Linux文件系统预读(一)

概述

本博客主要阐述最新内核(linux-3.12)的文件系统预读及时的设计和实现，可能有理解不全面的地方，望各位能多多指教。

所谓预读，是指文件系统为应用程序一次读出比预期更多的文件内容并缓存在page cache中，这样下一次读请求到来时部分页面直接从page cache读取即可。当然，这个细节对应用程序透明，应用程序可能的感觉唯一就是下次读的速度会更快，当然这是好事。

由于应用程序的访问行为存在多样性加上作者对预读的把握不是非常深入，因此，难免存在不是非常精确的地方，望多赐教。我们会通过设置几个情境来分析预读的逻辑。

情境1：顺序读

// 事例代码

{

 ...

 f = open("file", ....);

 ret = read(f, buf, 4096);

 ret = read(f, buf, 2 \* 4096);

 ret = read(f, buf, 4 \* 4096);

 ...

}

该场景非常简单：打开文件，共进行三次读（且是顺序读），那让我们看看操作系统是如何对文件进行预读的。

**Read 1**

第一次进入内核读处理流程时，在page cache中查找该offset对应的页面是否缓存，因为首次读，缓存未命中，触发一次同步预读：

static void do\_generic\_file\_read(struct

 file \*filp, loff\_t \*ppos,

 read\_descriptor\_t \*desc,

 read\_actor\_t actor) {

 ......

 for (;;) {

 ......

 cond\_resched();

find\_page:

 // 如果没有找到，启动同步预读

 page = find\_get\_page(mapping, index);

 if (!page) {

 page\_cache\_sync\_readahead(

 mapping, ra, filp,

 index,

 last\_index - index

 );

该同步预读逻辑最终进入如下预读逻辑：

// 注意: 这里offset 和req\_size其实是页面数量

static unsigned long ondemand\_readahead(

 struct address\_space \*mapping,

 struct file\_ra\_state \*ra,

 struct file \*filp,

 bool hit\_readahead\_marker,

 pgoff\_t offset,

 unsigned long req\_size)

{

 unsigned long max =

 max\_sane\_readahead(ra->ra\_pages);

 // 第一次读文件，直接初始化预读窗口即可

 if (!offset)

 goto initial\_readahead;

 ......

initial\_readahead:

 ra->start = offset;

 ra->size =

 get\_init\_ra\_size(req\_size, max);

 // ra->size 一定是>= req\_size的，这个由

 get\_init\_ra\_size保证

 // 如果req\_size >= max,那么ra-

 >async\_size = ra\_size

 ra->async\_size = ra->size > req\_size ? ra->size - req\_size : ra->size;

readit:

 /\*

 \* Will this read hit the readahead marker made by itself?

 \* If so, trigger the readahead marker hit now, and merge

 \* the resulted next readahead window into the current one.

 \*/

 if (offset == ra->start &&

 ra->size == ra->async\_size) {

 ra->async\_size =

 get\_next\_ra\_size(ra, max);

 ra->size += ra->async\_size;

 }

 return ra\_submit(ra, mapping, filp);

}

读逻辑会为该文件初始化一个预读窗口：

**(ra->start, ra->size, ra->async\_size)**

本例中的预读窗口为(0,4,3)，初始化该预读窗口后调用ra\_submit提交本次读请求。形成的读窗口如下图所示：



图中看到，应用程序申请访问PAGE 0，内核一共读出PAGE0 ~PAGE3，后三个属于预读页面，而且PAGE\_1被标记为PAGE\_READAHEAD，当触发到该页面读时，操作系统会进行一次异步预读，这在后面我们会仔细描述。

等这四个页面被读出时，第一次读的页面已经在pagecache中，应用程序从该page中拷贝出内容即可。

**Read 2**

接下来应用程序进行第二次读，offset=4096， size=8192。内核将其转化为以page为单位计量，offset=1，size=2。即读上面的PAGE1和PAGE2。

