Android OS Practice Lab 4-基于socket通信的云文件系统

李炎 1400012951

## Background And General Description：

随着网络技术日新月异的发展，很多场景下网络带宽已经不再是网络服务的瓶颈。这一现象也使得云文件系统的普及得到了可能。而另一方面，人们日常生活中使用的终端设备中，常用的只是一些媒体资料，或一些图像资料等，剩余的数据大多数是以备份的形式存放在存储介质上，某种程度上讲这其实是一种浪费。本lab以此为出发点，旨在搭建一个Android系统下的云文件系统，定期扫描设备，当文件系统的使用率达到算法容忍的下界后将一些不常用的文件驱逐出设备，为用户腾出更多的使用空间。

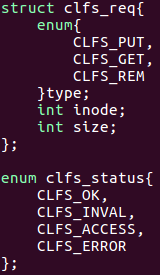
## Part A：初始化并挂载文件系统

简便起见，本lab中我们开展实验的目标文件系统是Ext2文件系统。该文件系统以一个SD卡镜像为载体，挂载到系统的一个目录下。本部分按照Write up中所描述的一步一步开展即可。具体几个步骤如下：

1. 修改内核编译的配置选项，使其支持Ext2文件系统拓展，并重新编译内核。
2. 生成一个SD卡镜像。此步骤我使用了Android SDK包tools目录下的mksdcard工具。
3. 使用busybox工具格式化SD卡。
4. 将格式化完毕的文件系统挂载到/mnt/sdcard目录下

## Part B：云端服务器

为方便起见，本lab中所用到的云文件系统服务器我直接在虚拟机的宿主机上完成。这样虚拟机可以直接通过IP地址10.0.0.2直接与宿主机建立TCP连接。TCP连接的握手信号定义如下：



结构体clfs\_req是client端发给server端的请求信号，枚举类enum定义了三种请求动作，分别是请求将文件驱逐到云服务器、将文件从云服务器取回、将文件从云服务器删除。Inode字段是一个文件的唯一标示，由内核中的inode结构体的i\_ino字段唯一给出（后文还会提到）。size是本次传输的数据长度。

结构体clfs\_status是server端返回给client端的状态码，对应于server是否允许client建立并完成这次传输。

Server端的服务器程序用socket编程实现。几个关键的函数有：

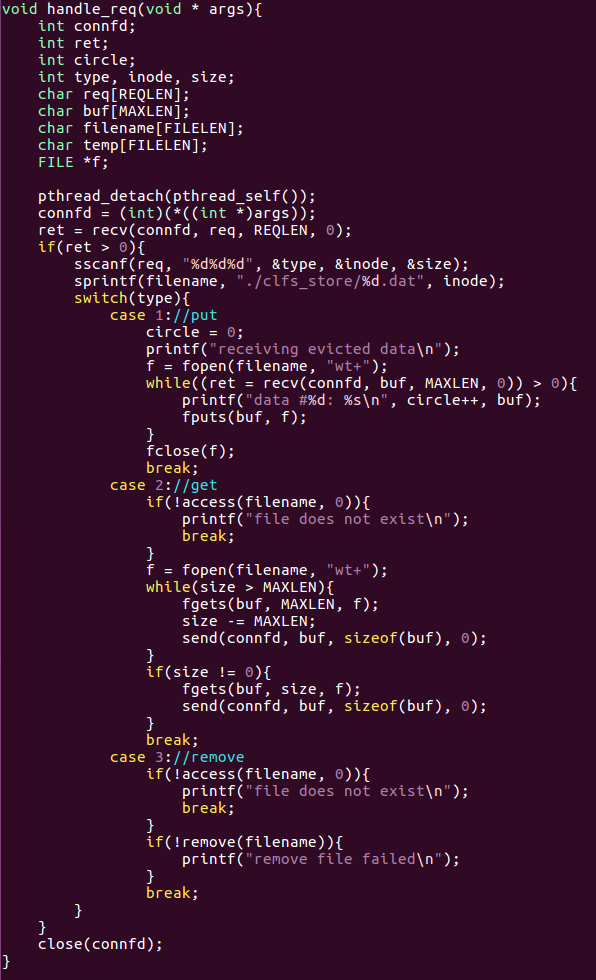
1，int open\_listenfd(int port);

