Principles of Operating Systems	**Name (Print):**	
Fall 2019	**Seat:**	SEAT
Final	**Left person:**	
12/13/2019	**Right person:**	

Time Limit: 8:00am – 10:00pm

- **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.**
- **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.**

Problem	Points	Score
1	10	
2	15	
3	15	
4	15	
5	17	
6	15	
7	4	
Total:	91	

1. Operating system interface

   (a) (10 points) Write code for a simple program that implements the following pipeline:

   ```
   cat main.c | grep "main" | wc
   ```

   I.e., you program should start several new processes. One for the `cat main.c` command, one for `grep main`, and one for `wc`. These processes should be connected with pipes that `cat main.c` redirects its output into the `grep "main"` program, which itself redirects its output to the `wc`.

   ```
   forked pid:811
   forked pid:812
   fork failed, pid:-1
   ```

2. Processes and system calls

   Alice is implementing a fork bomb, i.e., she tries to create as many processes in xv6 as possible.

```
#include "types.h"
#include "stat.h"
#include "user.h"

int
main(int argc, char *argv[])
{
  int pid;

  for(;;) {
        pid = fork();
        if(pid == -1) {
                printf(1, "fork failed, pid:%d\n", pid);
                exit();
        } else if (pid) {
                printf(1, "forked pid:%d\n", pid);
        } else {
                for (;;) {
                        sleep(1);
                }
        };
  }
  exit();
}
```

   (a) (5 points) She boots into xv6 and right away starts her program `forkbomb` in shell. She sees the following output:

```
$ forkbomb
forked pid:4
forked pid:5
forked pid:6
...
forked pid:61
forked pid:62
forked pid:63
forked pid:64
fork failed, pid:-1
```

   This means that her program forked 61 times. She realizes that xv6 kernel has an array of proc data structures of size 64, but still she is confused: why did she fork only 61 times? Please explain why.

   In xv6, process identifiers start with 1. When the system boots, the init process is created and it gets pid of 1, the process itself first runs the userinit code and then executes the `exec()` system call (`exec()` however, doesn't change the pid of the process). The init process creates shell that gets pid of 2. Alice then starts her forkbomb program that gets

pid of 3. The first forked child gets pid of 4. Alice keeps forking until she uses all processes in the proc array, i.e., up until pid 64.

(b) (10 points) Alice quickly changes the size of the array to 4096, reboots, and runs her program again. How many times she will be able to fork now? Explain your reasoning.

Each process consumes some number of physical pages, and eventually xv6 will run out of physical memory. To understand how long it will take you need to estimate a number of physical pages used for each new process. A process needs a page for 1) kernel stack, 2) text and data sections (one page), 3) guard page, 4) stack page, 5) root of the page table page (page table directory), and N pages for page table pages. What is this N? Each process maps all memory from 0 to PHYSTOP, and a small BIOS region at the very top of the address space. One page table page maps 4MBs of virtual memory, so our formula is: (PHYSTOP + (size of BIOS area))/(4096*1024) = 56 + 7 = 63. Hence, the total pages for one fork is 5 + 63 = 68.

Now lets estimate how much physical memory is available when Alice starts her first fork. The total number of pages on the allocators free list is PHYSTOP - kernel.end (we don't ask you to know the exact value kernel end, but a quick check with readelf is a bonus, it's 0x801154a8 virtual or 0x1154a8 physical, if we round it up to the next page it's 0x116000). Then the system has a total of 57066 pages on the allocator list. When the system boots, each CPU creates a kernel page table, lets assume, Alice runs a 2 CPU system and all CPUs besides the first one need a page for the kernel stack. This means that 63x2 + 1 pages are used before the system starts the init process. Before the first child is forked, 3 other processes are created: init, sh, and forkbomb, this is 68x3. The total number of pages used is 63x2 + 1 + 68x3 = 331, the total free pages left is 57066 - 331 = 56735.

Now we have all the data to estimate how many times Alice's program will fork: 56735/68 = 834.

3. Context switch

   The `swtch()` function that implements the core of the context switch saves only 4 registers on the stack

```
.globl swtch
swtch:
  movl 4(%esp), %eax
  movl 8(%esp), %edx

  # Save old callee-saved registers
  pushl %ebp
  pushl %ebx
  pushl %esi
  pushl %edi

  # Switch stacks
  movl %esp, (%eax)
  movl %edx, %esp

  # Load new callee-saved registers
  popl %edi
  popl %esi
  popl %ebx
  popl %ebp
  ret
```

   The context data structure has 5 registers:

```
struct context {
  uint edi;
  uint esi;
  uint ebx;
  uint ebp;
  uint eip;
};
```

   (a) (3 points) How does the EIP register gets saved and restored?

   The EIP is pushed on the stack when swtch() is called. Since it's pushed on the stack it's accessible as part of the `struct context` data structure, and it will be restored into the EIP hardware register when `swtch()` executes the `ret` instruction.

(b) (3 points) How does the kernel EAX register gets saved and restored?

The kernel EAX register is a caller-saved register, if the function that calls `swtch()` uses this register after the `swtch()` invocation, it will push the register on the stack before invoking `swtch()` and will pop it after.

(c) (3 points) How does user-level EAX register gets saved and restored?

The user EAX is pushed on the kernel stack inside the `alltrap()` function that pushes all registers on the kernel stack of the process. The register is accessible as part of the `trapframe` data structure. It is restored from the kernel stack on exit from the interrupt.

