Fall 2018 Mid Term Solution

*Chen Wang**

11/8/2019

Contents

1 OS Interfaces

This question is not covered in this midterm

2 Basic page tables

2.1 (5 points) Draw page table structure

Alice wants to construct a page table that maps virtual addresses 0x0, 0x1000and 0x2000 into physical addresses 0x1000, 0x2000, and 0x3000. Assume that the Page Directory Page is at physical address 0x0, and the Page Table Page is at physical address0x00001000 (which is PPN 0x00001).

Draw a picture of the page table Alice will construct (or alternatively simply write it down in the format similar to the one below):

Page Directory Page:

```
PDE 0: PPN=0x1, PTE_P, PTE_U, PTE_W
```

. . . all other PDEs are zero

The Page Table Page:

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

```
PTE 0: PPN=0x1, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x2, PTE_P, PTE_U, PTE_W
PTE 2: PPN=0x3, PTE_P, PTE_U, PTE_W
```

... all other PTEs are zero

Reference Solution:

Page Directory Page:

```
PDE 0: PPN=0x1, PTE_P, PTE_U, PTE_W
```

... all other PDEs are zero

The Page Table Page:

```
PTE 0: PPN=0x1, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x2, PTE_P, PTE_U, PTE_W
PTE 2: PPN=0x3, PTE_P, PTE_U, PTE_W
```

... all other PTEs are zero

3 Stack and calling conventions

Alice developed a program that has a function `foo()` that is called from two other functions `bar()` and `baz()`:

```
int foo(int a) {
  ...
}

int bar(int a, int b)
  ...
  foo(x);
  printf("bar:%d\n", x);
  ...
}

int baz(int a, int b, int c) {
  ...
  foo(x);
  printf("baz:%d\n", x);
  ...
}
```

While debugging her program Alice observes the following state when she hits a breakpoint of the program inside `foo()` (assume that the compiler does not inline invocations of `foo()`, `bar()`, and `baz()`, and follows the calling conventions that we've covered in the class)

```
The bottom of the stack:

0x8010b5b4: ...
0x8010b5b0: 0x00000003
0x8010b5ac: 0x00000002
0x8010b5a8  0x80102e80
0x8010b5a4: 0x8010b5b4
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
```

```
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4    <-- ebp
0x8010b590: 0x00000000    <-- esp
```

3.1 Explain Stack

(a) (5 points) Provide a short explanation for each line of the stack dump above (you canannotate the printout above).

Reference Solution

```
The bottom of the stack:

0x8010b5b4: ...        // ebp
0x8010b5b0: 0x00000003 // argument #2 to the function that called foo()'s caller
0x8010b5ac: 0x00000002 // argument #1 to the function that called foo()'s caller
0x8010b5a8  0x80102e80 // return address
0x8010b5a4: 0x8010b5b4 // ebp
0x8010b5a0: 0x80112780 // (local variable, argument to a funciton, or register
                          spill inside function that called foo)
0x8010b59c: 0x00000001 // arg to foo
0x8010b598: 0x80102e32 // return address for foo()
0x8010b594: 0x8010b5a4    <-- ebp
0x8010b590: 0x00000000    <-- esp (local variable, argument to a funciton, or
                                   register spill inside foo)
```

3.2 Analyze output

(b) (5 points) If Alice continues execution of her program what output will she see on thescreen (justify your answer).

Reference Solution

We know that foo() can be called from bar() or baz(), but we also know that the caller of foo()'s caller i.e., either bar() or baz(), got two arguments. Hence, it's bar(). And since we know that foo() got 0x1 as argument the string Alice will see on the screen should be `bar:1`

4 Xv6 process organization.

In xv6, in the address space of the process, what does the following virtual addresses contain?

4.1 (3 points) Virtual address 0x0

Reference Solution

The memory at virtual address 0x0 contains the text section (code) of the user process.

4.2 (3 points) Virtual address 0x80100000

Reference Solution:

The memory at virtual address 0x80100000 contains the text section (code) of the kernel. During the boot the kernel was loaded at physical address 0x100000 (1MB) and then later this address was mapped at 2GBs + 1MB or (0x80000000 + 0x100000).

4.3 (3 points) What physical address is mapped at virtual address 0x80000000

Reference Solution:

Physical address 0x0.

4.4 Physical address mapping lookup

(7 points) Is there a way for the kernel to find out what physical address is mapped at a specific virtual address? Provide an explanation and a code sketch (pseudocode is ok, no need to worry about correct C syntax). Your code should take a virtual address as an input and resolve it into the physical address that is mapped into that virtual address by the process page table (in your code feel free to re-use functions that are already implemented in the xv6 kernel).

Reference Solution:

In xv6 we can access the entire page table and the page tables contain information about how a virtual address mapps to the physical address. Therefore, we only need to go though the table to find out where the physical page lie in the page table and then we will be able to find out the virtual addresses that are directing to this physical address.

Please see the C file at https://github.com/StrayBird-ATSH/OperatingSystemCourseMidTermExams/blob/master/Fall%202018/Fall2018Mid-Virtual2Physical.c

5 Protection and isolation

5.1 Kenrel Memory Protection Explanation

(5 points) In xv6 all segments are configured to have the base of 0 and limit of 4GBs, which means that segmentation does not prevent user programs from accessing kernelmemory. Nevertheless, user programs can't read and write kernel memory. How (through what mechanisms) such isolation is achieved?

Reference Solution:

Above all, xv6 adopts page tables. That is, the kernel memory and user memory will reside in different pages. Also, each page has a flag indicating whether this page is for kernel or for user. Therefore, when a user program wants to access kernel program, it will access the kernel pages, and visiting a page with a kernel flag will trigger a fault. So achieved.

6 System calls

This question is not covered in this midterm