昵称：胡潇
园龄：4年9个月
粉丝：56
关注：2
+加关注

< 2018年7月 >

日	一	二	三	四	五	六
24	25	26	27	28	29	30
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签

随笔分类

- Data Structures
- Information Retrieval(1)
- Learning to Rank
- Leetcode(111)
- Operating System(2)
- The Art of Sequence

随笔档案

- 2015年9月 (4)
- 2015年8月 (7)
- 2015年7月 (3)
- 2015年6月 (12)
- 2015年5月 (14)
- 2015年4月 (11)
- 2015年3月 (10)
- 2015年2月 (16)
- 2015年1月 (24)
- 2014年12月 (11)
- 2014年11月 (11)

最新评论

1. Re:认真分析mmap：是什么 为什么 怎么用
mmap = mmp 23333
--刘铁铲
2. Re:认真分析mmap：是什么 为什么 怎么用
@goodyboy6mmap操作系统管理的一块内存，所有进程地址空间中的mmap段都会映射到这个内存块上，他不是某个进程私有的。...
--LYCode
3. Re:认真分析mmap：是什么 为什么 怎么用

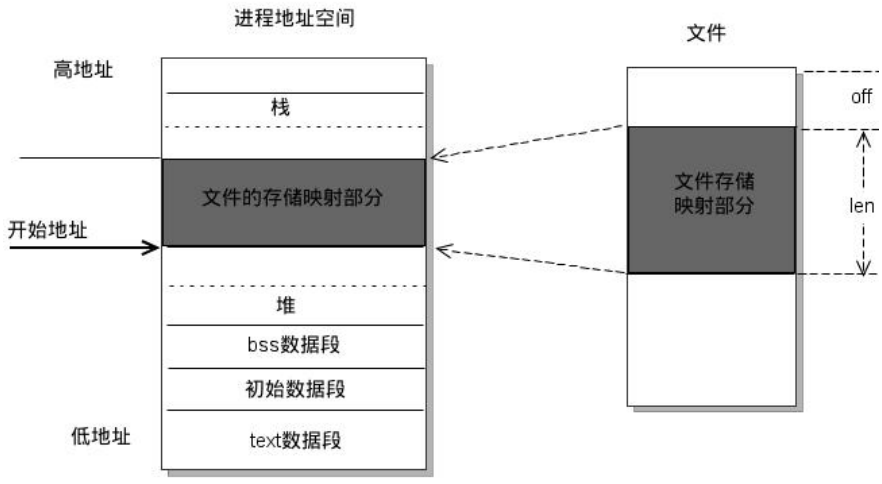
认真分析mmap：是什么 为什么 怎么用

阅读目录

回到顶部

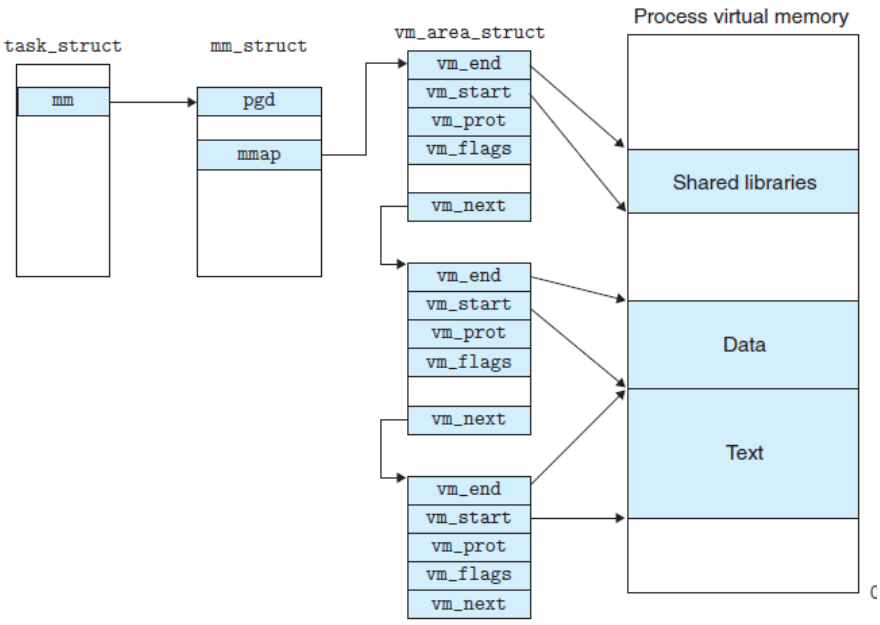
mmap基础概念

mmap是一种内存映射文件的方法，即将一个文件或者其它对象映射到进程的地址空间，实现文件磁盘地址和进程虚拟地址空间中一段虚拟地址的一一对映关系。实现这样的映射关系后，进程就可以采用指针的方式读写操作这一段内存，而系统会自动回写脏页面到对应的文件磁盘上，即完成了对文件的操作而不必再调用read,write等系统调用函数。相反，内核空间对这段区域的修改也直接反映用户空间，从而可以实现不同进程间的文件共享。如下图所示：



