CS 238P
Spring 2019
Midterm
05/15/2018
Time Limit: 6:30pm - 7:50pm

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 you 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.

Problem	Points	Score
1	30	
2	10	
3	35	
4	5	
Total:	80	

1. OS Interfaces

   Xv6 imlements the PIPE case in its shell in the following way:

```
8650   case PIPE:
8651     pcmd = (struct pipecmd*)cmd;
8652     if(pipe(p) < 0)
8653       panic("pipe");
8654     if(fork1() == 0){
8655       close(1);
8656       dup(p[1]);
8657       close(p[0]);
8658       close(p[1]);
8659       runcmd(pcmd>left);
8660     }
8661     if(fork1() == 0){
8662       close(0);
8663       dup(p[0]);
8664       close(p[0]);
8665       close(p[1]);
8666       runcmd(pcmd>right);
8667     }
8668     close(p[0]);
8669     close(p[1]);
8670     wait();
8671     wait();
8672     break;
```

   (a) (5 points) Explain what lines 8654–8659 are doing?

   Answer: Line 8654 forks a new process and checks if execution is inside the "child". If yes the code closes the standard output (descriptor 1) (line 8655), duplicates the write and of the pipe into the standard output (line 8656), and closes both read and write ends of the pipe (lines 8657 and 8658). It then execs the left part of the "pipe" with the `runcmd` function (line 8659).

(b) (10 points) Bob who likes to optimize his code decides that it's not necessary to fork twice, after all the `fork()` system call takes time. He re-writes his xv6 shell like this

```
case PIPE:
  pcmd = (struct pipecmd*)cmd;
  if(pipe(p) < 0)
    panic("pipe");
  if(fork1() == 0){
    close(1);
    dup(p[1]);
    close(p[0]);
    close(p[1]);
    runcmd(pcmd>left);
  }
  wait()
  close(0);
  dup(p[0]);
  close(p[0]);
  close(p[1]);
  runcmd(pcmd>right);
  break;
```

He runs it a couple of times and everything seems to work, so he shows his optimization to Alice, an experienced UNIX hacker. Alice however says that Bob's code is horribly wrong. Can you explain what she means, and what is wrong with Bob's code?

Answer: Bob's code waits for the left side of the pipe to finish. Since pipes have finite capacity, e.g., 512 bytes in xv6, some left sides finish successfully, but longer outputs fill the pipe completely before finishing. In this case, the left side will be waiting for the right side to clean up some space by reading data from the pipe. However, since the right side of the pipe does not start before the left finishes this never happens. TLDR: if left side of the pipe produces an output larger than the size of the pipe's buffer, Bob's code gets stuck forever.

(c) (15 points) Write a program that uses the UNIX system call API, as described in Chapter 0 of the xv6 book. The program `tee` that reads from standard input and writes to standard output and to a file that is passed as a command line argument.

2. Basic page tables.

   Consider the following 32-bit x86 page table setup.

   `%cr3 holds 0x00000000.`

   The Page Directory Page at physical address 0x00000000:

   `PDE 0: PPN=0x00001, PTE_P, PTE_U, PTE_W`

   ... all other PDEs are zero

   The Page Table Page at physical address 0x00001000 (which is PPN 0x00001):

```
PTE 0: PPN=0x00000, PTE_P, PTE_U, PTE_W
PTE 1: PPN=0x00001, PTE_P, PTE_U, PTE_W
```

   ... all other PTEs are zero

   (a) (5 points) What are all virtual addresses mapped by this page table?

   Answer: The page table maps two consequtive virtual pages (addresses 0x0-0x2000) to two consequtive physical pages (physical addresses 0x0-0x2000).

   (b) (5 points) Extend this page table to map virtual address 0x100000 which is 1MB.

   Answer: Lets assume that we would like to map the virtual address 0x100000 to physical address 0x0. Then we need to convert 0x100000 into 10:10:12 form to see which entries of the page table we need to use for this mapping. The top 10 bits are 0x0, middle ten are 0x100 (this is 1, and the lower 12 bits are 0x0. The page table should be extended with the following entry:

