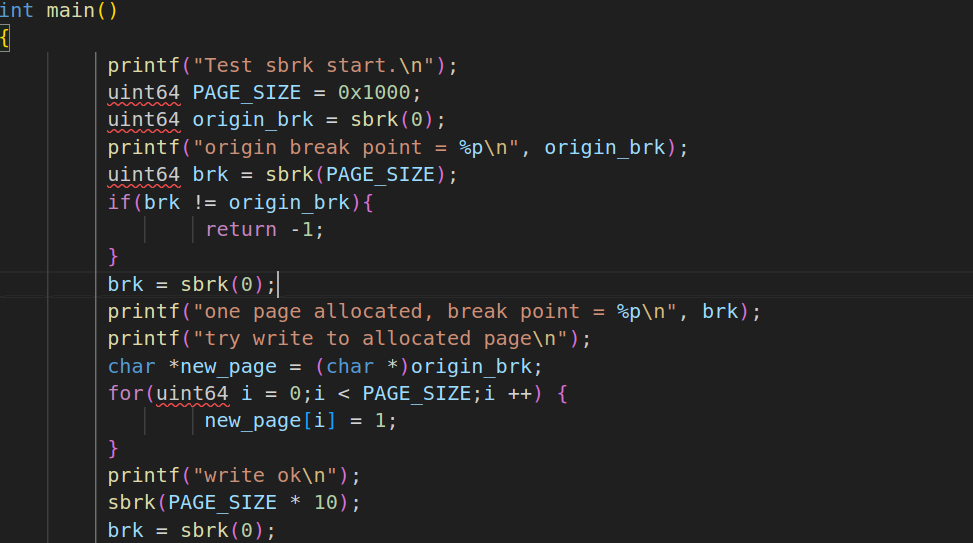
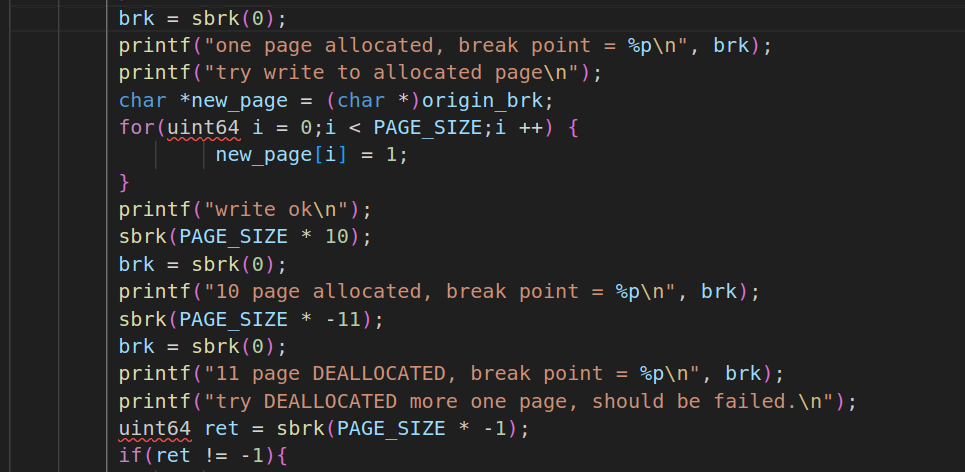
LAB4：