open\_clientfd函数接受一个指定监听端口的参数port，并在该端口上建立一个监听文件描述符，具体代码如下：



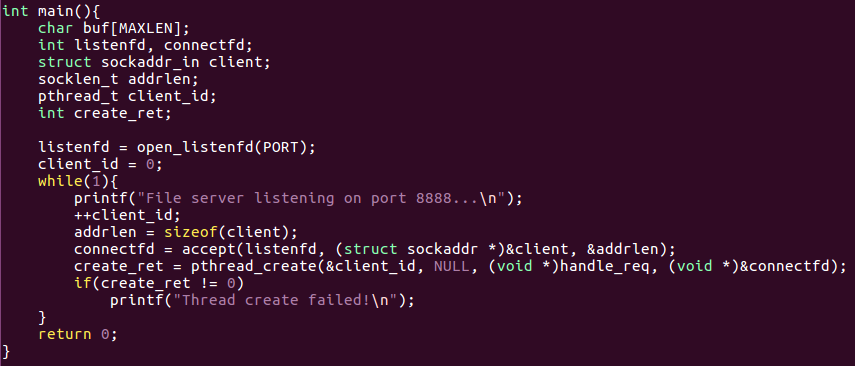
2，void handle\_req(void \*args);

handle\_req函数接受一个连接文件描述符为参数，首先服务一个来自client端的握手信号，然后根据握手信号的情况进行判断以及存在性和安全性检查，最后根据不同的情况接受来自client端发来的数据或者将云服务器的文件数据发送到client端。当从client端接收数据时，server将文件数据写入到云文件系统的clfs\_store文件夹下的inode\_id.dat文件中，文件名前缀即为文件inode的唯一标示。注意到处理来自client端请求时主线程创建一个子线程来处理这个请求，因此为了不阻塞主线程，handle\_req中使用了pthread\_detach(pthread\_self()) 具体代码如下：



3，int main(int argc, char \*\*argv);

main函数中首先调用open\_clientfd创建一个在8888端口上的监听描述符，进入无限循环不断接收来自client的数据请求。一旦接收到一个请求，就创建一个子线程处理这个连接，并进入下一轮循环继续处理别的连接请求。代码如下：



### Part C 驱逐机制

本部分拓展了ext2文件系统的文件操作，新增加了两个核心接口：

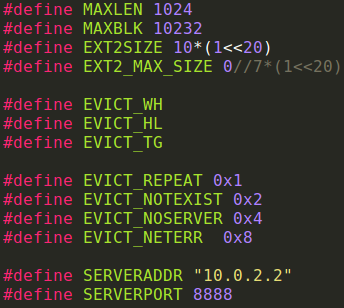
int ext2\_evict(struct inode \*i\_node);

int ext2\_fetch(struct inode \*i\_node);

