实验四

Lazy Page Allocation

目的：在程序之中，往往出现调用sbrk()获取空间，但实际上使用的却往往不多。这样会造成分配大量的的页表，浪费空间也浪费时间。因此，本项目将致力于修改sbrk的页表分配行为，使得在分配空间的时候，不会真正的分配空间，而只是记录分配标志。当程序首次尝试使用该区域的时候，CPU会产生页表缺失fault，这样kernel收到fault之后会分配相应的物理空间，并清空该区域，而且设置页表。

任务一：打印当前的页表。

任务二：sbrk在分配空间的时候，不会再分配页表。

任务三：修改trap.c，使得kernel在收到page fault的时候，会分配新的页表给对应的地址。

**具体操作：**

任务一：打印页表。

首先，我们需要了解页表的数据结构。

递归的调用打印程序，使得打印完整的页表。

Xv6使用三层页表，每页打印即可，采用迭代的算法。具体见代码

任务二：sbrk的分配缩减。

调用sbrk的时候，仅仅增加process size 也就是代码的myproc()->sz，函数返回一个旧的size。注意，它将不会分配存储空间，所以不要调用growproc()，尽管你还是需要增加进程的空间。此步改动将会使得整个程序在运行echo hi的时候直接陷入panic:uvmunmap。

任务三：懒人分配法

在trap里面更改相应page fault的解决办法，每当收到对应的fault就分配新的page，然后返回至用户态，让进程继续运行。所有的代码都恰好在printf之前，也就是说之前的代码都要执行，然后才是自己修改的代码。

每个进程都有自己的页表，growproc实际上是增加自己的sz，也就是va，从0线性增长，但是每次增长的时候都要在页表之中加入相应的page，进行相应的虚实映射。Kalloc的k是kernel的意思，uvmunmap是 usermode virtualmemory unmap的意思。既然是线性映射，那么实际上自己的va就反应了实际的sz，如果加上PGSIZE大于分配的sz，就返回error，与更改之前相同。

原来的growproc，就是用uvmalloc直接将自己的vm sz从sz增长到sz+n（实际上会多分配，因为要按照PGSIZE取整数）

具体来说，收到page fault之后，申请一个页的内存，然后做虚实地址的映射。注意vm要向下取整（PAGE），要不然会分配出现重新分配，页表项冗余的情况。

在和之前uvmalloc一样处理page fault分配存储空间失败的时候，一定要注意，要将之前分配的内存空间释放。

问题1：sbrk增加好做，减少呢？减少的是未分配的内存，不用管就行，但是如果减少的是已经分配的内存，怎么回收？现在的想法是，调用growproc。如果问题2解决的，growproc最终会调用uvmunmap，也就不会出现问题了。

问题2：还有一个问题，就算sbrk的n<0不管，最后进程停止的时候，也要uvmunmap，这个时候如果出现unmap的情况会报错，因此需要解决相应问题的措施。既然是没分配，那就没必要unmap，也就是说遇见这种情况直接跳过，注意，不能直接退出，要跳过而且继续执行unmap，因为未分配的块可能是不连续的，不能代表后面不存在没有释放的块。但是如果出现了找不到对应的pte，那就说明已经全部释放完毕了，但是还不可以退出！因为这个表（512项pte）已经结束了，但不代表剩下的表被释放了。注意代码中，不要使用\*pte=0，因为此时pte是空指针会进kerneltrap。

问题3：uvmcopy将复制不存在的page，这同样是因为它以sz为搜寻大小，进行批量复制。那么我们也需要同样的，由于源页表不存在的页表项对应的数据没有访问过（更不用说修改），那么也就没有必要进行复制，忽略不存在的表。

exec解释：

本实验中的vm对应操作不仅仅是简单的数据存储空间，在分配关键空间也非常重要，这是由kernel进行的操作，而如果只解决用户态的操作，势必会造成问题。而kernel调用vm的操作，与exec（进程相关）紧密相连，因此此部分进一步分析exec的操作。

具体操作：

* 新建页表：申请一级页表
* 加载文件：按照elf文件，不停的uvmalloc和loadseg以建立text与data区域
* 建立用户栈：申请两个页高地址作为用户栈，低地址作为guard（标志栈结束地址，此处页表PTE\_U为无效）
* 准备main数调用：将参数放入用户栈，更换页表。

Kernel缺页：usertrap无法解决。系统调用write时，直接发生在内核态，会使用copyin。对应会有walkaddr，直接绕过了usertrap，如果不在walkaddr中解决lazy的分配问题，就没有办法解决内核的内存分配问题。

Guard Page：栈顶之后，会有一个guard page，对应的页表要设成用户态不可访问。但是实际上当时分配了，创建进程的时候会手动就行两个页表的分配，其中一个就是guard page。我们在处理page fault的时候，因为没有考虑由于其它情况引起的类似lazy造成trap的情形（其实就是栈溢出），导致处理栈溢出的时候也允许程序继续下去，这样会出现remap，甚至kernel trap的问题。因此，在对应trap的时候，还要查看对应虚拟地址的页表项是否存在，存在的时候表示是真的出错了，而不是lazy。

