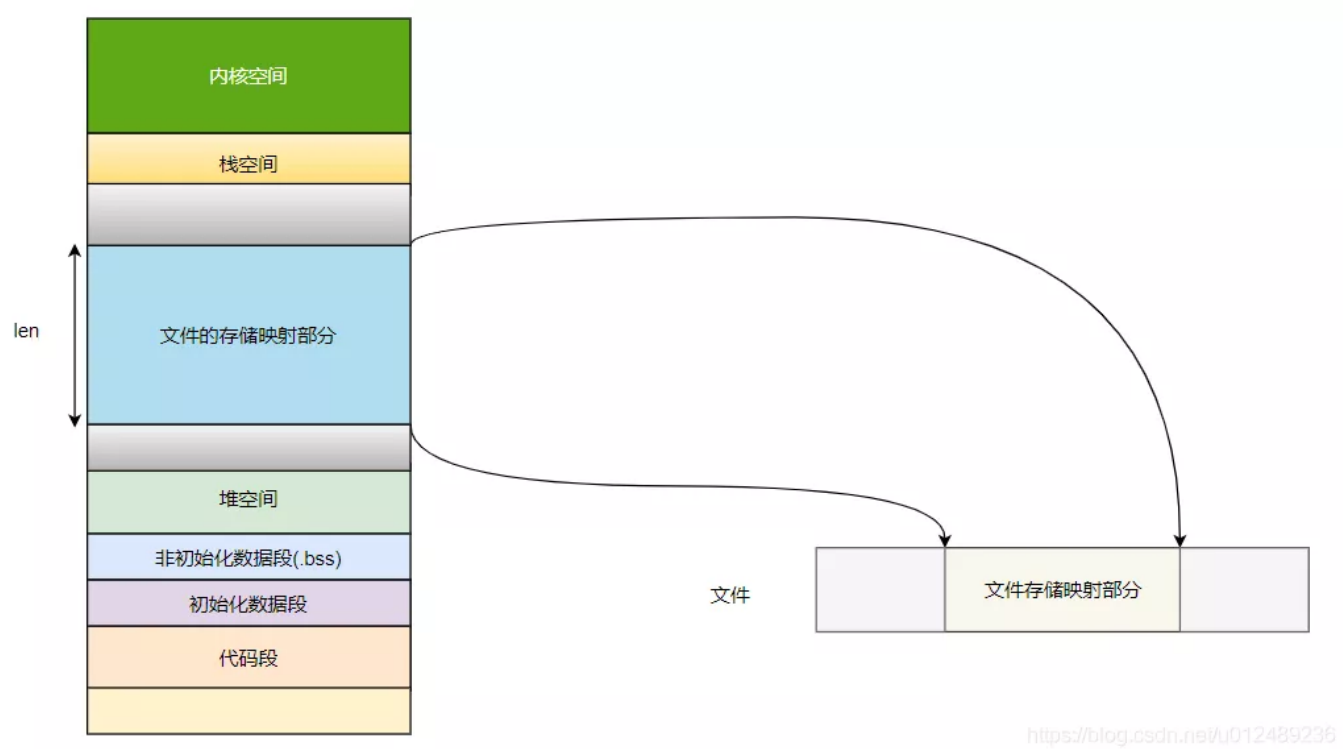
完全分析 Linux mmap 原理

Linux内核那些事 2022-02-13 19:29

内存映射是一个很有趣的思想，我们都知道操作系统分为用户态和内核态，用户态是不能直接和物理设备打交道，如果我们用户空间想访问硬盘的一块区域数据，则需要两次拷贝(硬盘->内核->用户)，但是内存映射的设计只需要发生一次拷贝，大大提高了读取数据的效率。那么内存映射的原理和内核是如何实现的呢？

1. 内存映射概念

内存映射，简而言之就是将用户空间的一段内存区域映射到内核空间，映射成功后，用户对这段内存区域的修改可以直接反映到内核空间，同样，内核空间对这段区域的修改也直接反映给用户空间，对于用户空间和内核空间两者之间需要进行大量数据传输等操作的话效率是非常高的。如下图所示



