Project #4 - xv6 Kmalloc & Anonymous mmap

1. Part 4a: A memory allocator for the kernel – the kmalloc

Your task will be to build a memory allocator for the kernel in xv6. Currently, xv6 only has a user-space allocator. This cannot be used by the kernel since the memory returned by the user-space allocator is freed when the process terminates; the kernel instead requires memory that will be stable beyond the lifetime of a process.

Your new kmalloc should look and function very similar to malloc. It should be able to allocate variable-sized chunks of memory. We recommend that you create a new source file named kmalloc.c and reuse most of umalloc.c. You will have to modify the morecore() call and replace how umalloc() asks the operating system for more memory.

```
umalloc.c:

 if(nu < 4096)
    nu = 4096;
  p = sbrk(nu * sizeof(Header));
  if(p == (char*)-1)
    return 0;
```

You might have also noticed a kalloc() call. kalloc does also let you allocate memory but only in units of 4096 bytes. If the allocation does not use all 4096 bytes of that allocation, then the rest is wasted. For any meaningful kernel, wasting up to 4K of memory for each malloc call in the kernel quickly becomes unsustainable.

The **kmalloc()** and **kmfree()** calls should look like this:

```
void *kmalloc(uint nbytes);
void kmfree(void *addr);.
```

You are free to manage the memory regions using algorithms similar to the user-space memory allocator. However kmalloc() cannot use **growproc()** to get more memory, so you will be using **kalloc()** to receive a free page to refill the free chunks in your kmalloc()

The goal of the kmalloc is to give your xv6 kernel the ability to allocate memory for kernel data structures without wasting space by using kalloc().

2. Size limitations

kalloc() cannot return more than 4096 bytes (a single page) at a time. It does not keep multiple contiguous pages of memory. As a result, your kmalloc() is also limited to obtaining a maximum of 4096 bytes at a time.

For the scope of this assignment, this is okay. In the next part (part b) you will be designing the kernel data structure that keeps track of memory regions mapped into each process using the **mmap()** call. You will have to keep this limitation of kmalloc() in mind as you think about how each process should keep track of these regions.

3. Tips

- To avoid doing redundant work, you are welcome to use most of the code used in umalloc.c. You will however have to modify how your allocator gets more memory.
- Remember that the kernel, which kmalloc belongs in, should not be invoking system calls.
- Add a check that panics the kernel if you do ask kmalloc() for more memory than the limitation allows for. (4096 bytes).

Example usage:

```
struct mystruct *s = kmalloc(sizeof(*s));

// do something with mystruct...

// later, perhaps in another function:
kmfree(s);
```

- You can refer to Section 8.7 of the book named "Programming Language in C" by Brian W. Kernighan & Dennis M. Ritchie to understand about the umalloc code.

4. For Testing Purposes

- Just add two system calls for kmalloc() and kmfree(), which will simply call the actual kernel implementation of kmalloc & kmfree.
- This is purely for testing purposes and it is enough if you just write wrappers for calling the kmalloc & kmfree from the user space.
- The test cases won't try to access the memory returned by kmalloc, rather just stress test them. (You can refer test_1.c)

5. Part 4b: mmap Part 1: Anonymous Mappings

Now that you've added a malloc for the kernel, xv6 is ready for **mmap()**! You'll be implementing a slightly simplified version of mmap compared to Linux, but the core concepts will be the same. Reading through the man pages for mmap will help you greatly.

- **mmap():**

Your new mmap system call should look like this:

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

This call creates a new mapping in the calling process's address space by utilizing the **allocuvm()** call in vm.c. It returns the starting address of the newly mapped memory region. The addr argument is a hint for the kernel to place the new memory region. If addr is NULL (0), then the kernel chooses the (page-aligned) address to create the mapping. If addr is not NULL (0), the kernel may or may not take the hint to place the new region at the nearest page-aligned address. If another mapping already exists there, then the kernel will pick another address. New regions will always begin at page-aligned addresses.

The functionality of prot, flags, fd, and offset argument will be implemented in the next assignment (Project #5).

As part of this system call, the kernel will need to have some way to keep track of the new memory mapping created. This is important to handle **munmap()** calls which will unmap memory regions, as well as to keep track of metadata such as permissions and type of memory region (Anonymous or file backed).

Finally, mmap() should zero out the entire newly mapped anonymous memory region and return the starting address.

Note: If at any point an error occurs, make sure to deallocate any data structures and free any memory regions before returning -1.

▪ **munmap():**

The munmap system call will look like this:

```
int munmap(void *addr, uint length);
```

The first step will be to verify that the address and length passed is indeed a valid mapping that can be removed. This will involve traversing your data structure for tracking mapped regions and finding the corresponding regions. You can assume that a correct call to munmap will always use the same starting address and length used in the mmap() call.

For anonymous regions no data is preserved by the kernel when unmapping, so you can simply clear out the memory region. Clearing out the memory is crucial so that another process cannot potentially see the old data written by mmap() if they happen to mmap() the same physical page of memory.

Then use the deallocuvm() call in vm.c to unmap the memory region. Finally, you can free (kmfree) the data structure element used to track the freed region.

6. Which data Structure to use?

We recommend using a linked list to keep track of mmap regions. Each process will have its own set of mappings, so you'll need to modify the process struct, 'struct proc' in the file proc.h. Each element in the list can represent a contiguous region of mapped memory and will keep track of the following metadata:

- starting address
- length
- region type (anonymous vs file-backed)
  - You will be implementing file-backed in the next project.

- offset
  - The offset into the file for file-backed memory regions.

- fd

  - the duplicate of the file descriptor passed into mmap() if it is a valid fd. We want to duplicate the fd passed in because the user is allowed to close the file descriptor while the region is still in use. You can duplicate a file descriptor by using filedup() found in file.c. There is an example of how it's used in sys_dup(). If the mmap() is for an anonymous region, this should be set to -1.

You are free to add more fields if required to the metadata. You will be using kmalloc to allocate memory to the data structure you are using. Also, remember to deallocate any memory you have allocated using kmalloc when the process terminates to prevent the memory leaks. Data structures inside the process struct belong to the kernel, so memory leaks here are even "worse" than ones inside applications, as there is no "process" that can be restarted for the kernel. The logical place to do this would be in freevm() in vm.c.

7. Tips

- It might make sense to use argptr() to parse the address argument for the mmap calls, however, take a closer look at the code for argptr() to see why it might not work.
- It's much simpler to only keep track of mmap regions instead of every address region mapped by the process.
- Creating some functions that dump the info contained in your data structure in a neat manner to stdout will make debugging much easier.
- Create helper functions to manage/access the data structure you use to keep track of mapped regions.

8. 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.

9. 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\
    _mkdir\
    _rm\
    _sh\
    _stressfs\
    _usertests\
    _wc\
    _zombie\
```

```
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)