Project #5 - xv6 Pagefault Handler & File-backed mmap

1. Part 5a: Pagefault Handler

(part of this assignment was inspired by https://pdos.csail.mit.edu/6.828/2012/homework/xv6-zero-fill.html)

Before we move on to adding file-backed mappings to our mmap() implementation, let's make xv6 a little closer to a full operating system such as Linux. Currently, the **mmap()** system call immediately allocates (kalloc) and map (mappages) all the pages of physical memory needed to fulfill the user request. In this part of the project, you'll be changing this behavior into lazy page allocation, where pages are only allocated when a pagefault occurs during access.

Lazy page allocation has a number of advantages over the current implementation:

- Some programs allocate memory but never use it, for example, to implement large sparse arrays. Lazy page allocation allows us to avoid spending time and physical memory allocating and mapping the entire region.
- We can allow programs to map regions bigger than the amount of physical memory available. Do remember that, one of the operating system's main goal is to provide the illusion of unlimited resources.

2. Causing Pagefaults

The first step is to modify your current **mmap()** implementation to remove the mappages call that map new memory region into the page table.

Remember *mmap()* will still have to "claim" the virtual addresses by increasing the size of the process. *mmap()* will also still have to add the necessary information to your memory region data structure. Notably, each region will still have a valid starting virtual address. Doing this will now cause any access to newly mmapped regions to pagefault. Try it and see!

Now that you are getting pagefaults, it's time to handle them! When a trap is generated in xv6, it is handled in the switch statement inside the file ***trap.c***. You can see examples of how xv6 is detecting traps such as system calls and the timer interrupt here. The *trapno* (trap number) is stored inside the trap frame and you can compare the value to the macros defined inside *traps.h*.

The next step is to add a pagefault handler in *trap.c* to attempt to recover from a pagefault. It helps here to create a function to solely handle pagefaults and pass the entire trapframe in as an argument. First, you'll be validating that the faulting address belongs to a valid mmap region stored inside your data structure. Then, you can map the single page around the faulting address into the page table.

3. Tips

- Use **PGROUNDDOWN(va)** to round the faulting virtual address down to a page boundary.
- *break* or *return* in order to avoid the *cprintf* and the *proc→killed = 1*.
- You'll need to call **mappages()**. In order to do this you'll need to delete the static in the declaration of mappages() in vm.c, and you'll need to declare mappages() in defs.h.
- you can check whether a fault is a page fault by checking if *tf→trapno* is equal to T_PGFLT in trap().
- Additionally, you are required to add the following debugging printf statements inside your *pagefault_handler(struct trapframe *tf)* method in *trap.c* which you will be adding,

```
void pagefault_handler(struct trapframe *tf)
{
     // Your code
     cprintf("============in pagefault_handler============\n");
     cprintf("pid %d %s: trap %d err %d on cpu %d "
          "eip 0x%x addr 0x%x\n",
          curproc->pid, curproc->name, tf->trapno,
          tf->err, cpuid(), tf->eip, fault_addr);

     // Your code
}
```

4. Part 5b: mmap Part 2: File-Backed Mappings

You can now support file-backed mmap regions. File-backed memory maps are the same as anonymous regions except that they are initialized with the contents of the file descriptor. They also can write the changes made to the memory region back to the file using the **msync()** system call, which you'll also implement in this part.

File-backed memory maps can be extremely useful in avoiding the high cost of writing changes to persistent storage. It lets the program manipulate the mapped file in memory as if it is a byte array, then write multiple changes back all at once. The tradeoff is that no changes made to file-backed memory regions are guaranteed to be persistent until **msync()** successfully returns. (Calling **munmap()** also does not guarantee durability).

Before beginning, we recommend you familiarize with the xv6 file system code.

5. Seeking in a file

Here is the mmap() prototype again for reference:

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

The *offset* argument to mmap() is describing the offset into the file that the file-backed region should be initialized from and written back to. Currently, xv6 does not support seeking to another offset in the file. You will have to add this function before proceeding with the rest of this part.

