CSE 536: Advanced Operating Systems

Assignment 2: Process Memory Management in xv6

Deadline: 11:59 PM on October 5, 2023

(100 points) + (20 bonus points)

1 Brief Description

In this assignment, you will implement on-demand paging and copy-on-write using page faults. This requires you to change how the xv6 OS currently statically allocates pages for a process, write a page fault handler to intercept and load pages on page faults, and swap pages to disk if the system is out of memory.

GitHub Repo. Please clone as follows:

git clone -b lab2-memory https://github.com/ASTERISC-Teaching/cse536-release.git

Deliverables. You will submit your **updated code** (in an archived file) and a **PDF document** containing answers to all questions asked in this handout. The code is worth 90 points, the PDF is worth 10 points.

Important note

- **If your assignment does not compile or your upload is corrupted, you will get a zero.** Always double-check your submission.
- **We provide some test-cases, but not all of them**. Please test your code rigorously.
- **Submit a zip file titled by your ASU username**. For instance, if your username is adil, your file should be adil.zip.

Suggested reading(s)

xv6: A Simple, Unix-like Teaching Operating System, Russ Cox, Franz Koshoek, and Robbert Morris.

2 Breakdown of Tasks in this Assignment

2.1 Enable On-Demand Binary Loading (5 points)

At process creation, the xv6 OS statically loads all process regions (e.g., text) located within the program's binary. In this task, our goal is to **set-up on-demand loading of a program binary's contents.**

A process is created using the `exec()` call in xv6 (located in `kernel/exec.c` file). The kernel keeps all relevant information about a process within its `proc` data structure (defined in `kernel/proc.h`).

Please follow the steps below (in `exec()`) to complete this task:

1. **Determine process on-demand status.** There are two processes in xv6 that are always executed, namely `init` and `sh`; hence, they can be statically allocated. We have defined a new boolean within `proc` and called it `ondemand`. Use it to track on-demand processes (i.e., all except for `init` and `sh`).

2. **Skip program section loading.** For all on-demand processes, modify the program section iterator (the *for* loop after reading the binary's ELF header in `exec()`) and stop xv6 from allocating physical pages for the program's sections or loading the sections from disk.

   Hints: `uvmalloc` and `loadseg` are important function calls for this task. Also, pay close attention to the `sz` variable and how it is used to decide virtual addresses of the process' stack region.

3. **Update print statements.** These are defined in `kernel/debug.c`.

   - If a process is on-demand, call `print_ondemand_proc()`, which accepts the process name.
   - Whenever you skip loading a program section, call `print_skip_section()`. This function accepts: (1) the process' name, (2) the section's virtual address, and (3) the section's size.

Testcases. You can test the correctness of your implementation within xv6's shell script as follows:

- **Test #1.** Run `make qemu`. This should work correctly since your `init` and `sh` processes are statically allocated. Hence, you should see the shell ($..) printed on the console.
- **Test #2.** Within the xv6 shell, run `ls` and compare its output with `output/test2-ls`.

Question-1

(6 points)

(a) What is the purpose of the `init` and `sh` processes in xv6?

(b) What is the role of `uvmalloc` and `loadseg` during process creation?

Important note

Please take a close look at `exec()`. If you understand how it works, the next task will become easy.

2.2 Design a Page Fault Handler (25 points)

At Test #2, the `ls` process incurs a page fault while accessing memory that is not loaded. Since the kernel does not handle page faults, it kills the process. Hence, our goal is now to **setup a page fault handler to load program binary contents on-demand.**

Suggested reading(s)

Lecture #7: Adding page fault handling to xv6's exception handler

Please follow the steps below to complete this task:

1. **Redirect page fault exceptions.** All exceptions during a process' execution are received by `usertrap()` (in `kernel/trap.c`). Redirect page fault exceptions to `page_fault_handler()` in `kernel/pfault.c`. Recall that page fault exceptions have specific codes in the `scause` register.

