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

Spring **UID:**

Time Limit: 9:10am – 10:30am

- **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	25	
2	5	
3	15	
4	15	
5	10	
Total:	70	

1. 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 9157?

Answer: Line 9157 closes the read end of the pipe. If it stays open nothing bad happens, the writer (left side of the pipe) can finish it's work and exits (normally, at this point this read end of the pipe will be closed). The reader (right side of the pipe) finishes it's work reading all data from the pipe and exits too. I.e., the pipe will be destroyed when all read and write ends are closed, under some exotic condition, when left side of the pipe closes it's write end and yet keeps running and the right side of the pipe exits the pipe remains open for a bit longer than necessary.

(b) (5 points) What happens if we comment out line 9158?

Answer: Same as above but now we are talking about a write end of the pipe.

(c) (5 points) What happens if we comment out line 9170?

Answer: If you remove one wait() the shell will wait for only one end of the pipe to finish, which means that if one side of the pipe finishes but the other doesn't it might keep printing something on the console. The shell however will print it's prompt (\>) which might be overwritten by the output. This might confuse the user.

(d) (10 points) Extend xv6 shell with a "double-pipe" command which connects two programs with two pipes in such a manner that they can send and reply to each other. For example, if you run `server <|> client` (we use `<|>` for the double-pipe) , the shell will start two programs, `client` and `server`, where the standard output of `server` is connected to the standard input of `client` and the standard output of `client` is connected to the standard input of `server`.

```
int fd1[2] = pipe()
int fd2[2] = pipe()

pid_t s = fork()
if (s == 0) // Child{
  close(1)
  dup(fd1[1])
  close(0)
  dup(fd2[0])
  close(fd1[0], fd1[1], fd2[0], fd2[0])
}

pid_t c = fork()
if (c == 0) // Child{
  close(0)
  dup(fd1[0])
  close(1)
  dup(fd2[1])
  close(fd1[0], fd1[1], fd2[0], fd2[0])
}
else // Parent{
  close(fd1[0], fd1[1], fd2[0], fd2[0])
  wait(s)
  wait(c)
}
```

2. More OS interface

   Alice adds the following program to xv6

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


int
main(void)
{
  int n;
  char *argv[] = { "echo", "hello", 0 };
  printf(1, "start\n");

  for(n = 0; n < 10; n++) {
    fork();
    exec("echo", argv);
  }

  printf(1, "end\n");
  exit();
}
```

   (a) (5 points) What are the possible outputs of this program (explain your answer)?

   Answer: In a normal case the output will be "start hello hello" as both the parent and child immediately call `exec()`, which replaces their program image. However, it would be nice to consider various failure paths when either of the `fork()` or `exec()` fail. If `fork()` fails there might be one "hello", if `exec()` fails there might be "end".

3. Asm, stack and calling conventions

   (a) (5 points) Whrite assembly code that implements the following invocation, make sure to save result into `ret` which is in `rbx` register (use x86 64bit calling convention)

```
ret = foo(1, 2)
```

   Answer:

```
mov rdi, 2
mov rsi, 1
call foo
mov rbx, rax
```

   (b) (10 points) Here are a couple of relevant bits of source code from the xv6 kernel

```
1362
1363 // Bootstrap processor starts running C code here.
1364 // Allocate a real stack and switch to it, first
1365 // doing some setup required for memory allocator to work.
1366 int
1367 main(void)
1368 {
1369   void *kinit1_end = (1024*1024*1024 > PHYSTOP) ? P2V(PHYSTOP)
1370                      : P2V(1024*1024*1024);
1371   kinit1(end, kinit1_end); // phys page allocator
...
1389 }
```

```
3125 // Initialization happens in two phases.
3126 // 1. main() calls kinit1() while still using entrypml4 to place just
3127 // the pages mapped by entrypml4 on free list.
3128 // 2. main() calls kinit2() with the rest of the physical pages
3129 // after installing a full page table that maps them on all cores.
3130 void
3131 kinit1(void *vstart, void *vend)
3132 {
3133   initlock(&kmem.lock, "kmem");
3134   kmem.use_lock = 0;
3135   freerange(vstart, vend);
3136 }

3150 void
3151 freerange(void *vstart, void *vend)
3152 {
3153   char *p;
3154   p = (char*)PGROUNDUP((uint64)vstart);
3155   for(; p + PGSIZE <= (char*)vend; p += PGSIZE)
3156     kfree(p);
3157 }
```

Draw the stack when you reach line 3155 and explain each line on the stack. Explain all assumptions you're making.

Answer:

```
old rbp saved in main              # note, we jumped to main, so no return address on the stack
16 bytes for storing kinit_end     # (optional) space for local variable in main
                                   # alternatively one can assume its in regisrter
                                   # 16 bytes for alignment (optional)
return address from kinit          # i.e., address of line 1372
rbp saved in kinit1                # stack frame
return address saved from kinit1   # i.e., address of line 3136
rbp saved inside freerange         # stack frame
16 bytes for storing *p            # (optional) space for local var
                                   # alternatively assume its in register

...