Add a new function with this prototype to file.c :

```
int fileseek(struct file* f, uint offset);
```

This function will change the offset field inside the struct file to allow xv6 to seek around in the file before doing a read or write operation.

You will need to acquire the inode lock inside your new function before modifying the offset field. This is to prevent any potential races. There are examples of how this is done in the other functions inside file.c.

6. Flags and Memory Region Permissions

mmap() also takes in protection bits and flags to customize the behavior of the memory region. In xv6, you are required to add the support for only few flags. You can define the following flags in a new header file **"mman.h"** to be included by both the *mmap()* implementation, as well as any user programs that uses *mmap()*:

Protection bits:

- *PROT_WRITE*:
  - Pages may be written. By default, pages may ***only*** be read.

Flags:

- *MAP_ANONYMOUS:*
  - The memory region will be anonymous. fd has to be set to -1.
- *MAP_FILE*:
  - The memory will be file-backed. fd **has** to be a valid file descriptor opened in read/write mode.
  - If any of these flags are set inconsistently or incorrectly, *return -1*.

Note: In Linux, *mmap()* has many more flags and features. See the man pages if you want to learn more. But it is out of scope for this project..!

7. System calls: mmap, munmap, and msync

The first step is to extend your **mmap()** and **munmap()** implementation from the previous project to handle the arguments related to files. This includes *int fd*, *int offset*, argument.

- mmap():
  - In mmap(), first determine which type of memory region is being mapped.
  - If the *fd* is greater than -1, error check the *fd* argument, and if it is valid, create a duplicate file descriptor to save in your data structure. Remember to increment the open count of the file. You will be reading in the contents from the file inside the page fault handler as the program tries to access that page.
  - If at any point an error occurs, make sure to deallocate any data structures and free any memory regions before returning -1.

- munmap():

For munmap() , there shouldn't be much changes. In addition to implantation you have done in the previous project, make sure to deallocate the duplicated file descriptor and decrement the open count using fileclose() before freeing your data structure. That's it!

- msync():

For the new msync() system call, it will look like this:

```
int msync(void* start_addr, int length):
```

The first step is to find the corresponding memory region. You can make the same assumptions as you did for munmap(): a correct call to msync will use the same *start_addr* and length used in mmap(). Then, write the contents of the memory region to the file descriptor you have stored in your data structure. Remember to use the offset stored in the region! If everything goes well, you can return 0. Otherwise, return -1 for error.

Since we are using lazy page allocation in mmap, some pages in the memory region could have never been touched! This means that during the msync() call, you would only want to write back the pages that are actually present in the process's page table. You can check if a page is mapped by calling the walkpgdir() function inside vm.c.

8. Optimizing msync()

In msync() the kernel can check the dirty bit of the pages within the mapped region and only write pages that have been modified to disk. For large regions and sparsely update regions, this can increase the performance significantly! xv6 already uses some bits inside the *pte*. You can find the macro definitions inside mmu.h.

The first step is to add some macros for extracting the dirty bit from the PTE. The xv6 book has a good picture and reference for where the dirty bit is located inside of the PTE. Then, you can update your msync() implementation to only write a page to disk if it satisfies the following conditions:

- It belongs to a valid mmap region.
- It is mapped into the process's address space.
- The dirty bit is set in the page's PTE.

Note: There are some other ways we can optimize msync() even more. One way is that we are writing each page to disk in separate filewrite() calls even if they are in fact contiguous. It would be a good idea to check for this so that we only call filewrite() once for each contiguous "chunk" of mapped and dirty mmap regions. You don't have to implement this unless you want to, but it's a good idea when designing a system to keep in mind the places where improvements can be made or where you are sacrificing something.

