Ka Tam (A20374415)
Jack Critzer (A20396230)
CS 450
4/5/2020
Programming Assignment 3

Part 2 System Calls

myV2p(uint va, int w) System Call

The design of myV2p( ) contains 3 parts, passing the parameters from the user space to the kernel space, going through the paging mechanism to retrieve the physical address, and returning the value into the user space for print out. The parameter for this system call is the unsigned int of virtual address (int only goes up to 2^16) and the write bit (0 for read, 1 for write).

To pass those arguments into the kernel space, there needs to be some modifications to the argint function because we are passing in an unsigned int. The modification is being implemented in syscall.c. Instead of getting the int from addr, we modified it to be uint. This allows the virtual address to be passed into the kernel successfully. Below is a code snippet for this functionality.

```
28  int
29  fetchuint(uint addr, uint *up)
30  {
31    struct proc *curproc = myproc();
32
33    if(addr >= curproc->sz || addr+4 > curproc->sz)
34      return -1;
35    *up = *(uint*)(addr);
36    return 0;
37  }
```

```
66  int
67  arguint(int n, uint *up)
68  {
69    return fetchuint((myproc()->tf->esp) + 4 + 4*n, up);
70  }
```

After successfully getting the argument, we can perform manipulations to the virtual address to get the physical address. The myV2p function happens in proc.c. The code snippet below demonstrates the function getting the current process from the operating system.

```
202  long myV2p(uint va, int w)
203  {
204    // get the current process's page directory
205    struct proc *curproc = myproc();
206    pde_t *pde;
207    pte_t *pgtab;
208
209    // print virtual address
210    cprintf("\n\nVirtual Address: %p\n", va);
```

The next part is the split the virtual address into offset, table index, and directory index. The offset is the right most 12 bits of the virtual address. The table index is the 10 bits to the left of offset. The directory index is the left most 10 bits of the virtual address. The page directory table can be obtained from the current process. The page directory entry is the entry at the page directory table of index, directory index, extracted from the virtual address.

```
212   // seperate virtual address into:
213   // 10 bit page directory index
214   // 10 bit page table index
215   // 12 bit offset
216   uint offset = ((va)&0xFFF);
217   uint table = PTX(va);
218   uint dir = PDX(va);
219
220   pde = &curproc->pgdir[dir];
```

The page table can be obtained from the page directory entry by taking the left 20 bits with PTE_ADDR( ) function. The 20 bit address is a physical address in the memory. But in order to be able to access it, we need to change that physical address to the corresponding virtual address with the function P2V( ).

```
222   // print page directory entry
223   cprintf("Page Directory: %x\n", *pde);
224
225   // get the page table from the page directory entry
226   pgtab = (pte_t *)P2V(PTE_ADDR(*pde));
227
228   // print page table entry
229   cprintf("Page Table: %x\n", pgtab[table]);
```

The next step is to get the corresponding page table entry from the page table. This can be done by accessing the element in the page table pointed by the table index extracted from the virtual address. Then the address can be obtained by shifting the bits right ward by 12; we will get the PPN by this operation. The physical address is obtained by combining the PPN and the offset from the virtual address.

```
231   // seperate page table entry:
232   // 20 bit ppn
233   // 12 bit flag
234   uint ppn = pgtab[table] >> 12;
235
236   // combine ppn and offset to make physical address
237   uint pa = ppn << 12;
238   pa += offset;
```

Lastly, we have to check if the address is readable or writable. This can be done by comparing the flag portion of the page table entry with the input operation. If the input operation is write but the flag says not writable, then we can return -1 as error. The same can be done for the addresses

that are not present. If the virtual address has no problem, then the physical address will be returned. Otherwise, there are some errors, which will return -1.

```
240   // check if the page table entry is writable
241   if (w && !(pgtab[table] & PTE_W)) {
242     return -1;
243   }
244   // examine the error condition
245   if (*pde & PTE_P && pgtab[table] & PTE_P) {
246     // print phsycial address
247     cprintf("Physical Address: %x\n", pa);
248     return (long)pa;
249   } else {
250     return -1;
251   }
```

