作业四、malloc()和mmap()

目录：

- malloc()
- mmap()
- 参考文献

1.malloc()

在剖析malloc()前，首先要看一下一个linux进程的线性地址空间的构成，如下图所示（图片来源：《深入理解计算机系统 第2版》）：

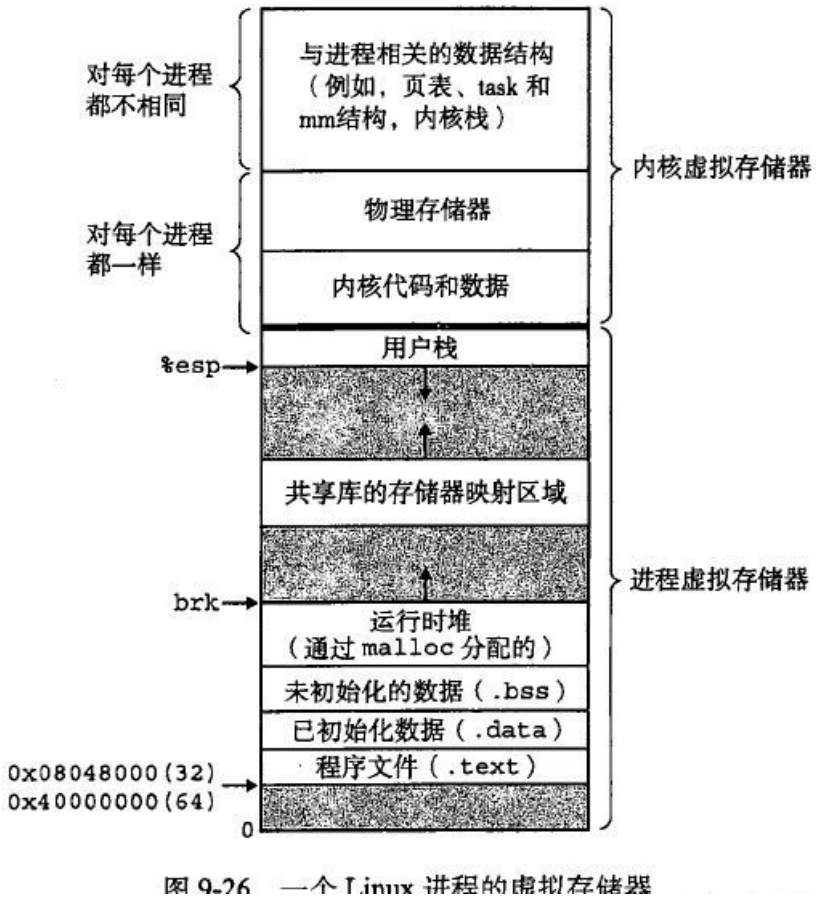


图 9-26 一个 Linux 进程的虚拟存储器

而malloc()正是用于在进程运行时动态分配内存。为了减少内存碎片引起的浪费，采用内存池的分配方法，首先分配较大的内存为堆，分为大小不同的内存块进行管理。malloc利用隐式链表，在分配时遍历链表，选择大小合适的内存分配。**内存分配时会调用brk或mmap系统，小于128k的用brk在堆中分配，大于128k的调用mmap系统在映射区分配**[1]。

进程控制块中的mm_struct 的 start_brk 和 brk 域分别保存了堆的开始和结束地址。当malloc()要分配的内存小于128k时，会调用brk()系统调用，brk()会将brk域往高地址推，从而在进程的虚拟地址空间中开辟出了一块新的空间。

当malloc()要分配的内存大于128k时，会调用mmap()系统调用，会在堆和栈之间的共享库存储器映射区域中分配一块虚拟内存。这里调用mmap()系统调用只是用来申请一块空间，并不会真的进行映射和共享，因此会将mmap()的输入参数flags中的MAP_ANONYMOUS位置为一，表示进行匿名映射。

