Operating Systems
Spring 2019
Final
06/12/2019
Time Limit: 4pm - 6pm

Name (Print):

- **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	5	
3	10	
4	20	
5	5	
6	5	
Total:	70	

1. Context switch. Ben is a student in cs238P. While preparing for the final exam he looks up the following question from the last year's exam:

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

   He then reads the answer to this question:

   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).

   (a) (5 points) Ben is confused by the answer. He desides to change xv6 and don't save `EBX` inside alltrap(). What will go wrong? Be specific, describe in details how the system misbehaves, and provide a concrete example.

   Answer: Several answers are accepted. First of all, Ben should be careful to implement his changes. He has to replace the `pushall` instruction with several `push` instructions that save all registers but `EBX`, and implement similar changes on the exit path, i.e., replaces `popal` with a bunch of `pop` instructions. Further, he has to change the definition of the trapframe data structure, i.e., remove the space taken by the `ebx`.

   It's ok to base your answer on the assumption that Ben forgets to change the trapframe data structure (in this case the kernel will crash ...)

   But if Ben does all the changes correctly, the user value of the `EBX` register does not get saved when the execution transitions into the kernel via an interrupt, and does not get restored on the exit path. Since kernel functions use the `EBX` register, sooner or later this will overwrite the user state of `EBX` and the user program will crash unpredictably or will exhibit some undefined behavior. For example, if `EBX` contains an array index, the changed value of `EBX` will result in an out-of-bounds access, e.g., for example to an unmapped region of memory, and a crash due to a translation of a virtual address that is not present in the page table.

   (b) (5 points) Now Ben decides to save `EBX` inside the `alltraps()` function, but don't save it inside the `swtch()` function. Again, be specific, describe in details how the system misbehaves, and provide a concrete example of how it crashes.

   Answer: The `swtch()` function implements the final steps of the context switch from a process to the scheduler and back. If the `EBX` register does not get saved and restored the value of the register will sooner or later be overwritten by the code of another process or the scheduler. I.e., sooner or later the code of the kernel will exhibit some unpredictable behavior, e.g., overwriting random data structures (e.g., if `EBX` contains a pointer to some data structure), accessing unmapped virtual addresses, repeating iterations of a loop (if `EBX` contains the loop index), etc.

(c) (5 points) Ben looks through the xv6 code trying to understand where the "context" data structure of each process is allocated. Can you help Ben by explaining this to him?

Answer: The context data structure is allocated by the `allocproc()` function on the kernel stack of the process. I.e., `allocproc()` allocates a page of memory from the kernel memory allocator for the kernel stack. It then uses top of this page for the trapframe, and the space right below the trapframe for the "context".

(d) (5 points) Ben continues preparing for the final exam. He understands that the "context" data structure serves the purpose of saving and restoring the callee registers and instruction pointer (`EIP`) of the process. Ben understands that for each process the "context" is initialized by the `allocproc()` function like this:

```
2509   sp -= sizeof *p->context;
2510   p->context = (struct context*)sp;
2511   memset(p->context, 0, sizeof *p->context);
2512   p->context->eip = (uint)forkret;
```

Can you explain why "context" is initialized with zeros?

Answer: The context contains the values of the callee saved registers (`edi`, `esi`, `ebx`, `ebp`). I.e., the `swtch()` function will load these values into the actual hardware registers switching from the scheduler to the newly allocated process. However, since the process didn't run yet, and moreover it does not expect any specific values in these registers, i.e., remember that the process will start from executing the `main()` function, that expects its arguments on the stack, but does not expect any specific values in registers, it's ok to initialize these registers with zeros. It would be fine to leave this memory uninitialized, since again the process does not expect anything in these registers, but this would introduce an information leakage vulnerability, e.g., imagine some process kept a data structure containing a password on the same physical page where context is now allocated.

(e) (5 points) Ben knows that the scheduler also has "context", but he is confused again. Can you explain at what point the "context" of the scheduler is initialized?

Answer: The context of the scheduler is initialized right when the scheduler context switches into the process for the first time. The context data structure of the scheduler is allocated on the stack of the scheduler process (i.e., it's not really allocated, the callee saved registers are simply pushed on the stack by the `swtch()` function, and the pointer to the top of the stack becomes the pointer to the "context" of the scheduler).

2. Synchronization

(a) (5 points) Along with the `acquire()` function, xv6 implements a different `acquiresleep()` function.

```
4621 void
4622 acquiresleep(struct sleeplock *lk)
4623 {
4624   acquire(&lk->lk);
4625   while (lk->locked) {
4626     sleep(lk, &lk->lk);
4627   }
4628   lk->locked = 1;
4629   lk->pid = myproc()->pid;
4630   release(&lk->lk);
4631 }
```

Can you explain the logic of the `acquiresleep()` function and the difference between `acquire()` and `acquiresleep()`?

Answer: The `acquiresleep()` function first acquires the `lk->lk` lock (line 4624). It then checks the `lk->locked` flag (line 4625), if it's set it goes to sleep passing the `lk->lk` lock as an argument to the `sleep()` function (this allows preventing the "lost wakeup" problem). It the `lk->locked` flag is not set, the function sets it in line 4628 (note, it's ok to do that, there is no race since `lk->lk` is already acquired). It finally releases the `lk->lk` lock (line 4630) and exits (at this point the "sleeplock " is acquired).

