Operating Systems 2
Programming Assignment 6: Paging

Rajiv Shailesh Chitale
CS21BTECH11051

This assignment implements optimizations for paging in xv6.
In trap.c, a switch-case statement checks the trap number to determine if a page fault occurred. If so, it calls pgflt_handler(). The address which caused the fault is read from the cr2 register. Using walkpgdir(), the corresponding page table entry is read. If the page table doesn't exist or the entry is invalid, then it is due to demand paging. If the entry is valid but read-only (write bit is 0), it will lead to copy-on-write.

Part-1: Demand Paging
1) In the function exec(), pages are allocated and the file is loaded for the amount of memory equal to filesz. This is done using allocuvm() and loaduvm(). This amount does not include the size of uninitialized data. But the size of the process, sz, is increased to handle memsize, which includes the uninitialized data along with read only text. This is done to provide space for future demand pages before the guard and stack pages.

2) When the corresponding trap due to invalid page occurs:
kalloc() is used to allocate kernel memory for a new frame. It is zeroed out using memset(). It is assigned to a user page using mappages(). This page corresponds to the faulting address.

Part-2: Copy-On-Write
In the original fork() function, copyuvm() was used to copy the pages and frames from parent to child. In the updated code for copy-on-write, these two functionalities are split into two functions- copy_pages() and copy_frame().

copy_pages() is called by fork() and enables sharing of frames. It returns a new page directory for the child process created using setupkvm(). It walks through the parent's valid page table entries and sets write flags to 0 (making pages read-only). Then it uses memmap() to create page table entries for the child pointing to the parent's frames. Fork() flushes TLB with lcr3().

copy_frame() is only called when a trap occurs from trying to write to a read-only page. It uses kalloc() to allocate kernel memory, memmove() to copy data from parent's frames, and mappages() to map the page to the new frame. It also sets the write flag for the page to 1.

To handle deallocation in the case of multiple forks, an array readers[] is set up to maintain the count of pages referring to a frame. Its size is PHYSTOP/PGSIZE which is the total number of frames. The number of readers of a frame is incremented within kalloc() and decremented within kfree(). A frame is only added to the freelist when the count of readers drops to 0.

A wrapper function add_readers() is also provided to increment, decrement and read the value outside. This uses locks for concurrency. This is used in copy_frame(). When only 1 reader remains, the pages are automatically converted to writable.

Observations:
I have also created an additional command ‘set’ to help with observations and debugging.

set i 1	prints *information* about demand paging, copy on write and forks
set d 0	switches off *demand* paging. (set d 1 will switch it back on)
set s 1	will print *syscall* names (except write). (set s 0 will switch it off)

