Operating System 2021 Machine Problem Site

The site provides the information for machine problems for Operating System Course offered in Spring 2021 at National Taiwan University

MP2 - Shared Memory

Description

In computer software, **shared memory** is a way of exchanging data between process. One process will create an area in memory which other processes can access. Since processes can access the shared memory area like regular working memory, this is a very fast way of inter-process communication (**IPC**).

Shared memory is also a method of conserving memory space by directing accesses to what would ordinarily be copies of a piece of data to a single instance instead, by using virtual memory mappings or with explicit support of the program in question.

Portable Operating System Interface (**POSIX**) provides a standardized shm_open application programming interface (API) for using shared memory. POSIX's IPC (part of the POSIX:XSI Extension) also provides the shared memory facility in sys/shm.h.

What's more, POSIX provides the mmap API for mapping files into memory, the one you might be more familiar with; a mapping can be shared, allowing the file's contents to be used as shared memory.

In MP2, you'll add `mmap` and `munmap` to xv6, focusing on memory-mapped (mmap-ed) files.

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.

You only need to build them with limited utility required by shared memory. We assume you have possessed the basic knowledge of **file descriptor** and **inode**, which are taught in System Programming.

Before Coding

After accepting MP2 assignment on GitHub Classroom, clone the repository to your machine and change directory under it.

```
$ git clone [mp2_repository_path]
$ cd [mp2_repository]
```

Pull the image from Docker Hub and start a container.

```
$ docker pull ntuos/mp2
$ docker run -it -v $(pwd)/xv6:/home/mp2/xv6 ntuos/mp2
```

Explanation

- Preliminary (35%)
    - Print a Page Table (10%+10%)
    - Generate a Page Fault (10%)
    - Add System Call Stubs (5%)
- Implementation (50%)
    - mmap (15%+20%)
    - munmap (10%+5%)
- Shared Virtual Memory (15%)

Preliminary

Print a Page Table (20%)

(10%) Define a function called `vmprint()`. It should take a `pagetable_t` argument, and print that page table in format described below.

Format

```
// 'Update'
page table 0x0000000087f6f000
 ..0: pte 0x0000000021fdac01 pa 0x0000000087f6b000
 .. ..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
 .. .. ..0: pte 0x0000000021fdb01f pa 0x0000000087f6c000
 .. .. ..1: pte 0x0000000021fda40f pa 0x0000000087f69000
 .. .. ..2: pte 0x0000000021fda01f pa 0x0000000087f68000
 ..255: pte 0x0000000021fdb801 pa 0x0000000087f6e000
 .. ..511: pte 0x0000000021fdb401 pa 0x0000000087f6d000
```

```
.. .. ..510: pte 0x0000000021fddc07 pa 0x0000000087f77000
.. .. ..511: pte 0x0000000020001c0b pa 0x0000000080007000
```

- The first line displays the argument to `vmprint`.
- After that, there is a line for each PTE, including PTEs that refer to page-table pages deeper in the tree.
- Each PTE line is indented by a number of `" .."` that indicates its depth in the tree.

> **Note**: It's `" .."`, not `".. "`. Otherwise, you may not pass the judge.

- Each PTE line shows:
    - The PTE index in its page-table page
    - The PTE bits
    - The physical address extracted from the PTE
- Don't print PTEs which are not valid.

To print your first process's page table, you can insert `if(p->pid==1) vmprint(p->pagetable)` in `kernel/exec.c` just before the `return argc`. Then, if you start xv6, it should print output like the above example, describing the page table of the first process at the point when it has just finished `exec()` ing `init`.

Your code might emit different physical addresses than those shown above. But the number of entries and the virtual addresses should be the same.

