# 实验要求

mmap和munmap system call允许UNIX程序详细使用它们的地址空间。它们可以被用来在进程间共享内存，映射文件到进程地址空间，作为用户级page fault方案的一部分，例如课堂上讨论的GC算法。在本实验中，你将添加mmap和munmap到xv6，聚焦于内存映射文件。

获取本实验xv6源码，并检出mmap分支：

git fetch、git checkout mmap、make clean

操作手册页（执行man 2 mmap）展示mmap的声明：

void \*mmap(void \*addr, size\_t length, int prot, int flags, int fd, off\_t offset);

mmap可以用多种方式调用，但本实验仅需要特性（与内存映射文件相关）的子集。你可以假设addr将总是为0，意味着kernel应该决定映射文件的虚拟地址。mmap返回该地址，或失败时返回0xffffffffffffffff。length是map的字节数；它可以与文件长度不同。prot表明内存是否应该被映射为可读、可写、可执行；你可以假设prot是PROT\_READ、PROT\_WRITE或两者都有。flags将要么是MAP\_SHARED，意味对映射内存的修改应该被写回文件，或为MAP\_PRIVATE，意味着它们会写回到文件。你无需实现flags中的任何其他位。fd是打开的、要映射文件的文件描述符。你可以假设offset为0（文件中要映射的起始点）。

映射相同MAP\_SHARED文件的进程没有共享物理内存是可以的。

munmap(addr, length)应该移除已知地址范围的mmap映射。如果进程已经更改了内存，且映射为MAP\_SHARED，更改应该首先被写到文件。munmap调用可能只覆盖mmap-ed区域的一部分，但你可以假设：要么unmap起始，要么unmap结束、要么整个区域（而不是在region中打孔）。

你可以实现足够的map和munmap功能来通过mmaptest测试程序。如果是mmaptest不用的mmap特性，你无需实现。

当你完成时，你可以看到如下输出：

$ mmaptest

mmap\_test starting

test mmap f

test mmap f: OK

test mmap private

test mmap private: OK

test mmap read-only

test mmap read-only: OK

test mmap read/write

test mmap read/write: OK

test mmap dirty

test mmap dirty: OK

test not-mapped unmap

test not-mapped unmap: OK

test mmap two files

test mmap two files: OK

mmap\_test: ALL OK

fork\_test starting

fork\_test OK

mmaptest: all tests succeeded

$ usertests

usertests starting

...

ALL TESTS PASSED

# 一些提示

添加\_mmaptest到UPROGS，添加mmap、munmap system calls，为了让user/mmaptest.c可以编译。现在，仅从mmap和munmap返回错误。我们在kernel/fcntl.h定义PROT\_READ等等。执行mmaptest，首次mmap调用失败。

懒填充页表，对page faults做出响应。mmap应该不分配物理内存或读取文件。而是，在usertrap page fault处理代码中，正如lazy page allocation实验中那样。lazy的原因是：保证mmap大文件较快，以及让mmap一个比物理内存大的文件成为可能。

跟踪每个进程mmap已经map的东西。定义一个结构对应VMA（virtual memory area），第15节课描述过，记录地址、长度、权限、文件等待。用于mmap创建的虚拟内存。因为xv6内核在内核中没有内存分配器，声明一个固定尺寸的VMA数组，按需从数组分配是可行的。16应该就足够了。

实现mmap：找一个进程地址空间中未使用区域来映射文件，添加一个VMA到进程映射区域表。VMA应该包含一个映射文件的struct file的指针；mmap应该提升文件索引数量，以便于当文件关闭时结构不会消失（提示：看filedup）。执行mmaptest：第一个mmap应该成功，但首次访问mmap-ed内存将导致page fault并杀掉mmaptest。

添加代码让mmap-ed区域的page-fault分配一页物理内存，读取相关文件的4096字节到该page，并映射它到用户地址空间。用readi读取文件，可以接收一个偏移参数（在文件中从此处读取），但你不得不lock/unlock传到readi中的inode。不要忘记设置page正确的权限。执行mmaptest；它应该可以获取第一个munmap。

实现munmap：找VMA对应地址范围，取消明确页的映射（提示：使用uvmunmap）。如果munmap移除之前mmap的所有页，它应该减少对应file的引用计数。如果一个未映射页更改了，且文件映射为MAP\_SHARED，把页写回文件，看filewrite找灵感。

你的实现应该只写回程序真正修改过、且为MAP\_SHARED的页。RISC-V PTE的dirty位表明这个页是否已经写过。然而mmaptest不会检查：非dirty页不会写回；因此你可以不看dirty标志位把数据写回去。

更改exit来unmap进程的已映射区域，就像调用了munmap一样。执行mmaptest；mmap\_test应该通过，但fork\_test可能不通过。

更改fork确保：child有和parent一样的映射区域。不要忘记提升VMA struct file的引用数量。在child的page fault handler，可以分配一个新的物理页，而不是共享parent的页。后者是更好的，但它需要更多实现工作。执行mmaptest；它应该通过mmap\_test和fork\_test。