Suggested reading(s)

Lecture #6 Slide #26 : Exception codes in `scause`

2. **Find faulting page address (PF_{addr}).** In `page_fault_handler()`, find PF_{addr} by reading the `stval` register. This register contains the exact byte-level address. For instance, if the fault is at 0x1234 address, it will contain 0x1234. Since memory allocation and loading occurs at the page-level, you must find the base address of the page (0x1000 in this example). *Hint: This can be achieved by right-shifting and left-shifting the page offset-related bits.*

3. **Load program binary page from disk.** We only configured xv6 to avoid loading program binary contents (subsection 2.1). Hence, PF_{addr} must be a page from the program binary that is not yet loaded. In `page_fault_handler()`, perform the following sub-steps to load the required page:

   - Iterate through each program section header (using the binary's ELF).
   - Find the section and offset within the binary that PF_{addr} belongs to.
   - Allocate a free physical page at PF_{addr} using `uvmalloc()`.
   - Load the required page from the program binary using `loadseg`.

   Hints: Look at how `exec()` (`kernel/exec.c`) iterates through the program headers while loading sections into memory.

4. **Update print statements.** In `page_fault_handler()`:

   - Call `print_page_fault()` with the process name and faulting address.
   - Whenever loading a segment from the program binary, call `print_load_seg()` with the faulting address, segment offset, and load size.

Testcases. You can test the correctness of your implementation within xv6's shell as follows:

- **Test #3.** Within the xv6 shell, run `echo Hello World` and compare its output with `output/test3-echo-hw`.
- **Bonus test.** If you run `cat README` in the xv6 shell, it will give you an error even after handling page faults. **If you can address this case (*entirely on your own*), you will get 5 extra points.**

2.3 Enable On-Demand Heap Memory (10 points)

In this task, your goal is to **load heap memory on-demand as well to efficiently use system memory.** To complete this task, follow the steps below:

1. **Avoid heap page allocation.** Heap allocation requests come to `growproc()` (in `kernel/proc.c`). At these requests, prevent the OS from allocating pages for on-demand processes (subsection 2.1).

2. **Track heap pages.** We have allocated a new data structure called `heap_tracker` within `proc`. Use this structure to track of every heap page allocated to the process in `growproc`. At this point, only track the virtual address of each heap page.

3. **Handle heap page fault.** Update the `page_fault_handler()` (subsection 2.2) as follows:

   - Use the `heap_tracker` to determine if PF_{addr} belongs to the heap regions.
   - If PF_{addr} belongs to the heap regions, skip checking the program binary in disk (step-3 in subsection 2.2).
   - Allocate a new physical page and map it to the faulted address.
     Hint: Look at how `growproc` allocates heap pages.

4. **Update print statements.** In `growproc`, call `print_skip_heap_region()` if you skip loading a heap region. This function accepts: (1) the process' name, (2) the heap region's starting address, and (3) the number of heap pages skipped.

Testcases. You can test the correctness of your implementation within xv6's shell as follows:

- **Test #4.** Run `test4-odheap` on the xv6 shell and compare output with `outputs/test4-odheap`.
- **Test #5.** Run `test5-odheap-big` on the xv6 shell and compare output with `outputs/test5-odheap-big`.

2.4 Implement Page Swapping to Disk for Heap Memory (30 points)

System memory is limited and the OS must swap pages to disk (on page faults) if the system is out of free memory. In this task, your goal is to **implement page swapping to disk for the heap memory of a process during a page fault**.

Suggested reading(s)

Lecture #7 Slide #16 : Page Replacement

Each process is allowed to keep only 100 pages of heap regions *resident* in memory. We define this limit as MAXRESHEAP. Hence, if the process exceeds MAXRESHEAP, the OS will automatically swap a page to disk at a page fault. We will implement the First-In-First-Out (FIFO) algorithm for page-swapping to disk.