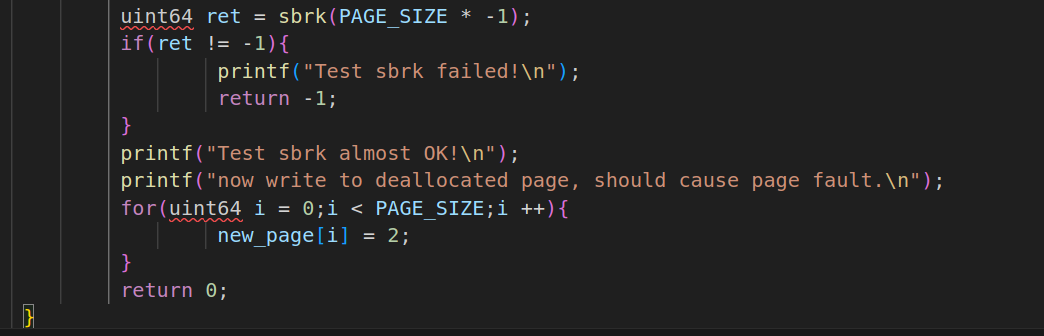
编程题：

第一题：\*\* 使用sbrk，mmap,munmap,mprotect内存相关系统调用的linux应用程序。

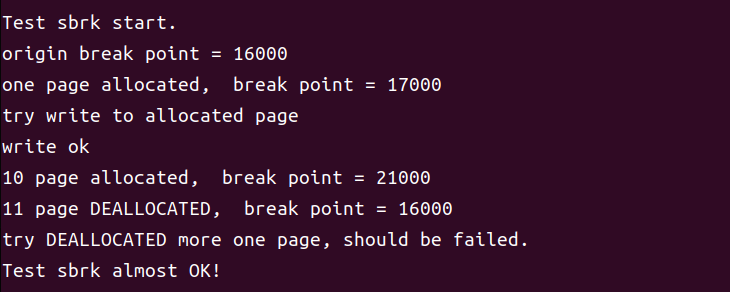
我们可以设计程序的代码如下：







把该程序放到user/src目录下，运行程序，我们能看到结果：



表明该程序使用了sbrk系统调用。



Mmap.c

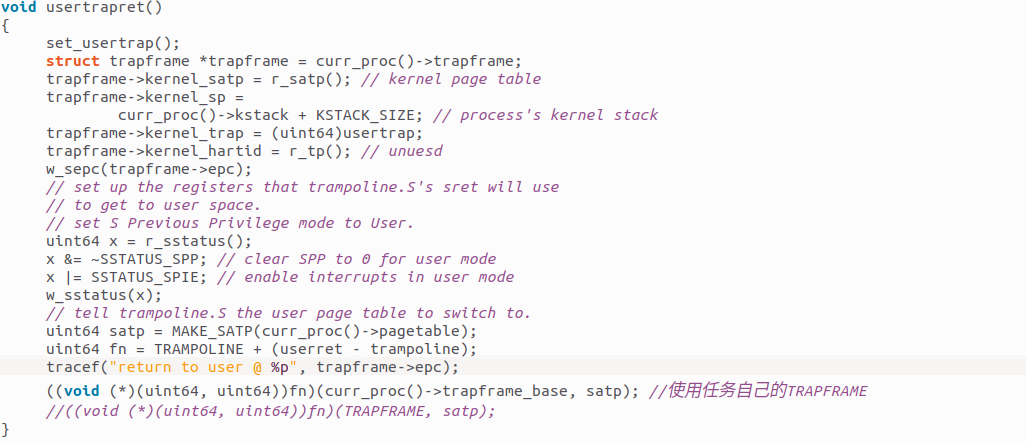


Mprotect.c

编程题二：

修改本章操作系统内核，实现任务和操作系统内核共用同一张页表的单页表机制。要实现任务和操作系统内核通用一张页表，需要了解清楚内核地址空间和任务地址空间的布局，然后为每个任务在内核地址空间中单独分配一定的地址空间。

在描述任务的struct proc中添加新的成员“kpgtbl”、“trapframe\_base”，前者用户保存内核页表，后者用于保存任务的TRAPFRAME虚地址。并增加获取内核页表的函数“get\_kernel\_pagetable()”。



