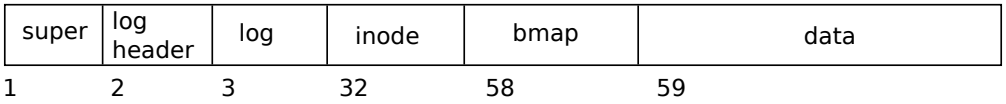
Principles of Operating Systems **Name (Print):** _____
Fall 2017
Final
12/13/2017
Time Limit: 8:00am - 10:00am

- **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 me if something is not clear 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	20	
2	5	
3	10	
4	10	
5	15	
6	5	
7	10	
8	5	
Total:	80	

1. File system

   Xv6 lays out the file system on disk as follows:



   Block 1 contains the super block. Blocks 2 through 31 contain the log header and the log. Blocks 32 through 57 contain inodes. Block 58 contains the bitmap of free blocks. Blocks 59 through the end of the disk contain data blocks.

   Ben modifies the function bwrite in bio.c to print the block number of each block written.

   Ben boots xv6 with a fresh fs.img and types in the command rm README, which deletes the README file. This command produces the following trace:

```
$ rm README
write 3
write 4
write 5
write 2
write 59
write 32
write 58
write 2
$
```

   (a) (5 points) Briefly explain what block 59 contains in the above trace. Why is it written?

   Block 59 contains the data for the "/" directory inode. Since Ben delets the file from the "/" the directory inode is updated.

   (b) (5 points) What does block 5 contain? Why is it written?

   Block 5 contains the copy of block 58 in the log. Xv6 file system frist writes the log to ensure atomicity of all file system updates, then it copies the log to the actual data blocks.

(c) (10 points) How many non-zero bytes are written to block 2 when it's written the first time and what are the bytes? (To get the full credit you have to explain what block 2 contains, and why each non-zero byte is written).

Block 2 contains the header of the log, or more specifically an integer (4 bytes) that contains the size of the log, and then an array of integers (4 bytes each) of size LOGSIZE that keep the actual block numbers for the blocks in the log. The log header data structure that is written to block 2 is defined as:

```
// Contents of the header block, used for both the on-disk header block
// and to keep track in memory of logged block# before commit.
struct logheader {
  int n;
  int block[LOGSIZE];
};
```

In our example, n is equal to 3, an then 3 integers are 59, 32, and 58. The total of 4 integers or 16 non-zero bytes are written to block 2.

If you want to be extra pedantic you can reason about non-zero integers that might be in the block array from previous log transactions. In practice, on a clean xv6 file system, the only one transaction that happened before wrote 2 blocks creating the console device. So the rest of the array is clean anyway.

2. Synchronization

(a) (5 points) Ben runs xv6 on a single CPU machine, he decides it's a good idea to get rid of the acquire() and release() functions, since after all they take some time but seem unnecessary in a single-CPU scenario. Explain if removal of these functions is fine.

No. In addition to acquire a spinlock to enter a critical section, `acquire()` disables interrupts to prevent an interrupt from entering the critical section and changing one of the protected data strutures concurrently with the process and other interrupts.

3. Process memory layout

Bob decides to implement the following xv6 program (hello)

```
#define PGSIZE  4096
int main(int argc, char *argv[]) {
  char buf[PGSIZE] = {0};
  printf(1, "Hello World!, %p\n", buf);
  exit();
}
```

(a) (10 points) When Bob runs it he encounters the following error message:

```
pid 3 hello: trap 14 err 7 on cpu 1 eip 0x22 addr 0x1fd0--kill proc
```

Can you help Bob understand the problem? (To receive full points, you should explain why this error happened, what does the error values mean and how Bob can fix his code)

Bob allocates an array of 4096 bytes on the stack

```
char buf[PGSIZE] = {0};
```

`Buf` is a local variable. It is allocated on the stack. Since xv6 stack is only one page (4096 bytes), and main already has 47 bytes on the stack (0x1fff-0x1fd0), the code tries to access the first element of the buffer when the compiler generates the code that initializes `buf` with zeroes. The first element of the buffer is actually on the guard page (0x1fd0). Hence, Bob's code triggers an exception and xv6 kills the process reporting a violation.

4. Virtual memory

Ben wants to know the address of physical pages that back up virtual memory of his process. He digs into the kernel source and comes across the V2P() macro that is frequently used in the kernel.

```
#define KERNBASE  0x80000000
#define V2P(a) (((uint) (a)) - KERNBASE)
```

He decides to try the V2P macro in his program (below), but encounters a crash.

```
void test(void) {
  int a;
  *(uint*)V2P(&a) = 0xaddb;
  printf(1, "I changed physical memory at %x\n", a);
}
int main(int argc, char *argv[]) {
  test();
  exit();
}
```

(a) (5 points) Explain what is going on and why Ben's program crashes.

(b) (5 points) Ben puts the code of the test() function inside a new system call trying to see if it works inside the kernel. Will it work (explain your answer)?

5. Demand paging

   Ben wants to extend xv6 with demand paging. Ben observes that some pages of user processes (heap, text, and stack) are not accessed that frequently, yet anyway they consume valuable physical memory. So Ben comes up with a plan to free these infrequently used pages by saving them to disk (swapping). For an idle page he plans to unmap it from the process page table, save content of the page to disk (i.e., in a special swap area on disk), and free the physical page back to the kernel, making it accessible for other processes. Obviously, Ben wants paging to be transparent. I.e., when a process accesses one of the swapped pages, Ben plans to catch an exception, allocate a new physical page, read old content of the page from disk, and fix the process page table in such a way that process can access the page like nothing happened.

   (a) (5 points) How can Ben unmap a page from the process address space? I.e., what changes to the process page table are required to catch an exception when the process tries to access a swapped page (hint: look at how guard page is implemented)?

(b) (10 points) Ben plans to catch the exception caused by an unmapped page access inside the trap() function. Provide a sketch for the code that implements the exception handling, reads page from disk, and maps it back into the process address space.

6. System call API

   Alice executes the following program

```
main() {
  char *msg = "bar\n";
  int pid = fork();
  if (pid)
    msg = "foo\n";
  else
    msg = "baz\n";
  write(1, msg, 4);
  exit(0);
}
```

   (a) (5 points) What are all possible outputs of this program? Explain your answer.

7. Process creation While editing the xv6 code, Jimmy accidentally erases the below section of code under fork() function on proc.c

```
2584 for(i = 0; i < NOFILE; i++)
2585    if(proc->ofile[i])
2586        np->ofile[i] = filedup(proc->ofile[i]);
```

   (a) (5 points) Explain what the above section of code does?

(b) (5 points) Explain what can go wrong without this piece of code? Quote a concrete example and explain the incorrect behavior.

8. cs143A. I would like to hear your opinions about cs143A, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

   (a) (1 point) Grade cs143A on a scale of 0 (worst) to 10 (best)?

   (b) (2 points) Any suggestions for how to improve cs143A?

   (c) (1 point) What is the best aspect of cs143A?

   (d) (1 point) What is the worst aspect of cs143A?