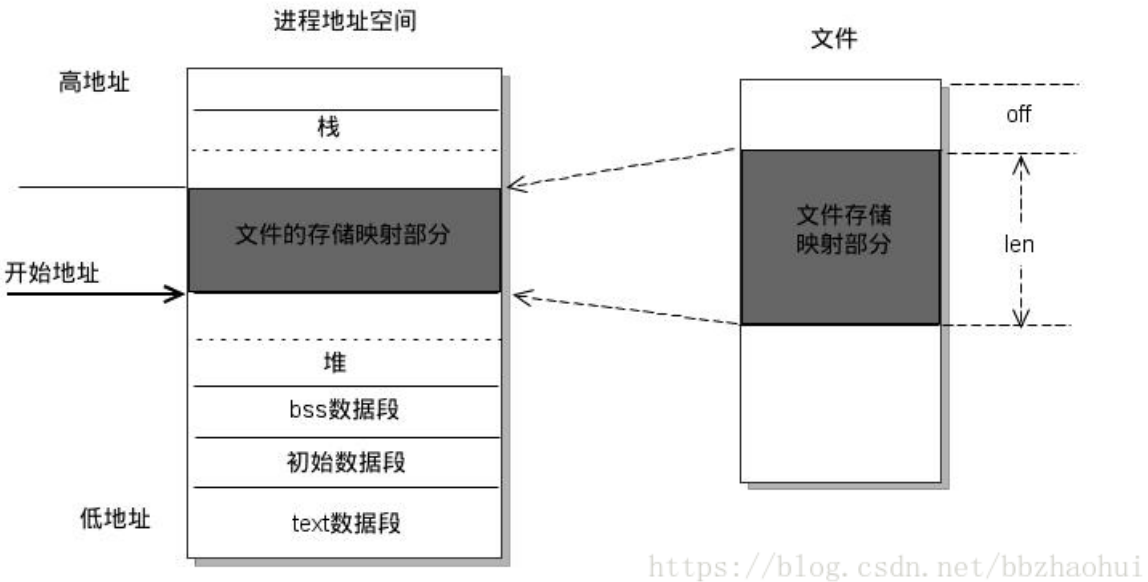
对于不同情况下，两者的调用对进程虚拟地址空间产生的效果，可以参考[2]中的举例和图解部分。

但无论是brk()或mmap()，开辟的空间都只是虚拟地址空间，并没有与内存物理地址建立真正的映射。只有当访问到该区域时，才会通过异常机制中的缺页中断，进行地址空间的映射。

那么为什么malloc要分以上两种情况呢？brk()分配的低地址内存需要在高地址内存释放后才能真正释放，这就意味着堆中的低地址空间虽然释放了，但只是显示为空闲状态，该部分映射的内存物理空间并没有还给操作系统，这就导致内存泄漏和碎片问题；但是由于这部分并没有还给操作系统，因此可重用，并且访问该部分很可能不会再产生缺页中断，这会降低时间开销。另一方面，在文件映射部分中，mmap()申请的每一部分都会以结构体的形式使用链表或者树形结构链接，管理这部分也是不小的开销。**结合brk()和mmap()两者的特点，小空间由brk()来申请，以减少碎片带来的影响；而大空间由mmap()来申请，可以减少管理相关数据结构带来的开销。可通过 mallopt(M_MMAP_THRESHOLD, <SIZE>) 来修改这个临界值(默认128k)。**

2.mmap()

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（图片来源：[3]）：



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，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。

mmap()系统调用原型如下：

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

- addr：如果不为NULL，内核会在此地址创建映射；否则，内核会选择一个合适的虚拟地址。大部分情况不指定虚拟地址，意义不大，而是让内核选择返回一个地址给用户空间使用。
- length：表示映射到进程地址空间的大小。

- prot：内存区域的读/写/执行属性。
- flags：内存映射的属性，共享、私有、匿名、文件等。
- fd：表示这是一个文件映射，fd是打开文件的句柄。如果是文件映射，需要指定fd；匿名映射就指定一个特殊的-1。
- offset：在文件映射时，表示相对文件头的偏移量；返回的地址是偏移量对应的虚拟地址。

其中flags可以有四种组合：

- 私有文件映射：多个进程使用同样的物理页面进行初始化，但是各个进程对内存文件的修改不会共享，也不会反映到物理文件中。比如对linux .so动态库文件就采用这种方式映射到各个进程虚拟地址空间中。
- **私有匿名映射：mmap会创建一个新的映射，各个进程不共享，主要用于分配内存(malloc分配大内存会调用mmap)。**
- 共享文件映射：多个进程通过虚拟内存技术共享同样物理内存，对内存文件的修改会反应到实际物理内存中，也是进程间通信的一种。
- 共享匿名映射：这种机制在进行fork时不会采用写时复制，父子进程完全共享同样的物理内存页，也就是父子进程通信。

sys_mmap()是mmap()的入口函数，经过多层调用，最后do_mmap()是整个mmap()的具体操作函数，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。具体步骤如下：