由上图可以看出，进程的虚拟地址空间，由多个虚拟内存区域构成。虚拟内存区域是进程的虚拟地址空间中的一个同质区间，即具有同样特性的连续地址范围。上图中所示的text数据段（代码段）、初始数据段、BSS数据段、堆、栈和内存映射，都是一个独立的虚拟内存区域。而为内存映射服务的地址空间处在堆栈之间的空余部分。

linux内核使用vm_area_struct结构来表示一个独立的虚拟内存区域，由于每个不同质的虚拟内存区域功能和内部机制都不同，因此一个进程使用多个vm_area_struct结构来分别表示不同类型的虚拟内存区域。各个vm_area_struct结构使用链表或者树形结构链接，方便进程快速访问，如下图所示：



vm_area_struct结构中包含区域起始和终止地址以及其他相关信息，同时也包含一个vm_ops指针，其内部可引出所有针对这个区域可以使用的系统调用函数。这样，进程对某一虚拟内存区域的任何操作需要用要的信息，都可以从vm_area_struct中获得。mmap函数就是要创建一个新的vm_area_struct结构，并将其与文件的物理磁盘地址相连。具体步骤请看下一节。

回到顶部

mmap内存映射原理

mmap内存映射的实现过程，总的来说可以分为三个阶段：

（一）进程启动映射过程，并在虚拟地址空间中为映射创建虚拟映射区域

1、进程在用户空间调用库函数mmap，原型：
```
void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

2、在当前进程的虚拟地址空间中，寻找一段空闲的满足要求的连续的虚拟地址

3、为此虚拟区分配一个vm_area_struct结构，接着对这个结构的各个域进行了初始化

4、将新建的虚拟区结构（vm_area_struct）插入进程的虚拟地址区域链表或树中

（二）调用内核空间的系统调用函数mmap（不同于用户空间函数），实现文件物理地址和进程虚拟地址的一一映射关系

5、为映射分配了新的虚拟地址区域后，通过待映射的文件指针，在文件描述符表中找到对应的文件描述符，通过文件描述符，链接到内核“已打开文件集”中该文件的文件结构体（struct file），每个文件结构体维护着和这个已打开文件相关各项信息。

6、通过该文件的文件结构体，链接到file_operations模块，调用内核函数mmap，其原型为：int mmap(struct file *filp, struct vm_area_struct *vma)，不同于用户空间库函数。

7、内核mmap函数通过虚拟文件系统inode模块定位到文件磁盘物理地址。

8、通过remap_pfn_range函数建立页表，即实现了文件地址和虚拟地址区域的映射关系。此时，这片虚拟地址并没有任何数据关联到主存中。

（三）进程发起对这片映射空间的访问，引发缺页异常，实现文件内容到物理内存（主存）的拷贝

注：前两个阶段仅在于创建虚拟区间并完成地址映射，但是并没有将任何文件数据的拷贝至主存。真正的文件读取是当进程发起读或写操作时。

9、进程的读或写操作访问虚拟地址空间这一段映射地址，通过查询页表，发现这一段地址并不在物理页面上。因为目前只建立了地址映射，真正的硬盘数据还没有拷贝到内存中，因此引发缺页异常。

10、缺页异常进行一系列判断，确定无非法操作后，内核发起请求调页过程。

11、调页过程先在交换缓存空间（swap cache）中寻找需要访问的内存页，如果没有则调用nopage函数把所缺的页从磁盘装入到主存中。

12、之后进程即可对这片主存进行读或者写的操作，如果写操作改变了其内容，一定时间后系统会自动回写脏页面到对应磁盘地址，也即完成了写入到文件的过程。

注：修改过的脏页面并不会立即更新回文件中，而是有一段时间的延迟，可以调用msync()来强制同步， 这样所写的内容就能立即保存到文件里了。

回到顶部

mmap和常规文件操作的区别

对linux文件系统不了解的朋友，请参阅我之前写的博文《从内核文件系统看文件读写过程》，我们首先简单的回顾一下常规文件系统操作（调用read/fread等类函数）中，函数的调用过程：