实现这样的映射后，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到对应的文件磁盘上，就可以完成对于文件的操作，而不需要再调用read/write等系统调用函数。同时，内核空间对于这段区域的修改也可以直接反映到用户空间，从而可以实现不同进程间的文件共享。

`mmap/munmap` 接口是常用的内存映射的系统调用接口，无论是在用户空间分配内存、读写大文件、连接动态库文件，还是多进程间共享内存，都可以看到其身影，其声明如下：

```
#include <sys/mman.h>
void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset);
int munmap(void *addr, size_t length);
```

条件：

mmap()必须以PAGE_SIZE为单位进行映射，而内存也只能以页为单位进行映射，若要映射非PAGE_SIZE整数倍的地址范围，要先进行内存对齐，强行以PAGE_SIZE的倍数大小进行映射。

参数说明：

- `start` ： 映射区的开始地址，设置为0时表示由系统决定映射区的起始地址。
- `length` ： 映射区的长度。//长度单位是 以字节为单位，不足一内存页按一内存页处理
- `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_EXEC: 表示映射的页面是可以执行的
  - PROT_READ：表示映射的页面是可以读取的
  - PROT_WRITE ： 表示映射的页面是可以写入的
  - PROT_NONE ： 表示映射的页面是不可访问的
- 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
  - MAP_SHARED：创建一个共享映射的区域，多个进程可以通过共享映射的方式来映射一个文件，这样其他进程也可以看到映射内容的改变，修改后的内容会同步到磁盘文件
  - MAP_PRIVATE：创建一个私有的写时复制的映射，多个进程可以通过私有映射方式来映射一个文件，其他的进程不会看到映射文件内容的改变，修改后也不会同步到磁盘中
  - MAP_ANONYMOUS：创建一个匿名映射，即没有关联到文件的映射
  - MAP_FIXED：
  - MAP_POPULATE：提前遇到文件内容到映射区
- fd：mmap映射释放和文件相关联，可以分为匿名映射和文件映射
  - 文件映射：将一个普通文件的全部或者一部分映射到进程的虚拟内存中。映射后，进程就可以直接在对应的内存区域操作文件内容!
  - 匿名映射：匿名映射没有对应的文件或者对应的文件时虚拟文件(如：/dev/zero)，映射后会把内存分页全部初始化为0。
- offset：被映射对象内容的起点

返回说明：

成功执行时，mmap()返回被映射区的指针，munmap()返回0。失败时，mmap()返回MAP_FAILED[其值为(void *)-1]，munmap返回-1。

根据文件关联性和映射区域示范共享等属性，其分为

	私有映射	**共享映射**
匿名映射	私有匿名映射(通常用于内存分配)，当使用大于128K内存时 fd = -1 且 flags = MAP_ANONYMOUS | MAP_PRIVATE	共享匿名映射(通常用于父子进程间共享) fd = -1 且 flags = MAP_ANONYMOUS | MAP_SHARED
文件映射	私有文件映射(通常用于动态库加载)	共享文件映射(通常用于内存映射IO、进程间通信)

2. 源码分析

查看 `mmap` 的系统调用的代码实现，其流程为 `sys_mmp_pg_off()` ，最终会调用达到 `do_mmap_pgoff` ，该函数使一个体系结构无关的代码，定义在 `mm/mmap.c` 中，

