Fall 2017 Mid Term Solution

Solution for CS 143A course at University of California, Irvine

Chen Wang[*]

11/8/2019

Contents

1 Basic page tables.

1.1 Address Mapping Explanation

(10 points) Illustrate the page table used by xv6 to map the kernel into the virtual address space of each process (draw a page table diagram and explain the page table entries). Specifically concentrate on one entry: the entry responsible for the translation of the first page of the kernel. Keep in mind that xv6 maps the kernel into the virtual address range starting above the second gigabyte of virtual memory. Note, that after xv6 is done booting,it xv6 uses normal 4KB, 32bit, 2-level page tables. You also have to recall the physical address of the first kernel page (look at the boot lecture or the kernel map), and the virtual address where this page is mapped. To make the example realistic, don't forget that xv6 allocates memory for it's page table directory and page tables from the kernel memory allocator.

Reference Solution:

The mapping relationship can be just found on the Lecture 10 - Kernel Page Table, the diagram is on the slide page 79. And for the page table diagram, this can be found at the slide page 82. As for the first page of the kernel, it is certainly at virtual address 0x80000000. We take its first 10 bits, and find out that this is at the 512th entry at the page table directory. Then we take the second 10 bits of the virtual address, which is 0, we can look into the 0th entry of the page table found through the 512th entry of the directory. Through the 0th entry of the table, we find out the physical address of the kernel.

2 Shell

Alice works on implementing a new shell for xv6. She implements a pipe command (e.g., ls|wc) like this:

[*]Undergraduate in Computer Engineering, Samueli School of Engineering, University of California, Irvine. (chenw23@uci.edu)

```
void
runcmd(struct cmd *cmd)
{
  ...
  switch(cmd->type){
  default:
    fprintf(stderr, "unknown runcmd\n");
    exit(-1);

    case '|': pcmd = (struct pipecmd*)cmd;
      int p[2];
      pipe(p);
      int pid = fork();
      if(pid == 0){//child process:left side
        close(1);
        dup(p[1]);
        close(p[1]);
        close(p[0]);
        runcmd(pcmd->left);
      }
      close(0);
      dup(p[0]);
      close(p[0]);
      close(p[1]);
      wait(NULL);
      runcmd(pcmd->right);
      break;
    }
    ...
}
```

2.1 Wrong Implementation Analysis

(a) (5 points) Her implementation always waits for left side to finish, but she is not sure if it's correct since she notices that the shell that xv6 implements (sh.c in the xv6 source tree) launches the right side right away. Can you come up with an example for which Alice's shell fails, while the xv6's is still correct? Explain your answer.

Reference Solution:

✓ ***This solution is proofread by Prof. Anton***

An example can be `ls|wc`, when the files to be listed is too many so that the pipe memory will get full.

Explanation:

In this version, the two ends of the pipe will execute sequentially. That is, the right end of the pipe will wait for the left end of the pipe to finish and exit and it will start to run. So suppose the left side take a long time to run, or the contents it puts into the pipe is so much, problems will occur in such circumstances.

3 OS isolation and protection

3.1 Memory Layout of xv6

(5 points) Explain the organization and memory layout of the xv6 process. Draw a diagram. Explain which protection bits are set by the kernel and explain why kernel does it.

Reference Solution:

It can be found from Lecture 04 - Linking and Loading. It is in the page 6 of the slides.

The third bit from the right end is the kernel bit. It prevents the user from modifying kernel data and prevents potential faults.

3.2 Memory isolation between processes

In xv6 individual processes are isolated, specifically they cannot access each others memory. Explain how this is implemented.

Reference Solution:

✓ ***This solution is proofread by Prof. Anton***

Each process has its own page table. When reading from a virtual address, MMU will find physical address by reading this process's page table. The pages of other processes are not mapped to this process's page table, therefore, the physical address of other processes will never be the translate result of this process.

4 OS organization

4.1 Kernel Base Understanding

(10 points) `KERNBASE` limits the amount of memory a single process can use, which might be irritating on a machine with a full 4 GB of RAM. Would raising `KERNBASE` allow a process to use more memory (explain your answer)?

Reference Solution:

No. Raising `KERNBASE` will leave kernel space less memory. However, page table and mappings reside in the kernel space. Therefore, shrinking the space that a kernel can use will also shrink the space a user process can use. So this will not allow a process to use more memory.