Demand paging in xv6

March 8, 2019

The goal of this assignment to understand the paging subsystem of an operating system. The first component you need to implement is demand paging. Demand paging lazily maps a physical page corresponding to a virtual page during the time of dereferencing. In addition to demand paging, you also need to implement swapping. If the system runs out of physical memory, then the swapping subsystem reuse an already used physical page by saving its contents to secondary storage (e.g., disk).

1 Environment

For this assignment, you need to install qemu.
Run, `sudo apt-get install qemu`
Please clone the assignment repo from `https://github.com/Systems-IIITD/xv6-paging.git`.
Run `make qemu`.
It will boot the xv6 OS in qemu emulator. You can also redirect the output to a file using, `make qemu > log.txt`.
The default process is the shell. Type `memtest1` to run memtest1.
Currently, it panics because the page fault handler is not implemented. After, you have correctly implemented everything, memtest1 and memtest2 should run normally.

2 Implementation

To implement these features: please look at the disk and memory management APIs.

2.1 Memory Management

Read "*Page Tables*" chapter from the xv6 book. All physical pages are mapped in the kernel virtual address (KVA) space. KVA start from the address KERNBASE. The physical pages are mapped in the kernel at the location KERN-

BASE. To get the KVA of a physical address x, we simply need to add KERNBASE to x. Important functions to look at:

- kalloc() : Allocate a physical page. kalloc return KVA; subtract the KERNBASE to get the physical page.
- kfree() : Free a physical page (accept KVA).
- walkpgdir() : Return the address of the page table entry corresponding to a virtual address in the input page table (pgdir).
- mappages(): Map physical page corresponding to a virtual page in the page table (pgdir) with the given permissions.
- allocuvm(): Allocate and map physical pages for virtual addresses between oldsz to newsz.

2.2 Disk management

Read "*File System*" chapter from the xv6 book. The disk layer reads and writes blocks on an IDE hard drive. On top of the device interface, a buffer cache layer exists, that caches the disk data in memory for faster access. All the disk accesses are done through buffer cache APIs. The IDE hard drive reads/ write data on the sector granularity, which is 512 bytes long. The buffer cache block size is same as the disk sector size. The first block (block 0) on disk is boot sector. The second block is superblock. xv6 block allocator maintains a free bitmap on disk, with one bit per block. A zero bit indicates that the corresponding block is free; a one bit indicates that it is in use. The block number of the first bitmap block is stored in the superblock `bmapstart` field.

You can look at the "Block allocator" section in the xv6-book for a detailed discussion.

The important functions to look at:

- balloc() : Allocate a free disk block.
- bfree() : Free a disk block.
- readsb() : Read super block.
- bget() : Lookup a block in the buffer cache. If not found allocate a buffer and returned the locked buffer.
- bread() : If not found in buffer cache read a block from disk.
- brelse() : Release a locked buffer. bget()/bread() returns a locked buffer, which must be released after use.
- bwrite() : Write a block to the disk.
- log_write() : Safely writes a block to the disk. Must preceded/succeeded with begin_op()/end_op().
- iderw() : Read/write blocks to disk device.

2.3 Swapping

Implement system call swap (sys_swap). sys_swap takes a user virtual address of the current process to swap. For swapping:

- Implement balloc_page(). balloc_page() allocates 4 KB consecutive disk space and returns the address of first disk block.
- Save the content of the virtual page to a disk page. It is okay for this assignment to use buffer cache APIs to read/write swapped blocks.
- Mark the page-table entry corresponding to the swapped virtual page as invalid.
- Save the block id of the swapped location in the page-table entry itself. Reserve some bits in the page-table entry to identify a swapped page.
- Invalidate the TLB corresponding to the swapped virtual page.
- Free the physical page.

For this part, you may need to implement the following routines: sys_swap, balloc_page, balloc_free, write_page_to_disk, read_page_from_disk, swap_page_from_pte. Please feel free to add new routines or change interfaces of these routines.

2.4 Demand paging

`malloc()` uses `sbrk` system call to allocate virtual pages. Currently, sbrk does not map any physical page to the allocated virtual pages. Application dereferences to a virtual page for the first time triggers a page fault. In the page fault handler, you need to allocate a physical page and map it to the corresponding faulty address. If you are unable to allocate a physical page, then you must have to swap a virtual page in the current process address space. For this assignment, you need to implement the following policy for finding a victim for replacement:

- Find a virtual page whose access bit is not set.
- If you are unable to find a virtual page in the above step, randomly reset the access bit of the 10% of total allocated pages. It is also okay if you reset the access bit of only one virtual page.
- Repeat the first step.

If the faulty virtual page was previously swapped then you have to restore the contents of the swapped page and free the swapped disk blocks.

For this part, you may need to implement the following routines: map_address, swap_page, select_a_victim, clearaccessbit, getswappedblk. Please feel free to add new routines or change interfaces of these routines.

2.5 Fork

Currently, fork system call copies entire data from the parent process. If the memory gets exhausted during the copying, then swap pages from the parent process to make sure that copy succeeds. You should look at: fork, copyuvm routines.

2.6 Process termination

Free all the swap blocks when the process exits. You should look at: freevm, deallocuvm.

2.7 Bstat

You should implement the bstat system call which returns the global count of swapped pages. Please adjust the `numallocblocks` properly, when the swapped blocks are allocated and deallocated.

2.8 Synchronization

Look at the implementation of `acquire`, `release`, `acquiresleep` and `releasesleep` in `spinlock.c` and `sleeplock.c`. You can use these APIs directly for synchronization. If you want to understand how these APIs works internally, read chapter-4 from the xv6 book.

3 Test cases

memtest1 and memtest2 are two test cases. memtest1 tests the demand paging functionality and memtest2 tests the fork and swap system call. memtest3 does bstat system call to collect the swap space usage. Ideally, memtest3 should return zero after the completion of memtest1 and memtest2. Don't start directly with memtest1. First, write small test cases (either add a new test case or change memtest1). E.g., write a test case that `malloc` one virtual page and dereference it. When you able to get it working, add more functionality, e.g., the swap system call. Write small tests to test these functionalities.

3.1 Design documentation

Answer the following questions in your design documentation. If we find that you have provided any misleading fact in your submission, you may get a negative mark. Answer all questions.

- Are you able to run memtest1 without any error?
- Are you able to run memtest2 without any error?

- Run memtest3 after memtest1 and memtest2. What is the output of memtest3?
- When do you update `numallocblocks`?
- What is the structure of your page table entry that stores the demand paging/swapping information?
- Does your scheme has a limitation on maximum swap size? If yes, what is it?
- How do you ensure synchronization in `balloc_page` and `balloc_free`?
- Write the pseudocode code of your `select_a_victim` implementation.
- Can concurrent calls to `sys_swap` swap the same physical page? If yes, how do you ensure synchronization among them?
- If you are not able to run `memtest1` without any error, write the summary of your implementation. If you can run `memtest1`, then you don't have to answer this question.

3.2 How to submit.

Submit the entire xv6-paging folder to the submission link. Upload your design document at the given link on Backpack.