本部分我将这两个函数的实现拆分为了几个模块，使程序有更强的结构性和易拓展性。具体如下：

1，头文件ext2\_evict.h

该头文件中定义了ext2\_evict定义的所有的函数的返回码，具体如下：



其中EVICT\_WH等三个宏是决定调用驱逐函数的下界，EVICT\_REPEAT等四个宏定义了调用相关的函数遇到错误时的返回码，具体意义如下：

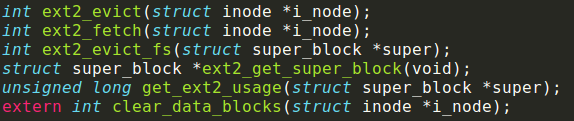
EVICT\_REPEAT：驱逐一个已经被驱逐的文件

EVICT\_NOTEXSIT：要驱逐的文件不存在

EVICT\_NOSERVER：无法与文件服务器建立TCP连接（文件服务器未启动）

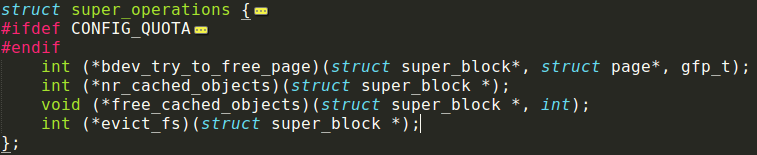
EVICT\_NETERR：建立TCP连接中发生其他错误

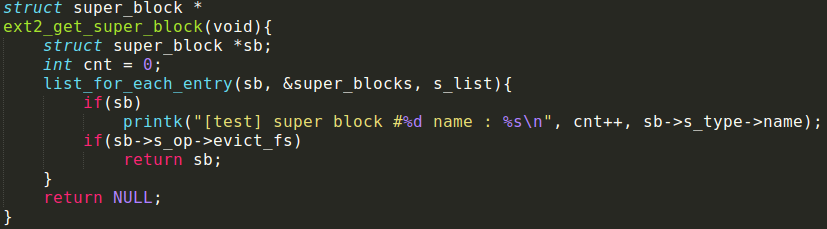
头文件中向操作系统开放了四个接口：



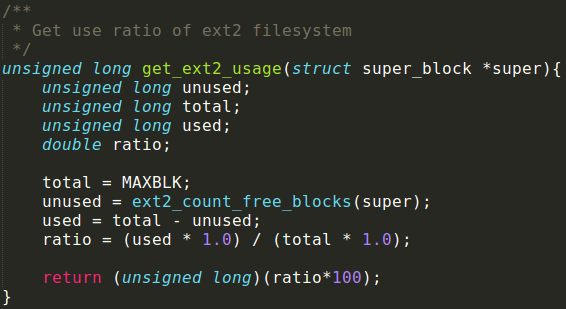
前两个接口前文已述。

函数ext2\_evict\_fs针对一个super\_block进行扫描，获得超级块的利用率，以及在利用率超出下界时决定驱逐哪些inode对应的文件。该函数是整个云文件系统的接口，同时也作为super\_block操作的一种添加到super\_operations中。如下图：

函数ext2\_get\_super\_block获得我们挂载的ext2分区的超级块。通过遍历所有的super\_block判断是否该super\_block（以下简称sb）是ext2的super\_block。代码如下：