首先我们来看看 `do_mmap()` ，是整个 `mmap()` 的具体操作函数

```
unsigned long do_mmap(struct file *file, unsigned long addr,
   unsigned long len, unsigned long prot,
   unsigned long flags, vm_flags_t vm_flags,
```

```
    unsigned long pgoff, unsigned long *populate)
{
 struct mm_struct *mm = current->mm;         //获取该进程的memory descriptor
 int pkey = 0;

 *populate = 0;
 //函数对传入的参数进行一系列检查，假如任一参数出错，都会返回一个errno
 if (!len)
  return -EINVAL;

 /*
  * Does the application expect PROT_READ to imply PROT_EXEC?
  *
  * (the exception is when the underlying filesystem is noexec
  *  mounted, in which case we dont add PROT_EXEC.)
  */
 if ((prot & PROT_READ) && (current->personality & READ_IMPLIES_EXEC))
  if (!(file && path_noexec(&file->f_path)))
   prot |= PROT_EXEC;
 //假如没有设置MAP_FIXED标志，且addr小于mmap_min_addr，因为可以修改addr，所以就需要将addr设为mmap_min_addr的页对齐
 if (!(flags & MAP_FIXED))
  addr = round_hint_to_min(addr);

 /* Careful about overflows.. */
 len = PAGE_ALIGN(len);          //进行Page大小的对齐
 if (!len)
  return -ENOMEM;

 /* offset overflow? */
```

```
if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
 return -EOVERFLOW;

/* Too many mappings? */
if (mm->map_count > sysctl_max_map_count)    //判断该进程的地址空间的虚拟区间数量是否超过了限制
 return -ENOMEM;

//get_unmapped_area从当前进程的用户空间获取一个未被映射区间的起始地址
addr = get_unmapped_area(file, addr, len, pgoff, flags);
if (offset_in_page(addr))        //检查addr是否有效
 return addr;

if (prot == PROT_EXEC) {
 pkey = execute_only_pkey(mm);
 if (pkey < 0)
  pkey = 0;
}

/* Do simple checking here so the lower-level routines won't have
 * to. we assume access permissions have been handled by the open
 * of the memory object, so we don't do any here.
 */
vm_flags |= calc_vm_prot_bits(prot, pkey) | calc_vm_flag_bits(flags) |
  mm->def_flags | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC;
//假如flags设置MAP_LOCKED，即类似于mlock()将申请的地址空间锁定在内存中，检查是否可以进行lock
if (flags & MAP_LOCKED)
 if (!can_do_mlock())
  return -EPERM;
```

```
if (mlock_future_check(mm, vm_flags, len))
 return -EAGAIN;

if (file) {          // file指针不为nullptr, 即从文件到虚拟空间的映射
 struct inode *inode = file_inode(file);   //获取文件的inode

 switch (flags & MAP_TYPE) {       //根据标志指定的map种类，把为文件设置的访问权考虑进去
 case MAP_SHARED:
  if ((prot&PROT_WRITE) && !(file->f_mode&FMODE_WRITE))
   return -EACCES;

  /*
   * Make sure we don't allow writing to an append-only
   * file..
   */
  if (IS_APPEND(inode) && (file->f_mode & FMODE_WRITE))
   return -EACCES;

  /*
   * Make sure there are no mandatory locks on the file.
   */
  if (locks_verify_locked(file))
   return -EAGAIN;

  vm_flags |= VM_SHARED | VM_MAYSHARE;
  if (!(file->f_mode & FMODE_WRITE))
   vm_flags &= ~(VM_MAYWRITE | VM_SHARED);

  /* fall through */
```

```
 case MAP_PRIVATE:
  if (!(file->f_mode & FMODE_READ))
   return -EACCES;
  if (path_noexec(&file->f_path)) {
   if (vm_flags & VM_EXEC)
    return -EPERM;
   vm_flags &= ~VM_MAYEXEC;
  }

  if (!file->f_op->mmap)
   return -ENODEV;
  if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
   return -EINVAL;
  break;

 default:
  return -EINVAL;
 }
} else {
 switch (flags & MAP_TYPE) {
 case MAP_SHARED:
  if (vm_flags & (VM_GROWSDOWN|VM_GROWSUP))
   return -EINVAL;
  /*
   * Ignore pgoff.
   */
  pgoff = 0;
  vm_flags |= VM_SHARED | VM_MAYSHARE;
```

```
  break;
 case MAP_PRIVATE:
  /*
   * Set pgoff according to addr for anon_vma.
   */
  pgoff = addr >> PAGE_SHIFT;
  break;
 default:
  return -EINVAL;
 }
}

/*
 * Set 'VM_NORESERVE' if we should not account for the
 * memory use of this mapping.
 */
if (flags & MAP_NORESERVE) {
 /* We honor MAP_NORESERVE if allowed to overcommit */
 if (sysctl_overcommit_memory != OVERCOMMIT_NEVER)
  vm_flags |= VM_NORESERVE;

 /* hugetlb applies strict overcommit unless MAP_NORESERVE */
 if (file && is_file_hugepages(file))
  vm_flags |= VM_NORESERVE;
}
//一顿检查和配置，调用核心的代码mmap_region
addr = mmap_region(file, addr, len, vm_flags, pgoff);
if (!IS_ERR_VALUE(addr) &&
    ((vm_flags & VM_LOCKED) ||
```

```
        (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE))
    *populate = len;
  return addr;
}
```