感谢第一次的预读，PAGE1和PAGE2目前已经在内存中了，但由于PAGE1被打上了PAGE\_AHEAD标记，读到该页面时会触发一次异步预读：

find\_page:

 ......

 page = find\_get\_page(mapping,

 index);

 if (!page) {

 page\_cache\_sync\_readahead(

 mapping, ra, filp,

 index,

 last\_index - index);

 page = find\_get\_page(

 mapping, index);

 if (unlikely(page == NULL))

 goto no\_cached\_page;

 }

 if (PageReadahead(page)) {

 page\_cache\_async\_readahead(

 mapping, ra, filp,

 page,index,

 last\_index - index);

 }

static unsigned long

ondemand\_readahead(

 struct address\_space \*mapping,

 struct file\_ra\_state \*ra,

 struct file \*filp,

 bool hit\_readahead\_marker,

 pgoff\_t offset,

 unsigned long req\_size)

{

 unsigned long max =

 max\_sane\_readahead(ra->ra\_pages);

 ........

 /\* 如果:

 \* 1. 顺序读(本次读偏移为上次读偏移 (ra->start) + 读大小(ra->size,包含预读量) -

 \* 上次预读大小(ra->async\_size))

 \* 2. offset == (ra->start + ra->size)???

 \*/

 if ((offset == (ra->start + ra->size - ra->async\_size)

 ||

 offset == (ra->start + ra->size))) {

 // 设置本次读的offset,以page为单位

 ra->start += ra->size;

 ra->size = get\_next\_ra\_size(

 ra, max);

 ra->async\_size = ra->size;

 goto readit;

 }

经历了第一次预读，文件的预读窗口状态为(ra->start,ra->size, ra->async\_size)=(0, 4, 3)，本次的请求为(offset,size)=(1, 2)，上面代码的判断条件成立，因此我们会向前推进预读窗口，此时预读窗口变为

**(ra->start,ra->size, ra->async\_size) = (4, 8, 8)**

由于本次是异步预读，应用程序可以不等预读完成即可返回，只要后台慢慢读页面即可。本次预读窗口的起始以及大小以及预读大小可根据前一次的预读窗口计算得到，又由于本次是异步预读，因此，预读大小就是本次读的页面数量，因此将本次预读的第一个页面(PAGE 4)添加预读标记。



由于上面的两次顺序读，截至目前，该文件在操作系统中的page cache状态如下：



**Read 3**

接下来应用程序进行第三次读，顺序读，范围是[page3, page6]，上面的预读其实已经将这些页面读入page cache了，但是由于page4被打上了*PAGE\_READAHEAD*标记，因此，访问到该页面时会触发一次异步预读，预读的过程与上面的步骤一致，当前预读窗口为(4,8,8)，满足顺序性访问特征，根据特定算法计算本次预读大小，更新预读窗口为(12,16,16)，新的预读窗口如下：



对该情境简单总结下，由于三次的顺序读加上内核的预读行为，文件的page cache中的状态当前如下图所示:



Linux文件系统预读(二)

概述

前一篇文章仔细描述了最简单的一种预读情况：单进程文件顺序读，且读大小不超过32页面，这里我们来看另外一种情境：单进程文件顺序读，读大小为256KB，看看预读逻辑如何处理这种情况。

情境2

照例首先给出事例代码：

{

 ...

 f = open("file", ....);

 ret = read(f, buf, 40 \* 4096);

 ret = read(f, buf, 16 \* 4096);

 ret = read(f, buf, 32 \* 4096);

 ...

}

事例代码中我们一共进行了三次读，顺序读，且读的大小不定，有超过最大预读量的，也有低于最大预读量的。

**Read 1**

毫无疑问，由于第一次读肯定未在缓存命中，前一篇博客告诉我们需要进行一次同步预读，需要初始化预读窗口

initial\_readahead:

 ra->start = offset;

 ra->size = get\_init\_ra\_size(req\_size, max);

 // ra->size 一定是>= req\_size的，这个由get\_init\_ra\_size保证

 // 如果req\_size >= max,那么ra->async\_size = ra\_size

 ra->async\_size = ra->size > req\_size ? ra->size - req\_size : ra->size;

readit:

 /\*

 \* Will this read hit the readahead marker made by itself?