函数get\_ext2\_usage通过计算一个sb中空闲的磁盘块和已经使用的磁盘块数目之比获得当前sb的占用率，返回一个0到100之间的整数，代码如下：

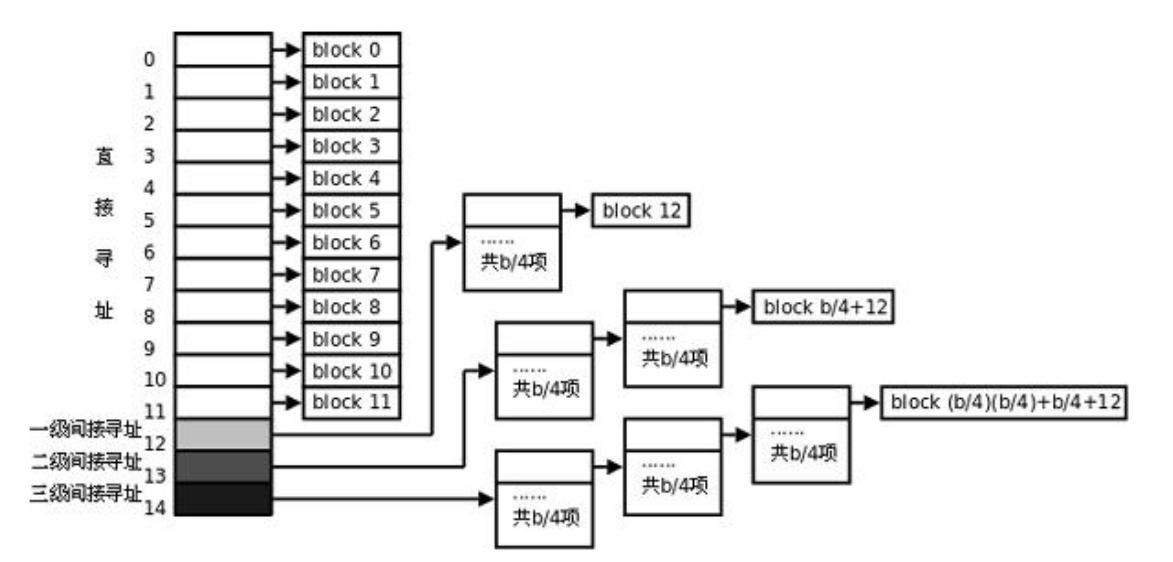


函数clear\_data\_blocks接受一个inode指针作为参数，释放掉该inode所对应文件的所有磁盘块，但是不修改inode的任何元数据。为了调用相关函数的方便，该函数定义在文件inode.c中。实现过程主要分如下几个步骤：

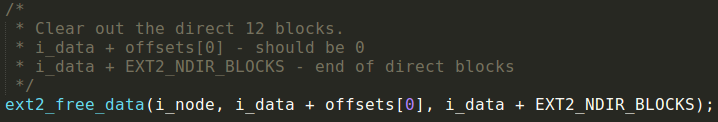
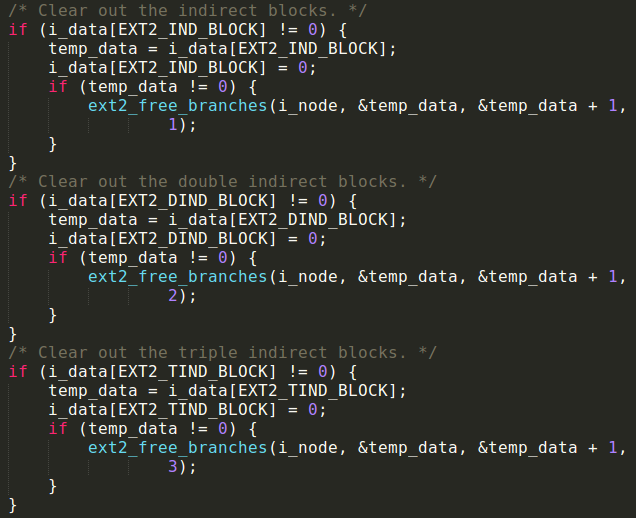
**Step 1**，将所有缓存在page cache中的磁盘页清除，调用内核的truncate\_inode\_pages调用：



**Step 2**，linux的inode的组织结构是这样的：一个inode对应15个磁盘块索引，前12个磁盘索引直接对应一个磁盘块。第13个索引指向一个一级索引表，该索引表索引到若干个磁盘块；第14个索引指向一个二级索引表，该索引表索引到若干一级索引表；第15个索引指向一个三级索引表，该索引表索引到若干二级索引表。如下图：