Trap.c



修改vm.c。



修改proc.h。



修改loader.c。



修改config.rs。

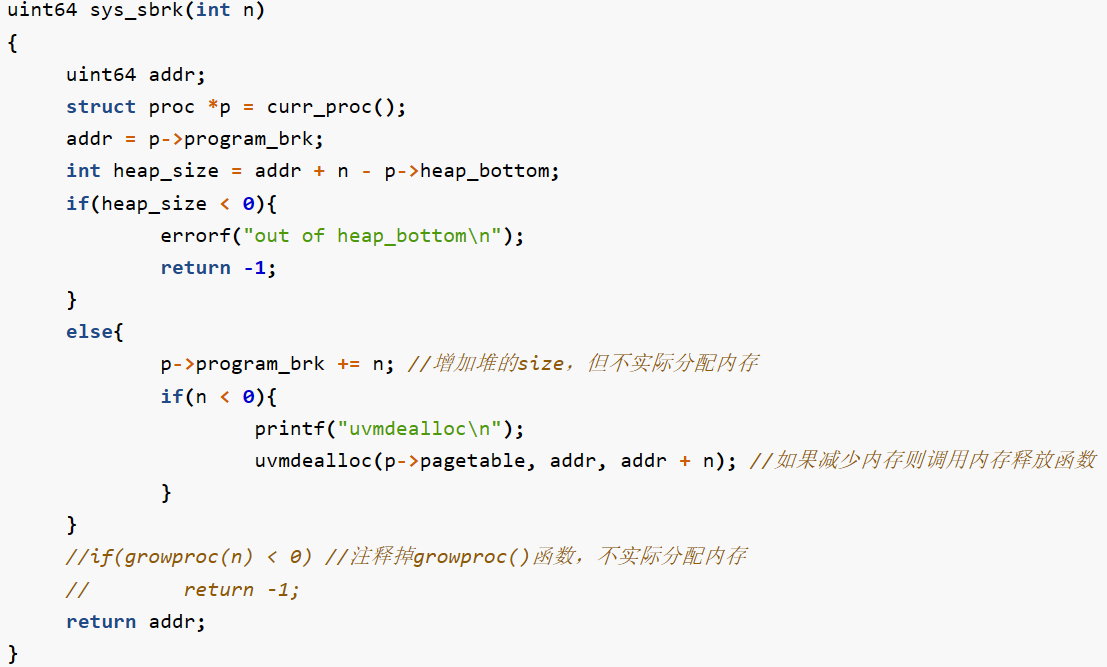
编程题三：

扩展内核，支持基于缺页异常机制，具有Lazy 策略的按需分页机制。

在页面懒分配（Lazy allocation of pages）技术中，内存分配并不会立即发生，而是在需要使用内存时才分配，这样可以节省系统的资源并提高程序的性能。

实现页面懒分配的思路是：当调用sbrk时不分配实际的页面，而是仅仅增大堆的大小，当实际访问页面时，就会触发缺页异常，此时再申请一个页面并映射到页表中，这时再次执行触发缺页异常的代码就可以正常读写内存了。

注释掉growproc()函数，增加堆的size。



*syscall.c*



修改loader.c。



修改trap.c。

问答题：

1、在使用高级语言编写用户程序的时候，手动用嵌入汇编的方法随机访问一个不在当前程序逻辑地址范围内的地址，比如向该地址读/写数据。该用户程序执行的时候可能会生什么？

触发越界访问异常(内中断) ---> trap进内核处理异常--->把程序标记为exit，切换为next进程之后os就不会管被标记为exit程序的事了，就在那让他停着。

2、用户程序在运行的过程中，看到的地址是逻辑地址还是物理地址？从用户程序访问某一个地址，到实际内存中的对应单元被读/写，会经过什么样的过程，这个过程中操作系统有什么作用？（站在学过计算机组成原理的角度）