 \* If so, trigger the readahead marker hit now, and merge

 \* the resulted next readahead window into the current one.

 \*/

 if (offset == ra->start && ra->size == ra->async\_size) {

 ra->async\_size = get\_next\_ra\_size(ra, max);

 ra->size += ra->async\_size;

 }

 return ra\_submit(ra, mapping, filp);

}

在初始化预读窗口中判断得出：ra->size=32 pages，即使应用程序要读的数量是40 pages，这样ra->async\_size = ra->size=32 pages，在readit逻辑判断成立，因此会重设ra->async\_size的值，根据计算应该是32 pages，而总的ra->size=初始值+ra->async\_size=64 pages。形成的预读窗口为(0, 64, 32)，如下图：



由于应用程序本次访问的实际页面是PAGE0 ~PAGE40（由于同步预读会全部在缓存命中），因此在访问过程中会碰到page32，此时触发一次异步预读，并向前推进预读窗口：

 /\* 如果:

 \*\* 1. 顺序读(本次读偏移为上次读偏移 (ra->start) + 读大小(ra->size,包含预读量) -

 \*\* 上次预读大小(ra->async\_size))

 \*\* 2. offset == (ra->start + ra->size)???

 \*/

 if ((offset == (ra->start + ra->size - ra->async\_size) ||

 offset == (ra->start + ra->size))) {

 // 设置本次读的

 ra->start += ra->size;

 ra->size = get\_next\_ra\_size(ra, max);

 ra->async\_size = ra->size;

 goto readit;

 }

更新后的当前预读窗口为(64, 32, 32)，如下：



因此，经过第一次读以后，该文件在内存中page cache状态如下图所示：



**Read 2**

由于第二次读只需读出page40 ~ page55，直接在page cache中命中，也不会触发一次异步预读，预读窗口也不会更新，因此，该过程非常简单。本次读完以后，文件在内存page cache的状态如下：



Read3

应用程序第三次读的范围为page56 ~ page87，由上图可知，这些均可以在page cache中命中，但是由于访问了PAGE64，因此会触发一次异步预读，且当前的预读窗口为(64, 32, 32)，根据上面的算法更新预读窗口为(96, 32, 32)，因此，本次预读完成以后，文件在page cache中的缓存状态如下：



总结

我们可以将本情境与上篇博客中描述的情境对比，其实可以发现两者思想完全一致，只是由于应用程序读的粒度不同导致了预读的粒度更大，仅此而已。

Linux文件预读(三)

概述

前面两篇关于文件系统的预读我们通过实例阐述了应用程序的顺序读而触发操作系统对文件大小预取情况，本篇博客我们主要描述预读如何解决交织读的问题。所谓的交织读指的是多线程（进程）读同一个打开的文件描述符，单个线程的顺序读在操作系统看来可能会变成随机读。同样我们还是结合实例来分析。

情境3：交织读

事例代码

{

 ......

 f = open("file", ......)

 pthread\_create(read\_file\_1, f, ...)

 pthread\_create(read\_file\_2, f, ...)

 ......

}

read\_file\_1(f)

{

 lseek(f, 0, SEEK\_SET);

 read(f, ..., 2 \* 4096);

 read(f, ..., 4 \* 4096)

 read(f, ..., 16 \* 4096)

}

read\_file\_2(f)

{

 lseek(f, 128 \* 4096, SEEK\_SET);

 read(f, ..., 2 \* 4096);

 read(f, ..., 4 \* 4096)

 read(f, ..., 16 \* 4096)