# Lower addresses
```

4. Relocation

Below is the `cat()` function from the cat program in xv6. It reads bytes from the file descriptor into a temporary buffer and then writes the buffer into standard output.

```
 1 #include "types.h"
 2 #include "stat.h"
 3 #include "user.h"
 4
 5 char buf[512];
 6
 7 void
 8 cat(int fd)
 9 {
10   int n;
11
12   while((n = read(fd, buf, sizeof(buf))) > 0) {
13     if (write(1, buf, n) != n) {
```

```
14        printf(1, "cat: write error\n");
15        exit();
16      }
17    }
18    if(n < 0){
19      printf(1, "cat: read error\n");
20      exit();
21    }
22  }
```

Here is a disassembly of the same function:

```
0000000000000000 <cat>:
       0: 55                          push   rbp
       1: 48 89 e5                    mov    rbp, rsp
       4: 41 54                       push   r12
       6: 53                          push   rbx
       7: 41 89 fc                    mov    r12d, edi
       a: ba 00 02 00 00              mov    edx, 0x200
       f: be 00 00 00 00              mov    esi, 0x0
      14: 44 89 e7                    mov    edi, r12d
      17: e8 00 00 00 00              call   0x1c <cat+0x1c>
      1c: 89 c3                       mov    ebx, eax
      1e: 85 c0                       test   eax, eax
      20: 7e 2b                       jle    0x4d <cat+0x4d>
      22: 89 da                       mov    edx, ebx
      24: be 00 00 00 00              mov    esi, 0x0
      29: bf 01 00 00 00              mov    edi, 0x1
      2e: e8 00 00 00 00              call   0x33 <cat+0x33>
      33: 39 d8                       cmp    eax, ebx
      35: 74 d3                       je     0xa <cat+0xa>
      37: be 00 00 00 00              mov    esi, 0x0
      3c: bf 01 00 00 00              mov    edi, 0x1
      41: b0 00                       mov    al, 0x0
      43: e8 00 00 00 00              call   0x48 <cat+0x48>
      48: e8 00 00 00 00              call   0x4d <cat+0x4d>
      4d: 78 0d                       js     0x5c <cat+0x5c>
      4f: 5b                          pop    rbx
      50: 41 5c                       pop    r12
      52: 5d                          pop    rbp
      53: 31 c0                       xor    eax, eax
      55: 31 d2                       xor    edx, edx
      57: 31 f6                       xor    esi, esi
      59: 31 ff                       xor    edi, edi
      5b: c3                          ret
      5c: be 00 00 00 00              mov    esi, 0x0
      61: bf 01 00 00 00              mov    edi, 0x1
      66: b0 00                       mov    al, 0x0
      68: e8 00 00 00 00              call   0x6d <cat+0x6d>
      6d: e8 00 00 00 00              call   0x72 <main>
```

(a) (15 points) Which instructions use relocation and why will they need it? Use instruction addresses like 0x55 to refer to `xor edx, edx`.

Answer: Here the observation is that you don't really need to understand what assmebly code is doing, it's enough to understand what causes relocations: accesses to global variables and calls to global (non-static) functionsi

1. 0xf mov instruction uses a global address, this shold be the address of the `buf` (global

variable) as it's loaded into the `esi` register (second argument) of the `read()` function. Compiler generates `e8 00 00 00 00`, i.e., zeroes for the address

2. 0x17 call of a global (non-static) function `read()`
3. 0x24 this is a bit hard as you have to guess which function is called at 0x2e but if you look that it takes "1" as the first argument, and it passes what the first function (`read()`) returned as the third argument we know it's `write()`, so here buf is accessed by the `mov` instruction
4. 0x2e call of a global function `write()`
5. 0x37 mov puts the global string into `rsi` (second argument)
6. 0x43 call of a global function `printf()`
7. 0x48 call of a global function `exit()`
8. 0x5c mov puts the global string into `rsi`
9. 0x68 `printf()` global function
10. 0x6d `exit()` global function

There is a couple of shady spots, e.g., why we set `al` to 0 before invoking `printf()` is it a global address? One can reason it's not as the instruction just uses one byte for an immediate value. In practice it's part of the `printf()` calling convention `al` holds the upper bound on how many SSE/XMM registers were used for arguments, from 0 to 8.

5. Page tables

On boot time xv6 uses the following page table:

