CS 238P **Name (Print):**

Winter 2018
Midterm
02/20/2018
Time Limit: 3:30pm - 4:50am

- **Don't forget to write your name on this exam.**
- **This is an open book, open notes exam. But no online or in-class chatting.**
- **Ask us if you something is confusing in the questions.**
- **Organize your work**, in a reasonably neat and coherent way, in the space provided. Work scattered all over the page without a clear ordering will receive very little credit.
- **Mysterious or unsupported answers will not receive full credit**. A correct answer, unsupported by explanation will receive no credit; an incorrect answer supported by substantially correct explanations might still receive partial credit.
- If you need more space, use the back of the pages; clearly indicate when you have done this.

Problem	Points	Score
1	10	
2	10	
3	5	
4	10	
5	5	
6	5	
Total:	45	

1. Basic page tables.

   Consider the following 32-bit x86 page table setup.

   `%cr3 holds 0x00000000.`

   The Page Directory Page at physical address 0x00000000:

   ```
   PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W
   PDE 1: PPN=0x00002, PTE_P, PTE_U, PTE_W
   PDE 2: PPN=0x00001, PTE_P, PTE_U, PTE_W
   ```

   ... all other PDEs are zero

   The Page Table Page at physical address 0x00001000 (which is PPN 0x00001):

   ```
   PTE 0: PPN=0x00003, PTE_P, PTE_U, PTE_W
   PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
   ```

   ... all other PTEs are zero The Page Table Page at physical address 0x00002000:

   ```
   PTE 0: PPN=0x00005, PTE_P, PTE_U, PTE_W
   PTE 1: PPN=0x00004, PTE_P, PTE_U, PTE_W
   ```

   ... all other PTEs are zero

   (a) (5 points) What are all virtual addresses mapped by this page table?
   Solution.
   https://aftabhussain.github.io/documents/teaching/uci/cs238p/spring2019/cs238p-winter18-midterm-sol-q1.pdf

   (b) (5 points) What is the virtual address of the page table directory? Solution. None, we have not mapped any virtual address to 0x0, the physical address of the page table directory.

2. 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(...);
  ...
}

int baz(int a, int b, int c) {
  ...
  foo(...);
  ...
}
```

   While debugging her program Alice observes the following state after pausing execution of the program inside `foo()` (assume that the compiler does not inline invocations of `foo()`, `bar()`, and `baz()`:

```
The bottom of the stack:

0x8010b5b8: ...
0x8010b5b4: 0x00010074
0x8010b5b0: 0x00000002
0x8010b5ac: 0x00000001
0x8010b5a8  0x80102e80
0x8010b5a4: 0x8010b5b8
0x8010b5a0: 0x80112780
0x8010b59c: 0x00000001
0x8010b598: 0x80102e32
0x8010b594: 0x8010b5a4    <-- ebp
0x8010b590: 0x00000000    <-- esp
```

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

   Solution. *The key is to follow the frame pointers, and infer which function's info is being stored between them by looking at the size of the allocations between each frame pointer.*

```
The bottom of the stack:

0x8010b5b8: ...
0x8010b5b4: 0x00010074    <-- 3rd argument, c
0x8010b5b0: 0x00000002    <-- 2nd argument, b
0x8010b5ac: 0x00000001    <-- 1st argument, a
```

```
0x8010b5a8  0x80102e80    <-- return address of baz
0x8010b5a4: 0x8010b5b8    <-- ebp
0x8010b5a0: 0x80112780    <-- 2nd argument, b
0x8010b59c: 0x00000001    <-- 1st argument, a
0x8010b598: 0x80102e32    <-- return address of bar
0x8010b594: 0x8010b5a4    <-- ebp
0x8010b590: 0x00000000    <-- esp
```

(b) (5 points) Can Alice make a conclusion if foo() is called from the context of bar() or baz() (explain your answer)?

Solution. No. There is no frame pointer showing that foo has been called. The sizes of the two frames determined from the stack can only correspond to the functions bar (2 args and 1 return address) and baz (3 args and 1 return address).

3. Process organization.

   (a) (5 points) xv6 processes have the following memory layout created as part of the `exec()` function. First, the kernel allocates pages for the kernel text and data (not that these pages are both executable and writable). Then xv6 allocates two pages: stack and guard. The guard page is made is placed between the stack and the rest of the program to make sure that if the stack overflows the operating system can catch an exception caused by the access to the guard page and terminate the program early.

   Alice thinks that the guard page mechanism is bulletproof, i.e., there is no way for a C program to overflow the stack and start overwriting the program text and data. Is she right, i.e., is it possible to write a C program that escapes the guard page mechanism and accidentally overwrites the text section of the program (provide an example).

   Solution. Yes, it is possible to write a C program that escapes the guard page mechanism. If a C program has a local variable that is of size greater than 2 pages, we would skip the guard page and overwrite the text and data section.

4. Physical and virtual memory allocation

   (a) (5 points) Xv6 uses 234MB of physical memory. But how does it keep track of available physical memory? Specifically, explain the following: the xv6 memory allocator (kalloc()) always returns a virtual address, but how does the allocator know which physical page to use for each virtual address it allocates?

   Solution. Initially kinit1() constructs a linked-list of 4KB pages in the (end,P2V(4MB)) range with the initial pagetable. Once the new pagetable is initialized, kinit2() constructs a linked-list of 4KB pages from (P2V(4MB), P2V(PHYSTOP)). Each virtual page(4KB) in the range is linearly mapped to a physical page and is arranged in a chain like fashion to create a freelist of pages. Kalloc() takes a virtual page from the freelist and returns it to the caller.

   (b) (5 points) Xv6 defines the V2P() macro that allows the kernel to convert between virtual and physical addresses:

   ```
   #define V2P(a) (((uint) (a)) - KERNBASE)
   ```

   Does V2P() macro work for virtual addresses that belong to the user part of the address space (i.e., a virtual address inside the user data or stack)? Explain your answer.

   Solution. (a - KERNBASE) would give a negative address and wrap around to a high address in the physical memory and that may/may not be the correct physical address for that virtual page. In order to for the user to access virtual addresses, we need to walk the pagedir of the process to find out the physical address of the corresponding virtual page

5. Exec and fork

   (a) (5 points) 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";
  int pid = fork();

  if(pid != 0){

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

    message = "bbb\n";
    exec("echo", echoargv);
  }

  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 output this program produces (explain your answer)?

   "aaa" "Hello" or "Hello" "aaa". After fork, either the parent or the child runs first (no defined ordering).

6. 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 entrypgdir 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,
};
```

   (a) (5 points) Explain whether Bob's change will work?

   Solution. No. If the 0-4 MB mapping is removed, the kernel will not boot as the remaining few instructions of entry.S are still in the 0-4 MB physical/virtual range.