逻辑地址

这个过程需要经过页表的转换，操作系统会负责建立页表映射。实际程序执行时的具体VA到 PA的转换是在CPU的MMU之中进行的。

操作系统的作用: 完成地址虚拟地址到物理地址的映射

3、覆盖、交换和虚拟存储有何异同，虚拟存储的优势和挑战体现在什么地方？

相同点: 都是为了提高内存空间的利用率

不同点: 覆盖需要程序员自己写程序来确定覆盖的顺序与位置，并且是程序内的覆盖

交换是由os完成的，是程序间的交换，换出旧进程，运行新进程，不过在换出的时候依赖I/O速度

虚拟存储是由os完成的，核心思想是按需分配

优势: 允许程序员编写不受物理内存大小限制的程序，程序编程简单，隔离性与安全性强，内存利用率高

挑战: 页面置换依赖于I/O速度、选择恰当的页面置换算法

4、什么是局部性原理？为何很多程序具有局部性？局部性原理总是正确的吗？为何局部性原理为虚拟存储提供了性能的理论保证？

(1)、什么是局部性原理？

局部性分为时间局部性与空间局部性

1. 时间局部性

如果一个数据项被访问，那么它在不久的将来可能再次被访问。这是因为循环和频繁的数据访问模式导致的。

1. 空间局部性

如果一个数据项被访问，那么存储在其附近地址的数据项很快也可能被访问。这通常是由于数据结构的顺序存储（如数组）和编程中的顺序执行造成的。

(2)、为何很多程序具有局部性

1. 这是由编程语言决定的

编程语言里面包含了大量的循环语句

1. 常用的连续存储的数据结构

比如数组之类的

(3)、局部性原理一定是正确的吗？

在大多数情况下是正确的，但也有可能不生效，比如大量的goto语句跳转可能会导致局部性原理失效

(4)、为何局部性原理为虚拟存储提供了性能的理论保证？

当程序的地址被访问后，其或者其旁边的地址很有可能会被再次访问，而这些地址访问通常频繁的落在一页或几页上，这就导致OS不需要经常的进行页面置换，以此提高了效率

5、一条load指令，最多导致多少次页访问异常？尝试考虑较多情况。

考虑多级页表的情况。首先指令和数据读取都可能缺页。因此指令会有3次访存，之后的数据读取除了页 表页缺失的3次访存外，最后一次还可以出现地址不对齐的异常，因此可以有7次异常。若考更加极端的 情况，也就是页表的每一级都是不对齐的地址并且处在两页的交界处（straddle），此时一次访存会触 发2次读取页面，如果这两页都缺页的话，会有更多的异常次数。

6、如果在页访问异常中断服务例程执行时，再次出现页访问异常，这时计算机系统（软件或硬件）会如何处理？这种情况可能出现吗？

我们实验的os在此时不支持内核的异常中断，因此此时会直接panic掉，并且这种情况在我们的os中这种情况不可能出现。像linux系统，也不会出现嵌套的page fault。

7、全局和局部置换算法有何不同？分别有哪些算法？

全局置换算法会在所有任务中进行页面置换，而局部置换算法只会在该任务内部进行页面置换

全局页面置换算法: 工作集置换算法，缺页率置换算法。

局部页面置换算法: 最优置换算法、FIFO置换算法、LRU置换算法、Clock置换算法。

8、简单描述OPT、FIFO、LRU、Clock、LFU的工作过程和特点 (不用写太多字，简明扼要即可)

OPT：选择一个应用程序在随后最长时间内不会被访问的虚拟页进行换出。性能最佳但无法实现。

FIFO：由操作系统维护一个所有当前在内存中的虚拟页的链表，从交换区最新换入的虚拟页放在表尾，最久换入的虚拟页放在表头。当发生缺页中断时，淘汰/换出表头的虚拟页并把从交换区新换入的虚拟页加到表尾。实现简单，对页访问的局部性感知不够。

LRU：替换的是最近最少使用的虚拟页。实现相对复杂，但考虑了访存的局部性，效果接近最优置换算法。

Clock：将所有有效页放在一个环形循环列表中，指针根据页表项的使用位（0或1）寻找被替换的页面。考虑历史访问，性能略差于但接近LRU。

LFU：当发生缺页中断时，替换访问次数最少的页面。只考虑访问频率，不考虑程序动态运行。

10.Clock算法仅仅能够记录近期是否访问过这一信息，对于访问的频

度几乎没有记录，如何改进这一点？

如果想要改进这一点，可以将Clock算法和计数器结合使用。具体做法是为每个页面设置一个计数器，记录页面在一段时间内的访问次数，然后在置换页面时，既考虑页面最近的访问时间，也考虑其访问频度。当待缓存对象在缓存中时，把其计数器的值加1。同时，指针指向该对象的下一个对象。若不在缓存中时，检查指针指向对象的计数器。如果是0，则用待缓存对象替换该对象；否则，把计数器的值减1，指针指向下一个对象。如此直到淘汰一个对象为止。由于计数器的值允许大于1，所以指针可能循环多遍才淘汰一个对象。

11.哪些算法有belady现象？思考belady现象的成因，尝试给出说明

OPT和LRU等为何没有belady现象。

FIFO算法、Clock算法。

页面调度算法可分为堆栈式和非堆栈式，LRU、LFU、OPT均为堆栈类算法，FIFO。Clock为非堆栈类算法，只有非堆栈类才会出现Belady现象。

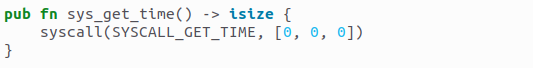
14.请问一个任务处理 10G 连续的内存页面，需要操作的页表实际大致占用多少内存(给出数量级即可)？

