CS3500: Operating Systems Lab 4

Chester Rebeiro

August 28 2020

Submission Guidelines
Your submission folder should contain all the files where you have made modifications and the screenshot of the output for each of the problem.The submission folder should be named as **{Rollno}.zip**.Please submit the assignment to the moodle before deadline.

1 Introduction

One of the many neat tricks an O/S can play with page table hardware is lazy allocation of user-space heap memory. Xv6 applications ask the kernel for heap memory using the sbrk() system call.*sbrk*() allocates physical memory and maps it into the process's virtual address space. However, there are programs that use sbrk() to ask for large amounts of memory but never use most of it, for example to implement large sparse arrays. To optimize for this case, sophisticated kernels allocate user memory lazily. That is, sbrk() doesn't allocate physical memory, but just remembers which addresses are allocated. When the process first tries to use any given page of memory, the CPU generates a page fault, which the kernel handles by allocating physical memory, zeroing it, and mapping it. You'll add this lazy allocation feature to xv6

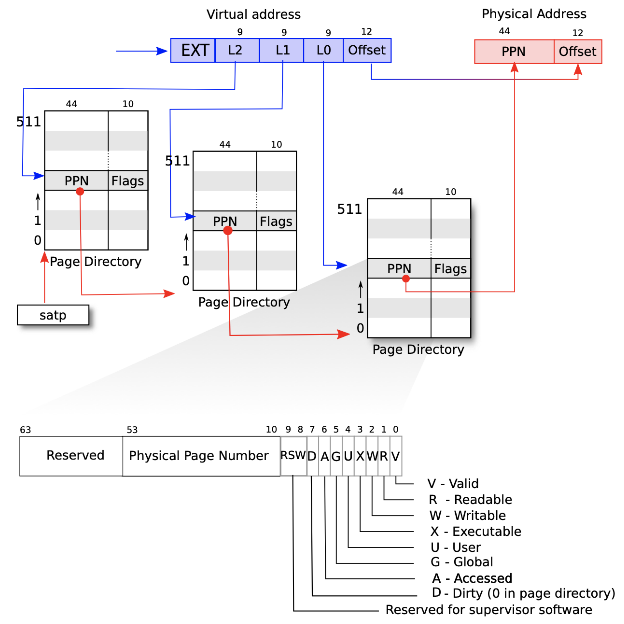

Figure 3.2: RISC-V page table hardware.

in this lab.

Few pointers to know about which will come useful while solving this lab:

- Look at the x86 paging hardware diagram for reference.
- you will need to understand the following files: *defs.h*, , *proc.h*, *sysproc.c*, *trap.c*, *vm.c*
  - The file *defs.h* acts as the header file for several parts of the kernel code
  - The file *trap.c* contains trap handling code, including page faults
  - The file *proc.h* contains various definitions and macros pertaining to virtual address translation and page table structure
  - The file *vm.c* contain most of the logic for memory management in the xv6 kernel
- *mappages* function installs mappings into a page table for a range of virtual addresses to a corresponding range of physical addresses. It does this separately for each virtual address in the range , at page intervals. For each virtual address to be mapped,*mappages* calls *walkpgdir* to find the address of the PTE for that address. It then initializes the PTE to hold the relevant physical page number, the desired permissions (PTE_W and/or PTE_U), and PTE_P to mark the PTE as valid
- *walkpgdir* takes a page directory (first-level page table) and returns a pointer to the page table entry for a particular virtual address. It optionally will allocate any needed second-level page tables.

2 Print Page Table (20 marks)

- The first task is to implement a function that prints the contents of a page table.
- Define the function in *kernel/vm.c* having following prototype:***void vmprint(pagetable_t)***.
- Insert a call to *vmprint* in *exec.c* to print the page table for the first user process; its output should be as below.
- Insert the function prototype in *defs.h*
- **Sample Output**

```
page table 0x0000000087f6e000
..0: pte 0x0000000021fda801 pa 0x0000000087f6a000
.. ..0: pte 0x0000000021fda401 pa 0x0000000087f69000
.. .. ..0: pte 0x0000000021fdac1f pa 0x0000000087f6b000
.. .. ..1: pte 0x0000000021fda00f pa 0x0000000087f68000
.. .. ..2: pte 0x0000000021fd9c1f pa 0x0000000087f67000
..255: pte 0x0000000021fdb401 pa 0x0000000087f6d000
.. ..511: pte 0x0000000021fdb001 pa 0x0000000087f6c000
.. .. ..510: pte 0x0000000021fdd807 pa 0x0000000087f76000
.. .. ..511: pte 0x000000002000200b pa 0x0000000080008000
```

Note:

- The first line prints the address of the argument of vmprint.
- Each PTE line shows the PTE index in its page directory, the pte, the physical address for the PTE.
- The output should also indicate the level of the page directory: the top-level entries are preceeded by "..", the next level down with another "..", and so on. You should not print entries that are not mapped.
- You can use the idea from *freewalk* function in *vm.c*

3 Eliminate allocation from sbrk() (10 marks)

- Your Next but very task is to delete page allocation from the *sbrk*(*n*) system call implementation, which is the function *sys_sbrk*() in *sysproc.c*.
- The *sbrk*(*n*) system call grows the process's memory size by *n* bytes, and then returns the start of the newly allocated region (i.e., the old size).
- Your new sbrk(n) should just increment the process's size ($myproc()->sz$) by *n* and return the old size.

- Remember to delete the memory allocation but you still need to increase the process size
- **Sample Output**
  Make this modification, boot xv6, and type *echohi* to the shell. You should see something like this:

  ```
  init: starting sh
  $ echo hi
  usertrap(): unexpected scause 0x000000000000000f pid=3
  sepc=0x0000000000001258 stval=0x0000000000004008
  va=0x0000000000004000 pte=0x0000000000000000
  panic: uvmunmap: not mapped
  ```

Note: Make sure you understand why the page fault occurs.

4 Lazy allocation (40 marks)

- Modify the code in *trap.c* to respond to a page fault from user space by mapping a newly-allocated page of physical memory at the faulting address, and then returning back to user space to let the process continue executing.
- You should add your code just before the printf call that produced the "*usertrap*() : ..." message so that when you call *echohi* it should be able to allocate new page.
- **Sample Output** $ echo hi statement should work correctly.

Note: you will find some additional problems that have to be solved to make it work correctly

- You can check whether a fault is a page fault by seeing if *r_scause*() is 13 or 15 in *usertrap*().
- Look at the arguments to the *printf*() in *usertrap*() that reports the page fault, in order to see how to find the virtual address that caused the page fault.
- Steal code from *uvmalloc*() in *vm.c*, which is what *sbrk*() calls (via *growproc*()). You'll need to call *kalloc*() and *mappages*().
- Use *PGROUNDDOWN*(*va*) to round the faulting virtual address down to a page boundary.
- *uvmunmap*() will panic; modify it to not panic if some pages aren't mapped.

- Use your print function from above to print the content of a page table.
- If you see the error "incomplete type proc", include "*proc.h*" and "*spinlock.h*".