Operating Systems **Name (Print):**

Fall 2018
Final
12/11/2018
Time Limit: 4pm - 6pm

- **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 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.
- **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.**

Problem	Points	Score
1	25	
2	10	
3	5	
4	10	
5	20	
6	10	
7	5	
Total:	85	

1. Context switch

   (a) (5 points) Provide a high-level description of what happens during the context switch?

   Answer:

   There are two possible cases when xv6 switch from one process to another: 1) when the timer interrupt arrives, 2) when the process is going to sleep inside a system call and yields it's CPU time slides. In both cases the mechanics of the context switch are similar: the process enters the kernel with either an interrupt or a system call (in xv6 the system calls are invoked with the `int` instruction, so the path is very much the same). If the timer interrupt arrives and the process is already in the kernel, the hardware saves only 3 values on the kernel stack, since there is no change of privilege level and hence SS and CS don't have to be saved. If however the process is running in the user-space (CPL = 3) the hardware pushes 5 values on the kernel stack. Note, that execution cannot continue on the user stack, and hence the hardware locates the kernel stack by looking up the TSS. The hardware fetches the CS and EIP of the interrupt handler from the IDT, pushes 5 values on the stack, loads new CS segment from the GDT (the execution continues with CPL = 0) and jumps to the entry point of the handler.

   The handler saves state of the user process with the `alltraps` assembly function, pushes the pointer to the trapframe on the stack and invokes the C `traps()` function. Inside trap() the kernel context switches into the scheduler with the `swtch()` function. The scheduler picks the next process to run from the array of all processes, and switches into that process by invoking `swtch()` again.

   More details of how the state is saved on the stack are needed here ...

   (b) (5 points) During the context switch, where is the user register EAX saved (i.e., the value of the EAX register that was used by the preempted user program)?

   Answer:

   On the kernel stack, inside the trap frame, proc->tf->eax.

(c) (5 points) The register `EBX` gets saved twice, once by the `popal` instruction in the `alltraps()` function and second in the `swtch()` function. Can you explain why do we need to save it twice?

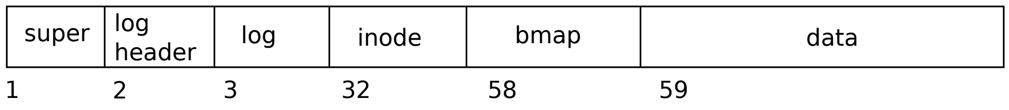
Answer:

First, EBX value of the user-process is saved on the trapframe inside the proc->tf->ebx field of the trapframe data structure. Then since EBX is callee saved register, the kernel cannot assume that it's value will be presumed while some other process is switched to run on the CPU. Hence xv6 pushes it on the stack as part of the context data structure (proc->contex->ebx).

(d) (10 points) In xv6 during the context switch the process first switches to the scheduler and then switches to the next process. Ben thinks it is wasteful to switch into the scheduler, and decides to change xv6 to switch directly to the next process. What changes he needs to implement? Be specific: describe the data structures that need to be changed, explain why your changes are correct, provide a sketch for the code that implements it.

2. 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 boots xv6 with a fresh fs.img and starts a program that opens a file which is represented by the inode 0 and writes one byte into it. The inode's dentry[0] contains 76.

(a) (5 points) Which blocks will be written as the result of the write system call?

Answer:

```
block 3               // Log data (bitmap)
block 4               // Log data (actual byte)
block 5               // Log data
block 2               // Log header
block 58              // Bitmap
block 595             // Actual byte (any number between 59 and ... are fine as answer)
block 32              // Inode
block 2               // Log header
```

(b) (5 points) Which block will be written more than once?
Block #2 that contains the log header.

3. Interrupts and Exceptions

(a) (5 points) Describe what happens when a timer interrupt arrives while a process executes in the kernel, i.e., after entering the kernel with a system call. Will it be preempted?

Answer:

If the timer interrupt arrives and the process is already in the kernel, the hardware saves only 3 values on the kernel stack, since there is no change of privilege level and hence SS and CS don't have to be saved. The rest of the interrupt path is the same: the hardware locates the interrupt handler through the IDT, and jumps to it. The interrupt handler goes through two assembly functions: `vector32` and `alltraps` that save current state of the registers on the kernel stack forming a valid trapframe, it then passes control to the `trap()` function that will continue with the timer interrupt and will enter the context switch by invoking the `swtch() function`.

4. Scheduler

   (a) (5 points) Alice is booting into an xv6 system. How many processes are running when she sees the shell command prompt after the boot (be specific: what these processes are)?