(d) (3 points) How does the kernel ESP register gets saved and restored?

The kernel ESP register is saved inside the `swtch()` function as a pointer to the context of currently running process, i.e., the following line

```
# Switch stacks
movl %esp, (%eax)
```

saves ESP into `proc->context` making this pointer point to the top of the stack of the process that was running before the `swtch()` was invoked.

It is restored with the following line that loads `proc->context` into ESP

```
movl %edx, %esp
```

(e) (3 points) How does user-level ESP register gets saved and restored?

The user ESP pointer is saved as part of the interrupt transition (i.e., the hardware pushes 5 things on the stack, when interrupt arrives and one of them is ESP).

It is restored with the `iret` instruction when execution returns into user-level.

4. System calls

   (a) (10 points) What does the user stack looks like when the `read()` system call is invoked, i.e., when the execution is already in kernel and it reaches the `sys_read()` function. Draw a diagram, provide a short description for every value on the stack. Remember the `read()` system call has the following signature:

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

   ```
   |  3rd arg (int)    |  | stack growth direction
   |  2nd arg (void *) |  v
   |  1st arg (int)    |
   |  return addr      |
   ```

   User calls the `read()` function that internally invokes `int` instruction. The return address for that invocation is on the stack (note this function does not maintain the stack frame as it's automatically generated in the `usys.S` file). The arguments for the function are pushed on the stack before invocation.

   (b) (5 points) If the execution is inside a system call, e.g., inside the `sys_read()` function, and we count from the bottom of the kernel stack (here the top of the stack is pointed by the ESP register, and bottom is the end of the kernel stack page), bytes 0-3 from the bottom contain the `ss` (stack segment of the user program when it entered the kernel with the system call), bytes 4-7 contiain the user ESP value, etc.. Then what do bytes 24-27 contain (explain your answer)?

   System call number, which is accessible as `tf->trapno`.

5. Global Descriptor Table (GDT)

   (a) (5 points) How GDT is used in xv6, i.e., what role does it play in the system?

   Xv6 uses flat segment model, i.e., all code and data segemnts are configured with base 0x0, and the limit of 0xffffffff. Xv6 uses GDT for two purposes: 1) to ensure that user code runs with privilege level 3, and 2) to keep track of location of the kernel stack of currently running process (i.e., during the interrupt we don't trust user code to have a meaningful value of ESP, hence, the hardware fetches the kernel ESP from the TSS segment that points to the TSS table that has a pointer to the kernel stack.

   (b) (5 points) How many global descriptor tables xv6 creates?

   Xv6 creates one GDT per physical CPU. This is needed as each CPU may run a process, and hence it needs a unique kernel stack if interrupt or a system call is executed on that CPU.

(c) (7 points) Explain lines 1870–1874 in the `switchuvm()` function (be specific).

```
1859 void
1860 switchuvm(struct proc *p)
1861 {
1862   if(p == 0)
1863     panic("switchuvm: no process");
1864   if(p->kstack == 0)
1865     panic("switchuvm: no kstack");
1866   if(p->pgdir == 0)
1867     panic("switchuvm: no pgdir");
1868
1869   pushcli();
1870   mycpu()->gdt[SEG_TSS] = SEG16(STS_T32A, &mycpu()->ts,
1871                                 sizeof(mycpu()->ts)-1, 0);
1872   mycpu()->gdt[SEG_TSS].s = 0;
1873   mycpu()->ts.ss0 = SEG_KDATA << 3;
1874   mycpu()->ts.esp0 = (uint)p->kstack + KSTACKSIZE;
1875   // setting IOPL=0 in eflags *and* iomb beyond the tss segment limit
1876   // forbids I/O instructions (e.g., inb and outb) from user space
1877   mycpu()->ts.iomb = (ushort) 0xFFFF;
1878   ltr(SEG_TSS << 3);
1879   lcr3(V2P(p->pgdir));  // switch to processs address space
1880   popcli();
1881 }
```

Every time xv6 switches between processes it updates the pointer to the kernel stack (each process has its own kernel stack) in the TSS table.

Line 1870 loads the TSS segment with the base of the address at which `mycpu()->ts` is located.

Line 1873 configures the stack segment SS for CPL 0 to point to the kernel data segment.

Line 1874 sets the kernel stack pointer in the TSS to point to the kernel stack of the current process (`p->kstack + KSTACKSIZE`).

6. Interrupts

(a) (5 points) Can an interrupt preempt execution of a system call, i.e., can the interrupt be delivered and processed why the system executes a system call (explain your answer)?

Yes, xv6 configures vector 64 in the IDT (64 is used for system calls) to leave the interrupts enabled.

(b) (5 points) Can an interrupt preempt execution of another interrupt (explain your answer)?

No, xv6 configures all other vectors in the IDT besides 64 to disable interrupts on the interrupt transition. Xv6 never re-enables interrupts on the interrupt processing path (it will always use push and pop CLI). The only exception is a non-maskable interrupt that can be delivered even when interrupts are disabled. Xv6 however does not handle non-maskable interrupts (if it ends up running on the hardware that delivers an NMI it will panic inside the `trap()` function.

(c) (5 points) Xv6 creates a kernel stack for each process. Why can't we simply create one kernel stack per physical CPU?

Xv6 allows processes to sleep inside system calls. I.e., a process can enter the kernel with a system call, e.g., `read()` and yield execution to another process. Some state of the first process is still saved on the kernel stack of that process.

7. 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) What is the best aspect of cs143A?

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

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