因此在释放一个inode对应的所有磁盘块时主要分四步，即分别释放各级索引表对应的磁盘块。

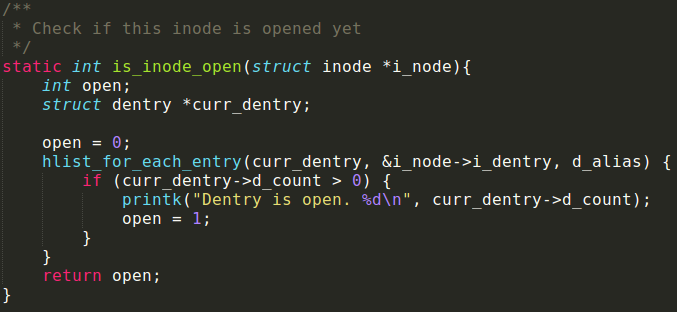
1. 直接索引的12个磁盘块内核提供了接口ext2\_free\_data：  
   
2. 间接索引的三个表项使用ext2\_free\_branches。先分别检查该inode是否含有间接索引的blocks，如果是则进行空间释放操作。代码如下：  
   

以上即为几个自定义的函数，在接下来的操作中会用到。

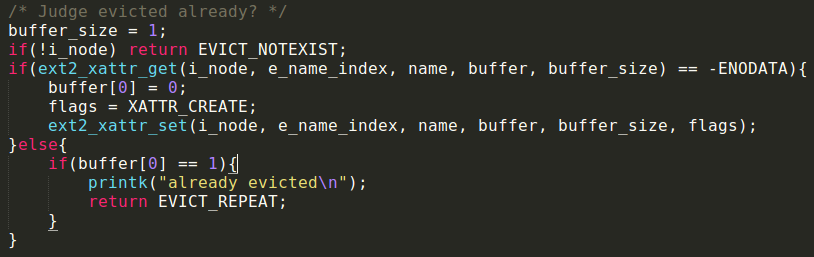
下面来看驱逐机制的具体实现：

以ext2\_evict为例，实现的过程中大致分为如下几步：

**Step 1**：检查该inode是否已经被打开。如果被打开则不进行驱逐（显然我们是不能将用户正在用手机浏览的图片不加判断地直接驱逐的）。函数is\_inode\_open实现如下，即遍历所有该inode对应的dentry：

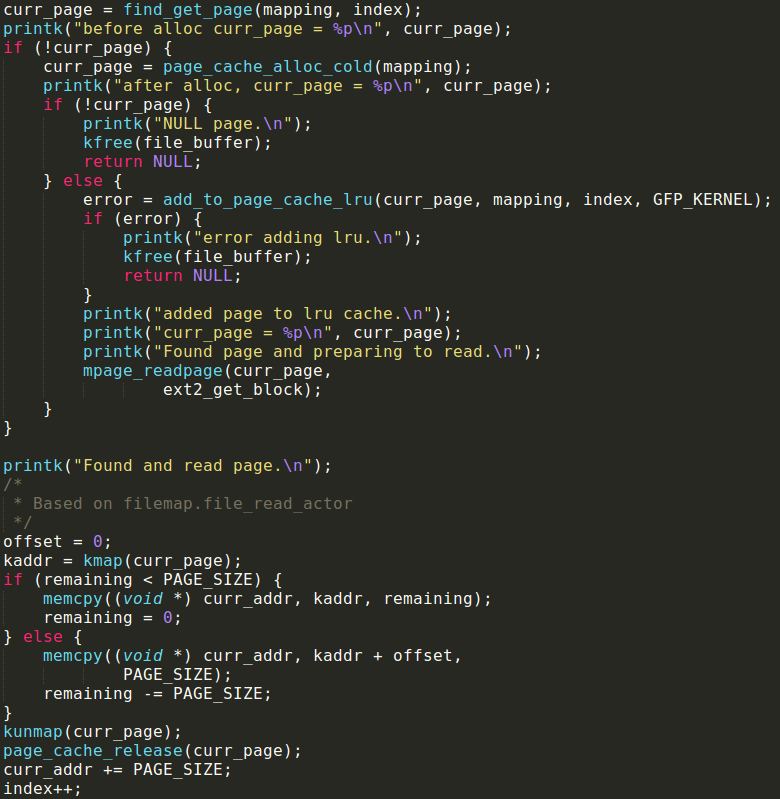


**Step 2**：检查该文件是否已经被驱逐，如果已经被驱逐则返回，否则标记该文件已经被驱逐。标记是通过kernel提供的拓展属性机制（xattr）实现的。拓展属性是不存在于inode中的另一种形式的元数据。为此我们设置一种叫做“evicted”的拓展属性，用来标记该文件是否已经被驱逐（这个属性在打开文件时会用到，如果发现即将打开的文件已经被驱逐，则调用ext2\_fetch从云服务器取回数据）：

