PA3_Part 2

Chen En Lee, A20435695

Sarbani Bhattacharyya, A20383075

Manual Page

Name:

myV2p() - Get physical address and check readability and writability

Synopsis:

myV2p(int address, int operation)   operation: 1 – read, 2- write

Description:

Pass an address and an operation. Print out the physical address of this virtual address in this current process by adding virtual address to the base physical address of the process. Then, print out the page of this address and the ability to write or read to this page according to the operation passed into this system call. If the passing address is too large or the passing operation is not 1 or 2, print out the exception.

Return Value:

0

Exception:

Passing address too large.

Passing address less than 0.

Passing operation is not 1 or 2.

Example:

1. myV2p(100, 1)
2. myV2p(200, 2)

Manual Page

Name:

hasPages()   - Check the page information for the process

Synopsis:

hasPages(int pid)

Description:

Find the process from process table by checking the pid. Then run through all pages for this process to get how many pages has been used and how many of them are readable and writable.

Return value:

0

Exception:

Passing void to the system call

Passing a non-existed pid to the system call

Example:

hasPages(getpid())

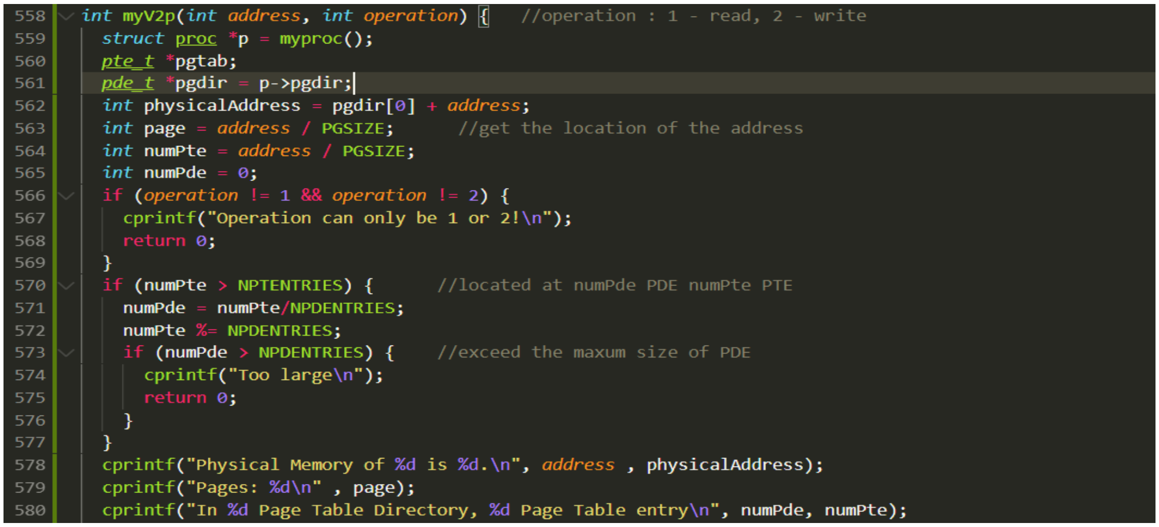
Different kind of information:

By checking the "mmu.h" file, we find there are three kind of flags. PTE_P, PTE_W, and PTE_U. PTE_P represent the valid page directory. PTE_W represents the writable page. PTE_U represents the user page which means it's readable. So, we can provide information includes valid page directory, writable pages, and readable pages via these flags.