hasPage() System Call (implemented by Jack Critzer)

```
// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  struct trapframe *tf;        // Trap frame for current syscall
  struct context *context;     // swtch() here to run process
  void *chan;                  // If non-zero, sleeping on chan
  int killed;                  // If non-zero, have been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  char name[16];               // Process name (debugging)

  // properites for alsoNice
  int timeslice;               // the number of timeslices this process has requested to exec for
  int curr_slice;              // the current amount of timeslices this process has been running for

  // properties for hasPages

  uint data_text_end;
  uint stack_start;
  uint heap_start;
};
```

```
// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.

// start of guard page
sz = PGROUNDUP(sz);

// add 1 page to get stack_start
curproc->stack_start = sz + PGSIZE;


if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
  goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;

// sz points to end of stack, start of heap
curproc->heap_start = sz;
```

```
// search ptable for process with PID = pid
// return process struct if found, 0 otherwise
struct proc*
findProc(int pid) {
  struct proc *p;

  acquire(&ptable.lock);

  // iterate through page table
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    // process in page table
    if(p->pid == pid){
      // release lock and return process
      release(&ptable.lock);
      return p;
    }
  }
  // process not found
  release(&ptable.lock);
  return 0;

}
```

```
int
sys_haspages(void)
{
  struct proc *p;
  int pid;
  pde_t *pgdir;
  pte_t *pte;

  if(argint(0, &pid) < 0)
    return 0;

  // get process with pid
  p = (struct proc*) findProc(pid);

  // process not found
  if(!p){
    cprintf("No process with PID %d", pid);
    return -1;
  }

  pgdir = p->pgdir;

  // top of process address space
  //uint top_process = 0xFFFFF000;

  // process segments
  uint data_text_end = p->data_text_end;
  uint stack_start = p->stack_start;
  uint heap_start = p->heap_start;

  cprintf("This process, %s, has the following pages:\n\n", (char *) p->name);
  cprintf("Virtual Address  |   Permissions   |   Segment\n\n");
  cprintf("-----------------------------------------------\n\n");

  // iterate thru page directory, i points at current page offset
  for(uint i=0; i < KERNBASE; i += PGSIZE) {

    // get page table entry
    pte = walkpgdir(pgdir, (void *) i, 0);

    // page not mapped or present bit not set, go to next page
```

```
    else if (i >= heap_start && (perm_bits & PTE_U)) {
        segment = "HEAP";
    }
    // kernel page
    else if (i >= KERNBASE) {
        segment = "KERNEL";
    }

    cprintf(" %x            |       %s        |    %s\n", i, perm_arr, segment);
  }

  cprintf("-----------------------------------------------\n\n");

  return 1;
}


// add char to str
void add_char (char *str, char c)
  {
    for (;*str;str++);
    *str++ = c;
    *str++ = 0;
  }
```

```
 // iterate thru page directory, i points at current page offset
 for(uint i=0; i < KERNBASE; i += PGSIZE) {

   // get page table entry
   pte = walkpgdir(pgdir, (void *) i, 0);

   // page not mapped or present bit not set, go to next page
   if(!pte || !(*pte & PTE_P))
     continue;

   // get permission bits
   int perm_bits = ((int)*pte) & 0x7;

   char perm_arr[3] = "";

   // check if R, W, U flags are set
   // if yes, add permission to array, else add -

   perm_bits & PTE_P ? add_char(perm_arr, 'R') : add_char(perm_arr, '-');

   perm_bits & PTE_W ? add_char(perm_arr, 'W') : add_char(perm_arr, '-');

   perm_bits & PTE_U ? add_char(perm_arr, 'U') : add_char(perm_arr, '-');


   // determine segment page is in
   char *segment = (char *) 0;

   // data/text segment
   if(i < data_text_end) {
       segment = "DATA/TEXT";
   }
   // guard page, skip
   else if (i < stack_start) {
       segment = "GUARD PAGE";
   }
   // stack page
   else if  (i < heap_start) {
       segment = "STACK";
   }
   // heap page
```

```
// allocate text/data section of process starting at VA 0

// sz starts at 0

// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
  if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
    goto bad;
  if(ph.type != ELF_PROG_LOAD)
    continue;
  if(ph.memsz < ph.filesz)
    goto bad;
  if(ph.vaddr + ph.memsz < ph.vaddr)
    goto bad;
  if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
    goto bad;
  if(ph.vaddr % PGSIZE != 0)
    goto bad;
  if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
    goto bad;
}
iunlockput(ip);
end_op();
ip = 0;

// finished allocation data + text
// set data_text_end to current sz
curproc->data_text_end = sz;


// Allocate two pages at the next page boundary.
// Make the first inaccessible.  Use the second as the user stack.

// start of guard page
sz = PGROUNDUP(sz);

// add 1 page to get stack_start
curproc->stack_start = sz + PGSIZE;
```