## 文件I/O编程作业

1. **描述利用fcntl()（以lock\_set （）函数为例）加解、锁的过程。**

答： （1）、 对flock结构体进行赋值，确定文件的状态（包括文件锁）

1. 、在对文件锁进行操作前，先判断文件锁是否可以上锁等操作
2. 、如果lock.l\_type（锁状态）不是解锁状态，则要判断是读取锁还 是写入锁
3. 、在文件进行阻塞式上锁或解锁之前，要对l\_type重新赋值，避免 F\_GETLK修改l\_type的值
4. 、根据不同的type值进行阻塞式上锁或解锁

**2.根据自己的理解描述select（）、poll()及epoll（）三个函数的异同点。**

答：

**相同点：**（1）、都能监听多个文件描述符

（2）、他们将等待有timeout参数指定的超时时间，直到一个或多个文件描述符上有事件返回，返回值为文件描述符数量，返回0表示事件没有发生。

**不同点：**1、事件集：都通过某种结构体变量来告诉内核监听哪些文件描述符上的哪些事件。用该结构类型的参数获取内核处理的结果。

1. 、select的参数类型没有将文件描述和事件绑定，仅是文件描述符集合。
2. 、poll的参数类型pollfd,将文件描述符和事件都定义在其中，任何事件都被统一处理，使编程接口简洁，并且内核修改每次都是pollfd结构体的revents成员，而events保持不变。因此下次应用程序调用poll时应用程序不需要重置pollfd的事件集参数。但由于每次select和poll调用都返回整个用户注册事件的集合（就绪的+未就绪的）所以应用程序索引就绪文件描述符的时间复杂度是O（n）;
3. poll在内核中维护一个事件表，并提供独立的系统调用epoll\_ctl(),添加、删除、修改事件。故，每次epoll\_wait调用events参数仅用来返回就绪事件，不需要反复从用户空间读入这些事件，所以应用程序索引就绪文件描述符的时间复杂度是O（1）；
4. 工作模式：select和poll工作在TL模式下；epoll工作在ET模式。
5. select和poll采用轮询的方式，epoll\_wait采用回调函数的方法。

**3、SSD与机械硬盘区别，SSD用了那些机械硬盘没有的技术？**

答： 固态硬盘：是由控制单元和固态存储单元组成的硬盘。

机械硬盘：存在机械结构，读取和写入时由磁头在转动的盘片上转动寻找文件所在扇区，类似以前的VCD播放碟片。

SSD有闪存（flash memory）技术和DRAM技术

**4、那些DB对SSD做了优化？**

答：基于SSD的优化就是解决erase-before-write产生的写入放大的问题，不同类型的IO分离，减少写操作带来的性能影响。

**5、优化原理？**

答：1.将sequential logging修改为In-page logging，避免对相同位置的反复擦写。

2.通过缓存写入的方式将大量的in-place update随机写入合并为少量顺序写入。

3.利用SSD随机读写能力高的特点，减少写增加读，从而达到整体性能的提升。

**扩展题**

**Epoll（）是否使用用户态和内核态共享内存，请分析说明。**

答：

epoll\_wait实现的功能是将有关的内容从内核态拷贝到用户态，并不存在内核态共享内存。可以查看epoll的源码。

Epoll\_wait源码：

/\*

\* Implement the event wait interface for the eventpoll file. It is the kernel

\* part of the user space epoll\_wait(2).

\*/

asmlinkage long sys\_epoll\_wait(int epfd, struct epoll\_event \_\_user \*events,

int maxevents, int timeout)

{

int error;

struct file \*file;

struct eventpoll \*ep;

DNPRINTK(3, (KERN\_INFO "[%p] eventpoll: sys\_epoll\_wait(%d, %p, %d, %d)\n",

current, epfd, events, maxevents, timeout));

/\* The maximum number of event must be greater than zero \*/

if (maxevents <= 0 || maxevents > MAX\_EVENTS) // 检查maxevents参数

return -EINVAL;