```
1450 // The boot page table used in entry.S and entryother.S.
1451 // Page directories (and page tables) must start on page boundaries,
1452 // hence the __aligned__ attribute.
1453 // PTE_PS in a page directory entry enables 1Gbyte pages.
1454 __attribute__((__aligned__(PGSIZE)))
1455 pdpte_t identitymap[NPDPTENTRIES] = {
1456   // Map VA's [0, 1GB) to PA's [0, 1GB)
1457   [0] = (0) | PTE_P | PTE_W | PTE_PS,
1458 };
1459
1460 __attribute__((__aligned__(PGSIZE)))
1461 pdpte_t kernmap[NPDPTENTRIES] = {
1462   // Map VA's [KERNBASE, KERNBASE+1GB) to PA's [0, 1GB)
1463   [PDPTX(KERNBASE)] = (0) | PTE_P | PTE_W | PTE_PS,
1464 };
1465
1466 __attribute__((__aligned__(PGSIZE)))
1467 pml4e_t entrypml4[NPML4ENTRIES] = {
1468   // Flags below should be added with "|" and not with "+",
1469   // however, "|" seems to complex for the link editor,
1470   // so the compiler refuses to compile the code.
1471   // The use of "+" is valid since PTE_* are only single bits.
1472   [0] = V2P(identitymap) + PTE_P + PTE_W,
1473   [PML4X(KERNBASE)] = V2P(kernmap) + PTE_P + PTE_W,
1474 };
```

Some additional defines that might be helpful:

```
0870 // Page directory and page table constants.
0871 #define NPML4ENTRIES 512 // # page map level 4 entries
0872 #define NPDPTENTRIES 512 // # page directory pointer table entries
0873 #define NPDENTRIES 512 // # directory entries per page directory
0874 #define NPTENTRIES 512 // # PTEs per page table
0875 #define PGSIZE 4096 // bytes mapped by a page
...
0885 // Page table/directory entry flags.
0886 #define PTE_P 0x001 // Present
0887 #define PTE_W 0x002 // Writeable
0888 #define PTE_U 0x004 // User
0889 #define PTE_PS 0x080 // Page Size
```

(a) (10 points) You need to boot on the x86 64-bit CPU which is identical to the one we discussed in class but does not support 1GB pages. How do you need to change this page table to make everything work. Provide definition of a new page table, use C code.

Answer: High-level plan is PML4 -¿ PML3 -¿ PML2 (Each entry maps 2MB).

```
__attribute__((__aligned__(PGSIZE)))
pdpte_t pml2identitymap[NPDPTENTRIES] = {
  // Map VA?s [0, 1GB) to PA?s [0, 1GB)
  [0] = 0 | PTE_P | PTE_W ,
  [1] = (1 << (12+9)) | PTE_P | PTE_W,
  [2] = 2 * (1 << (12+9)) | PTE_P | PTE_W,
  [3] = 3 * (1 << (12+9)) | PTE_P | PTE_W,
  ...
  [511] = 511 * (1 << (12+9)) | PTE_P | PTE_W,
};
```

```
__attribute__((__aligned__(PGSIZE)))
pdpte_t pml2kernmap[NPDPTENTRIES] = {
  // Map VA?s [0, 1GB) to PA?s [0, 1GB)
  [0] = 0 | PTE_P | PTE_W ,
  [1] = (1 << (12+9)) | PTE_P | PTE_W,
  [2] = 2 * (1 << (12+9)) | PTE_P | PTE_W,
  [3] = 3 * (1 << (12+9)) | PTE_P | PTE_W,
  ...
  [511] = 511 * (1 << (12+9)) | PTE_P | PTE_W,
};

1454 __attribute__((__aligned__(PGSIZE)))
1455 pdpte_t identitymap[NPDPTENTRIES] = {
1456 // Map VA?s [0, 1GB) to PA?s [0, 1GB)
1457 [0] = V2P(pml2identitymap) | PTE_P | PTE_W ,
1458 };
1459
1460 __attribute__((__aligned__(PGSIZE)))
1461 pdpte_t kernmap[NPDPTENTRIES] = {
1462 // Map VA?s [KERNBASE, KERNBASE+1GB) to PA?s [0, 1GB)
1463 [PDPTX(KERNBASE)] = V2P (pml2kernmap) | PTE_P | PTE_W ,
1464 };
1465
1466 __attribute__((__aligned__(PGSIZE)))
1467 pml4e_t entrypml4[NPML4ENTRIES] = {
1468 // Flags below should be added with "|" and not with "+",
1469 // however, "|" seems to complex for the link editor,
1470 // so the compiler refuses to compile the code.
1471 // The use of "+" is valid since PTE_* are only single bits.
1472 [0] = V2P(identitymap) + PTE_P + PTE_W,
1473 [PML4X(KERNBASE)] = V2P(kernmap) + PTE_P + PTE_W, 1474 };
```

Extra space page