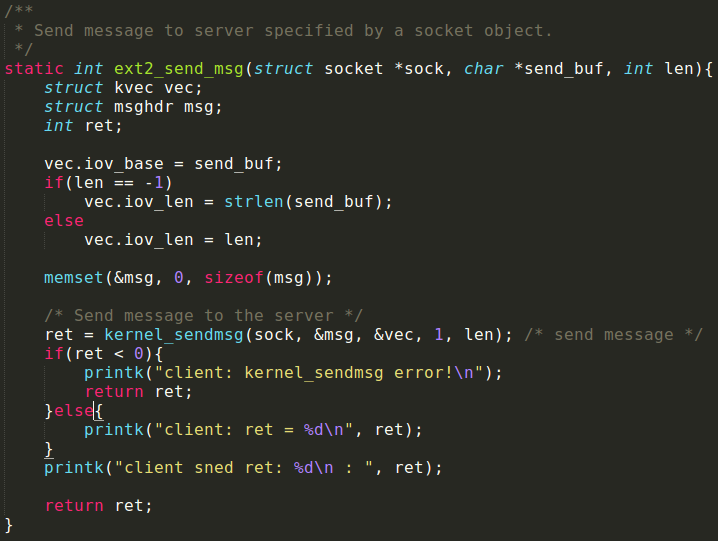


**Step 3**：由给定inode获取文件数据。主要分如下几个步骤：

1. 检查该inode对应的文件页是否在page cache中。调用内核接口find\_get\_page，该函数接受inode->mapping和index两个参数，检查索引为index的page是否在缓存中。
2. 如果不在缓存中，则调用page\_cache\_alloc\_cold分配一个page，然后调用add\_to\_page\_cache\_lru将之添加到page cache中，最后调用mpage\_readpage读取page所对应的文件数据。
3. 调用kmap将相应的page映射到当前的虚拟地址空间，即由一个物理地址直接得到对应当前进程的虚拟地址。
4. 创建一个长度为file size的字符数组，每读一页就使用memcpy将该页的内容附加到该字符数组的末尾。
5. 返回该字符数组，或根据不同的情况返回相应的错误码。  
   关键代码如下：



**Step 4**：创建socket并向server发出send请求，准备发送文件数据。用户空间创建socket的流程前文叙述server端编写时已经提及，内核空间的操作大同小异，只是为了内核的安全考虑做了一些附加的操作。值得一提的是，版本为3.10的linux kernel已经默认关闭了内核空间创建socket的开关，如果想无视之，必须在配置文件中更改相关配置选项。以发送为例，具体代码如下：