do_mmap() 根据用户传入的参数做了一系列的检查，然后根据参数初始化 vm_area_struct 的标志 vm_flags ， vma->vm_file = get_file(file) 建立文件与vma的映射，mmap_region() 负责创建虚拟内存区域:

```
unsigned long mmap_region(struct file *file, unsigned long addr,
    unsigned long len, vm_flags_t vm_flags, unsigned long pgoff)
{
  struct mm_struct *mm = current->mm;      //获取该进程的memory descriptor
  struct vm_area_struct *vma, *prev;
  int error;
  struct rb_node **rb_link, *rb_parent;
  unsigned long charged = 0;

  /* 检查申请的虚拟内存空间是否超过了限制 */
  if (!may_expand_vm(mm, vm_flags, len >> PAGE_SHIFT)) {
    unsigned long nr_pages;

    /*
     * MAP_FIXED may remove pages of mappings that intersects with
     * requested mapping. Account for the pages it would unmap.
     */
    nr_pages = count_vma_pages_range(mm, addr, addr + len);
```

```
 if (!may_expand_vm(mm, vm_flags,
    (len >> PAGE_SHIFT) - nr_pages))
  return -ENOMEM;
}

/* 检查[addr, addr+len)的区间是否存在映射空间，假如存在重合的映射空间需要munmap  */
while (find_vma_links(mm, addr, addr + len, &prev, &rb_link,
        &rb_parent)) {
 if (do_munmap(mm, addr, len))
  return -ENOMEM;
}

/*
 * Private writable mapping: check memory availability
 */
if (accountable_mapping(file, vm_flags)) {
 charged = len >> PAGE_SHIFT;
 if (security_vm_enough_memory_mm(mm, charged))
  return -ENOMEM;
 vm_flags |= VM_ACCOUNT;
}

//检查是否可以合并[addr, addr+len)区间内的虚拟地址空间vma
vma = vma_merge(mm, prev, addr, addr + len, vm_flags,
  NULL, file, pgoff, NULL, NULL_VM_UFFD_CTX);
if (vma)       //假如合并成功，即使用合并后的vma，并跳转至out
 goto out;
//如果不能和已有的虚拟内存区域合并，通过Memory Descriptor来申请一个vma
vma = kmem_cache_zalloc(vm_area_cachep, GFP_KERNEL);
```

```
if (!vma) {
 error = -ENOMEM;
 goto unacct_error;
}
//初始化vma
vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
vma->vm_flags = vm_flags;
vma->vm_page_prot = vm_get_page_prot(vm_flags);
vma->vm_pgoff = pgoff;
INIT_LIST_HEAD(&vma->anon_vma_chain);

if (file) { //假如指定了文件映射
 if (vm_flags & VM_DENYWRITE) {    //映射的文件不允许写入，调用deny_write_accsess(file)排斥常规的文件操作
  error = deny_write_access(file);
  if (error)
   goto free_vma;
 }
 if (vm_flags & VM_SHARED) {    //映射的文件允许其他进程可见，标记文件为可写
  error = mapping_map_writable(file->f_mapping);
  if (error)
   goto allow_write_and_free_vma;
 }

 //递增File的引用次数，返回File赋给vma
 vma->vm_file = get_file(file);
 error = file->f_op->mmap(file, vma);  //调用文件系统指定的mmap函数
 if (error)
```