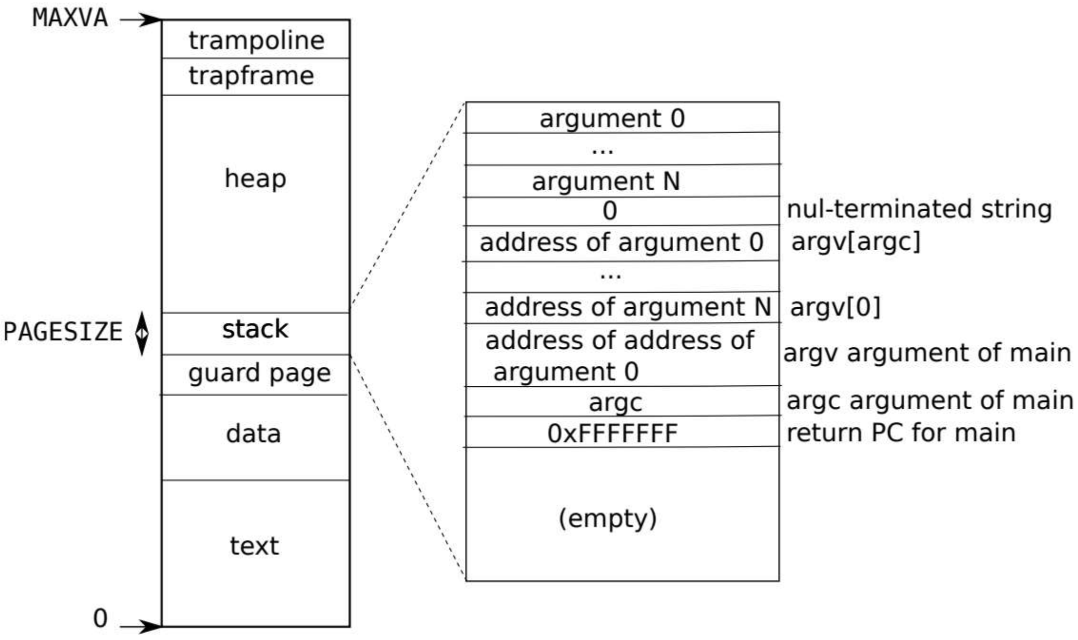
In the above example, the top-level page-table page has mappings for entries 0 and 255. The next level down for entry 0 has only index 0 mapped, and the bottom-level for that index 0 has entries 0, 1, and 2 mapped.

Hint

- Read **Chapter 3** of the xv6 book, and related files:
    - `kernel/memlayout.h`, which captures the layout of memory.
    - `kernel/vm.c`, which contains most virtual memory code.
- You can put `vmprint()` in `kernel/vm.c`.
- Use the macros at the end of the file `kernel/riscv.h`.
- The function `freewalk` in `kernel/vm.c` may be inspirational.
- Define the prototype for `vmprint` in `kernel/defs.h` so that you can call it from `kernel/exec.c`.
- Use `%p` in your `printf` calls to print out full 64-bit hex PTEs and addresses.
- ~~File under `material/` may help.~~

(10%) The figure below shows a process's user address space, with its initial stack. **In report, explain the output of `vmprint` in terms of the figure below and answer questions:**

- What does page 0 contain?
- What is in page 2?
- When running in user mode, could the process read/write the memory mapped by page 1? Why?



Note: Remember to comment out or remove the additional code you just added in `kernel/exec.c` before `git push`, but keep `vmprint`. TA will judge your `vmprint()` and it also helps you debugging in later sections.

Generate a Page Fault (10%)

Lazy allocation of user-space heap memory is one of the many neat tricks an OS can play with page table hardware.

xv6 applications ask the kernel for heap memory using the `sbrk()` system call, which is implemented at the function `sys_sbrk()` in `kernel/sysproc.c`. In xv6 kernel, `sbrk()` allocates physical memory and maps it into the process's virtual address space. It can take a long time for a kernel to allocate and map memory for a large memory request. For example, consider that a gigabyte consists of 262,144 4096-byte pages; that's a huge number of allocations.

In addition, some programs allocate more memory than they actually use (e.g., to implement sparse arrays), or allocate memory well in advance of use.

To allow `sbrk()` to complete more quickly in these cases, sophisticated kernels allocate user memory lazily. That is, `sbrk()` doesn't allocate physical memory, but just **remembers which user addresses are allocated and marks those addresses as invalid in the user page table**.

When the process first tries to use any given page of lazily-allocated memory, the CPU generates a page fault, which the kernel **handles by allocating physical memory, zeroing it, and**

mapping it.

You will need to add lazy allocation feature to `mmap` in later sections. For this section, you simply need to eliminate allocation from `sbrk()` and to understand what happens to xv6.

Delete page allocation from the `sbrk(n)` system call implementation (`sys_sbrk()` in `kernel/sysproc.c`). The `sbrk(n)` system call grows the process's memory size by `n` bytes, and then returns the start of the newly allocated region (i.e., the old size). Your new `sbrk(n)` should just increment the process's size (`myproc()->sz`) by `n` and return the old size. It should not allocate memory.