9. Tips

- Remember to clear the page allocated using kalloc() using a memset
- The hardware will always set the dirty bit on a modification, so make sure to clear the dirty bit after the initial fileread() into the page.
- Be careful of permissions when dealing with file descriptors.

10. Running Tests

- Use the following script file for running the tests:
```
prompt> ./test-mmap.sh -c -v
```

- If you implemented things correctly, you should get some notification that the tests passed. If not …

- The tests assume that xv6 source code is found in the *src/* subdirectory. If it's not there, the script will complain.

- The test script does a one-time clean build of your xv6 source code using a newly generated makefile called *Makefile.test*. You can use this when debugging (assuming you ever make mistakes, that is), e.g.:

```
prompt> cd src/
prompt> make -f Makefile.test qemu-nox
```

- You can suppress the repeated building of xv6 in the tests with the '-s' flag. This should make repeated testing faster:
```
prompt> ./test-threads.sh -s
```

- You can specifically run a single test using the following command
```
./test-mmap.sh -c -t 7 -v
```
The specifically runs the test_7.c alone.

- The other usual testing flags are also available. See the testing appendix for more details.

- This project will have few hidden test cases apart from the provided test cases.

11. Adding test files inside xv6

- Inorder to run the test files inside xv6, manually copy the test files (*test_1.c, test_2.c* etc…) inside xv6/src directory.(preferably in a different name like *xv6test_1.c*, etc…)
- Make the necessary changes to the Makefile

```
UPROGS=\
    _cat\
    _echo\
    _forktest\
    _xv6test_1\
    _grep\
    _init\
    _kill\
    _ln\
    _ls\
```

```
EXTRA=\
    mkfs.c xv6test_1.c ulib.c user.h cat.c echo.c forktest.c
grep.c kill.c\
    ln.c ls.c mkdir.c rm.c stressfs.c usertests.c wc.c zombie.c\
    printf.c umalloc.c\
    README dot-bochsrc *.pl toc.* runoff runoff1 runoff.list\
    .gdbinit.tmpl gdbutil\
```

- Now compile the xv6 using the command "*make clean && make qemu-nox*".

```
prompt> make clean && make qemu-nox
```

- Once it has compiled successfully and you are inside xv6 prompt, you can run the test.

```
$
$ xv6test
```

- You can also add your own test cases to test your solution extensively.
- Once you are inside xv6 qemu prompt in your terminal, if you wish to shutdown xv6 and exit qemu use the following key combinations: press Ctrl-A, then release your keys, then press X. (Not Ctrl-X)

Appendix – Test options

The run-tests.sh script is called by various testers to do the work of testing. Each test is actually fairly simple: it is a comparison of standard output and standard error, as per the program specification.

In any given program specification directory, there exists a specific tests/ directory which holds the expected return code, standard output, and standard error in files called n.rc, n.out, and n.err (respectively) for each test n. The testing framework just starts at 1 and keeps incrementing tests until it can't find any more or encounters a failure. Thus, adding new tests is easy; just add the relevant files to the tests directory at the lowest available number.

The files needed to describe a test number n are:

- n.rc: The return code the program should return (usually 0 or 1)
- n.out: The standard output expected from the test
- n.err: The standard error expected from the test
- n.run: How to run the test (which arguments it needs, etc.)
- n.desc: A short text description of the test
- n.pre (optional): Code to run before the test, to set something up
- n.post (optional): Code to run after the test, to clean something up

There is also a single file called pre which gets run once at the beginning of testing; this is often used to do a more complex build of a code base, for example. To prevent repeated time-wasting pre-test activity, suppress this with the -s flag (as described below).

In most cases, a wrapper script is used to call ***run-tests.sh*** to do the necessary work.

The options for run-tests.sh include:

- -h (the help message)
- -v (verbose: print what each test is doing)
- -t n (run only test n)
- -c (continue even after a test fails)
- -d (run tests not from tests/ directory but from this directory instead)
- -s (suppress running the one-time set of commands in pre file)