}

事例代码中创建了两个线程同时读文件file，每个线程均是顺序读，让我们看看操作系统的预读是如何处理这种情况的。因为多线程的执行顺序可能是多种多样的，我们只列举一种执行流并解释，线程1 read 1，线程2 read 1，线程2 read 2，线程1 read 2，线程1 read 3，线程2 read3。

**线程1 Read 1**

线程1读文件的前两个页面，由于尚未缓存命中，因此会触发文件系统的一次同步预读，确定预读窗口为

**(ra->start, ra->size, ra->async\_size) = (0, 4, 2)**

形成的预读窗口如下：



**线程2 Read 1**

线程2读文件的128和129两个页面，由于这两个页面也尚未缓存在page cache中，也必须启动一次同步预读，这里会更改上面的预读窗口为(128, 4, 2)，更新后的预读窗口如下：



线程2 Read 2

由于本次读和上次读是顺序读，且本次访问的4个页面有两个缓存命中，但由于访问了PAGE 130，而该页面又被打上了异步预读标记，因此在访问页面130的时候会触发一次异步预读，更新预读窗口为（132， 8， 8），如下：



由于本次会访问4个页面，因此PAGE 132也会被访问，从而又触发一次异步预读，更新预读窗口为（140, 16, 16），最终形成的预读窗口如下：



线程2两次读 read1 和read 2以后形成的page cache状态如下所示：



**线程1 Read 2**

接下来线程1进行第二次读，范围是PAGE 2 ~ PAGE 5，由于线程1 read 1将PAGE 2 和PAGE 3已经预读进page cache，因此可直接命中，但在访问PAGE 2的时候会触发一次异步预读，所以这里会更新预读窗口，但很不幸，预读窗口保存的是线程2的预读状态，因此本次访问和之前的预读窗口并不连续，因此我们必须想办法来恢复线程1的之前的预读状态，会触发下面的执行逻辑：

if (hit\_readahead\_marker) {

 pgoff\_t start;

 rcu\_read\_lock();

 //计算本次应该从哪个页面开始读

 // 计算的方法是:从上次的offset开始查找，找到第一个没有缓存在page cache 的页面

 start = radix\_tree\_next\_hole(&mapping->page\_tree, offset+1,max);

 rcu\_read\_unlock();

 // 如果没有找到或者与本次读的偏移相差甚大，那么其实无需再读了

 if (!start || start - offset > max)

 return 0;

 ra->start = start;

 ra->size = start - offset; /\* old async\_size \*/

 ra->size += req\_size;

 ra->size = get\_next\_ra\_size(ra, max);

 //既然是异步预读，那读出的所有页面均是提前读的，因此设置async\_size = size

 ra->async\_size = ra->size;

 goto readit;

 }

这里恢复线程1的预读窗口方法也比较简单：从本次预读的页面开始向后搜索，找到第一个没有缓存在page cache的页面，本例中是page4，然后以此为本次预读的起始页面号，并可以计算出上次的预读窗口大小（page 4 – page 2 = 2），根据这两个值便可确定本次预读窗口为(4, 8, 8)。
更新后的预读窗口如下图所示：



在访问页面4时，会再次出发异步预读，更新预读窗口为（8, 8, 8），如下图所示：



因此，线程1经过read 1 和read 2，形成的page cache状态如下：



**线程1 Read 3**

线程1第三次读的页面是PAGE 6 ~ PAGE 13，全部在缓存命中，但在访问PAGE 8的时候会触发一次异步预读，更新预读窗口为(16, 16, 16)。

在线程1经历了三次读以后，page cache的状态如下图所示：



**线程2 Read 3**

线程2第三次读页面是PAGE 134 ~ PAGE 141，这些全在缓存中命中，但是访问PAGE 140时会触发一次异步预读。更新预读窗口，但是很不幸，之前的预读窗口是线程1的，因此我们必须搜寻才能恢复线程2的预读窗口，搜寻过程之前已经描述，这里不再啰嗦，恢复出线程2的预读窗口为(156, 32,32)。因此，总的来看，由于线程2的三次读形成的page cache状态如下：