Try to guess what the result of this modification will be: What will break?

Hint

- Read **Section 4.6** of the xv6 book and related file `kernel/trap.c` .
- You should delete the call to `growproc()` .
- You still need to increase process's size.

After making modifications, boot xv6 and type `echo hi` to the shell. You should see something like this:

```
init: starting sh
$ echo hi
usertrap(): unexpected scause 0x000000000000000f pid=3
            sepc=0x00000000000012b4 stval=0x0000000000004008
panic: uvmunmap: not mapped
```

The "usertrap(): ..." message is from the user trap handler in `kernel/trap.c` . It has caught an exception that it does not know how to handle. The "stval=0x0..04008" indicates that the virtual address that caused the page fault is 0x4008. **In report, explain why this page fault occurs.**

Note: Remember to undo your revision in `sbrk(n)` before `git push` . No code will be graded in this section.

Add System Call Stubs (5%)

Now we start explain engineering details on `mmap` and `munmap` . We provide `user/mp2test.c` to help you undertake the mission step by step. Each test in `mp2test` will be based on the assumption that you have already passed previous tests. Our judge will do the same way, so do not skip sections. See sample execution

1. `mmap`

If you run `man 2 mmap` in Linux platform, the manual page shows this declaration for `mmap` :

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

You can also try it in our container (outside xv6).

`mmap` can be called in many ways, but MP2 only requires a subset of its features relevant to **memory-mapping a file**.

2. `munmap`

You may notice that the manual page for `mmap` (run `man 2 mmap`) also shows this declaration for `munmap` :

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

Add `$U/_mp2test` to `UPROGS` in Makefile. Run `make qemu` and you will see that the compiler cannot compile `user/mp2test.c` , because the user-space stubs for the system call `mmap` and `munmap` don't exist yet.

To add system calls, you need to give xv6:

- User-space stubs for the system calls
    - Add prototypes for the system calls to `user/user.h`
    - Add stubs to `user/usys.pl`
    - Add syscall numbers to `kernel/syscall.h`

  `Makefile` invokes the Perl script `user/usys.pl` , which produces `user/usys.S` . The actual system call stubs, which use the RISC-V `ecall` instruction to transition to the kernel.

- Kernel-space implemetation of the system calls
    - Add `sys_mmap()` and `sys_munmap()` functions in `kernel/sysproc.c` .

  For now, just return errors from `sys_mmap()` and `sys_munmap()` . Leave the implementation in later sections.

Hint

- Read **Chapter 2**, **Section 4.3** and **Section 4.4** of the xv6 book.
- Each of these functions should use its argument in a variable in the `proc` structure (see `kernel/proc.h`).
- The functions to retrieve system call arguments from user space are in `kernel/syscall.c` .

- You can see other system calls as examples in `kernel/sysproc.c` .
- If you're stopped by type issues, solve it.
- If you want to know what each field represents, see SPECs below.

Run `mp2test` , which will fail at the first `mmap` call but give you some informational messages.

Implementation

If you try doing the implementation, **you need to briefly explain how you manage VMA in process's user address space in report**, depending on how far you go.

- Following sections are expected to be **DIFFICULT**. You are encouraged to discuss with other classmates, but do not share code or report. **Write the code on your own** and **write the report in your own words**.
- **If your explanation does not correspond to your code, you are also suspected of committing plagiarism.** TAs will decide whether you can get partial score for corresponding section, or fail this course.
- List your assumptions, if any. Of course, any assumption should not violate the SPECs.
- **If your code was based on some critical assumptions and you lack them in report, TAs will decide whether you get partial score or get 0 for corresponding section.** In some severe cases, you may be suspected of committing plagiarism.

Note: Make sure you fully understand Print a Page Table and Generate a Page Fault sections before moving on.

SPECs

1. `mmap`

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

- `addr` will always be zero in MP2.
- `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*; i.e., `prot` is `PROT_READ` or `PROT_WRITE` or both.
- `flags` will be one of the following bits:
    - `MAP_SHARED` : modifications to the mapped memory should be written back to the file.
    - `MAP_PRIVATE` : modifications should not be written back.