   `PTE 256: PPN=0x00000, PTE_P, PTE_U, PTE_W`

3. Stack and calling conventions.

   xv6 implements exec() system call like this:

```
#include "types.h"
#include "param.h"
#include "memlayout.h"
#include "mmu.h"
#include "proc.h"
#include "defs.h"
#include "x86.h"
#include "elf.h"

int
exec(char *path, char **argv)
{
  char *s, *last;
  int i, off;
  uint argc, sz, sp, ustack[3+MAXARG+1];
  struct elfhdr elf;
  struct inode *ip;
  struct proghdr ph;
  pde_t *pgdir, *oldpgdir;
  struct proc *curproc = myproc();

  begin_op();

  if((ip = namei(path)) == 0){
    end_op();
    cprintf("exec: fail\n");
    return 1;
  }
  ilock(ip);
  pgdir = 0;

  // Check ELF header
  if(readi(ip, (char*)&elf, 0, sizeof(elf)) != sizeof(elf))
    goto bad;
  if(elf.magic != ELF_MAGIC)
    goto bad;

  if((pgdir = setupkvm()) == 0)
    goto bad;

  // Load program into memory.
  sz = 0;
  for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
      goto bad;
```

```
    if(ph.type != ELF_PROG_LOAD)
      continue;
    if(ph.memsz < ph.filesz)       goto bad;
    if(ph.vaddr + ph.memsz < ph.vaddr)
      goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
      goto bad;
    if(ph.vaddr % PGSIZE != 0)
      goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
      goto bad;
  }
  iunlockput(ip);
  end_op();
  ip = 0;

  // Allocate two pages at the next page boundary.
  // Make the first inaccessible.  Use the second as the user stack.
  sz = PGROUNDUP(sz);
  if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
  clearpteu(pgdir, (char*)(sz  2*PGSIZE));
  sp = sz;

  // Push argument strings, prepare rest of stack in ustack.
  for(argc = 0; argv[argc]; argc++) {
    if(argc >= MAXARG)
      goto bad;
    sp = (sp  (strlen(argv[argc]) + 1)) & ~3;
    if(copyout(pgdir, sp, argv[argc], strlen(argv[argc]) + 1) < 0)
      goto bad;
    ustack[3+argc] = sp;
  }
  ustack[3+argc] = 0;

  ustack[0] = 0xffffffff;  // fake return PC
  ustack[1] = argc;
  ustack[2] = sp  (argc+1)*4;  // argv pointer

  sp = (3+argc+1) * 4;
  if(copyout(pgdir, sp, ustack, (3+argc+1)*4) < 0)
    goto bad;

  // Save program name for debugging.
  for(last=s=path; *s; s++)
    if(*s == /)
      last = s+1;
  safestrcpy(curproc>name, last, sizeof(curproc>name));
```

```
6694
6695   // Commit to the user image.
6696   oldpgdir = curproc>pgdir;
6697   curproc>pgdir = pgdir;
6698   curproc>sz = sz;
6699   curproc>tf>eip = elf.entry
6700   curproc>tf>esp = sp;
6701   switchuvm(curproc);
6702   freevm(oldpgdir);
6703   return 0;
6704
6705  bad:
6706   if(pgdir)
6707     freevm(pgdir);
6708   if(ip){
6709     iunlockput(ip);
6710     end_op();
6711   }
6712   return 1;
6713 }
```

(a) (5 points) Which lines in this code create the guard page? Explain what these lines do.

Answer: The following lines create the guard and stack pages:

```
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(pgdir, sz, sz + 2*PGSIZE)) == 0)
6666     goto bad;
6667   clearpteu(pgdir, (char*)(sz  2*PGSIZE));
6668   sp = sz;
```

Line 6665 allocates two pages right above the address `sz` that is used by the process for text and data (loaded from the ELF file). The `clearpteu()` function clears the “user accessible” flag for one of the pages making it a guard page. Line 6668 remembers where the stack memory is allocated by saving it into the `sp` pointer.

(b) (5 points) What is the purpose of the guard page?

Answer: The guard page allows the operating system to catch an exception when the program runs out of memory on the page allocated for the stack. Without the guard page the program would silently corrupt it's own data section that would overlap with the stack and would crash in some misterious ways.

(c) (5 points) Bob decides that he wants to allocate the stack in the first page of the process' virtual address space (i.e., 0x0 - 0xfff). What does he need to change in the `exec()`'s implementation above? Write the code for the new lines.

Answer: Bob deletes lines 6662-6668, and instead adds the following lines right between lines 6641-6642

```
if((sz = allocuvm(pgdir, sz, sz + PGSIZE)) == 0)
  goto bad;
sp = sz;
```

(d) (10 points) Bob shows his new code to Alice, but she again says that it's horribly wrong. Can you explain what is wrong with Bob's code and why even if it works on some programs it will sooner or later fail.

Answer: Bob's code boots and shell starts, and even some commands like

```
cat README
```

work, but something like `ls` doesn't. This is because user-level programs in xv6 are linked to have their text section at address 0x0, right where Bob wants to have stack. For some lucky programs the fact that stack overwrites part of the program at the top of the first page (or when the program text is smaller than one page) the code runs. But in general as soon as the stack modifies the program's code (text section) the program fails in some unpredictable manner.

(e) (10 points) After working for a while Alice and Bob fix Bob's implementation of `exec()`. Bob is happy, he also says that his code is better since it saves one page that is normally wasted for the guard page. Do you think he is right, and if you allocate the stack at address 0x0-0xfff (which is 4KB) the guard page is not needed? Explain your answer.

Answer: Alice fixes the Makefile to link all xv6 programs to start at the second page, e.g.,

```
 _%: %.o $(ULIB)
-       $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
+       $(LD) $(LDFLAGS) -N -e main -Ttext 0x1000 -o $@ $^
```

Now Bob's code works, and he is right, his code doesn't need a guard page since even if stack overflows it wraps around the address space and tries to write addresses near the very top of the address space (e.g., 0xfffffffc). These addresses however are mapped by the kernel as not "user-accessible" (they are mapped as part of the 4th region in the memory map, i.e., the I/O region), and hence the kernel catches an exception and kills the process.

4. Physical and virtual memory allocation

   (a) (5 points) Xv6 uses 234MB of physical memory. But how does it keep track of available physical memory? Specifically, explain the following: the xv6 memory allocator (kalloc()) always returns a virtual address, but how does the allocator know which physical page to use for each virtual address it allocates?

   Answer: The allocator always returns one of the virtual addresses from the pool of virtual pages that was constructed when the kernel booted (i.e., `kinit1()` and `kinit2()` functions). These functions map these virtual pages one-to-one to physical pages, but with a 2GB shift, i.e., the virtual address is always equals physical address of the page plus 2GB. The allocator itself does not know anything about physical addresses of the pages it manages, however, the rest of the kernel knows that this is how the pool was constructed and hence it uses the 2GB shift for converting between physical and virtual page addresses.