Lab: mmap (hard)

The mmap and munmap system calls allow UNIX programs to exert detailed control over their address spaces. They can be used to share memory among processes, to map files into process address spaces, and as part of user-level page fault schemes such as the garbage-collection algorithms discussed in lecture. In this lab you'll add mmap and munmap to xv6, focusing on memory-mapped files.

Fetch the xv6 source for the lab and check out the mmap branch:

```
$ git fetch
$ git checkout mmap
$ make clean
```

The manual page (run `man 2 mmap`) shows this declaration for mmap:

```
void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
```

`mmap` can be called in many ways, but this lab requires only a subset of its features relevant to memory-mapping a file. You can assume that `addr` will always be zero, meaning that the kernel should decide the virtual address at which to map the file. `mmap` returns that address, or 0xffffffffffffffff if it fails. `length` is the number of bytes to map; it might not be the same as the file's length. `prot` indicates whether the memory should be mapped readable, writeable, and/or executable; you can assume that `prot` is `PROT_READ` or `PROT_WRITE` or both. `flags` will be either `MAP_SHARED`, meaning that modifications to the mapped memory should be written back to the file, or `MAP_PRIVATE`, meaning that they should not. You don't have to implement any other bits in `flags`. `fd` is the open file descriptor of the file to map. You can assume `offset` is zero (it's the starting point in the file at which to map).

It's OK if processes that map the same `MAP_SHARED` file do **not** share physical pages.

`munmap(addr, length)` should remove mmap mappings in the indicated address range. If the process has modified the memory and has it mapped `MAP_SHARED`, the modifications should first be written to the file. An `munmap` call might cover only a portion of an mmap-ed region, but you can assume that it will either unmap at the start, or at the end, or the whole region (but not punch a hole in the middle of a region).

> You should implement enough `mmap` and `munmap` functionality to make the `mmaptest` test program work. If `mmaptest` doesn't use a `mmap` feature, you don't need to implement that feature.

When you're done, you should see this output:

```
$ 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 -q
usertests starting
...
ALL TESTS PASSED
$
```

Here are some hints:

- Start by adding `_mmaptest` to `UPROGS`, and `mmap` and `munmap` system calls, in order to get `user/mmaptest.c` to compile. For now, just return errors from `mmap` and `munmap`. We defined `PROT_READ` etc for you in `kernel/fcntl.h`. Run `mmaptest`, which will fail at the first mmap call.
- Fill in the page table lazily, in response to page faults. That is, `mmap` should not allocate physical memory or read the file. Instead, do that in page fault handling code in (or called by) `usertrap`, as in the lazy page allocation lab. The reason to be lazy is to ensure that `mmap` of a large file is fast, and that `mmap` of a file larger than physical memory is possible.
- Keep track of what `mmap` has mapped for each process. Define a structure corresponding to the VMA (virtual memory area) described in Lecture 15, recording the address, length, permissions, file, etc. for a virtual memory range created by `mmap`. Since the xv6 kernel doesn't have a memory allocator in the kernel, it's OK to declare a fixed-size array of VMAs and allocate from that array as needed. A size of 16 should be sufficient.
- Implement `mmap`: find an unused region in the process's address space in which to map the file, and add a VMA to the process's table of mapped regions. The VMA should contain a pointer to a `struct file` for the file being mapped; `mmap` should increase the file's reference count so that the structure doesn't disappear when the file is closed (hint: see `filedup`). Run `mmaptest`: the first `mmap` should succeed, but the first access to the mmap-ed memory will cause a page fault and kill `mmaptest`.
- Add code to cause a page-fault in a mmap-ed region to allocate a page of physical memory, read 4096 bytes of the relevant file into that page, and map it into the user address space. Read the file with `readi`, which takes an offset argument at which to read in the file (but you will have to lock/unlock the inode passed to `readi`). Don't forget to set the permissions correctly on the page. Run `mmaptest`; it should get to the first `munmap`.
- Implement `munmap`: find the VMA for the address range and unmap the specified pages (hint: use `uvmunmap`). If `munmap` removes all pages of a previous `mmap`, it should decrement the reference count of the corresponding `struct file`. If an unmapped page has been modified and the file is mapped `MAP_SHARED`, write the page back to the file. Look at `filewrite` for inspiration.
- Ideally your implementation would only write back `MAP_SHARED` pages that the program actually modified. The dirty bit (`D`) in the RISC-V PTE indicates whether a page has been written. However, `mmaptest` does not check that non-dirty pages are not written back; thus you can get away with writing pages back without looking at `D` bits.
- Modify `exit` to unmap the process's mapped regions as if `munmap` had been called. Run `mmaptest`; `mmap_test` should pass, but probably not `fork_test`.
- Modify `fork` to ensure that the child has the same mapped regions as the parent. Don't forget to