1、进程发起读文件请求。

2、内核通过查找进程文件符表，定位到内核已打开文件集上的文件信息，从而找到此文件的inode。

3、inode在address_space上查找要请求的文件页是否已经缓存在页缓存中。如果存在，则直接返回这片文件页的内容。

4、如果不存在，则通过inode定位到文件磁盘地址，将数据从磁盘复制到页缓存。之后再次发起读页面过程，进而将页缓存中的数据发给用户进程。

总结来说，常规文件操作为了提高读写效率和保护磁盘，使用了页缓存机制。这样造成读文件时需要先将文件页从磁盘拷贝到页缓存中，由于页缓存处在内核空间，不能被用户进程直接寻址，所以还需要将页缓存中数据页再次拷贝到内存对应的用户空间中。这样，通过了两次数据拷贝过程，才能完成进程对文件内容的获取任务。写操作也是一样，待写入的buffer在内核空间不能直接访问，必须要先拷贝至内核空间对应的主存，再写回磁盘中（延迟写回），也是需要两次数据拷贝。

而使用mmap操作文件中，创建新的虚拟内存区域和建立文件磁盘地址和虚拟内存区域映射这两步，没有任何文件拷贝操作。而之后访问数据时发现内存中并无数据而发起的缺页异常过程，可以通过已经建立好的映射关系，只使用一次数据拷贝，就从磁盘中将数据传入内存的用户空间中，供进程使用。

总而言之，常规文件操作需要从磁盘到页缓存再到用户主存的两次数据拷贝。而mmap操控文件，只需要从磁盘到用户主存的一次数据拷贝过程。说白了，mmap的关键点是实现了用户空间和内核空间的数据直接交互而省去了空间不同数据不通的繁琐过程。因此mmap效率更高。

回到顶部

mmap优点总结

由上文讨论可知，mmap优点共有一下几点：

1、对文件的读取操作跨过了页缓存，减少了数据的拷贝次数，用内存读写取代I/O读写，提高了文件读取效率。

2、实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉。

3、提供进程间共享内存及相互通信的方式。不管是父子进程还是无亲缘关系的进程，都可以将自身用户空间映射到同一个文件或匿名映射到同一片区域。从而通过各自对映射区域的改动，达到进程间通信和进程间共享的目的。

同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。

4、可用于实现高效的大规模数据传输。内存空间不足，是制约大数据操作的一个方面，解决方案往往是借助硬盘空间协助操作，补充内存的不足。但是进一步会造成大量的文件I/O操作，极大影响效率。这个问题可以通过mmap映射很好的解决。换句话说，但凡是需要用磁盘空间代替内存的时候，mmap都可以发挥其功效。

回到顶部

mmap相关函数

函数原型

```
void *mmap(void *start, size_t length, int prot, int flags,
int fd, off_t offset);
```

返回说明

成功执行时，mmap()返回被映射区的指针。失败时，mmap()返回MAP_FAILED[其值为(void *)-1]， error被设为以下的某个值：

返回错误类型

参数

start：映射区的开始地址

length：映射区的长度

prot：期望的内存保护标志，不能与文件的打开模式冲突。是以下的某个值，可以通过or运算合理地组合在一起

prot

flags：指定映射对象的类型，映射选项和映射页是否可以共享。它的值可以是一个或者多个以下位的组合体

flag

fd：有效的文件描述词。如果MAP_ANONYMOUS被设定，为了兼容问题，其值应为-1

offset：被映射对象内容的起点

相关函数

```
int munmap( void * addr, size_t len )
```

成功执行时，munmap()返回0。失败时，munmap返回-1，error返回标志和mmap一致；

该调用在进程地址空间中解除一个映射关系，addr是调用mmap()时返回的地址，len是映射区的大小；

当映射关系解除后，对原来映射地址的访问将导致段错误发生。

