1 Problem 1 - Printing Page Table Entries

The functions defined for this part are in *kernel/vm.c* - *vmprint()* and utillity function *_print_valid_pte()*.

vmprint(), _print_valid_pte() in kernel/vm.c

```
// Function to print the valid pagetable entries for a process recursively
void
_print_valid_pte(pagetable_t pagetable, char* linestart)
{
  int len = strlen(linestart);
  linestart[len] = '.';
  linestart[len+1] = '.';
  linestart[len+2] = ' ';
  linestart[len+3] = '\0';

  // there are 2^9 = 512 PTEs in a page table.
  for(int i = 0; i < 512; i++){
    pte_t pte = pagetable[i];
    if(pte & PTE_V) {
      printf("%s%d: pte %p pa %p\n", linestart, i, pte, PTE2PA(pte));
    }
    if((pte & PTE_V) && (pte & (PTE_R|PTE_W|PTE_X)) == 0) {
      // this PTE points to a lower-level page table.
      uint64 child = PTE2PA(pte);
      _print_valid_pte((pagetable_t)child, linestart);
    }
}

linestart[len] = '\0';
}

// Function to print the pagetable info of a process in the given format
void
vmprint(pagetable_t pagetable)
{
  char linestart[12] = "";
  printf("page table %p\n", pagetable);
  _print_valid_pte(pagetable, linestart);
}
```

The function *vmprint()* takes a pagetable as input and prints all valid page table entries (PTEs) at all levels of the pagetable, with proper indentation to indicate the level of the PTE.
linestart is a *char** (string) variable that stores the string to be printed at the beginning of each page table entry line to manifest the indentation. It is assigned a size of 12 because there are at most three levels of PTEs and each deeper level is indicated by an additional ".. ", hence requiring at most 9 characters plus one for the null character at the end.

The *_print_valid_pte()* function works in a recursive way just like *kernel/vm.c/freewalk()*. At the beginning of the function, linestart string is appended with ".. " and at the end of the function one ".. " is stripped from the end of linestart. Thus depending on the depth of the recursion those many number of ".. " gets added to the beginning of the line printed.
The function traverses through the $2^9 = 512$ entries at the given level of the pagetable and prints the page table entry if it is valid. The presence of *PTE_R* or *PTE_W* or *PTE_X* bits indicates that the PTE is at level 0 (leaf). So, if the PTE is valid and not a leaf, the functions goes into a page table one level deeper, as pointed by the address in the PTE at current level.

2 Problem 2 - Eliminate Allocation from sbrk()

The code for this part was done in *kernel/sysproc.c/sys_sbrk()*.

kernel/sysproc.c/sys_sbrk()

```
uint64
sys_sbrk(void)
{
  int addr;
  int n;

  if(argint(0, &n) < 0)
    return -1;
  addr = myproc()->sz;

  //if(growproc(n) < 0)
  //  return -1;

  myproc()->sz += n;
  // Allocation happens lazily
  // De-allocation alone happens normally
  if(n < 0){
    uvmdealloc(myproc()->pagetable, addr, addr + n);
  }

  return addr;
}
```

The system call handler *sys_sbrk()* handles both allocation (if $n > 0$) and de-allocation (if $n < 0$). n is added to the size of the process to reflect the system call request in the size of the process.

If $n < 0$, the required de-allocation happens through a call to *uvmdealloc()* which is present in *kernel/vm.c* in the *sys_sbrk()* function itself so that the free memory is available for being allocated to other processes.

If $n > 0$, the required allocation happens lazily (i.e) a page is allocated only when that memory location is accessed by the process. This is done in problem 3 in *kernel/trap.c* and is not done in *sys_sbrk()*. Hence the *grwoproc()* call, which took care of both allocation and de-allocation earlier, is commented and only de-allocation code is added to the function. The allocation is logically conveyed to the process through the change in process size alone.

3 Problem 3 - Lazy Allocation

The code for this part is mainly in *kernel/trap.c/usertrap()*.

kernel/trap.c/usertrap()

```
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();
```

```
  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else if(r_scause() == 13 || r_scause() == 15){
    // Page Load/Store Fault
    uint64 vaddr = r_stval();
    uint64 grdown_vaddr = PGROUNDDOWN(vaddr);

    // If a virtual address is accessed without it being allocated
    if(vaddr >= p->sz) {
      printf("usertrap(): pagefault: invalid memory access to %p\n", vaddr);
      p->killed = 1;
      goto end;
    }

    char* mem = kalloc();
    if(mem == 0){
      printf("usertrap(): pagefault: out of memory when trying to lazy allocate to %p\n", vaddr);
      p->killed = 1;
      goto end;
    }

    memset(mem, 0, PGSIZE);
    if(mappages(p->pagetable, grdown_vaddr, PGSIZE, (uint64)mem, PTE_W|PTE_X|PTE_R|PTE_U) != 0){
      printf("usertrap(): pagefault: error while mapping physical page for %p\n", vaddr);
      kfree(mem);
      p->killed = 1;
      goto end;
    }

  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

end:
  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();
```

```
  usertrapret();
}
```

r_scause() gets the contents of the scause register which stores an identifier indicating the cause of trap or interrupt. If it is 13 or 15 it corresponds to load or store page fault respectively. We add an else if clause for this case before the error message being printed to handle the allocation of pages which previously were not physically allocated by *sys_sbrk()* but the size of the process alone got increased. Thus when that memory location is accessed through a load or store operation, only then is a physical page allocated leading to a **"lazy allocation"**.

The address that caused the page fault is obtained from the stval register and rounded down to the nearest multiple of PGSIZE lesser than or equal to it using *PGROUNDDOWN* macro. It is verified that the virtual address indeed is valid (i.e) the address is a part of the process' virtual memory map. Then a physical page is allocated through a *kalloc()* call and mapping from the rounded down virtual address that caused the fault to the physical page is performed thorugh *mappages()*. Any error while allocating physical memory or mapping is conveyed thorugh an error message and process killed.

There are minor modifications done to *uvmunmap()* function so that it does not panic when a virtual address is not mapped to a physical page (to support lazy allocation) :

kernel/vm.c/uvmunmap()

```
// Remove npages of mappings starting from va. va must be
// page-aligned. The mappings must exist.
// Optionally free the physical memory.
void
uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
{
  uint64 a;
  pte_t *pte;

  if((va % PGSIZE) != 0)
    panic("uvmunmap: not aligned");

  for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
    if((pte = walk(pagetable, a, 0)) == 0)
      panic("uvmunmap: walk");
    // if((*pte & PTE_V) == 0)
    //   panic("uvmunmap: not mapped");
    if(PTE_FLAGS(*pte) == PTE_V)
      panic("uvmunmap: not a leaf");
    if((*pte & PTE_V) != 0 && do_free){
      uint64 pa = PTE2PA(*pte);
      kfree((void*)pa);
    }
    *pte = 0;
  }
}
```

The check for the asserting the validity of the PTE is commented out and that check is added to the case where physical memory is made free so that *kfree()* does not act on invalid addresses and raises panic/error.