写。devfs不支持目录的创建和删除，也不支持link和unlink。

然后是procfs的设计，每个线程对应一个文件，以pid命名。文件里的内容包括线程名，总的cpu个数，当前线程在哪个cpu上，当前线程的堆区起始地址(十进制)，当前线程的堆区结束地址(十进制)，当前线程的堆区大小(十进制)。procfs支持的最大线程数量为1024。和devfs类似，procfs不支持目录的创建和删除，也不支持link和unlink。

最后用vfs的API对上述三个文件系统进行抽象。将不同的文件系统挂载到不同的挂载点，vfs管理挂载点，并且将路径名翻译成不同文件系统里的文件名。比如/dev/ramdisk0会被翻译成/ramdisk0传递给devfs, /mnt/a/b会被翻译成/a/b传递给blkfs。

最后实现shell，shell支持的命令有这些：ls, cd, mkdir, rmdir, rm, ln, touch。注意所有的路径均为绝对路径

ls: 不带参数。列举当前目录下的文件和子目录。如果在/路径下ls，还会列出所有的挂载点。

cd: 格式如 cd /proc, cd /mnt, cd /a/b。 只支持绝对路径。

mkdir: 格式如 mkdir /a, mkdir /mnt/a之类的，不支持递归创建目录，即上层目录必须存在。如 mkdir /a/b,则 /a必须存在且是目录。

rmdir: 格式如 rmdir /a, rmdir /mnt/a之类的，不支持递归删除目录很文件，即删除的目录必须是空目录。

rm: 格式如 rm /a.txt, rm /mnt/b.txt之类的，既可以删除文件也可以删除链接。

ln: 格式如 ln oldpath newpath. 如 ln /a.txt /b.txt. 其中/a.txt必须存在, /b.txt必须不存在。

touch: 格式如 touch /a.txt 用于创建一个新的空文件。

cat: 格式如 cat /a.txt 用于输出该文件里的所有内容。

write: 格式如 write filename offset content. 如 write /a.txt 10 helloworld 就是向/a.txt从offset为10开始的地方写入helloworld，其中offset为十进制表示。可以通过终端互写，如在tty1中 write /dev/tty2 0 hello, 就可以在tty2中看到效果。