**Step 5**：释放本地文件数据

直接调用前文所述的ext2\_free\_data接口，释放掉所有的block。

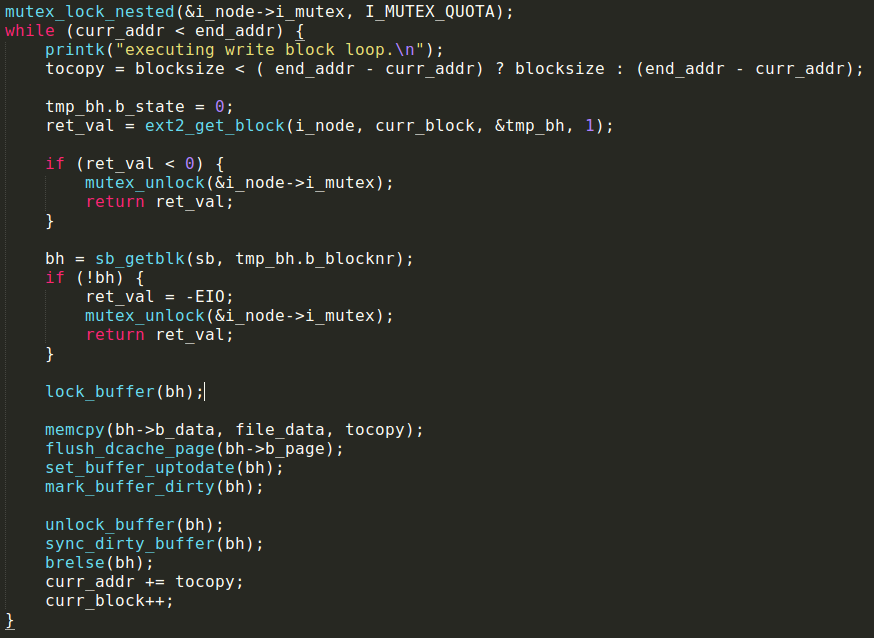
**Step 6**：关闭socket文件描述符。

ext2\_fetch的实现类似，只是二者一个是读文件，另一个是写文件。因此只有相应IO操作有不同。我编写了write\_blocks函数，用来处理fetch过程中向文件中写数据的操作。fetch的实现分为如下步骤：

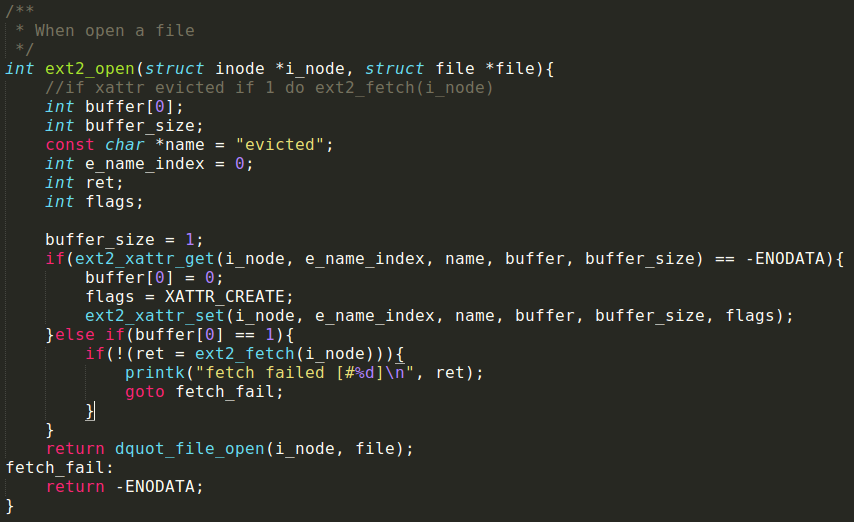
**Step 1**：检查“evicted”拓展属性。如果未被驱逐则直接返回。

**Step 2**：创建socket，向server发起TCP/IP连接请求。

**Step 3**：从server接收到数据，并将之写入文件块中。首先根据inode元数据获得该文件大小，然后调用ext2\_get\_block获得相应的文件块，并将之映射到一个buffer\_head（以下简称bh）。然后调用sb\_getblk为该bh分配相应的磁盘块。bh的data字段存储了该bh对应的数据。使用memcpy将从server收到的数据的相应段落拷贝到该缓冲区。然后通过set\_buffer\_toupdate设置该bh的toupdate标记，并调用mark\_buffer\_dirty将该缓冲区标记为dirty。最后使用sync\_dirty\_buffer将已经改动的bh写回到磁盘。核心代码如下：



