# Fall 2018 Mid Term Solution

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

*Chen Wang**

*11/8/2019*

## Contents



# 1 OS Interfaces

## 1.1 (a) (5 points) Program Analysis

Heres a program that uses the UNIX system call API, as described in Chapter 0 of the xv6 book:

```
#include "param.h"
#include "types.h"
#include "user.h"
#include "syscall.h"
int main() {
  char * message = "aaa\n";
```

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

```
  int pid = fork();

  if(pid != 0){

    char *echoargv[] = { "echo", "Hello\n", 0 };

    message = "bbb\n";
    exec("echo", echoargv);
    write(1, message, 4);

  }

  write(1, message, 4);
  exit();
}
```

Assume that fork() succeeds, that file descriptor 1 is connected to the terminal when the program starts, and echo program exists. What possible outputs this program can produce (explain your answer)?

***Reference Solution:***

The proccess forks() and execs() the "echo" program that prints "Hello" inside the parent. Since exec() overloads the address space of the parent the write(1, message, 4)line never gets executed (we assume that "echo" exists and exec() succeeds). The child prints "aaa". Two possible outputs depending on the order in which parent and child execute are

```
aaa
Hello
```

or

```
Hello
aaa
```

### 1.2 (b) (10 points) Multi-level pipe

Write a program that uses the UNIX system call API, as described in Chapter 0 of the xv6 book. The program forks and creates a pipeline of 10 stages. I.e., each stage is a separate process. Each two consequtive stages are connected with a pipe, i.e., the standard output of each stage is connected to the standard input of the next stage. Each stage reads a character from its standard input and sends it to the standard output. The last stage outputs the character it reads from the pipe to the standard output.

***Reference Solution:***

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

## 2 Basic page tables

### 2.1 (5 points) Draw page table structure

Alice wants to construct a page table that maps virtual addresses 0x0, 0x1000 and 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 address 0x00001000 (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:

```
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 break point 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 can annotate 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 the screen (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 look-up

(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 (pseudo-code 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 maps 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 Kernel 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 kernel memory. 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

## 6.1 System Call arguments

(a) (5 points)

How do system calls access their arguments that are passed from the user level? After all, all system calls are declared as void functions that return an integer, i.e., visibly they don't take any arguments. For example, the `read()` system call is declared as:

```
6131 int
6132 sys_read(void)
```

But user code can invoke it as

```
int read(int, void*, int);
```

# 7 Physical and virtual memory allocation

## 7.1 V2P macro usage explanation

(5 points) What is the purpose of the V2P macro? Specifically, the `allocuvm()` function(see the listing below) uses `kalloc()` to allocate and map a region of memory into the address space of a process. Explain, why the `V2P` macro is used in line 1946 below?

```
1926 int
1927 allocuvm(pde_t *pgdir, uint oldsz, uint newsz)
1928 {
1929   char *mem;
1930   uint a;
1931
1932   if(newsz >= KERNBASE)
1933     return 0;
1934   if(newsz < oldsz)
1935     return oldsz;
1936
1937   a = PGROUNDUP(oldsz);
1938   for(; a < newsz; a += PGSIZE){
1939     mem = kalloc();
1940     if(mem == 0){
1941       cprintf("allocuvm out of memory\n");
1942       deallocuvm(pgdir, newsz, oldsz);
1943       return 0;
1944     }
1945     memset(mem, 0, PGSIZE);
1946     if(mappages(pgdir, (char*)a, PGSIZE, V2P(mem), PTE_W|PTE_U) < 0){
1947       cprintf("allocuvm out of memory (2)\n");
1948       deallocuvm(pgdir, newsz, oldsz);
1949       kfree(mem)
1950       return 0;
1951     }
1952   }
1953   return newsz;
1954 }
```

***Reference Solution:***

It translates a virtual address into physical address.

Because the `mappages` needs to check whether a physical page is mapped or not, or whether it is available. This translations gives the `mappages` function the location of the physical page needed and this function will do the mapping work.

# 8 Initial page tables

Bob looks at the piece of code in entry.S where the initial page tables are set and thinks he doesn't need the entry that maps the 0-4MB of virtual page to 0-4MB of physical page. Accordingly he modifies the entry pgdir as below.

```
__attribute__((__aligned__(PGSIZE)))
pde_t entrypgdir[NPDENTRIES] = {
  // Map VA's [KERNBASE, KERNBASE+4MB) to PA's [0, 4MB)
  [KERNBASE>>PDXSHIFT] = (0) | PTE_P | PTE_W | PTE_PS,
};
```

## 8.1 (5 points) Explain what will go wrong with Bob's change?

***Reference Solution:***

Because when the system initially boots, there was no kernel section in the upper 2GB and the codes are in the first 4MB space. Therefore, before the new page table is successfully mapped, we still need the mapping of the first 4MB to keep the code running smoothly.