    You don't have to implement any other bits in `flags` .
- `fd` is the open file descriptor of the file to map.

- `offset` will always be zero in MP2; that is, we always map the file from the starting point of the mmap-ed file.

What we expect from your `mmap` call: The kernel should decide the **virtual address** where to map the file. `mmap` returns that address, or `0xffffffffffffffff` if it fails.

2. `munmap`

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

`munmap` should remove `mmap` mappings in address range indicated by `addr` and `length`. If the mappings has been mapped `MAP_SHARED` and the process has modified them, the modifications should first be written to the file.

Bare `mmap` (15%)

To keep track of what `mmap` has mapped for each process, **define a VMA (virtual memory area) structure for xv6 yourself**, recording the address, length, permissions, file, etc. for a virtual memory range created by `mmap`. You can refer to Linux's VMA.

Find the **unused region** in the process's address space in which to map the file, and **add the 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

- Read **Section 8.13** of the xv6 book.
- Recall what you have written in report for Print a Page Table section.
- You can define your VMA structure wherever is convenient for you. Of course, it must not affect the `Makefile`.
- 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.
- Write your implementation code in `sys_mmap()`, the function you just added in Add System Call Stubs section.
- TA defined `PROT_READ` etc for you in `kernel/fcntl.h`.

If all goes well, by running `mp2test`, the first test `mmap bare` should succeed, but the mmap-ed memory will cause page fault (and thus lazy allocation) and kill `mp2test`. See sample execution

> **Note**: In `mp2test.c`, you can comment out some code such as `munmap` system call, or insert some print functions, to adjust the test.

`mmap` with Lazy Allocation (20%)

Fill in 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` in `kernel/trap.c`. As mentioned in Generate a Page Fault section, 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.

Modify the code in `kernel/trap.c` to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing.

To handle page fault in a mmap-ed region:

- Add code to allocate a page of physical memory.
- Read 4096 bytes of the relevant file into that page.
- Map it into the user address space.
- Set the permissions correctly on the page.

Hint

- `kernel/trap.c` catches an exception, particularly to respond a page fault in MP2.
- Read the explanation in Generate a Page Fault section again, to see how **lazy allocation** mechanism works in xv6.
- Recall what you have written in report for Print a Page Table and Generate a Page Fault sections.
- You can check whether a fault is a page fault by seeing if `r_scause()` is **13** or **15** in `usertrap()`.
- `r_stval()` returns the RISC-V `stval` register, which contains the virtual address that caused the page fault.
- Use `PGROUNDDOWN()` to round the faulting virtual address down to a page boundary.
- Read the file `kernel/kalloc.c`, which contains code for allocating and freeing physical memory.
- Steal code from `uvmalloc()` in `kernel/vm.c`, which is what `sbrk()` calls (via `growproc()`). You'll need to call `kalloc()` and `mappages()`.
- Handle out-of-memory correctly: If `kalloc()` fails in the page fault handler, kill the current process.
- Read the file with `readi`:

  ```c
  int readi(struct inode *ip, int user_dst, uint64 dst, uint off, uint n);
  ```

  It takes an offset argument at which to read in the file. If `user_dst==1`, then `dst` is a user virtual address; otherwise, `dst` is a kernel address.

- You will have to lock/unlock the inode passed to `readi`. That is, caller must hold `ip->lock`.
- Use your `vmprint` function to print the content of page table for debugging.
- If the kernel crashes, look up `sepc` in `kernel/kernel.asm`.
- If you see the error "incomplete type proc", include `spinlock.h` then `proc.h`.
- If `uvmunmap()` panics, modify it to not panic when some pages aren't mapped.

Run `mp2test`. It should pass the test `mmap lazy` and stop at the first `munmap` test. See sample execution

Bare `munmap` (10%)

Find the VMA for the address range and unmap the specified pages. If `munmap` removes all pages of a previous `mmap`, it should decrement the reference count of the corresponding `struct file`. If the page to be unmapped has been modified and the file is mapped `MAP_SHARED`, write the page back to the file.

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, TA will not check that non-dirty pages are not written back; thus you can get away with writing pages back without looking at D bits.

Hint

- Similar to what you've done in Bare mmap section, write your implementation code in `sys_munmap()`.
- You may want to use `uvmunmap`.
- Look at `filewrite` in `kernel/file.c` for inspiration.