大致占用10G/512=20M 内存。

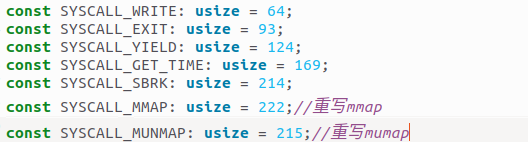
实验题：

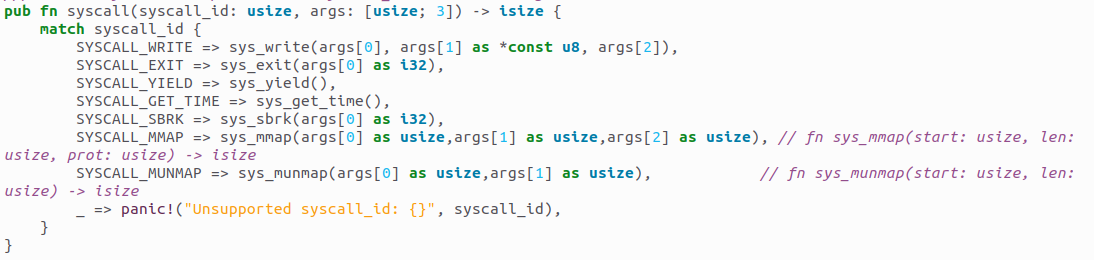
实践作业：

重写 sys\_get\_time：

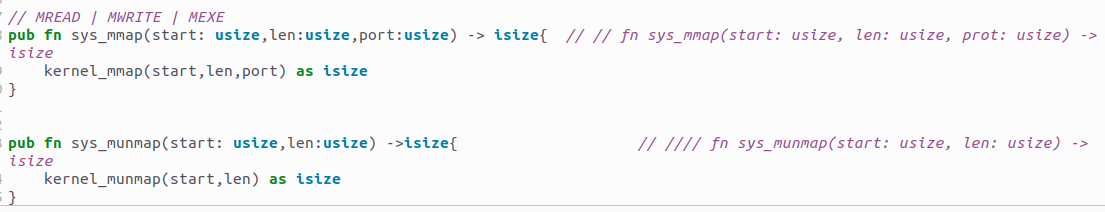


mmap 和 munmap 匿名映射：

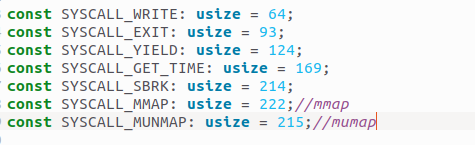


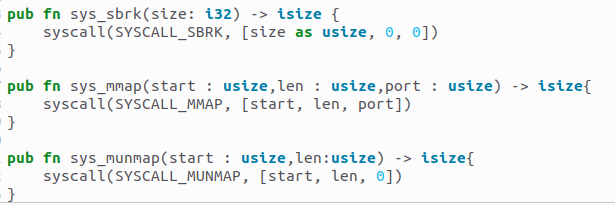


修改mod.rs，增加mmap和munmap。

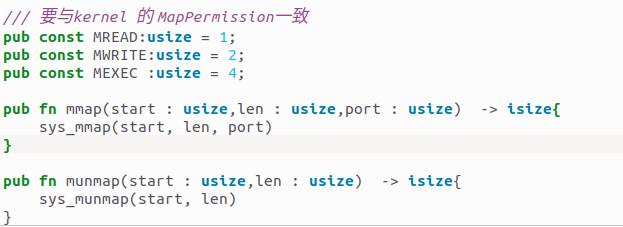


修改process。

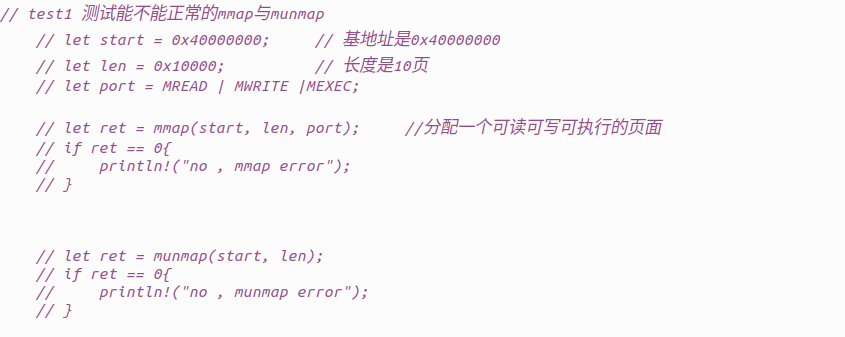




修改syscall.rs。



修改lib.rs。



修改测试文件。