CS 140 Lab Report 5

Dale Sealtiel T. Flores
2023-11373
THX/WXY

1. QEMU maps physical address `0x100000` to the write register of a virtual shutdown I/O device.Writing the magic value `0x5555` to this register causes QEMU to shut down.
   Using the code in `sys_shutdown` of `kernel/sysproc.c` as a guide, modify the xv6 `exec` syscall such that the virtual page containing virtual address `0x100000` of all user processes calling exec would be mapped to the physical page containing physical address `0x100000` with reads and writes allowed in user mode—this should enable a user process that executes `(*(volatile uint32 *) 0x100000) = 0x5555;` to shut QEMU down. Ensure that user processes are able to terminate gracefully—you may need to modify `proc freepagetable` in `kernel/proc.c` for this. Show all relevant changes made (with corresponding filenames) via code screenshots or snippets, and briefly describe what each change does. Ensure that your changes are properly committed and pushed to your Github Classroom repository. Include the user program and `Makefile` changes you used to test your implementation.

   Answer:
   This change modifies the `exec` function in `exec.c` to map the virtual address and physical address to `0x100000`.

   ```
   if(mappages(pagetable, 0x100000, PGSIZE, 0x100000, PTE_R | PTE_W | PTE_U) < 0) goto bad;
   ```

   Figure 1: `exec()` change in `exec.c`

   This change is important to ensure shutdown gets unmapped.

   ```
   // physical memory it refers to.
   void
   proc_freepagetable(pagetable_t pagetable, uint64 sz)
   {
     uvmunmap(pagetable, TRAMPOLINE, 1, 0);
     uvmunmap(pagetable, TRAPFRAME, 1, 0);
     uvmunmap(pagetable, 0x100000, 1, 0);
     uvmfree(pagetable, sz);
   }
   ```

   Figure 2: `proc_freepagetable` change in `proc.c`

2. Explain in detail how xv6 ensures that accessing the `kernel_satp` field of the `trapframe` field of the active PCB results in the correct memory location despite the user page table still being in use.

Answer: xv6 ensures that kernel_satp is accessed correctly by mapping TRAPFRAME in both user and kernel page tables at the same address. This makes it that when a trap occurs, it guarantees that the trapframe is accessible under the user page table and is still accessible even if you switch to the kernel page table.

3. Explain how uservec is still able to execute sfence.vma zero, zero right after csrw satp, t1 despite the page table changing after csrw satp, t1 is executed.

   Answer: It still able to execute sfence.vma zero, zero because xv6 maps trampoline at the same address in both user and kernel page tables. This makes sure that the CPU never loses access to it even after switching page tables.

4. exec sets a stack guard page when setting up the user page table. Explain what a stack guard page is for, and how exactly exec allocates the said page for it to serve as one. Cite all references used.

   Answer: A stack guard page is an unmapped page placed below the user stack to catch whenever a stack overflow occurs. Whenever a a stack overflow occurs, the guard page will trigger a page-fault exception to avoid overwriting other memory.
   When exec allocates and initializes a user stack. It just allocates just one stack page then it places an inaccessible page just below the stack page to serve as the guard page. (pp 40-41)
   Reference: https://pdos.csail.mit.edu/6.S081/2022/xv6/book-riscv-rev3.pdf

5. Create a system call vaspace with syscall number 23 that prints the base addresses of all valid virtual pages of the invoking process along with their read, write, execute, and user permissions. Check until just before MAXVA. Include a corresponding user mode wrapper function. For each valid virtual page, a line formatted as <b>: <r><w><x><u> should be printed in increasing order of virtual addresses where each token is as follows (sample line is 0x0: RWX-):
   - <b>: Base virtual address of virtual page in hex with 0x prefix, no leading zeros
   - <r>: R if the virtual page has read permissions, or - otherwise
   - <w>: W if the virtual page has write permissions, or - otherwise
   - <x>: X if the virtual page has execute permissions, or - otherwise
   - <u>: U if the virtual page has its user bit set, or - otherwise

   Additionally, create user/vaspace.c with the code in Code Block 1, run it (ensuring your change for Item 1 is present), and take a screenshot of the output. For each page in the output, describe what the purpose of the page is and justify your answer with the permission bits set, other lines of the output, and, if necessary, additional information from the xv6 codebase.
   Show all relevant changes made (with corresponding filenames) via code screenshots or snippets, and briefly describe what each change does. Ensure that your changes are properly committed and pushed to your Github Classroom repository.