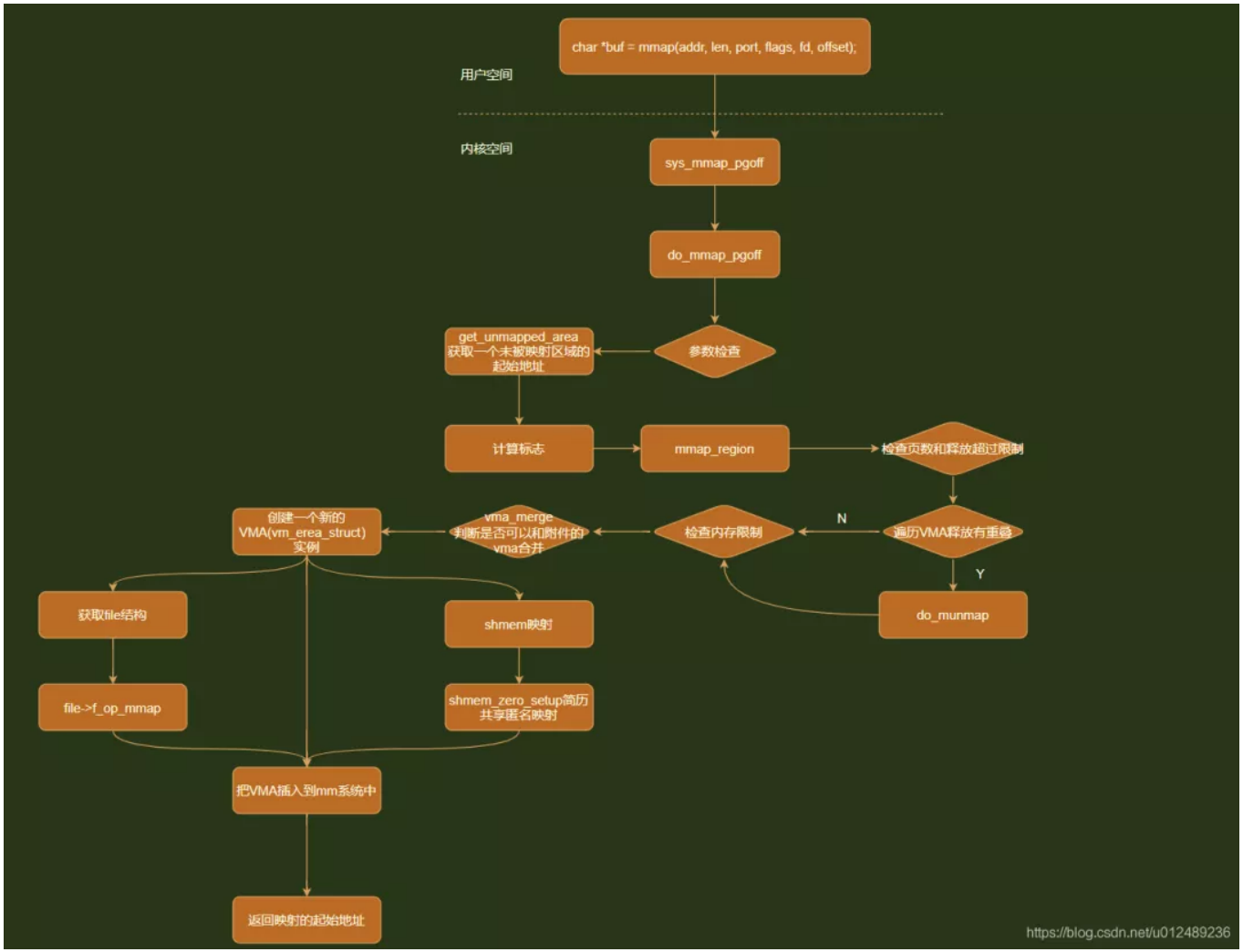
```
    goto unmap_and_free_vma;

  /* Can addr have changed??
   *
   * Answer: Yes, several device drivers can do it in their
   *         f_op->mmap method. -DaveM
   * Bug: If addr is changed, prev, rb_link, rb_parent should
   *      be updated for vma_link()
   */
  WARN_ON_ONCE(addr != vma->vm_start);

  addr = vma->vm_start;
  vm_flags = vma->vm_flags;
 } else if (vm_flags & VM_SHARED) {
  error = shmem_zero_setup(vma);  //假如标志为VM_SHARED，但没有指定映射文件，需要调用shmem_zero_setup(),实际映射
  if (error)
   goto free_vma;
 }
 //将申请的新vma加入mm中的vma链表
 vma_link(mm, vma, prev, rb_link, rb_parent);
 /* Once vma denies write, undo our temporary denial count */
 if (file) {
  if (vm_flags & VM_SHARED)
   mapping_unmap_writable(file->f_mapping);
  if (vm_flags & VM_DENYWRITE)
   allow_write_access(file);
 }
 file = vma->vm_file;
out:
```

```
 perf_event_mmap(vma);
 //更新进程的虚拟地址空间mm
 vm_stat_account(mm, vm_flags, len >> PAGE_SHIFT);
 if (vm_flags & VM_LOCKED) {
  if (!((vm_flags & VM_SPECIAL) || is_vm_hugetlb_page(vma) ||
     vma == get_gate_vma(current->mm)))
   mm->locked_vm += (len >> PAGE_SHIFT);
  else
   vma->vm_flags &= VM_LOCKED_CLEAR_MASK;
 }

 if (file)
  uprobe_mmap(vma);

 /*
  * New (or expanded) vma always get soft dirty status.
  * Otherwise user-space soft-dirty page tracker won't
  * be able to distinguish situation when vma area unmapped,
  * then new mapped in-place (which must be aimed as
  * a completely new data area).
  */
 vma->vm_flags |= VM_SOFTDIRTY;

 vma_set_page_prot(vma);

 return addr;

unmap_and_free_vma:
 vma->vm_file = NULL;
```

```
 fput(file);

 /* Undo any partial mapping done by a device driver. */
 unmap_region(mm, vma, prev, vma->vm_start, vma->vm_end);
 charged = 0;
 if (vm_flags & VM_SHARED)
  mapping_unmap_writable(file->f_mapping);
allow_write_and_free_vma:
 if (vm_flags & VM_DENYWRITE)
  allow_write_access(file);
free_vma:
 kmem_cache_free(vm_area_cachep, vma);
unacct_error:
 if (charged)
  vm_unacct_memory(charged);
 return error;
}
```

