cs5460/6460: Operating Systems **Name (Print):**

Spring **UID:**

Time Limit: 8:00am – 10:00am

- **Don't forget to write your name on this exam, and put your uid on each page.**
- **This is an open book, open notes exam. But no devices, we allow calculators and kindle readers that can save paper but can't access the web (well, we know kindle can access the web, but please don't do it for the sake of the trees.**
- **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 (yep, you're reading this twice!).**

Problem	Points	Score
1	10	
2	10	
3	15	
4	20	
5	10	
6	20	
7	10	
Total:	95	

1. Not again: Shell and OS interfaces

   xv6 implements a shell similar to the one you built in HW1, the following is the PIPE command

```
9104 // Execute cmd. Never returns.
9105 void
9106 runcmd(struct cmd *cmd)
9107 {
9108   int p[2];
9109   struct backcmd *bcmd;
9110   struct execcmd *ecmd;
9111   struct listcmd *lcmd;
9112   struct pipecmd *pcmd;
9113   struct redircmd *rcmd;
9114
9115   if(cmd == 0)
9116     exit();
9117
9118   switch(cmd->type){
9119   default:
9120     panic("runcmd");
9121
9122   case EXEC:
9123     ecmd = (struct execcmd*)cmd;
9124     if(ecmd->argv[0] == 0)
9125       exit();
9126     exec(ecmd->argv[0], ecmd->argv);
9127     printf(2, "exec %s failed\n", ecmd->argv[0]);
9128     break;
9129
9130   case REDIR:
9131     rcmd = (struct redircmd*)cmd;
9132     close(rcmd->fd);
9133     if(open(rcmd->file, rcmd->mode) < 0){
9134       printf(2, "open %s failed\n", rcmd->file);
9135       exit();
9136     }
9137     runcmd(rcmd->cmd);
9138     break;
9139
...
9149
9150   case PIPE:
9151     pcmd = (struct pipecmd*)cmd;
9152     if(pipe(p) < 0)
9153       panic("pipe");
9154     if(fork1() == 0){
9155       close(1);
9156       dup(p[1]);
9157       close(p[0]);
9158       close(p[1]);
9159       runcmd(pcmd->left);
9160     }
9161     if(fork1() == 0){
9162       close(0);
9163       dup(p[0]);
9164       close(p[0]);
```

```
9165       close(p[1]);
9166       runcmd(pcmd->right);
9167     }
9168     close(p[0]);
9169     close(p[1]);
9170     wait();
9171     wait();
9172     break;
...
9179   }
9180   exit();
9181 }
```

(a) (5 points) What happens if we comment out line 9164? Note: While the question is almost identical to the one on midterm there are some important differences and I want you to understand them by answering nearly the same question. .

Answer: Line 9164 closes the read end of the pipe (`p[0]`). Similar to the midterm nothing really bad happens. The process on the right side of the pipe is reading from the pipe. So when the process on the left side is done writing the data and closes it's write end, the process on the right correctly detects that communication via the pipe is done and exits. The pipe will be closed when the right side exits.

(b) (5 points) What happens if we comment out line 9165?

Answer: Line 9165 closes the write end of the pipe (`p[1]`). This becomes really problematic for the process on right side of the pipe, i.e., the side that is reading from the pipe: even when the process on left side is done writing to the pipe and closes it's write end this `p[1]` end remains open. Hence the right side of the pipe cannot detect that there will be no more writes to the pipe. The code will wait on the pipe trying to read more data forever. Even though the process on the left side of the pipe exits, the shell will continue waiting for the program on the right side and will never print the new command prompt.

