Winter 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 Translation through an example

(5 points) Illustrate organization of the x86, 4K, 32bit 2-level page tables through a simple example. Assume that the hardware translates the virtual address '0xc04005' (binary 0b1100 0000 0100 0000 0000 0101) into the physical address '0x55005'. The physical addresses of the page table directory and the page table (Level 2) involved in the translation of this virtual address are 0x8000 and 0x2000. Draw a diagram, provide a short explanation.

Reference Solution:

Well, it should be simple. Just follow the diagram on page 65 of the slide Lecture 10 - Kernel Page Table. Fill in the actual address number and it should be OK.

2 Shell

Xv6 shell implements a pipe command (e.g., ls — wc) as follows:

```
8650 case PIPE:
8651 pcmd = (struct pipecmd*)cmd;
8652 if(pipe(p) < 0)
8653   panic("pipe");
8654 if(fork1() == 0){
8655   close(1);
```

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

```
8656   dup(p[1]);
8657   close(p[0]);
8658   close(p[1]);
8659   runcmd(pcmd>left);
8660 }
8661 if(fork1() == 0){
8662   close(0);
8663   dup(p[0]);
8664   close(p[0]);
8665   close(p[1]);
8666   runcmd(pcmd>right);
8667 }
8668 close(p[0]);
8669 close(p[1]);
8670 wait();
8671 wait();
8672 break;
```

2.1 Pipe Analysis

(a) (5 points) Why does the child process that runs the left-side of the pipe close file descriptor1 and why does the child process that runs the right-side of the pipe close file descriptor0?

Reference Solution:

The file descriptor 1 is the standard output. So the left side process is the input of the pipe. It closes its standard output so that in the next `dup` command, the input of the pipe can be directed to the output of this process.

Likewise, the file descriptor 0 is the standard input So the right side process is the output of the pipe. It closes its standard input so that in the next `dup` command, the output of the pipe can be directed to the input of this process.

2.2 Problem Analysis

(b) (5 points) It looks that in the sh.c code above after the first fork() (at line 8654) both parent and child will reach the second fork() (line 8661) creating two child processes. Both child processes will start reading from the pipe and will try to execute the right side of the pipe. This seems wrong. Can you explain what is happening?

Reference Solution:

This is not true because in the `runcmd` function, the process will be executing a new program and they will exit rather than returning to the following codes in this page.

3 OS isolation and protection

3.1 User vs. kernel isolation

Reference Solution:

The user bit is not set in either page directory entry or page table entry (or both) for all translations that allow accessing pages of the kernel.

3.2 Page Flag Design

(5 points) Imagine you have hardware that is identical to x86, but does not have a user bit in the page tables. What changes need to be made to xv6 to ensure isolation of the kernel from user-processes?

Reference Solution:

If the Kernel is always in the front of physical memory, that is it starts at address 0 and goes till address phystop, we could enable segmentation to ensure that the physical memory that the Kernel is in can not be reached by the process.

4 OS organization.

Imagine you want to optimize xv6 to run a large number of very small processes. A realistic example can be a web server that implements a Facebook's login page—you have to isolate each user in its own process, otherwise a single exploit from any user reveals accounts of all users going through the login page, but at the same time each process is very small (it just sends out a simple HTML page). Entire logic of such web server program can fitin 2-3K of memory.

4.1 Page count

(10 points) You start by analyzing the overheads involved into creating a process. How many pages of memory are allocated when xv6 creates a smallest process? Count both user-level and kernel resources.

Reference Solution:

Since the process is small, we assume each part of the user-level program can fit in one page.

For user-level: One for text, one for data, one for heap, one for stack, and 1 for the Page table directory, 1 for the Page table. Totally at least 6. For kernel-level: 0 ~ 234MB, and we know that a page maps 4MB of space, that is it needs `234MB/4MB` pages

```
234 / 4 + 6
```

```
## [1] 64.5
```

Therefore, in total there should be 65 pages.

4.2 Optimizing Suggestion

(10 points) Suggest a set of changes to xv6 aimed at minimizing the number of pages that are required for creating very small processes, e.g., the once that are 1K in size.

Reference Solution:

Carefully analyze what system calls are used by these web page processes. Then in the mapped kernel space, omit out the functions that are not used by these small programs. It should be common that the small programs should only use limited functions of the system - otherwise, they won't be small.