`mmap_region()` 调用了 `call_mmap(file, vma)`， `call_mmap` 根据文件系统的类型选择适配的 `mmap()` 函数，我们选择目前常用的ext4， `ext4_file_mmap()` 是ext4对应的mmap, 功能非常简单，更新了file的修改时间(file_accessed(flie))，将对应的operation赋给 `vma->vm_flags` ，后面的文件系统章节在学习这块。

通过分析mmap的源码我们发现在调用 `mmap()` 的时候仅仅申请一个 `vm_area_struct` 来建立文件与虚拟内存的映射，并没有建立虚拟内存与物理内存的映射。假如没有设置 `MAP_POPULATE` 标志位，Linux并不在调用 `mmap()` 时就为进程分配物理内存空间，直到下次真正访问地址空间时发现数据不存在于物理内存空间时，触发 `Page Fault` 即缺页中断，Linux才会将缺失的Page换入内存空间。其代码流程图如下所示



3. 应用场景

对于传统的linux系统文件操作是如何的呢？首选我们来看看工作流是如何的，其流程如下图所示

其特点为

- 使用页缓存机制，提高读写效率和保护磁盘
- 读文件时，先将文件从磁盘拷贝到缓存，由于页缓存区是在内核空间，不能被用户空间直接访问，所以需要将页缓存区数据再次拷贝到用户空间，有2次文件拷贝工作