increment the reference count for a VMA's `struct file`. In the page fault handler of the child, it is OK to allocate a new physical page instead of sharing a page with the parent. The latter would be cooler, but it would require more implementation work. Run `mmaptest`; it should pass both `mmap_test` and `fork_test`.

Run `usertests -q` to make sure everything still works.

Submit the lab

This completes the lab. Make sure you pass all of the make grade tests. If this lab had questions, don't forget to write up your answers to the questions in answers-*lab-name*.txt. Commit your changes (including adding answers-*lab-name*.txt) and type make handin in the lab directory to hand in your lab.

Time spent

Create a new file, `time.txt`, and put in it a single integer, the number of hours you spent on the lab. Don't forget to `git add` and `git commit` the file.

Submit

You will turn in your assignments using the submission website. You need to request once an API key from the submission website before you can turn in any assignments or labs.

After committing your final changes to the lab, type **`make handin`** to submit your lab.

```
$ git commit -am "ready to submit my lab"
[util c2e3c8b] ready to submit my lab
 2 files changed, 18 insertions(+), 2 deletions(-)

$ make handin
tar: Removing leading `/' from member names
Get an API key for yourself by visiting https://6828.scripts.mit.edu/2022/handin.py/
Please enter your API key: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 79258  100   239  100 79019    853   275k --:--:-- --:--:-- --:--:--  276k
$
```

`make handin` will store your API key in *myapi.key*. If you need to change your API key, just remove this file and let **`make handin`** generate it again (*myapi.key* must not include newline characters).

If you run **`make handin`** and you have either uncomitted changes or untracked files, you will see output similar to the following:

```
 M hello.c
?? bar.c
?? foo.pyc
Untracked files will not be handed in.  Continue? [y/N]
```

Inspect the above lines and make sure all files that your lab solution needs are tracked i.e. not listed in a line that begins with **`??`**. You can cause `git` to track a new file that you create using `git add filename`.

If **`make handin`** does not work properly, try fixing the problem with the curl or Git commands. Or you can run `make tarball`. This will make a tar file for you, which you can then upload via our web interface.

- Please run `make grade` to ensure that your code passes all of the tests
- Commit any modified source code before running `make handin`
- You can inspect the status of your submission and download the submitted code at https://6828.scripts.mit.edu/2022/handin.py/

Optional challenges

- If two processes have the same file mmap-ed (as in `fork_test`), share their physical pages. You will need reference counts on physical pages.
- Your solution probably allocates a new physical page for each page read from the mmap-ed file, even though the data is also in kernel memory in the buffer cache. Modify your implementation to use that physical memory, instead of allocating a new page. This requires that file blocks be the same size as pages (set `BSIZE` to 4096). You will need to pin mmap-ed blocks into the buffer cache. You will need worry about reference counts.
- Remove redundancy between your implementation for lazy allocation and your implementation of mmap-ed files. (Hint: create a VMA for the lazy allocation area.)
- Modify `exec` to use a VMA for different sections of the binary so that you get on-demand-paged executables. This will make starting programs faster, because `exec` will not have to read any data from the file system.
- Implement page-out and page-in: have the kernel move some parts of processes to disk when physical memory is low. Then, page in the paged-out memory when the process references it.