```
int msync( void *addr, size_t len, int flags )
```

一般说来，进程在映射空间的对共享内容的改变并不直接写回到磁盘文件中，往往在调用munmap（）后才执行该操作。

可以通过调用msync()实现磁盘上文件内容与共享内存区的内容一致。

回到顶部

mmap使用细节

1、使用mmap需要注意的一个关键点是，mmap映射区域大小必须是物理页大小(page_size)的整倍数（32位系统中通常是4k字节）。原因是，内存的最小粒度是页，而进程虚拟地址空间和内存的映射也是以页为单位。为了匹配内存的操作，mmap从磁盘到虚拟地址空间的映射也必须是页。

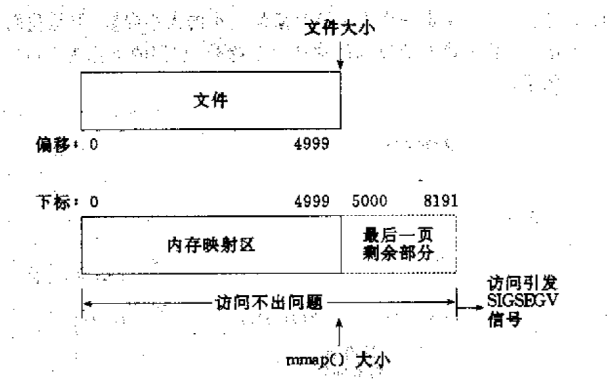
2、内核可以跟踪被内存映射的底层对象（文件）的大小，进程可以合法的访问在当前文件大小以内又在内存映射区以内的那些字节。也就是说，如果文件的大小一直在扩张，只要在映射区域范围内的数据，进程都可以合法得到，这和映射建立时文件的大小无关。具体情形参见“情形三”。

3、映射建立之后，即使文件关闭，映射依然存在。因为映射的是磁盘的地址，不是文件本身，和文件句柄无关。同时可用于进程间通信的有效地址空间不完全受限于被映射文件的大小，因为是按页映射。

在上面的知识前提下，我们下面看看如果大小不是页的整倍数的具体情况：

情形一：一个文件的大小是5000字节，mmap函数从一个文件的起始位置开始，映射5000字节到虚拟内存中。

分析：因为单位物理页面的大小是4096字节，虽然被映射的文件只有5000字节，但是对应到进程虚拟地址区域的大小需要满足整页大小，因此mmap函数执行后，实际映射到虚拟内存区域8192个 字节，5000~8191的字节部分用零填充。映射后的对应关系如下图所示：



此时：

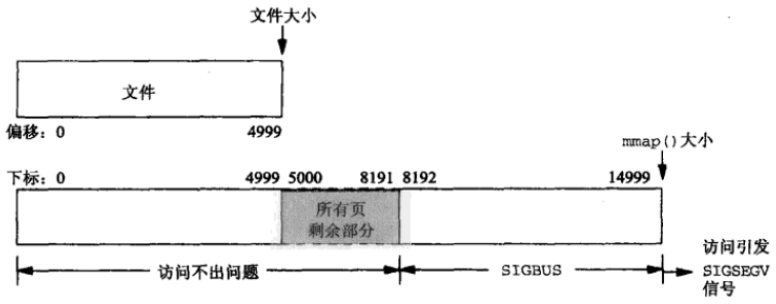
（1）读/写前5000个字节（0~4999），会返回操作文件内容。

（2）读字节5000~8191时，结果全为0。写5000~8191时，进程不会报错，但是所写的内容不会写入原文件中 。

（3）读/写8192以外的磁盘部分，会返回一个SIGSECV错误。

情形二：一个文件的大小是5000字节，mmap函数从一个文件的起始位置开始，映射15000字节到虚拟内存中，即映射大小超过了原始文件的大小。

分析：由于文件的大小是5000字节，和情形一一样，其对应的两个物理页。那么这两个物理页都是合法可以读写的，只是超出5000的部分不会体现在原文件中。由于程序要求映射15000字节，而文件只占两个物理页，因此8192字节~15000字节都不能读写，操作时会返回异常。如下图所示：



此时：

（1）进程可以正常读/写被映射的前5000字节(0~4999)，写操作的改动会在一定时间后反映在原文件中。

（2）对于5000~8191字节，进程可以进行读写过程，不会报错。但是内容在写入前均为0，另外，写入后不会反映在文件中。

（3）对于8192~14999字节，进程不能对其进行读写，会报SIGBUS错误。

（4）对于15000以外的字节，进程不能对其读写，会引发SIGSEGV错误。

情形三：一个文件初始大小为0，使用mmap操作映射了1000*4K的大小，即1000个物理页大约4M字节空间，mmap返回指针ptr。

53 1

分析：如果在映射建立之初，就对文件进行读写操作，由于文件大小为0，并没有合法的物理页对应，如同情形二一样，会返回SIGBUS错误。

但是如果，每次操作ptr读写前，先增加文件的大小，那么ptr在文件大小内部的操作就是合法的。例如，文件扩充4096字节，ptr就能操作ptr ~ [(char)ptr + 4095]的空间。只要文件扩充的范围在1000个物理页（映射范围）内，ptr都可以对应操作相同的大小。

这样，方便随时扩充文件空间，随时写入文件，不造成空间浪费。

分类: Operating System

标签: operating system， file system

胡潇
关注 - 2
粉丝 - 56
+加关注

« 上一篇： 从内核文件系统看文件读写过程
» 下一篇： 【Leetcode】【Easy】Compare Version Numbers

posted @ 2015-07-20 10:35 胡潇 阅读(70395) 评论(21) 编辑 收藏