2. Context switch

Below is the code of the `swtch` function from xv6:

```
3050 # Context switch
3051 #
3052 # void swtch(struct context **old, struct context *new);
3053 # 3054 # Save the current registers on the stack, creating 3055 # a struct
3054 # Save the current registers on the stack, creating
3055 # a struct context, and save its address in *old.
3056 # Switch stacks to new and pop previously-saved registers.
3057
3058 .globl swtch
3059 swtch: 3060   # Save old callee-saved registers 3061   pushq %rbp 3062
3060   # Save old callee-saved registers
3061   pushq %rbp
3062   pushq %rbx
3063   pushq %r12
3064   pushq %r13
3065   pushq %r14
3066   pushq %r15
3067
3068   # Switch stacks
3069   movq %rsp, (%rdi)
3070   movq %rsi, %rsp
3071
3072   # Load new callee-saved registers
3073   popq %r15
3074   popq %r14
3075   popq %r13
3076   popq %r12
3077   popq %rbx
3078   popq %rbp
3079   ret
```

Imagine you're running a single-CPU xv6 system.

(a) (5 points) What is stored in the `rdi` register, when the `swtch` function is executed for the **second time** since the system boot?

Answer: The very first invocation of `swtch` switches from the scheduler to the process, so the second invocation will be switching from the process back to the scheduler. The `swtch()` function takes two arguments: 1) a pointer to a pointer to a context data structure in the `rdi` register – this pointer is updated to point to the stack of the process from which we are switching away (`swtch()` saves the context on the stack), 2) a pointer to the context we are switching to in `rsi` register.

Since this is the second invocation of `swtch()`, `rdi` contains the pointer to the `context` filed in the process data structure that represents the current process. This pointer is updated to point to the stack on which the context is saved.

(b) (5 points) What is stored in the `rsi` register, same as above, the `swtch` function executes for the **second time** since the system boot?

Answer: See above, i.e., `rsi` contains the pointer to the context we are switching to. More specifically, the context of the scheduler of the current CPU.

3. Processes

Here is the definition of the data structure that represents a process in xv6.

```
2352 // Per-process state
2353 struct proc {
2354   uint64 sz; // Size of process memory (bytes)
2355   pml4e_t* pml4; // Page table
2356   char *kstack; // Bottom of kernel stack for this process
2357   enum procstate state; // Process state
2358   int pid; // Process ID
2359   struct proc *parent; // Parent process
2360   struct trapframe *tf; // Trap frame for current syscall
2361   struct context *context; // swtch() here to run process
2362   void *chan; // If non-zero, sleeping on chan
2363   int killed; // If non-zero, have been killed
2364   struct file *ofile[NOFILE]; // Open files
2365   struct inode *cwd; // Current directory
2366   char name[16]; // Process name (debugging)
2367 };
```

(a) (5 points) When the process is not running, i.e., the kernel context switched the process out while switching to a different process, where does the `context` field of the process struct point to?

Answer: In xv6 the context is always saved at the top of the process' kernel stack.

(b) (5 points) When the process is not running, i.e., the kernel context switched the process out, where does the `tf` field point to?

Answer: `tf` also points to the kernel stack. When the process is preempted with an interrupt (or enters the kernel) with a system call, the trapframe is created on the kernel stack by the `vectorXX` and `alltraps` functions.

(c) (5 points) What is the role of the `chan` field and how is it used in the kernel?

Answer: The kernel uses `chan` as an unique identifier for sleeping and waking up processes. I.e., the wakeup function goes through the list of all processes in the SLEEPING state and wakes them up, i.e., makes them runnable if their `chan` field equals the argument passed to the `wakeup()` function.

4. Kernel organization

   Imagine you've booted into the xv6 shell on a two-CPU xv6 system.

   (a) (5 points) How many stacks are allocated in the system (explain how each stack is used)?

   Answer: Temporary boot stack, one kernel stack for each CPU to run the scheduler, a pair of kernel and user stacks for each of the processes running in the system (if we booted into shell, it's init and shell so two). The total is $1 + 2 + 4 = 7$ stacks.

   (b) (5 points) How many IDTs are allocated in the system (explain how each IDT is used)?

   Answer: One, xv6 uses one global IDT.

   (c) (5 points) How many GDTs are allocated in the system (explain how each GDT is used)?

   Answer: One per CPU. A per-CPU GDT holds a pointer to the per-CPU TSS which will hold the pointer to the kernel stack of the currently running process.

