OS Lab – 3

Madhav Bhardwaj (CS23B035)

August 2025

Problem 1

```c
// helper function for printing entry
void vmprinthelper(int level, pagetable_t pt)
{
    for (int index = 0; index < 512; index++)
    {
        pte_t pte = pt[index];
        if (PTE_V & pte)  // check if valid
        {
            for (int i = 0; i < level; i++)
            {
                printf(".. ");
            }
            printf("%d: pte %p pa %p\n", index, (void *)pte, (void *)PTE2PA(pte));

            if (!PTE_LEAF(pte) && level <= 2)
                vmprinthelper(level + 1, (pagetable_t)PTE2PA(pte));
        }
    }
}

void vmprint(pagetable_t pagetable)
{
    printf("page table %p\n", pagetable);
    vmprinthelper(1, pagetable);
}
```

Explanation

- The above code is present in vm.c
- The RISC-V page table has 3 levels, with up to 512 entries per level (Sv39).

- The function `vmprinthelper()` iterates through all entries.
- If a page table entry is valid (`PTE_V` bit set), it is printed.
- Indentation using "`..`" visually represents the depth in the hierarchy.
- If the entry is not a leaf (i.e., points to another page table) which is identified by the protection bits, recursion continues.
- Leaf entries (actual virtual-to-physical mappings) terminate recursion.

Problem 2

- Added an entry in proc.h to store the address of the physical page:

  ```
  struct usyscall *pid_page;
  ```

- for every process there is a user space mapping of this pid_page. this is achieved first allocating a page to store pid and then map this page in the user space below the trapframe.
- In proc.c, when a process is initialized `allocproc` is called. hence creating a page in this function and then map it in `proc_pagetable`.

  ```
  p->pid_page = (struct usyscall *)kalloc();
  if (p->pid_page == 0) {
    freeproc(p);
    release(&p->lock);
    return 0;
  }
  ```

  After allocation the `if` block checks if the page is allocated successfully. if it is not, then cleanup is required. `freeproc` is called to free all the resources of that process. Unlocks the process lock, since `allocproc()` (where this code is written) holds the process lock while setting up.

- later in `proc_pagetable` function, the following lines are added. Here the actual mapping and the assignment is done.

  ```
  if (mappages(pagetable, USYSCALL, PGSIZE, (uint64)(p->pid_page), PTE_R | PTE_U) < 0) {
    uvmunmap(pagetable, USYSCALL, 1, 0);
    uvmfree(pagetable, 0);
    return 0;
  }
  (p->pid_page)->pid = p->pid;
  ```

  `mappages` call sets up a mapping in the process's page table.If mapping fails, the kernel must undo any partial setup. `uvmunmap` Removes the virtual-to-physical mapping at address `USYSCALL`. `uvmfree` function frees the entire page table of this process.

- in `proc_freepagetable` function, the following line is added to remove the mapping.

```
uvmunmap(pagetable, USYSCALL, 1, 0);
```

Also, in `freeproc` function, while freeing a process, the page must be freed.

```
if(p->pid_page)
    kfree((void*)p->pid_page);
p->pid_page = 0;
```

Problem 3

- As with any system call in `xv6`, the following steps were carried out:

  1. In `user.h`, the function prototype for `pgaccess` was added so that user programs can invoke it.
  2. In `usys.pl`, an entry was added so that the user-level stub is generated.
  3. In `syscall.h`, a new syscall number was defined for `SYS_pgaccess`.
  4. In `syscall.c`, the syscall table was updated to map the syscall number to the kernel handler function.
  5. Finally, the actual implementation was placed in `sysproc.c`.

```
uint64
sys_pgaccess(void)
{
  uint64 base;
  int num_of_pages;
  uint64 mask;
  argaddr(0, &base);
  argint(1, &num_of_pages);
  argaddr(2, &mask);
  uint64 temp_mask = 0;

  pagetable_t current_pagetable = myproc()->pagetable;
  for (int i = 0; i < num_of_pages; i++) {
    pte_t *pte = walk(current_pagetable, base + i * PGSIZE, 0);
    if (pte == 0)
      return -1;
    temp_mask |= (*pte & PTE_A) ? (1UL << i) : 0;
    *pte &= (~PTE_A);
  }
  copyout(current_pagetable, mask, (char *)&temp_mask, sizeof(temp_mask));
  return 0;
}
```

- The logic of this function is:

  1. It first extracts the system call arguments: the starting virtual address (`base`), the number of pages to check, and the user-space address where the result mask should be written.
  2. It walks through the process's page table for each page in the range, checks if the `Accessed` (`PTE_A`) bit is set, and accordingly sets the corresponding bit in a temporary mask.

3. After recording, it clears the `PTE_A` bit, so that future accesses can be detected afresh.
4. Finally, it copies the mask back to user space using `copyout`.