CS 333: Operating Systems Lab
Autumn 2018

Lab 6: munching on memory

Goal

In this lab we will understand how xv6 handles memory for user processes and the kernel.

Before you begin

- Download the source code for xv6 from the following url:
  `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-public.tar.gz`
  Make sure you read the README to understand how to boot into a xv6-based system.
- Download, read and use as reference the xv6 source code companion book.
  url: `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-rev10.pdf`
- The xv6 OS book is here:
  `http://www.cse.iitb.ac.in/~puru/courses/autumn18/labs/xv6-book-rev10.pdf`

0. Breaking the ice

In this exercise, you will walk through the xv6 code to understand how xv6 manages memory. xv6 uses 32-bit virtual addresses, resulting in a virtual address space of 4GB. It uses paging to manage its memory allocations with page size of 4KB and a two level page table structure.

- Look up description of Figures 1-3, 2-1, and 2-2, in the xv6 book, on pages 23, 30, and 31 respectively.
- A list of relevant function names and macros is as follows: `kalloc, kfree, allocuvm, growproc, walkpgdir, PDX, PTE_ADDR, v2p, p2v, mappages ...`

You should read the relevant code of above functions and macros, and explain the following in a file `memory.txt`. Keep your answers brief and precise.

1. Explain the code in each of the lines 3177 and 3178.
   ```
   3177 r->next = kmem.freelist;
   3178 kmem.freelist = r;
   ```
2. Consider the following sequence of statements executed in `walkpgdir` function. Explain what each line in this code is trying to do and how it achieves it.
   ```
   1740 pde = &pgdir[PDX(va)];
   1742 pgtab = (pte_t*)p2v(PTE_ADDR(*pde));
   1753 return &pgtab[PTX(va)];
   ```

Additional Reading: These can be read offline after the lab.

- The function `userinit` (line 2520) creates the first user process. For the user part of the memory, the function `inituvm` (line 1886) allocates one physical page of memory, copies the `init` executable into that memory, and sets up a page table entry for the first page of the user virtual address space.
- In `fork` (line 2580), once a child process is allocated, its memory image is setup as a complete copy of the parent's memory image by a call to `copyuvm` (line 2035). This function walks through the entire address space of the parent in page-sized chunks, gets the physical address of every page of the parent using a call to `walkpgdir`, allocates a new physical page for the child using `kalloc`, copies the contents of the parent's page into the child's page, adds an entry to the child's page table using `mappages`, and returns the child's page table.
- `allocuvm` (line 1927) allocates physical pages from the kernel's free pool via calls to `kalloc`, and sets up page table entries. Likewise `deallocuvm` (line 1961) deallocates pages and adds them to kernel's pool via calls to `kfree`.

1. Walking down memory lane

(a) Implement a system call `get_va_to_pa` to return the physical address mapping of a virtual address.
`int get_va_to_pa(uint va, uint* pa, int* flag)`

If a physical mapping exists for the virtual address `va`, the system call `get_va_to_pa` returns the physical address using the pointer `pa` and read/write `PTE` flag using `flags`, and returns 1. If a valid `V2P` mapping does not exist, the call should return 0.

A sample program `v2p.c` has been given to test your implementation.

(b) Implement the following functions in a user program.
`int getpusz()` // return size of user virtual address space of a process with valid mappings.
`int getpksz()` // return size of kernel virtual address space of a process with valid mappings.

Hints: For each of these function calls, find the valid `VA` to `PA` mappings using the `get_va_to_pa` system call and count the total allocated physical pages. Note this can be done at the granularity of a page.

Implement the following system call,
`int getpsz()` // return size of the (user) process as stored in the PCB.

Note: You are also given a sample program `proc-size.c`. Implement all the above functions in `proc-size.c` and test your implementations.

(c) **Self-study. Do this part now or later. Ignore it at your own peril.**
Implement a function `print_vm_block()` which prints contiguous virtual address regions with the following information. For all valid `V2P` mappings, it should print
`[<VA-start address> <VA-end-address>] <R/W-flag> <size>`.