   (d) (5 points) How many TSSs are allocated in the system (explain how each TSS is used)?

   Answer: One per CPU. A per-CPU TSS holds a pointer to the kernel stack of the currently running process. Every time xv6 switches from one process to another it updates the TSS on the current CPU to point to the kernel stack of the current process.

5. Interrupts and exceptions

   Here is a listing of the `tvinit()` function which initializes IDT in xv6:

```
3366 void
3367 tvinit(void)
3368 {
3369   int i;
3370
3371   for(i = 0; i < 256; i++)
3372     SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);
3373   SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, vectors[T_SYSCALL], DPL_USER);
3374
3375   initlock(&tickslock, "time");
3376 }
```

   (a) (5 points) What is the role of line 3373 above (explain your answer)?

   Answer: This line achieves two things: 1) it enables invocation of the system call via the `int 64` instruction by setting the descripor privilege level to `DPL_USER`, 2) it configures the IDT to leave the interrupts enabled after entering the kernel with the syscall (in xv6 system calls are running with system calls enabled), all other interrupts disable subsequent interrupts).

   (b) (5 points) What happens if the user code invokes a timer interrupt like `int 32` (explain your answer)?

   Answer: Since all interrupts besides the system call are configured with `DPL_KERNEL` (btw, it would be nice to use `DPL_KERNEL` in the code instead of 0) the user cannot invoke the timer interrupt with the `int` instruction. Such attempt will trigger the general protection fault exception.

6. xv6 process and memory organization

   Every time xv6 creates a new process quite a bit of memory is wasted, i.e., the memory which is allocated for the nodes of the page table that are involved in mapping the kernel part of the address space. It's wasted as the mappings are the same across all processes.