评论列表

#1楼 2015-07-20 11:12 王洪旭

good

支持(0) 反对(0)

#2楼 2015-07-20 11:31 myg

楼主能否举一些使用mmap的例子？

支持(0) 反对(0)

#3楼[楼主] 2015-07-20 12:32 胡潇

@ myg
您好，我下篇文章会写使用mmap的方法建立信息检索系统中的大规模倒排索引文件，也会附上我写的代码。不过，其中并不包括进程间通信共享内存的例子。如果您对进程间通信更感兴趣，望您再参阅其他的书籍文章。

支持(0) 反对(0)

#4楼 2015-07-21 10:22 wm007

mongoDB （ 3.0以前版本 ）、rocketMQ 都用到了 mmap

支持(0) 反对(0)

#5楼 2015-07-23 12:07 foreach_break

楼主辛苦了.

1. linux是否称文件描述符比“句柄”更佳.

2. mmap之所以快，是因为建立了页到用户进程的虚地址空间映射，以读取文件为例，避免了页从内核态拷贝到用户态。

3. 除第2点之外，mmap映射的页和其它的页并没有本质的不同.
所以得益于主要的3种数据结构的高效，其页映射过程也很高效：
(1) radix tree，用于查找某页是否已在缓存.
(2) red black tree ，用于查找和更新vma结构.
(3) 双向链表，用于维护active和inactive链表，支持LRU类算法进行内存回收.

4. mmap不是银弹.
(1) 对变长文件不适合.
(2) 如果更新文件的操作很多，mmap避免两态拷贝的优势就被摊还，最终还是落在了大量的脏页回写及由此引发的随机IO上.

所以在随机写很多的情况下，mmap方式在效率上不一定会比带缓冲区的一般写快.

5. 之前接触的mongodb版本无法控制对内存的使用，所以当数据比较大时，其内存使用难以控制，一切的内存 -> 页 -> swap 步骤都交给了操作系统，难以进行有效调优.频繁的页swap会让mongodb的优势荡然无从.

have fun :]

支持(8) 反对(0)

#6楼[楼主] 2015-07-23 21:33 胡潇

@ foreach_break
非常非常感谢您，我需要学习的地方还有很多。
您说的几点我会仔细学习研究，得到些心得再回来和您讨论。
再次感谢，我会尽量提升文章深度和准确性，望您继续关注！

支持(0) 反对(0)

#7楼 2015-07-24 00:41 foreach_break

@ 胡潇
你很有天赋，我会关注。

同时那些只是我的理解。你不要盲从 :]

支持(0) 反对(0)

#8楼 2015-07-24 09:45 请叫我头头哥

@ foreach_break
阿弥陀佛

支持(0) 反对(0)

#9楼 2015-07-24 12:13 foreach_break

@ 请叫我头头哥
头头哥神出鬼没的..

支持(0) 反对(0)

#10楼 2016-04-09 11:31 oyld

引用

同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。

C不是映射到不同虚拟空间中吗？为什么感觉A和B都可以共享C映射的虚拟空间了，这里没有弄明白。

支持(1) 反对(0)

#11楼 2016-08-02 22:42 水滴_石穿

mark.good

支持(0) 反对(0)

#12楼 2016-11-15 15:27 zhao01

博主说mmap“实现了用户空间和内核空间的高效交互方式。两空间的各自修改操作可以直接反映在映射的区域内，从而被对方空间及时捕捉”，这我不太理解。
mmap可以用于两个用户态程序的通信，这个好理解。可是如何用mmap做到用户态程序和内核的通信呢？这个要怎么做到呢？
难道是，内核先去查看用户态程序的vm_area_struct链表，再去查询用户态程序的页表，来确定用户态程序mmap到的物理内存在哪里吗？然后再直接读取或修改这块内存，来和用户态程序交互？不知道你说的是不是这个意思？

支持(0) 反对(0)

#13楼 2017-02-22 15:23 柔软的胖

man mmap，发现一段demo程序，其中设置length和offset的代码，看的不是太明白。offset和len是用户输入参数。在调用mmap时，对这两个参数做了处理。为什么要这样处理呢？