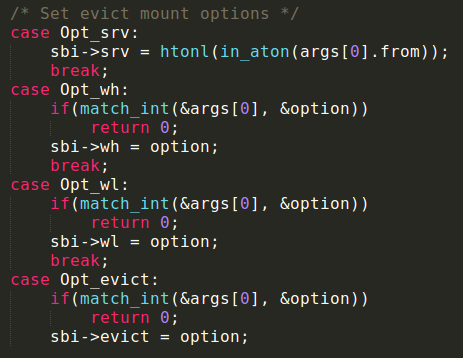
**Step 4**：修改ext2文件系统打开文件的基础操作，即先检查该inode对应的磁盘块是否被驱逐。核心代码如下：



以上便是evict和fetch的主要实现流程。值得一提的是，在这个过程中要特别注意同步问题，即合适地使用内核相应的锁结构，在合适的场合进行上锁与解锁。

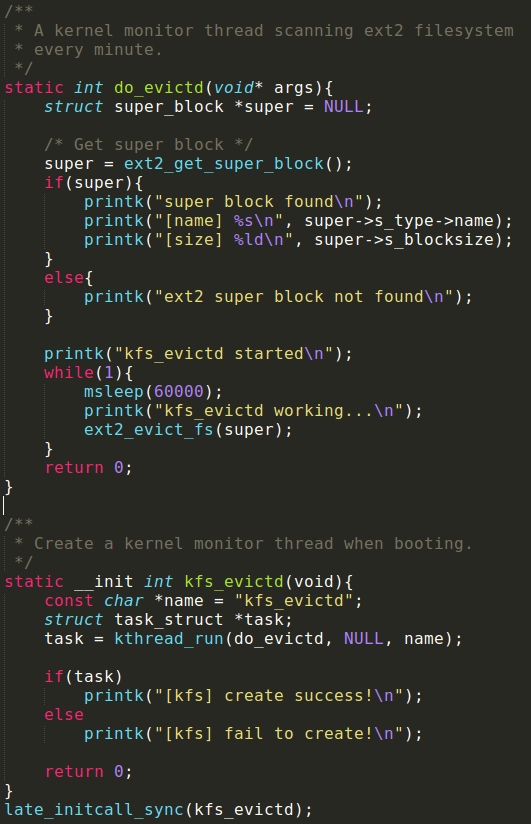
### Part D 监视文件系统

前文所述的evict\_fs即为监视文件系统的接口。在evict\_fs文件系统中实现了clockhand算法。在一个给定的super\_block中，一个inode可以根据一个index索引到。因此我将clockhand对应的整数值作为根目录inode的一个拓展属性“clockhand”来进行记录。如writeup中所说，在相应的情形下进行evict操作。此步骤中需要先修改mount的参数，以支持srv，wl，wh，evict等几个文件驱逐阈值。如下图：



具体evict\_fs的实现参见附件中的代码。

在系统执行时需要保证一个守护进程一直在监视ext2分区的磁盘使用率并执行相应的驱逐操作。使用kernel的kthread\_run可以启动一个线程。同时需要添加\_\_init关键字修饰，并用late\_initcall\_sync保证系统初始化时执行相关代码。（writeup中说的是fs\_initcall，但是我没有尝试成功，查了资料似乎是初始化顺序的问题，于是改成了比较靠后的初始化顺序即late\_initcall\_sync）。代码如下：



### Part E 测试Cloud FS

参见附件中output文件

### 总结

本lab是本学期安卓操作系统实习的收官之作（加分项的没时间写了QWQ）。这个lab中实现了一个真正意义上可以使用的云文件系统，用到了linux kernel中的许多机制，比如拓展属性，socket通信，vfs层以及更加底层的文件操作等。大大提高了阅读和编写工程代码的能力，以及查阅资料的能力。

总体来讲，Android 操统实习的这几个Lab让我对操作系统有了更深的认识，这门课是好课，只是不适合在一个课很多的学期选择。