   Alice argues it would be nice to fix this ineficiency in the kernel and figure out a way to share those mappings across all processes in the system. She looks at the following xv6 functions involved in setting up the kernel part of the page table.

```
1885 // Set up kernel part of a page table.
1886 pml4e_t*
1887 setupkvm(void)
1888 {
1889   pml4e_t *pml4;
1890   struct kmap *k;
1891
1892   if((pml4 = (pml4e_t*)kalloc()) == 0)
1893     return 0;
1894   memset(pml4, 0, PGSIZE);
1895   if (P2V(PHYSTOP) > DEV_P2V(DEVSPACE))
1896     panic("PHYSTOP too high");
1897   for(k = kmap; k < &kmap[NELEM(kmap)]; k++)
1898     if(mappages(pml4, k->virt, k->phys_end - k->phys_start,
1899                 (uint64)k->phys_start, k->perm) < 0) {
1900         freevm(pml4);
1901         return 0;
1902     }
1903   return pml4;
1904 }
...
1800 // Create PTEs for virtual addresses starting at va that refer to
1801 // physical addresses starting at pa. va and size might not
1802 // be page-aligned.
1803 static int
1804 mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm)
1805 {
1806   char *a, *last;
1807   pte_t *pte;
1808
1809   a = (char*)PGROUNDDOWN((uint64)va);
1810   last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
1811   for(;;){
1812     if((pte = walkpml4(pml4, a, 1, 0)) == 0)
1813       return -1;
1814     if(*pte & PTE_P)
1815       panic("remap");
1816     *pte = pa | perm | PTE_P;
1817     if(a == last)
1818       break;
1819     a += PGSIZE;
1820     pa += PGSIZE;
1821   }
1822   return 0;
1823 }
...
1775 // Return the address of the PTE in page table pml4
1776 // that corresponds to virtual address va. If alloc!=0,
1777 // create any required page table pages.
```

```
1778 // On failure (return value equals 0), missing is set to the level
1779 // in which the table was not present.
1780 static pte_t *
1781 walkpml4(pml4e_t *pml4, const void *va, int alloc, int *missing)
1782 {
1783   return walkpglevel(pml4, va, alloc, 4, missing);
1784 }
...
1731 // Return the address of the PTE in page table of the specified level
1732 // that corresponds to virtual address va. If alloc!=0,
1733 // create any required page table pages.
1734 static pte_t *
1735 walkpglevel(uint64 *pgdir, const void *va, int alloc, int level,
1736 int *missing)
1737 {
1738   uint64 *entry;
1739   uint64 *next_pgdir;
1740
1741   switch (level){
1742   case 4:
1743     entry = &pgdir[PML4X(va)];
1744     break;
1745   case 3:
1746     entry = &pgdir[PDPTX(va)];
1747     break;
1748   case 2:
1749     entry = &pgdir[PDX(va)];
1750     break;
1751   case 1:
1752     return &pgdir[PTX(va)];
1753   default:
1754     panic("walkpglevel");
1755   }
1756
1757   if(*entry & PTE_P){
1758     next_pgdir = (uint64*)P2V(PTE_ADDR(*entry));
1759   } else {
1760     if(!alloc || (next_pgdir = (uint64*)kalloc()) == 0){
1761       if (missing)
1762         *missing = level;
1763       return 0;
1764     }
1765     // Make sure all those PTE_P bits are zero.
1766     memset(next_pgdir, 0, PGSIZE);
1767     // The permissions here are overly generous, but they can
1768     // be further restricted by the permissions in the page table
1769     // entries, if necessary.
1770     *entry = V2P(next_pgdir) | PTE_P | PTE_W | PTE_U;
1771   }
1772   return walkpglevel(next_pgdir, va, alloc, level - 1, missing);
1773 }
...
1850 // There is one page table per process, plus one that?s used when
1851 // a CPU is not running any process (kpml4). The kernel uses the
1852 // current process?s page table during system calls and interrupts;
1853 // page protection bits prevent user code from using the kernel?s
1854 // mappings.
1855 //
1856 // setupkvm() and exec() set up every page table like this:
```

```
1857 //
1858 // 0..KERNBASE: user memory (text+data+stack+heap), mapped to
1859 // phys memory allocated by the kernel
1860 // KERNBASE..KERNBASE+EXTMEM: mapped to 0..EXTMEM (for I/O space)
1861 // KERNBASE+EXTMEM..data: mapped to EXTMEM..V2P(data)
1862 // for the kernel?s instructions and r/o data 1863 //
1863 // data..KERNBASE+PHYSTOP: mapped to V2P(data)..PHYSTOP,
1864 // rw data + free physical memory
1865 // 0xfe000000..0: mapped direct (devices such as ioapic)
1866 //
1867 // The kernel allocates physical memory for its heap and for user memory
1868 // between V2P(end) and the end of physical memory (PHYSTOP)
1869 // (directly addressable from end..P2V(PHYSTOP)).
1870
1871 // This table defines the kernel?s mappings, which are present in
1872 // every process?s page table.
1873 static struct kmap {
1874 void *virt;
1875 uint64 phys_start;
1876 uint64 phys_end;
1877 int perm;
1878 } kmap[] = {
1879   { (void*)KERNBASE, 0, EXTMEM, PTE_W}, // I/O space
1880   { (void*)KERNLINK, V2P(KERNLINK), V2P(data), 0}, // kern text+rodata
1881   { (void*)data, V2P(data), PHYSTOP, PTE_W}, // kern data+memory
1882   { (void*)DEVBASE, DEVSPACE, 0, PTE_W}, // more devices
1883 };
```

(a) (20 points) Change the above functions to implement the fix Alice is talking about. Note, there are multiple ways and strategies for implementing this change. Feel free to suggest your own but justify why it makes sense.

Answer: The high level idea here is to create a page table that maps the kernel part of the address space once, e.g., like when we create the very first `kpml4` page table that we use when xv6 is not running any process but runs the scheduler. And then later use existing pml3 nodes and simply copy (re-use) pointers to those pml3 nodes in every process.