```
offset = atoi(argv[2]);
/* offset for mmap() must be page aligned */
pa_offset = offset & ~(sysconf(_SC_PAGE_SIZE) - 1);

if (argc == 4) {
len = atoi(argv[3]);
/* Can't display bytes past end of file */
if (offset + len > sb.st_size)
len = sb.st_size - offset;
} else {
```

53 1

```
/* No length arg ==> display to end of file */
len = sb.st_size - offset;
}
addr = mmap(NULL, len + offset - pa_offset, PROT_READ,
MAP_SHARED, fd, pa_offset);
```

支持(0) 反对(0)

#14楼 2017-04-04 17:44 lisa_520

nice

支持(0) 反对(0)

#15楼 2017-06-13 14:18 sys0613

楼主，不知道您还关注这篇博文不，我遇到了小问题。。。例如我文件204800大小，我每次映射102400大小到内存中，我参数是第一次a=mmap（0，102400，PROT_READ,MAP_PRIVATE|MAP_NORESERVE,文件句柄，0）；执行munmap（a,102400）;前面步骤没有报错，我继续执行第二次mmap（0，102400，PROT_READ,MAP_PRIVATE|MAP_NORESERVE,文件句柄，102398）；我想继续读取文件，并回退了2个位置，继续读取，为何就报error=-1了，我查看错误码是22，Invalid argument，关键是哪个参数错了呢？第二个参数是4096的倍数就行了吧，最后一个参数没必要是4096倍数吧？多谢

支持(0) 反对(0)

#16楼 2017-09-07 16:06 guang_blog

问个问题，linux两个进程同时使用mmap写满同样大小的大文件（超过物理内存+swap）的过程中，一开始运行正常，过一会后，两个进程均出现SIGBUS信号，为什么？

支持(0) 反对(0)

#17楼 2017-09-18 12:07 长安怎乱

不错不错，收藏了。

推荐下，RocketMQ 源码解析 14 篇：http://www.yunai.me/categories/RocketMQ/?cnblog&601

晏

支持(0) 反对(0)

#18楼 2017-09-18 12:07 长安怎乱

不错不错，收藏了。

推荐下，RocketMQ 源码解析 14 篇：http://www.yunai.me/categories/RocketMQ/?cnblog&601

晏

支持(0) 反对(0)

#19楼 2018-04-10 16:20 goodyboy6

“同时，如果进程A和进程B都映射了区域C，当A第一次读取C时通过缺页从磁盘复制文件页到内存中；但当B再读C的相同页面时，虽然也会产生缺页异常，但是不再需要从磁盘中复制文件过来，而可直接使用已经保存在内存中的文件数据。”
这里面说的“可直接使用已经保存在内存中”这个内存是系统的分配给进程的内存吧？分配给进程的内存是受保护的，其他进程是不能访问的。

支持(0) 反对(0)

#20楼 2018-06-05 10:00 LYCode

@ goodyboy6
mmap操作系统管理的一块内存，所有进程地址空间中的mmap段都会映射到这个内存块上，他不是某个进程私有的。

支持(0) 反对(0)

#21楼 2018-06-29 15:27 刘铁铲

mmap = mmp 23333

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。

最新IT新闻

:

· SUSE Linux以25亿美元出售给EQT Partners
· 改进GitHub工作流的15个建议
· 为什么科技产品总是喜欢用蓝色？
· 从这些订阅号生产者的焦虑中 我们还看到了什么
· “无人驾驶先驱”莱万多夫斯基低调复出 曾流星般崛起又陨落
» 更多新闻...

最新知识库文章

:

· 如何提升你的能力？给年轻程序员的几条建议
· 程序员的那些反模式
· 程序员的宇宙时间线
· 突破程序员思维
· 云、雾和霭计算如何一起工作
» 更多知识库文章...