We have already implemented a space within the disk (fs.img) for you to write your pages that cannot be kept in memory. This is called the page swap area (PSA), and it is located between blocks 32 to 1032 (1000 blocks). These blocks are defined as PSASTART and PSAEND, respectively.

To complete this task, please follow the steps below:

1. **Track heap page load time on page faults.** Use the `read_current_timestamp()` to find the current time when you load a heap page. Then, update the `last_load_time` field within the heap page's corresponding entry in `heap_tracker` with the current time.

2. **Track total in-memory (or resident) heap pages.** Each `proc` has a `resident_heap_pages` field. Update this field whenever `page_fault_handler()` loads a new heap page.

3. **Evict victim page to a free disk region.** If `resident_heap_pages` is equal to MAXRESHEAP and a new heap page must be loaded, `page_fault_handler()` calls `evict_page_to_disk()` to evict a resident heap page to disk. Update the `evict_page_to_disk()` as follows:

- *Find free PSA blocks.* We need a free region of the PSA to store the page. Since each PSA block is 1KB, we need 4 blocks to store a 4KB page. For simplicity, use contiguous blocks (e.g., 0 - 3). Track the starting block of the PSA where the heap page will be swapped to in the `startblock` field of `heap_tracker`. Also, track each used block inside the `psa_tracker` data structure (defined in `kernel/pfault.c`) to avoid overwriting used blocks on subsequent disk writes.
- *Find a victim page.* Since your policy is FIFO, find the heap page that was loaded the longest time ago using `heap_tracker`.
- *Copy victim page to kernel memory.* While the heap page is mapped to a process' address space, it is not currently mapped to the xv6 kernel. Hence, we must copy the victim page from user address space to the kernel's using the `copyin()` function. This function accepts the process' L2 page table address (stored in `proc`), a kernel page, a user virtual address, and size.
  Hint: You can allocate a kernel page using `kalloc()`.
- *Write victim page to the chosen PSA blocks.* This requires using block I/O functions, namely `bread()`, `bwrite()`, and `brelse()`. A template is provided in `evict_page_to_disk()`.
- *Unmap the victim page.* Once a page is swapped to disk, unmap its entry from the user's address space using the `uvmunmap` function. This ensures there will be a page fault if the page is accessed.

4. **Retrieve a swapped page from disk.** On a subsequent page fault, the OS might have to retrieve a heap page that was swapped to disk. Write this code in `retrieve_page_from_disk()`.

   - Use a heap page's `heap_tracker` entry to check if it is stored in disk. If yes, find its block locations on the disk using the entry's `startblock`.
   - Read this page from disk using `bread()` onto a kernel page.
   - Copy from kernel to user page using `copyout()`.

5. **Update print statements.** `print_evict_page` and `print_retrieve_page` with their correct arguments. For each, the first argument should be the process virtual address being evicted or retrieved, and the second argument should be the starting block number in disk.

Testcases. You can test the correctness of your implementation within xv6's shell as follows:

- **Test #6.** Run `test6-pswap1` on the xv6 shell and compare output with `outputs/test6-pswap1`.
- **Test #7.** Run `test7-pswap2` on the xv6 shell and compare output with `outputs/test7-pswap2`.

Question-2

(4 points)

(a) Can you describe how the page swap area is reserved within the filesystem image by inspecting mkfs.c?

2.5 BONUS: Implement the Working Set Algorithm (15 points)

We have discussed the working set algorithm (WSA) in class. Replace the FIFO page swap algorithm with WSA in subsection 2.4. Test your algorithm robustly and also upload your testcases.

2.6 Implement Copy-On-Write (CoW) during Fork (20 points)

The `fork()` system call clones the current process (called the parent) into a new process (called the child) from the position where the parent executes `fork()`. The fork system call duplicates the entire memory of the parent process, resulting in separate, identical memory spaces for the parent and child processes. This system call is already implemented by xv6. Your goal is to **extend fork() with the copy-on-write (CoW) optimization discussed in class**.