Design

These two methods has been implemented in "proc.c" file

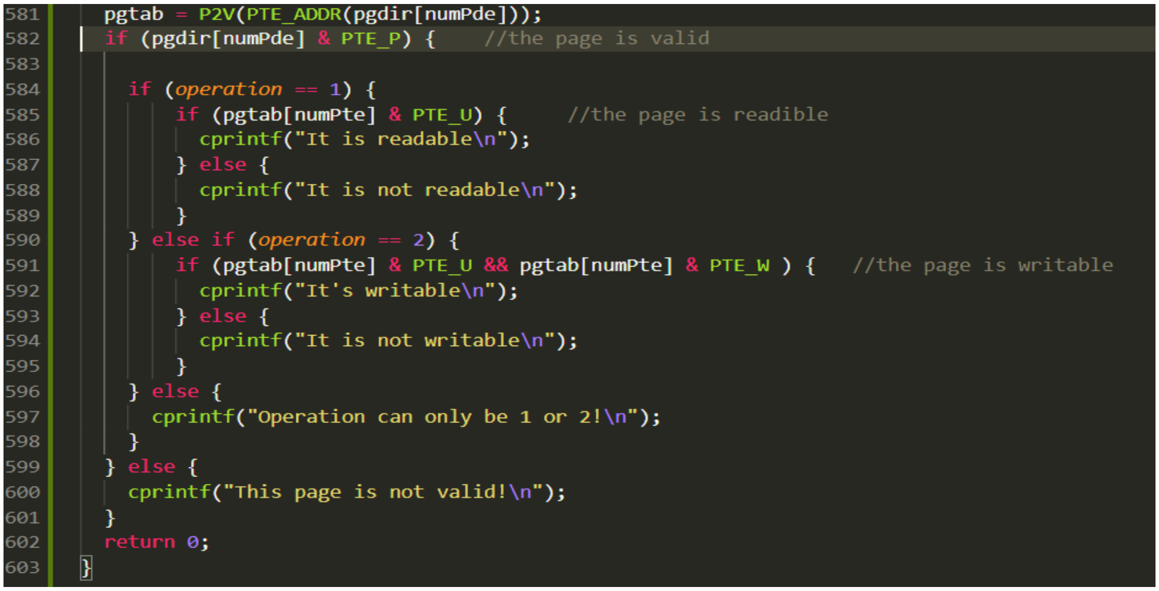
myV2p:



```c
int myV2p(int address, int operation) {   //operation : 1 - read, 2 - write
  struct proc *p = myproc();
  pte_t *pgtab;
  pde_t *pgdir = p->pgdir;
  int physicalAddress = pgdir[0] + address;
  int page = address / PGSIZE;      //get the location of the address
  int numPte = address / PGSIZE;
  int numPde = 0;
  if (operation != 1 && operation != 2) {
    cprintf("Operation can only be 1 or 2!\n");
    return 0;
  }
  if (numPte > NPTENTRIES) {       //located at numPde PDE numPte PTE
    numPde = numPte/NPDENTRIES;
    numPte %= NPDENTRIES;
    if (numPde > NPDENTRIES) {    //exceed the maxum size of PDE
      cprintf("Too large\n");
      return 0;
    }
  }
  cprintf("Physical Memory of %d is %d.\n", address , physicalAddress);
  cprintf("Pages: %d\n" , page);
  cprintf("In %d Page Table Directory, %d Page Table entry\n", numPde, numPte);
```

To get the physical address of the virtual address, we need to get the base address of the process first. Then add these two addressed together to get the physical address of the virtual address.

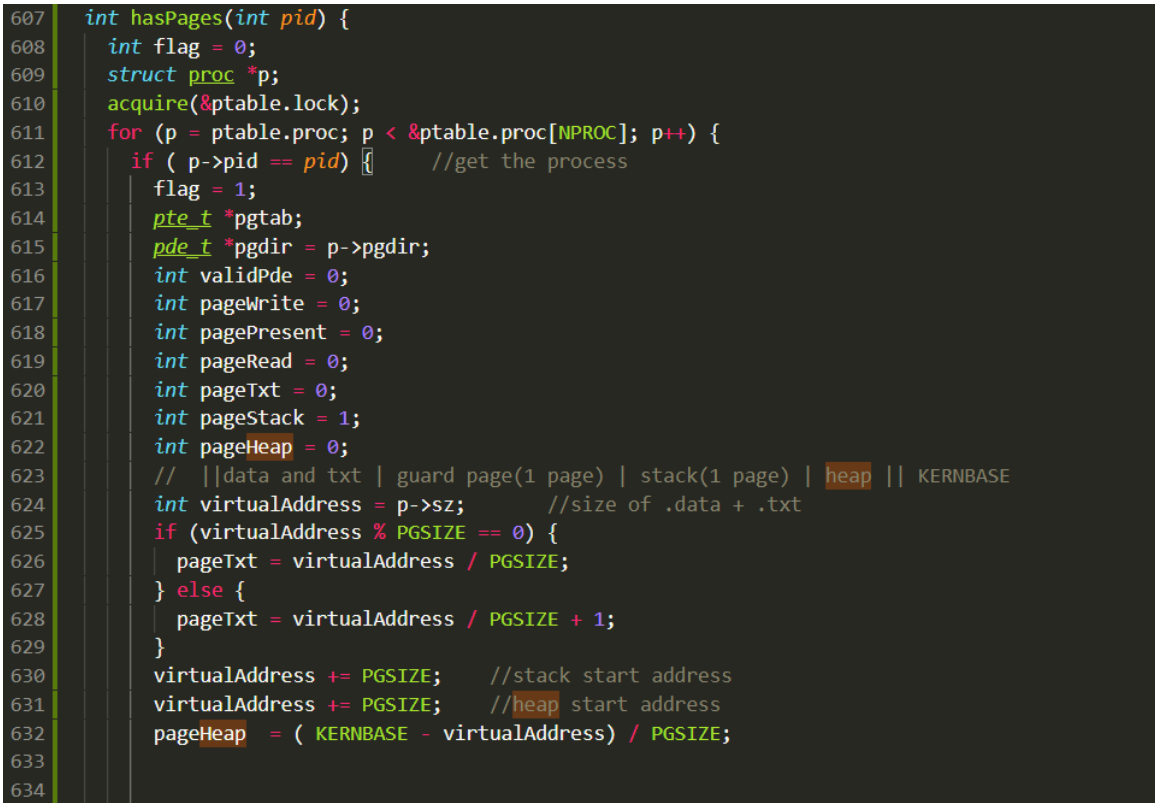
By dividing the address, we can get the location of the virtual address at which PDE and which PTE.



```c
  pgtab = P2V(PTE_ADDR(pgdir[numPde]));
  if (pgdir[numPde] & PTE_P) {    //the page is valid

    if (operation == 1) {
      if (pgtab[numPte] & PTE_U) {      //the page is readible
        cprintf("It is readable\n");
      } else {
        cprintf("It is not readable\n");
      }
    } else if (operation == 2) {
      if (pgtab[numPte] & PTE_U && pgtab[numPte] & PTE_W ) {    //the page is writable
        cprintf("It's writable\n");
      } else {
        cprintf("It is not writable\n");
      }
    } else {
      cprintf("Operation can only be 1 or 2!\n");
    }
  } else {
    cprintf("This page is not valid!\n");
  }
  return 0;
}
```