mydemandPage:
Observations with N = 5000:

```
global addr from user space: B20
pgdir entry num:0, Pgt entry num: 0, Virtual address: 0x0, Physical address: 0xdee2000, W-bit: 1
pgdir entry num:0, Pgt entry num: 1, Virtual address: 0x1000, Physical address: 0xdfbc000, W-bit: 1
pgdir entry num:0, Pgt entry num: 7, Virtual address: 0x7000, Physical address: 0xdedf000, W-bit: 1
pgdir entry num:0, Pgt entry num: 0, Virtual address: 0x0, Physical address: 0xdee2000, W-bit: 1
pgdir entry num:0, Pgt entry num: 1, Virtual address: 0x1000, Physical address: 0xdfbc000, W-bit: 1
pgdir entry num:0, Pgt entry num: 2, Virtual address: 0x2000, Physical address: 0xdfbd000, W-bit: 1
pgdir entry num:0, Pgt entry num: 7, Virtual address: 0x7000, Physical address: 0xdedf000, W-bit: 1
pgdir entry num:0, Pgt entry num: 0, Virtual address: 0x0, Physical address: 0xdee2000, W-bit: 1
pgdir entry num:0, Pgt entry num: 1, Virtual address: 0x1000, Physical address: 0xdfbc000, W-bit: 1
pgdir entry num:0, Pgt entry num: 2, Virtual address: 0x2000, Physical address: 0xdfbd000, W-bit: 1
pgdir entry num:0, Pgt entry num: 3, Virtual address: 0x3000, Physical address: 0xdfbf000, W-bit: 1
pgdir entry num:0, Pgt entry num: 7, Virtual address: 0x7000, Physical address: 0xdedf000, W-bit: 1
pgdir entry num:0, Pgt entry num: 0, Virtual address: 0x0, Physical address: 0xdee2000, W-bit: 1
pgdir entry num:0, Pgt entry num: 1, Virtual address: 0x1000, Physical address: 0xdfbc000, W-bit: 1
pgdir entry num:0, Pgt entry num: 2, Virtual address: 0x2000, Physical address: 0xdfbd000, W-bit: 1
pgdir entry num:0, Pgt entry num: 3, Virtual address: 0x3000, Physical address: 0xdfbf000, W-bit: 1
pgdir entry num:0, Pgt entry num: 4, Virtual address: 0x4000, Physical address: 0xdfc0000, W-bit: 1
pgdir entry num:0, Pgt entry num: 7, Virtual address: 0x7000, Physical address: 0xdedf000, W-bit: 1
Printing final page table:
pgdir entry num:0, Pgt entry num: 0, Virtual address: 0x0, Physical address: 0xdee2000, W-bit: 1
pgdir entry num:0, Pgt entry num: 1, Virtual address: 0x1000, Physical address: 0xdfbc000, W-bit: 1
pgdir entry num:0, Pgt entry num: 2, Virtual address: 0x2000, Physical address: 0xdfbd000, W-bit: 1
pgdir entry num:0, Pgt entry num: 3, Virtual address: 0x3000, Physical address: 0xdfbf000, W-bit: 1
pgdir entry num:0, Pgt entry num: 4, Virtual address: 0x4000, Physical address: 0xdfc0000, W-bit: 1
pgdir entry num:0, Pgt entry num: 5, Virtual address: 0x5000, Physical address: 0xdfc1000, W-bit: 1
pgdir entry num:0, Pgt entry num: 7, Virtual address: 0x7000, Physical address: 0xdedf000, W-bit: 1
Value: 2
```

Initially the pages with entry numbers 0, 1 and 7 are loaded. As further pages are demanded, the pages are allocated with entry numbers 2, 3, 4 and 5 in that order. This order corresponds to the linear access pattern of the data in the program. Notice, that entry 6 is the guard page and entry 7 is the stack.

For N= 3000, the stack was at 5
For N= 10000, the stack was entry number 12

myCOW:
This program allocates data to a process from indices 0 to 2N. After the fork, the parent writes from N to 3N, whereas the child process writes from 0 to N. Observations from myCOW:

For N=3000, Initial pages are demanded and allocated to the parent till entry 6 (entry 11 is stack). All the write bits are 1. After the fork, the write bits are set to 0. The only bits which remain 1 correspond to entry numbers 0, 3, 11. This is because the child accessing the start of the array (entry 0), and the parent accessing the Nth element (entry 3) cause copy-on-write.

Entry 11 is the stack. The parent continues to access elements in a linear fashion, causing more copy-on-writes. There are also changes in physical addresses upon copy-on-write.

Learnings

- ➔ I learnt about the ELF format and its segments, and understood the difference between memsz and filesz.
- ➔ I learnt how the functions memmove() and memmap() are implemented and used to copy data and map frames.
- ➔ I got a better idea about the different kinds of traps. And I learnt how to distinguish between demand paging and copy-on-write which result in the same page fault.
- ➔ I realized how to implement COW in the case of multiple forks, using a count of readers.
- ➔ I saw and used locks to maintain concurrency in kernel data structures. I understood the implementation of kfree() and kalloc() with the free list.
- ➔ I learnt about the cr2 register for finding the faulting address, and lcr3() function to flush the TLB.
- ➔ I got comfortable with working with a large number of files, with source files, header files and extern variables.
- ➔ While debugging, I also went through and understood parts of sh, fork, wait, exit and sleep.