/\* Verify that the area passed by the user is writeable \*/

// 检查用户空间传入的events指向的内存是否可写

if (!access\_ok(VERIFY\_WRITE, events, maxevents \* sizeof(struct epoll\_event))) {

error = -EFAULT;

goto eexit\_1;

}

/\* Get the "struct file \*" for the eventpoll file \*/

error = -EBADF;

file = fget(epfd); // 获取epfd对应的eventpoll文件的file实例，file结构是在epoll\_create中创建的

if (!file)

goto eexit\_1;

/\*

\* We have to check that the file structure underneath the fd

\* the user passed to us \_is\_ an eventpoll file.

\*/

error = -EINVAL;

if (!is\_file\_epoll(file))

goto eexit\_2;

/\*

\* At this point it is safe to assume that the "private\_data" contains

\* our own data structure.

\*/

ep = file->private\_data;

/\* Time to fish for events ... \*/

// 核心处理函数

error = ep\_poll(ep, events, maxevents, timeout);

eexit\_2:

fput(file);

eexit\_1:

DNPRINTK(3, (KERN\_INFO "[%p] eventpoll: sys\_epoll\_wait(%d, %p, %d, %d) = %d\n",

current, epfd, events, maxevents, timeout, error));

return error;

}

上述epoll\_wait函数调用了ep\_poll函数

ep\_poll函数源码：

static int ep\_poll(struct eventpoll \*ep, struct epoll\_event \_\_user \*events,

int maxevents, long timeout)

{

int res, eavail;

unsigned long flags;

long jtimeout;

wait\_queue\_t wait;

/\*

\* Calculate the timeout by checking for the "infinite" value ( -1 )

\* and the overflow condition. The passed timeout is in milliseconds,

\* that why (t \* HZ) / 1000.

\*/

jtimeout = (timeout < 0 || timeout >= EP\_MAX\_MSTIMEO) ?

MAX\_SCHEDULE\_TIMEOUT : (timeout \* HZ + 999) / 1000;

retry:

write\_lock\_irqsave(&ep->lock, flags);

res = 0;

if (list\_empty(&ep->rdllist)) {

/\*

\* We don't have any available event to return to the caller.

\* We need to sleep here, and we will be wake up by

\* ep\_poll\_callback() when events will become available.

\*/

init\_waitqueue\_entry(&wait, current);

add\_wait\_queue(&ep->wq, &wait);

for (;;) {

/\*

\* We don't want to sleep if the ep\_poll\_callback() sends us

\* a wakeup in between. That's why we set the task state

\* to TASK\_INTERRUPTIBLE before doing the checks.

\*/

set\_current\_state(TASK\_INTERRUPTIBLE);

if (!list\_empty(&ep->rdllist) || !jtimeout)

break;

if (signal\_pending(current)) {

res = -EINTR;

break;

}

write\_unlock\_irqrestore(&ep->lock, flags);

jtimeout = schedule\_timeout(jtimeout);

write\_lock\_irqsave(&ep->lock, flags);

}

remove\_wait\_queue(&ep->wq, &wait);

set\_current\_state(TASK\_RUNNING);

}

/\* Is it worth to try to dig for events ? \*/

eavail = !list\_empty(&ep->rdllist);

write\_unlock\_irqrestore(&ep->lock, flags);

/\*

\* Try to transfer events to user space. In case we get 0 events and

\* there's still timeout left over, we go trying again in search of

\* more luck.

\*/

if (!res && eavail &&

!(res = ep\_events\_transfer(ep, events, maxevents)) && jtimeout)

goto retry;

return res;

}

**ep\_send\_events**函数向用户空间发送就绪事件。

ep\_send\_events()函数将用户传入的内存简单封装到ep\_send\_events\_data结构中，然后调用ep\_scan\_ready\_list()将就绪队列中的事件传入用户空间的内存。

用户空间访问这个结果，进行处理。

参考博客： <https://www.cnblogs.com/apprentice89/p/3234677.html>

<https://blog.csdn.net/zhaobryant/article/details/80557262>