In CoW-optimized fork, all the physical pages of the parent are mapped as *Read-Only (RO)* in both the parent and the child process's page tables, at fork-time. When either the parent or the child writes to one of these shared RO pages, the OS (1) allocates and maps a new page (with write permission) for the writing process and (2) copies contents from the shared page to this new page.

A CoW forked process can also fork() more CoW-enabled processes. All of these processes are said to belong to the same CoW group. On the other hand, a CoW-enabled forked process, can also do a normal (non-CoW) fork, where the child process will not share memory with its parent.

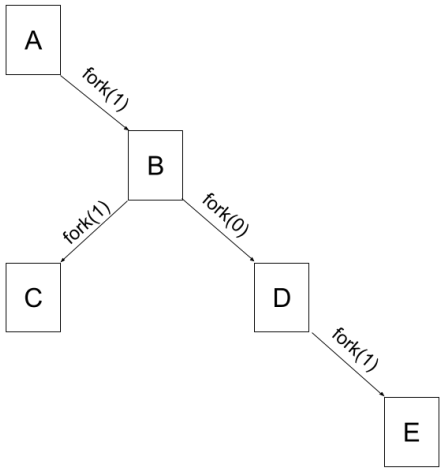

Figure 1: Processes A, B & C belong to the same CoW group, whereas D & E belong to another

Modifying fork() for CoW. The fork() system call has been modified to accept an integer. A fork(0) will denote a regular fork(), whereas a fork(1) will denote a CoW fork. Your task is to modify fork() to handle the copy-on-write case.

1. Implement the `uvmcopy_cow()` function (defined in cow.c) to map the parent process's memory, in both the parent and the child, to be Read-Only.
   Hint: Check how `uvmcopy()` is implemented in xv6.

2. Call `uvmcopy_cow()` in fork() and also set the metadata (`cow_group` and `cow_enabled`) on the proc structure to track processes belonging to the same group. For simplicity, you can use the process ID (pid) of the parent process of a group as the group ID.

3. Track the shared pages in a CoW group. You will have to make sure that the memory shared by the

processes in a CoW group, is freed only once, when the last process belonging to the group exits. (Note: Otherwise it would lead to a double free, and affect with memory allocator in kalloc.c)

Enabling CoW in the page fault handler. When a process belonging to a CoW group tries to write to a shared page, it will trigger a write page fault. On this fault, your task is to:

1. Allocate a new page for the faulted process.
2. Copy the contents of memory from the shared page to the newly allocated page.
3. Map the newly allocated page in the faulted process' page table, with *Write* permission.

Implement the above mentioned functionality in the `copy_on_write()` function provided in cow.c, and call it appropriately to handle the page fault.

Adding debug statement. Add the `print_copy_on_write()` function in the `copy_on_write()` function, and provide a pointer to the writing process and the faulting virtual address as the arguments.

Important note

The following helper functions are provided in cow.c for tracking shared memory regions.

1. **add_shmem(group, pa):** Add a physical address (pa) to the `shmem` array entry of the specified cow_group (group). It does not the address if it is already present.
2. **is_shmem(group, pa):** Checks if a physical address (pa) is present in `shmem` array at the specified cow_group (group); returns 1 if true, and 0 otherwise.

Testcases. You can test the correctness of your implementation within xv6's shell as follows:

- **Test #8.** Run `test8-cow1` on the xv6 shell and compare output with `outputs/test8-cow1`.
- **Test #9.** Run `test9-cow2` on the xv6 shell and compare output with `outputs/test9-cow2`.

3 Submitting your Assignment

1. Please zip the entire provided code directory and submit it to the Canvas under "Assignment 2: CODE".
2. Please create a PDF document and upload it to canvas under "Assignment 2: DOCUMENT".