CS692_Lab[4]

1: Page walk

The first piece needed to implement memory checkpointing is going through each PTE. To do this, I must do the following:

```
for each vma
     for each pgd
          for each pud
               for each pmd
                    for each pte
                         //do
```

I found a kernel function `walk_page_range()` in `mm/pagewalk.c` which does this page walking for me. I just have to pass it a start and end address and a `struct mm_walk` which has pointers to my callback functions. I set my callback function for `walk->pte_entry`, so, at every valid pte in the given range, my function is called (one can set call back for every level if they want). This function only walks thought valid entries.

2: Write to file

For this I used `filp_open` and `vfs_write` kernel utility functions.

3: Full checkpoint

Implementing full checkpoint was straight forward. I just have to check if the page is not a zero page with `pte_page(*pte) == PAGE_ZERO(0)` and then write the page contents to the file.

3: Incremental checkpoint

For this, the only way to implement is by using the pte bits managed by MMU. But, I cannot simple use an unused bit to OR it with _PAGE_BIT_DIRTY inside the MMU function that sets the dirty bit. Because apparently MMU somehow keeps track of clean pages and only calls the function (that sets the dirty bit) only the first time the page is written to.

Then I looked at the _PAGE_BIT_SOFT_DIRTY bit. Linux has this bit exactly for this checkpointing purpose. It is used to track if a page was written to since the last reset. Resetting this is done by write protecting the page (marking the page as read only). This will cause a page fault when ever the page is written to again so MMU will set the soft dirty bit again. The page is already mapped so there won't be a major overhead. The actual dirty bit is not touched.

This soft dirty bit resetting and tracking is already implemented in Linux which the user can access from proc to monitor page writes. So, looked at `fs/proc/task_mmu.c` to implement this mechanism in incremental checkpoint function.

To do incremental checkpointing, I am resetting the soft bit every time the page is saved during checkpointing. For the consecutive calls, I can just check this bit and only save the pages that have soft bit set and then clear it again. To implement this, I have to first call mmu function `mmu_notifier_invalidate_range_start()` before starting the page walk (to mark these ptes as invaid) and `mmu_notifier_invalidate_range_end()` once everything is done (to make them as valid again). I pass the start and end addresses given by user to them. Inside page walk, I reset the soft dirty bit and mark the page as read only with and `pte_clear_soft_dirty()` and `pte_wrprotect()`. This will cause a page fault when the page is next modified and MMU also sets soft bit again. Finally, I also called `flush_tlb_mm()` to flush the TLB entries of this process. If I don't call mmu notifier and flush TLB, the CPU might access cache and TLB which contains old information.

4: syscall usage

My implementation of inc_cp_range takes 3 arguments. First 2 are start and end address (end address is 1 byte after the last address) and a third count argument. The name of the fill will have this value. If the count is 0, it will take a full checkpoint regardless if the page is dirty.

5: Example

Here is an example I am allocating 4 pages worth of data on the heap and then calling the inc_cp_range. Then modified a byte and called the syscall again.

```
ubuntu@lab4:~/test$ ./a.out
start addr - 0x92d010
ubuntu@lab4:~/test$ ls -lh
total 44K
-rwxrwxr-x 1 ubuntu ubuntu 8.6K Apr  5 15:27 a.out
-rw-rw-r-- 1 ubuntu ubuntu  20K Apr  5 15:28 cp_0
-rw-rw-r-- 1 ubuntu ubuntu 4.0K Apr  5 15:28 cp_1
-rw-rw-r-- 1 ubuntu ubuntu  568 Apr  5 15:27 test.c
-rw-rw-r-- 1 ubuntu ubuntu  268 Apr  5 14:07 test2.c
ubuntu@lab4:~/test$
```

The second checkpoint (cp_1) is only 1 page size.

Another example where I did not modify any data.

```
ubuntu@lab4:~/test$ ./a.out
start addr - 0x18e5010
ubuntu@lab4:~/test$ ls -lh
total 40K
-rwxrwxr-x 1 ubuntu ubuntu 8.6K Apr  5 15:29 a.out
-rw-rw-r-- 1 ubuntu ubuntu  20K Apr  5 15:29 cp_0
-rw-rw-r-- 1 ubuntu ubuntu    0 Apr  5 15:29 cp_1
-rw-rw-r-- 1 ubuntu ubuntu  570 Apr  5 15:28 test.c
-rw-rw-r-- 1 ubuntu ubuntu  268 Apr  5 14:07 test2.c
ubuntu@lab4:~/test$
```

The second checkpoint (cp_1) is 0kb.

Note: The file size is 5 * PAGE SIZE because the start address is not aligned.

I included 2 test programs test1 and test2. Test 1 allocates 4 pages of int values. Calls the syscall twice and modifies 1 byte of data before 2nd call. Test 2 takes a full checkpoint 0x0 to 0xffffffffffff (48bit).