   This change adds the vaspace syscall and assign it to number 23

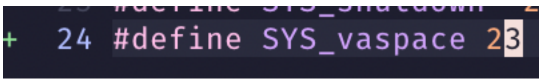

Figure 3: syscall.h change

These changes add the prototype and adds the array mapping for the vaspace syscall

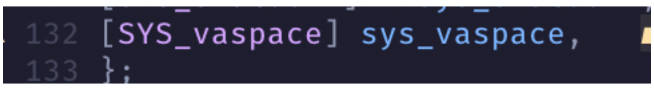

Figure 4: syscall.c change 1

```c
extern uint64 sys_vaspace(void);
```

Figure 5: `syscall.c` change

This change implements the `vaspace` syscall

```c
uint64
sys_vaspace(void)
{
  struct proc *p = myproc();
  for(uint64 va = 0; va < MAXVA; va += PGSIZE){
    pte_t *pte = walk(p->pagetable, va, 0);
    if(pte && (*pte & PTE_V)){
      char r = (*pte & PTE_R) ? 'R' : '-';
      char w = (*pte & PTE_W) ? 'W' : '-';
      char x = (*pte & PTE_X) ? 'X' : '-';
      char u = (*pte & PTE_U) ? 'U' : '-';
      printf("0x%lx: %c%c%c%c\n", va, r, w, x, u);
    }
  }
  return 0;
}
```

Figure 6: `sysproc.c` change

These changes implement a wrapper function for the `vaspace` syscall

```perl
entry("vaspace");
```

Figure 7: `usys.pl` change

```c
+ int vaspace(void);
```

Figure 8: `user.h` change

This change adds the user program to test the `vaspace` syscall given by Code Block 1 in the specs.

```makefile
$U/_vaspace\
```

Figure 9: `MAKEFILE` change

This is the output of the `vaspace` syscall

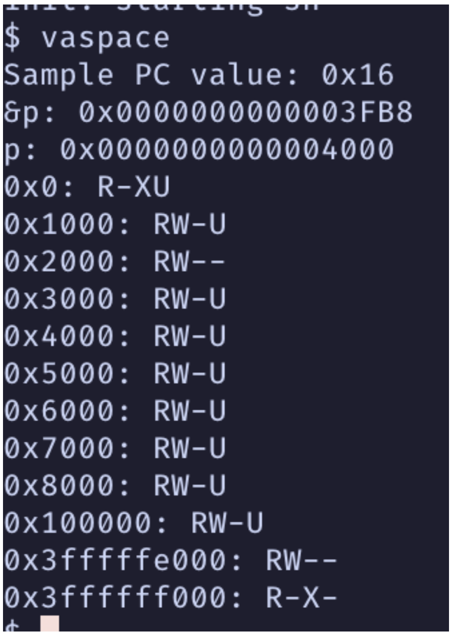

Figure 10: Output of `vaspace` syscall

6. Explain how xv6 page fault handling stays consistent with cumulative memory allocations via `sbrk` and how the handling already accommodates stack memory accesses.

> **Answer:** By analyzing `sys_sbrk` in `sysproc.c`, we can see from the comments that `sbrk` lazily increases the size of the memory but doesn't allocate memory. If the process uses the memory, `vmfault()` will allocate it. This keeps the memory allocations consistent.

7. The `fork` syscall is able to copy the data of parent to the child via `uvmcopy`. Explain what each line of `uvmcopy` does and what it is for. Exclude lines containing only braces.

Answer:
This declares the variables used in the function `uvmcopy`. pte is the page table entry, pa is the physical address, i is a loop counter, and flags are the permission bits of the page.

```
int
uvmcopy(pagetable_t old, pagetable_t new, uint64 sz)
{
  pte_t *pte;
  uint64 pa, i;
  uint flags;
  char *mem;
```

Figure 11: declaration of variables

This loop iterates through each page in the parent's page table. It starts from the first page and goes up to the size of the parent's memory, incrementing by the page size each iteration. During the loop, it checks if the PTE is valid, if it is not valid, it goes to the next page. If it is valid, it gets the physical address (pa) and the permission bits (flags) of the page. If it fails it goes to `err` which will be in the next figure. If it is successful, it will go to `memmove` which copies the contents of parent's page to the child's page. If it is successful, it uvmcopy returns 0. If it fails, it goes to `err`.

```
for(i = 0; i < sz; i += PGSIZE){
  if((pte = walk(old, i, 0)) == 0)
    continue;   // page table entry hasn't been allocated
  if((*pte & PTE_V) == 0)
    continue;   // physical page hasn't been allocated
  pa = PTE2PA(*pte);
  flags = PTE_FLAGS(*pte);
  if((mem = kalloc()) == 0)
    goto err;
  memmove(mem, (char*)pa, PGSIZE);
  if(mappages(new, i, PGSIZE, (uint64)mem, flags) != 0){
    kfree(mem);
    goto err;
  }
}
return 0;
```

Figure 12: for loop that iterates through each page in the parent's page table

This will unmap all the pages removing all allocations done by uvmcopy and then returns -1.

```
err:
  uvmunmap(new, 0, i / PGSIZE, 1);
  return -1;
```

Figure 13: gets the page table entry of the current page