The code can add a `is_first` flag to make sure the code works as normal for the first page table, and either pass a pointer to the reference `kpml4` page table or use it as a global variable to copy the mappings if the address falls above or equal to the kernel range (i.e., `KERNBASE`).

```
1804 mappages(pml4e_t *pml4, void *va, uint size, uint64 pa, int perm, int first, pml4* kpml4)
1805 {
1806   char *a, *last;
1807   pte_t *pte;
1808
1809   a = (char*)PGROUNDDOWN((uint64)va);
1810   last = (char*)PGROUNDDOWN(((uint64)va) + size - 1);
1811   for(;;){
        if(first == 0 && a >= KERNBASE) {
          pml4[PML4X(a)] = kpml4[PML4X(a)];
        } else{
1812      if((pte = walkpml4(pml4, a, 1, 0)) == 0)
1813        return -1;
1814      if(*pte & PTE_P)
1815        panic("remap");
1816      *pte = pa | perm | PTE_P;
       }
```

```
1817      if(a == last)
1818        break;
1819      a += PGSIZE;
1820      pa += PGSIZE;
1821    }
1822    return 0;
1823  }
```

You then patch the invocation of `mappages()` in `setupkvm()` to pass `first` equal to 0, and `kpml4`. Of course other invocations of `mappages()` have to be patched too, as well as user page table clean up routines (i.e., we do not deallocate page table nodes in the range above `KERNBASE`).

7. File systems

Below is the code of the `exec()` function that implements exec system call:

```
6609 int
6610 exec(char *path, char **argv)
6611 {
6612   char *s, *last;
6613   int i, off;
6614   uint64 argc, sz, sp, ustack[3+MAXARG+1];
6615   struct elfhdr elf;
6616   struct inode *ip;
6617   struct proghdr ph;
6618   pml4e_t *pml4, *oldpml4;
6619   struct proc *curproc = myproc();
6620
6621   begin_op();
6622
6623   if((ip = namei(path)) == 0){
6624     end_op();
6625     cprintf("exec: fail\n");
6626     return -1;
6627   }
6628   ilock(ip);
6629   pml4 = 0;
6630
6631   // Check ELF header
6632   if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
6633     goto bad;
6634   if(elf.magic != ELF_MAGIC)
6635     goto bad;
6636
6637   if((pml4 = setupkvm()) == 0)
6638     goto bad;
6639
6640   // Load program into memory.
6641   sz = 0;
6642   for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
6643     if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
6644       goto bad;
6645     if(ph.type != ELF_PROG_LOAD)
6646       continue;
6647     if(ph.memsz < ph.filesz)
6648       goto bad;
6649     if(ph.vaddr + ph.memsz < ph.vaddr)
6650       goto bad;
6651     if((sz = allocuvm(pml4, sz, ph.vaddr + ph.memsz)) == 0)
6652       goto bad;
6653     if(ph.vaddr % PGSIZE != 0)
6654       goto bad;
6655     if(loaduvm(pml4, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
6656       goto bad;
6657   }
6658   iunlockput(ip);
6659   end_op();
6660   ip = 0;
6661
6662   // Allocate two pages at the next page boundary.
6663   // Make the first inaccessible. Use the second as the user stack.
6664   sz = PGROUNDUP(sz);
```

```
6665   if((sz = allocuvm(pml4, sz, sz + 2*PGSIZE)) == 0)
6666     goto bad;
6667   clearpteu(pml4, (char*)(sz - 2*PGSIZE));
6668   sp = sz;
6669   ...
```

(a) (10 points) Why is code between lines 6621 and 6659 is executed under the file system journaling transaction (i.e., between `begin_op()/end_op()`? After all it seems that the code is only reading the ELF file? While this transaction is needed?

Answer: Despite the fact that `exec()` does not change the inode of the file, it actually only reads it it has to lock it by incrementing a reference counter to the inode. This quarantees that the inode is alive, i.e., not deleted by another process. However, in case the inode is actually deleted concurrently by another process the actual removal from the file system will happen when the inode counter drops to 0, i.e., when it's released (line 6658). The removal of the inode will trigger writes to the disk – clean up of all blocks used by the inode (writes to the free block area), release of the inode in the inode array (writes to the inode area). To ensure consistency of the file system in case of a power failure, all these writes should be performed as a transaction.