The main difference between `acquire()` and `acquiresleep()` is that `acquire()` is spinning on the lock until it acquires it (this is good for short critical section, e.g., a program on another CPU is holding the lock for a short period of time). The `acquiresleep()` function on the other hand allows the process to sleep instead of spinning. This is a natural choice in the cases when the lock will be held for a long period of time by another process in the system (e.g., in parts of the file system). Spinning would be wasteful in this case since the lock might be acquired for several time "ticks".

3. The `sleep()` function allows the process to sleep for a requested number of ticks (timer interrupts).

```
3814 int
3815 sys_sleep(void)
3816 {
3817   int n;
3818   uint ticks0;
3819
3820   if(argint(0, &n) < 0)
3821     return -1;
3822   acquire(&tickslock);
3823   ticks0 = ticks;
3824   while(ticks - ticks0 < n){
3825     if(myproc()->killed){
3826       release(&tickslock);
3827       return 1;
3828     }
3829     sleep(&ticks, &tickslock);
3830   }
3831   release(&tickslock);
3832   return 0;
3833 }
```

(a) (5 points) What does it mean for the process to go to "sleep()" in line 3829 above? Be specific, describe what happens to the process and why it's not running?

Answer: The process is not running because the `sleep()` function sets the process state to `SLEEPING` and then context switches into the scheduler picking the next process to run. Note that the scheduler will not pick a process in the `SLEEPING` state to run, hence the process will not be context switched into by the scheduler until it is woken up by the `wakeup()` function.

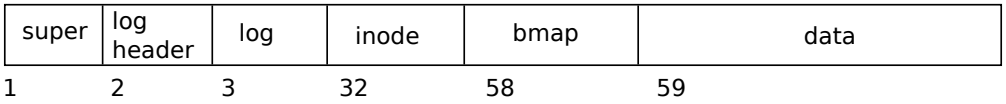
(b) (5 points) At what point this process (which part of the kernel code) "wakes up" the sleeping process?

Answer: The process is waken up from the timer interrupt handler from inside the `trap()` function. Specifically the following code wakes up every process sleeping on the address of the `ticks` variable.

```
3413   switch(tf>trapno){
3414   case T_IRQ0 + IRQ_TIMER:
3415     if(cpuid() == 0){
3416       acquire(&tickslock);
3417       ticks++;
3418       wakeup(&ticks);
3419       release(&tickslock);
3420     }
3421     lapiceoi();
3422     break;
```

4. 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.

   (a) (5 points) Every file system transaction that changes the file system write one disk block twice. What is this block (whats its block number) and why is it written twice?

   Answer: Block 2 is written twice since it contains the log header. The log header is first updated when all the other blocks are added to the log with the size of the transaction. And then after transaction is installed, the log header is updated with zero.

   (b) (5 points) If the `inode` is defined as follows, what is the maximum number of inodes this file system can have

```
4073 #define NDIRECT 12
...

4077 // Ondisk inode structure
4078 struct dinode {
4079   short type;           // File type
4080   short major;          // Major device number (T_DEV only)
4081   short minor;          // Minor device number (T_DEV only)
4082   short nlink;          // Number of links to inode in file system
4083   uint size;            // Size of file (bytes)
4084   uint addrs[NDIRECT+1];// Data block addresses
4085 };
```

   Answer: To count up the number of inodes supported by the file system we have to see how many times the `dinode` structure can fit into the area of the file system dedicated for keeping the inodes (i.e., from block 32 to block 58). The total size of this area is $(58 - 32) * 512 = 26 * 512 = 13312$ bytes.

   The size of the `dinode` data structure is 4 members of type `short` which is 2 bytes, and 14 members of type `uint` which is 4 bytes, or 64 bytes. Then the total number of inodes will be $13312/64 = 208$

(c) (5 points) What is the maximum file size the xv6 file system supports?

Answer: The xv6 supports the file size of 12 direct block and $512/4 = 128$ indirect blocks. Or $(12 + 128) * 512 = 71680$ bytes.

(d) (5 points) Alice wants to double the maximum file size xv6 supports, what does she have to do? Be specific.

Answer: Alice has to change the definition of the `dinode` structure. One way to do this is to double the number of direct and indirect blocks like this

```
4084   uint addrs[2*(NDIRECT+1)];// Data block addresses
```

An equivalent but slightly better way would be to change the definition of `NDIRECT` like this:

```
4073 #define NDIRECT 12*2
...
4084   uint addrs[NDIRECT+1*2];// Data block addresses
```

This allows her to have 24 direct blocks and 2 indirect blocks. She then has to change the `bmap()` function that converts logical offsets within the file to actual block numbers on disk to accomodate for this change. For example, she can add the following lines right after line 5435

```
5436   if(bn < 2*NINDIRECT){
5437     // Load indirect block, allocating if necessary.
5438     if((addr = ip>addrs[NDIRECT + 1]) == 0)
5439       ip>addrs[NDIRECT + 1] = addr = balloc(ip>dev);
5440     bp = bread(ip>dev, addr);
5441     a = (uint*)bp>data;
5442     if((addr = a[bn]) == 0){
5443       a[bn] = addr = balloc(ip>dev);
5444       log_write(bp);
5445     }
5446     brelse(bp);
5447     return addr;
5448   }
```

5. Processes

   (a) (5 points) Alice is running an xv6 system with two CPUs. Is it possible for the `init()` process which is created on the first CPU to run on the second CPU at some point in time? Be specific, explain why this may or may not happen.

   Answer: Yes, this is possible. Xv6 runs a scheduler on each physical CPU. Each scheduler goes through the table of processes and pics the one that can run (i.e., is in the `RUNNABLE` state). Sooner or later the `init` process will be picked up by the scheduler of the second CPU, will be context switched into, and will run.

6. 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?