1. 检查参数，并根据传入的映射类型设置vma的flags
2. 进程查找其虚拟地址空间，找到一块空闲的满足要求的虚拟地址空间
3. 根据找到的虚拟地址空间初始化vma
4. 设置vma->vm_file
5. 根据文件系统类型，将vma->vm_ops设为对应的file_operations
6. 将vma插入mm的链表中

do_mmap()函数代码[5]：

```
    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 list_head *uf)
{
        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;
    /* force arch specific MAP_FIXED handling in get_unmapped_area */
```

```
    if (flags & MAP_FIXED_NOREPLACE)
            flags |= MAP_FIXED;

    /* 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的页对齐后的地址 */
    if (!(flags & MAP_FIXED))
            addr = round_hint_to_min(addr);

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

    /* offset overflow? */
    if ((pgoff + (len >> PAGE_SHIFT)) < pgoff)
            return -EOVERFLOW;

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

    /* Obtain the address to map to. we verify (or select) it and ensure
     * that it represents a valid section of the address space.
     */
    /* 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);
/* 检查addr是否有效 */
    if (offset_in_page(addr))
            return addr;

/*  假如flags设置MAP_FIXED_NOREPLACE，需要对进程的地址空间进行addr的检查. 如果搜索发现存
在重合的vma，返回-EEXIST。
    这是MAP_FIXED_NOREPLACE标志所要求的
*/
    if (flags & MAP_FIXED_NOREPLACE) {
            struct vm_area_struct *vma = find_vma(mm, addr);

            if (vma && vma->vm_start < addr + len)
                    return -EEXIST;
    }

    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 */
        unsigned long flags_mask;

        if (!file_mmap_ok(file, inode, pgoff, len))
                return -EOVERFLOW;

        flags_mask = LEGACY_MAP_MASK | file->f_op->mmap_supported_flags;

        /*
          ...
          根据标志指定的map种类，把为文件设置的访问权考虑进去。
      如果所请求的内存映射是共享可写的，就要检查要映射的文件是为写入而打开的，而不
      是以追加模式打开的，还要检查文件上没有上强制锁。
      对于任何种类的内存映射，都要检查文件是否为读操作而打开的。
      ...
    */
    } 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;
    }
```

```
    addr = mmap_region(file, addr, len, vm_flags, pgoff, uf);
    if (!IS_ERR_VALUE(addr) &&
        ((vm_flags & VM_LOCKED) ||
         (flags & (MAP_POPULATE | MAP_NONBLOCK)) == MAP_POPULATE))
            *populate = len;
    return addr;
```

一般读写文件需要open、read、write，需要先将磁盘文件读取到内核cache缓冲区，然后再拷贝到用户空间内存区，涉及两次读写操作。mmap通过将磁盘文件映射到用户空间，当进程读文件时，发生缺页中断，给虚拟内存分配对应的物理内存，在通过磁盘调页操作将磁盘数据读到物理内存上，实现了用户空间数据的读取，整个过程只有一次内存拷贝。

另外，mmap()还可用于进程间大数据量通信。两个进程映射同一个文件，在两个进程中，同一个文件区域映射的虚拟地址空间不同。一个进程操作文件时，先通过缺页获取物理内存，进而通过磁盘文件调页操作将文件数据读入内存。另一个进程访问文件的时候，发现没有物理页面映射到虚拟内存，通过fs的缺页处理查找cache区是否有读入磁盘文件，有的话建立映射关系，这样两个进程通过共享内存就可以进行通信。

3.参考文献

[1]malloc的原理？brk系统调用和mmap系统调用的作用分别是什么？https://www.nowcoder.com/questionTerminal/10a65bc291bc4ebc93458f041215cea0?orderByHotValue=1&page=1&onlyReference=false

[2]Linux内存分配小结--malloc、brk、mmaphttps://blog.csdn.net/gfgdsg/article/details/42709943

[3]linux库函数mmap()原理https://blog.csdn.net/bbzhaohui/article/details/81665370

[4] Linux内存管理 (9)mmap(补充https://www.cnblogs.com/arnoldlu/p/9367253.html

[5]mmap源码分析https://blog.csdn.net/lggbxf/article/details/94012088