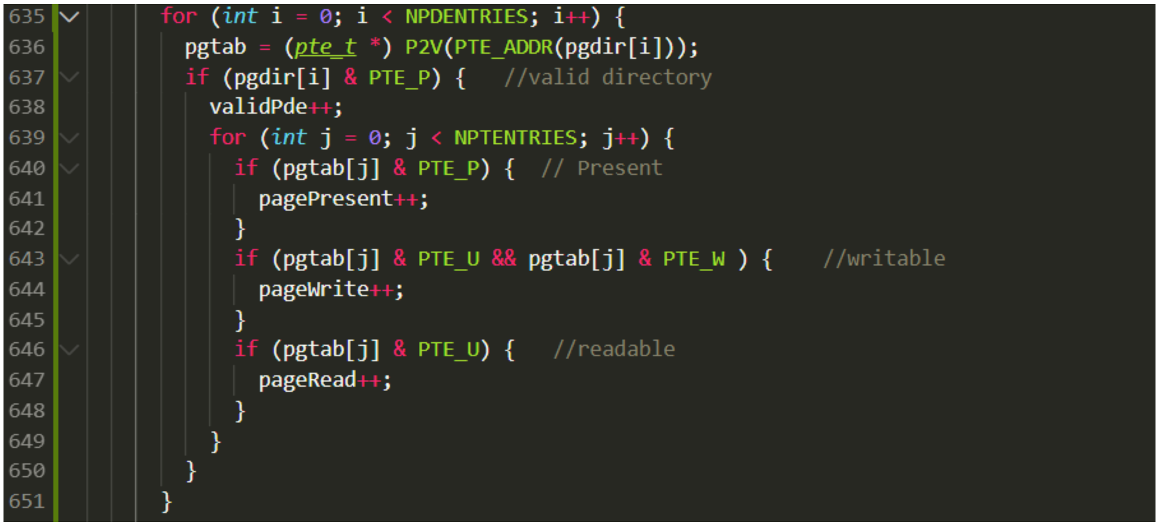
We can get the information from the flag (PTE_U, PTE_W, PTE_P) which is defined in “mmu.h” file.

hasPage:



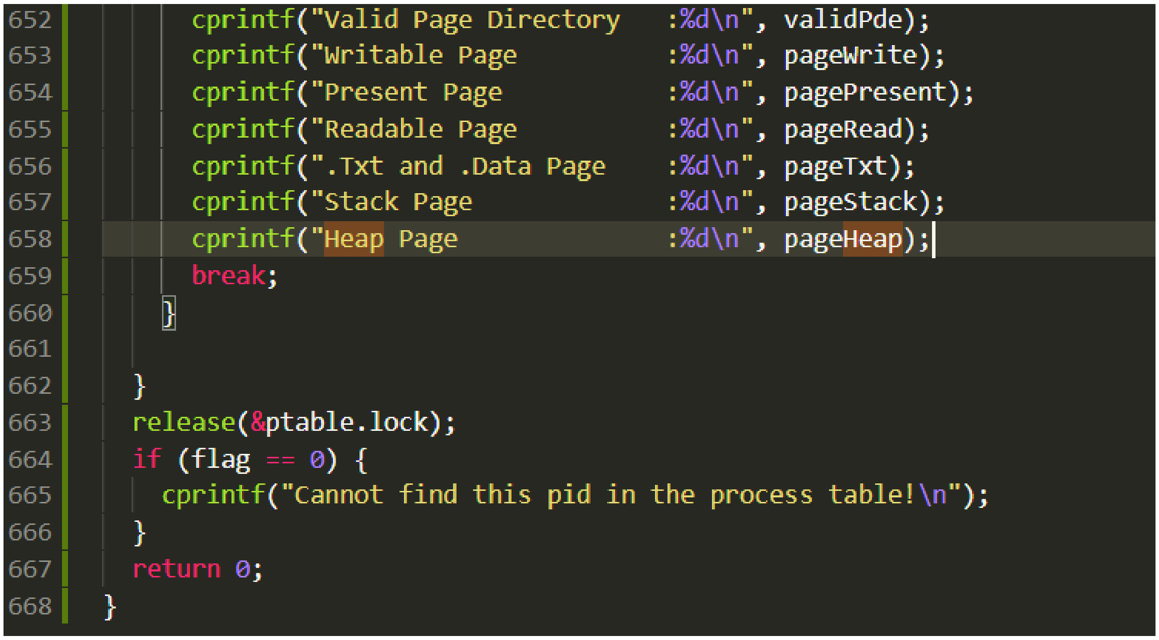
```c
int hasPages(int pid) {
  int flag = 0;
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
    if ( p->pid == pid) {     //get the process
      flag = 1;
      pte_t *pgtab;
      pde_t *pgdir = p->pgdir;
      int validPde = 0;
      int pageWrite = 0;
      int pagePresent = 0;
      int pageRead = 0;
      int pageTxt = 0;
      int pageStack = 1;
      int pageHeap = 0;
      //  ||data and txt | guard page(1 page) | stack(1 page) | heap || KERNBASE
      int virtualAddress = p->sz;       //size of .data + .txt
      if (virtualAddress % PGSIZE == 0) {
        pageTxt = virtualAddress / PGSIZE;
      } else {
        pageTxt = virtualAddress / PGSIZE + 1;
      }
      virtualAddress += PGSIZE;    //stack start address
      virtualAddress += PGSIZE;    //heap start address
      pageHeap  = ( KERNBASE - virtualAddress) / PGSIZE;


```

Loop through the process table to find the exact process. Set the variable counters validPde, pageWrite, pagePresent, pageRead. By checking the sz variable we can get the size of .txt and .data. According to the “exec.c”, we know there is a gard page (1 page) between .data and stack and we also know stack is 1 page. Heap address starts from the end of the stack address to the beginning of “KERNBASE”



```c
            for (int i = 0; i < NPDENTRIES; i++) {
                pgtab = (pte_t *) P2V(PTE_ADDR(pgdir[i]));
                if (pgdir[i] & PTE_P) {   //valid directory
                    validPde++;
                    for (int j = 0; j < NPTENTRIES; j++) {
                        if (pgtab[j] & PTE_P) {  // Present
                            pagePresent++;
                        }
                        if (pgtab[j] & PTE_U && pgtab[j] & PTE_W ) {    //writable
                            pageWrite++;
                        }
                        if (pgtab[j] & PTE_U) {   //readable
                            pageRead++;
                        }
                    }
                }
            }
```

Loop through all the page tables and page entries each by each and check it's PTE_P, PTE_U, and PTE_W flag. If matches, increases the counter.



```c
            cprintf("Valid Page Directory   :%d\n", validPde);
            cprintf("Writable Page          :%d\n", pageWrite);
            cprintf("Present Page           :%d\n", pagePresent);
            cprintf("Readable Page          :%d\n", pageRead);
            cprintf(".Txt and .Data Page    :%d\n", pageTxt);
            cprintf("Stack Page             :%d\n", pageStack);
            cprintf("Heap Page              :%d\n", pageHeap);
            break;
        }

    }
    release(&ptable.lock);
    if (flag == 0) {
        cprintf("Cannot find this pid in the process table!\n");
    }
    return 0;
}
```

Print out the information of the process. If cannot find the process in the process table, print the exception.