`VA-start-address` and `VA-end-address` are the virtual address of first and last bytes in a contiguous region respectively. `<R/W-flag>` denotes the read and write permissions of a contiguous block.(e.g., `r`, `rw`, etc.) A contiguous virtual address region is delineated by a missing mappings of VA address or by different `PTE` R/W permissions.

You are also given a sample program `vm-block.c`. Implement the function `print_vm_block()` in `vm-block.c` and test your implementation.

2: Lazy page allocation in xv6

Source and credits: Homework assignment of course 6.828, MIT

One of the many neat tricks an OS can play with page table hardware is lazy allocation of heap memory. xv6 applications ask the kernel for heap memory using the `sbrk()` system call. For example, this system call is invoked when the shell program does a `malloc` to allocate memory for the various tokens in the shell command. In the xv6 kernel we have given you, `sbrk()` allocates physical memory and maps it into the process's virtual address space. However, there are programs that allocate memory but never use it, for example to implement large sparse arrays. Sophisticated kernels delay allocation of each page of memory until the application tries to use that page – as signaled by a **page fault**. You are add this lazy allocation feature to xv6.

Step 1: Eliminate allocation from `sbrk()`

Your first task is to **remove page allocation/mapping from the `sbrk(n)` system call implementation**. The `sbrk(n)` system call grows the process's (virtual) 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 It should not allocate and map physical memory. However, you should still increase process size by `n` to trick the process into believing that it has the memory requested.

Make this modification to the code, boot xv6, and type `echo hi` to the shell. You should see something like this:

```
init:  starting sh
$ echo hi
pid 3 sh:  trap 14 err 6 on cpu 0 eip 0x12f1 addr 0x4004-kill proc
$
```

The "pid 3 sh: trap..." message is from the kernel trap handler in `trap.c`; it has caught a page fault (trap 14, or `T_PGFLT`), which the xv6 kernel does not know how to handle. Make sure you understand why this page fault occurs. The `addr 0x4004` indicates that the virtual address that caused the page fault is `0x4004.`
You will need to understand how xv6 gets to the faulting virtual address.

Step 2: Lazy Allocation

Modify the code in `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. That is, you must allocate a new memory page (which is free), add suitable page table entries, and return from the trap handler, so that the process can access the virtual address originally accessed.
Your code should enable an *correct* mapping. Part of the exercise is the demonstrate correctness in different scenarios.
Some helpful hints:

- Look at the `cprintf` arguments of the appropriate statement in `trap.c` to see how to find the virtual address that caused the page fault.
- Understand and steal code from `allocuvm()` in `vm.c`, which is what `sbrk()` originally uses.
- Use PGROUNDDOWN(`va`) to round the faulting virtual address down to a page boundary.
- Once you correctly handle the page fault, do `break` or `return` in order to avoid the `cprintf` and the `proc->killed = 1` statements.
- You will need to call `mappages()` from `trap.c` in order to map the newly allocated page. 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 `trap.c`. Add this declaration to `trap.c` before any call to `mappages()`: `int mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm);`
- You will have to check whether a fault is a page fault by using `tf->trapno`.
- You should check whether the page fault was actually due to a lazy allocated page or an actual page fault (For example - illegal memory access)

If all goes well, your lazy allocation code should result in `echo hi` working. You should get at least one page fault (and thus lazy allocation) in the shell, and perhaps two.

You need to implement another system call
After this, run the sample userspace programs `lazyvm1, lazyvm2, lazyvm3` provided. These programs call the system calls and the user space functions implemented in the previous part. The output should match with the sample output in `sample-outputs/afterlazy.txt`.

Submission Guidelines

- All submissions via moodle. Name your submission as: <**rollno_lab6**>**.tar.gz** item The tar should contain the following files in the following directory structure:

  ```
  <rollno_lab6>/
  |___ sysproc.c
  |___ trap.c
  |___ v2p.c
  ```

|___ `proc-size.c`
|___ `vm-block.c`
|___ `lazyvm1.c`
|___ `lazyvm2.c`
|___ `lazyvm3.c`
|___ <other modified files in xv6>

- **Deadline: Monday, 27th August 2018 - 05:00 PM.**