   Answer:

   Two: init and shell

   (b) (5 points) Alice is running a single-CPU xv6 system (a machine that has only one physical CPU). She creates five processes in the system. Is it possible that the same process runs for two consecutive scheduling periods, i.e., the process is running when the timer interrupt arrives, the timer interrupt handler forces a context switch into the scheduler, but the scheduler chooses the same process to run again? Explain your answer.

   Answer:

   Yes, it's possible, maybe all 4 other processes are waiting in the kernel, and hence are not in the RUNNABLE state. The scheduler will not be able to run them, and will come back to the original process.

5. Synchronization

   (a) (5 points) Alice removes `xchg()` from the `acquire()` function. Explain what happens when she boots xv6 on a multi-processor machine, i.e., machine with more than one physical CPUs.

   Answer:

   If you remove the loop that atomically sets the `locked` filed of the lock to 1 you will break the spinlock acquisition mechanisms—the spinlock will always be signalled as successfully acquired, two processes will be able to enter the same critical section simultaneously and will leave kernel data structures, e.g., process table, buffer cache, file table, etc., in an inconsistent state. The kernel will crash sooner or later.

   (b) (5 points) Now Alice decides to put the following code instead of the original `xchg()` loop in the `acquire()` function

```
for(;;) {
  if(!lk->locked)
  {
    lk->locked = 1;
    break;
  }
}
```

   She boots xv6 again on a multi-processor machine, explain what happens?

   Answer:

   Now the two lines (and several assembly instructions that are involved in implementing those lines): the one line that checks the `locked` filed and the line that sets it are not atomic. Sooner or later it will result in a race when two processes will try to acquire the same lock on two different CPUs.

(c) (10 points) Alice discovered that her CPU provides hardware support for symmetric encryption. In other words she can submit a pointer to a block of memory and a pointer to symmetric key to a hardware device that will encrypt the content of the block. Later she can get the block of memory encrypted. Unfortunately, the hardware encryption still takes tens of thousands of cycles, so she decides that instead of waiting for the encryption to finish, she can release the CPU for another process, and then receive an interrupt from the crypto device when encryption is done.

Help Alice to write a submission routine and the interrupt handler. Specifically, concentrate on correct use of synchronization primitives. Assume that Alice already implemented a low-level function `hw_encrypt()` that passes two pointers (buffer and key) to the crypto device and is now working on two functions: 1) submission function `encrypt_block()` that takes a pointer to the `cdev` structure that Alice uses to describe the state of the crypto device in her code, and two pointers (buffer and key) that she has to pass to the low-level `hw_encrypt()` function for encryption.

Your code should correctly handle synchronization between multiple processes that are trying to access the crypto device, and the interrupt handler that signals completion of encryption.

```
/* Low-level function to pass buffer and the key to hardware */
void hw_encrypt(void *buf, void *key);

struct cdev *c;

void encrypt_block(struct cdev *c, void *buf, void *key) {

  acquire(&c->lock);

  // Wait for device to become available
  while(c->flags != C_AVAIL){
    sleep(c, &c->lock);
  }

  // This process owns the device
  // Submit request for encryption
  cdev = c;
  hw_encrypt(buf, key);

  // Wait for request to finish.
  while(c->flags != C_DONE){
    sleep(c, &c->lock);
  }

  // Set device to available
  c->flags = C_AVAIL;
  // Wakeup all waiters
  wakeup(c);

  release(&c->lock);
```

```
    return;
  }

  /* Interrupt handler for the encryption device */
  void encrypt_int(void) {

    c = cdev;
    acquire(&c->lock);

    // Wake process waiting for this buf.
    c->flags = B_DONE;
    wakeup(c);

    release(&c->lock);

  }
```

6. Process memory layout

   Ben wrote a user program that grows its address space with 1 byte by calling the `sbrk()` system call. He runs the program and investigates the page table for the program before and after the call to sbrk().

   (a) (5 points) How much space has the kernel allocated?

   Answer:

   One page if it's first byte of the new page, or no pages if the page was previously allocated.

(b) (5 points) What does the pte for the new memory contain?

Answer:

The PTE for the new memory contains the physical frame number of the page that was allocated for the new page, and the flags: user accessible, writable, and present.

7. cs238P. I would like to hear your opinions about cs238P, so please answer the following questions. (Any answer, except no answer, will receive full credit.)

(a) (1 point) Grade cs238P on a scale of 0 (worst) to 10 (best)?

(b) (2 points) Any suggestions for how to improve cs238P?

(c) (1 point) What is the best aspect of cs238P?

(d) (1 point) What is the worst aspect of cs238P?