下面来看看使用内存映射文件读/写的流程，其流程图如下图所示

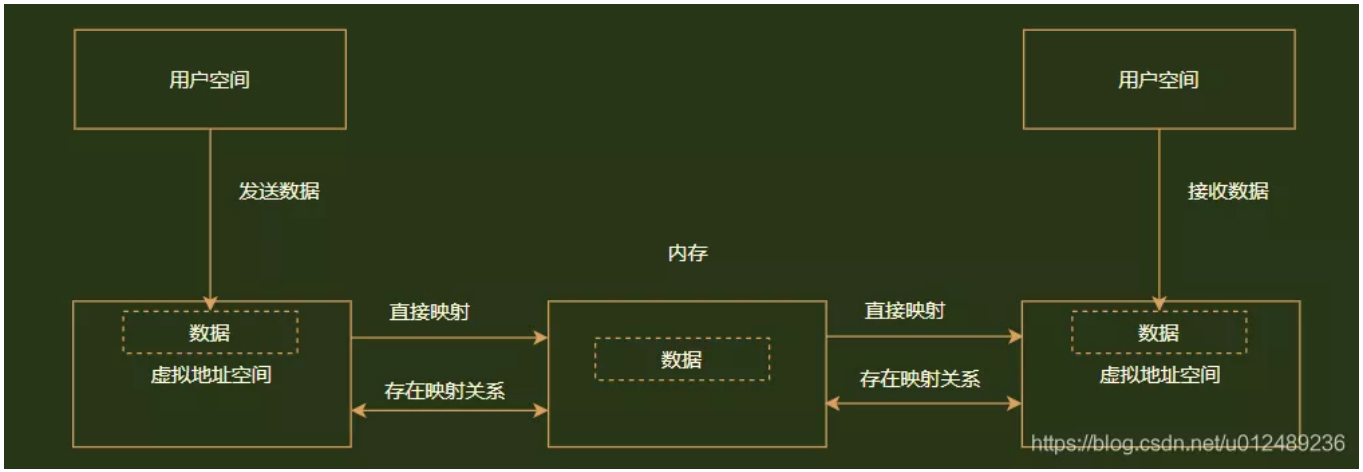
其特点为：

- 用户空间与内核空间的交互式通过映射的区域直接交互，用内存的读取代替I/O读写，文件读写效率高
- 数据拷贝次数少，对文件的读取操作跨过页缓存，减少了数据拷贝一次，效率提高
- 可实现高效的大规模数据传输

在Linux系统中，根据内存映射的本质和特点，其应用场景在于

- 1.实现内存共享，如跨进程通信
- 2.提高数据读/写效率：如读写操作

对于进程间的通信，其工作流程如下图所示



- 创建一块共享的接收区，实现地址映射关系
- 发送进程数据到自身的虚拟内存区域，数据拷贝1次

- 由于发送进程的虚拟地址空间与接收进程的虚拟内存地址存在映射关系，所以发送到的数据也存放到接收进程的虚拟内存中，即实现了跨进程间通信

4. 总结

内存映射的读写操作主要的过程如下：

- 创建虚拟映射区域，其在当前进程的虚拟地址空间中，寻找一段满足大小要求的虚拟地址，并且为此虚拟地址分配一个虚拟内存区域(vm_area_struct结构)，初始化该虚拟内存区域，插入到进程虚拟地址区域的链表和红黑树中
- 实现地址映射关系，建立页表，该过程在mmap函数中并未实现，此时只是创建了映射关系，并不将任何文件数据拷贝至主存中，真正的数据拷贝是通过进程发起读写操作时
- 进程访问该映射空间，实现文件内容到物理内存的数据拷贝，当进程读写访问该映射地址时，如果进程写操作改变了内容，并不会立即更新，而是一定时间后系统会自动会写脏数据到对应硬盘的地址空间

使用mmap来创建文件映射，由于只建立了进程地址空间VMA，并没有马上分配page cache和建立映射关系。那么就会导致一个问题，当创建一个很大的VMA，会频繁发生缺页中断。

内存映射机制mmap是POSIX标准的系统调用，有匿名映射和文件映射两种。

- 匿名映射使用进程的虚拟内存空间，它和malloc(3)类似，实际上有些malloc实现会使用mmap匿名映射分配内存，不过匿名映射不是POSIX标准中规定的。
- 文件映射有MAP_PRIVATE和MAP_SHARED两种。前者使用COW的方式，把文件映射到当前的进程空间，修改操作不会改动源文件。后者直接把文件映射到当前的进程空间，所有的修改会直接反应到文件的page cache，然后由内核自动同步到映射文件上。

相比于IO函数调用，基于文件的mmap的一大优点是把文件映射到进程的地址空间，避免了数据从用户缓冲区到内核page cache缓冲区的复制过程；当然还有一个优点就是不需要频繁的read/write系统调用。

原文： https://blog.csdn.net/u012489236/article/details/109709724