Run `mp2test`. The test `munmap bare` should pass. See sample execution

> **Note**: Remember to undo some changes you've made in `mp2test.c`.

Reclaim mmap-ed Files (5%)

OSes appear to be lazy, but the programmers seem right behind.

Conceptually, when the process is about to exit, it is **programmers' responsibility** to free allocated memory, to close non-standard file descriptors, etc. The classic POSIX programming guide Advanced Programming in the UNIX® Environment states:

> **When a process terminates, all of its open files are closed automatically by the kernel. Many programs (programmers) take advantage of this fact and don't explicitly close open files.**

Although that's not something a programmer should expect, every sane OS will clean up after process ends.

Therefore, your final mission of implementing the system calls `mmap` and `munmap` is to modify `exit` to unmap the process's mapped regions as if `munmap` had been called.

Run `mp2test`. The test `munmap exit` should pass. See sample execution

Shared Virtual Memory (15%)

Modify `fork` to ensure that the child has the same mapped regions as the parent.

~~In page fault handler of the child, **allocate a new physical page** for itself.~~
(Update) Allocate a new physical page for the child.

Hint

- Handle the parent-to-child memory copy correctly.
- Increment the reference count for a VMA's `struct file`.

Sharing a physical page with the parent would be cooler, but it would require more implementation work. You can try it in Bonus part.

Run `mp2test`. It should pass all tests.

Sample Execution

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

```
$ mp2test
mp2_test starting
test mmap bare
test mmap bare: PASS
test mmap lazy
test mmap lazy: PASS
test munmap bare
test munmap bare: PASS
test munmap exit
test munmap exit: PASS
test shared virtual memory
test shared virtual memory: PASS
mp2test: all tests succeeded
```

Bonus

- **Do not try it unless you think your life is too easy.**
- **TA won't answer any question about bonus part.**

Shared Physical Memory (20%)

If two processes get the same file mmap-ed, share their physical pages.

In Shared Virtual Memory part, our solution 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.

Hint

- This requires that file blocks be the same size as pages (set `BSIZE` to 4096).
- You need to pin mmap-ed blocks into the buffer cache.
- You need to worry about reference counts on physical pages.

In report, **explain how you manage kernel page table** and **describe the difference** between the implementation of Shared Virtual Memory and the one of Shared Physical Memory.

Submission

The score in summary is:

- Preliminary (35%)
  - Print a Page Table (10%+10%*)
  - Generate a Page Fault (10%*)
  - Add System Call Stubs (5%)
- Implementation (50%*)
  - mmap (15%+20%)
  - munmap (10%+5%)
- Shared Virtual Memory (15%)

*Need to write report.

Early Bird (5%)

Finish **Preliminary** part before **April 6th 23:59** (i.e., by the end of spring break).

Report

Submit your reports via Gradescope:

- MP2-Preliminary (Early Bird) **or** MP2-Preliminary
- MP2-Implementation
- MP2-Bonus

In order not to make confusion, MP2-Preliminary (Early Bird) will open until April 6th. Other three will open from April 7th.

- If you submit **MP2-Preliminary (Early Bird)**, when we judging source code, **your code in Preliminary part** (i.e. Print a Page Table and Add System Call Stubs) **before the deadline of Early Bird will be judged and graded**. (TA will checkout your repository to latest commit before the deadline).
- If you want to keep working on repository and don't want to affect grading, you can open a new branch and merge it back afterward. Leave your solution in `master` branch.
- Only if you submit **MP2-Bonus** will TA grade your Bonus part.

Source Code

Push your xv6 source code to your MP2 repository on GitHub.

- Make sure your xv6 can be compiled.
- Run `make clean` before you push.
- You can push any other files we do not request, but at your own risk. You will get 0 if xv6 cannot be compiled.

Reference

- POSIX — IEEE Standard Portable Operating System Interface for Computer Environments https://ieeexplore.ieee.org/document/8684566
- mmap(2) — Linux manual page https://man7.org/linux/man-pages/man2/mmap.2.html
- xv6 — A simple, Unix-like teaching operating system by MIT https://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf
- Linux Kernel - The Process Address Space — sathya's Blog https://sites.google.com/site/knsathyawiki/example-page/chapter-15-the-process-address-space
- Advanced Programming in the UNIX® Environment https://www.oreilly.com/library/view/advanced-programming-in/9780321638014/
