I Processes, stacks, and concurrency

1. [5 points]: Explain briefly one benefit of having one kernel stack per process (as in v6) rather than a single kernel stack (as in JOS).

Having a per-process stack allows the kernel to release the processor to another process when a system call needs to wait, for example while reading from the disk.

2. [5 points]: In v6's `getblk`, lines 4953 through line 4958 are protected by `spl6()`. Describe something that could go wrong if this protection was eliminated.

These lines handle the case where the requested block is not in the buffer cache, and there are no free buffers in `bfreelist` in which `getblk()` can store the block read from disk. `getblk()` sets `B_WANTED` to indicate it is waiting for a free buffer, and sleeps on `&bfreelist`.

If the `spl6()` were not there, a disk interrupt might occur after the check on line 4953 but before the sleep on line 4955. The interrupt might call `iodone()`, which might call `brelse()` to put a buffer on the free list. In that case, `brelse()` would call `wakeup(&bfreelist)` and clear `B_WANTED` before the interrupted process called `sleep()`. As a result the process would not wake up even though there is a free buffer.

3. [5 points]: Two v6 processes call `read()` at about the same time for different blocks that are not in the cache. There are many blocks on `bfreelist`, and no other active processes. As a result, there are two calls to `getblk()`, both of which allocate a free buffer in lines 4960 through 4975 at about the same time. Could the two processes end up taking the same buffer from the free list? If yes, how could that happen? If no, what prevents that from happening?

No, this could not happen. In V6, context switches between processes executing in the kernel can occur only during `sleep()` or just before a process returns to user space. As long as no interrupt-time code removes a buffer from `bfreelist` (and none does) there will no problem.

4. [5 points]: Suppose the B_BUSY flag were eliminated from the v6 source, so that for example lines 4941 through 4946 were deleted. Explain a scenario in which incorrect behavior would result.

In vague terms, the B_BUSY flag indicates that a process is actively using this buffer, for example because it is waiting for the result of a disk read to be put there. Eliminating the flag could result in kernel data structures being put in an inconsistent state.

For example, suppose a process calls `read()`, and the resulting `getblk()` call from `bread()` does not find the block in the cache. `getblk()` will put the buffer on the device's buffer list with the desired block number and device, but (so far) without correct contents. Then `bread()` will call `rkstrategy()`, which (among other things) sets the `av_forw` pointer in the `struct buf` to 0 and then waits for the disk read to complete. If, meanwhile, another process calls `getblk()` for the same block number, it will call `notavail()` from line 4948, which will crash because `av_forw` is zero.

5. [10 points]: It turns out that `savu` doesn't really need to save both `r5` and `sp`, as long as `retu` cooperates. Below you'll see a new version of `savu` that saves only `r5`. The `retu` code below is copied exactly from the v6 source; please modify it so that it works with the new `savu`. `retu` is called only by `swtch` (sheet 21) and `expand` (sheet 22).

```
/* the new savu: */
_savu:
        bis     $340,PS
        mov     (sp)+,r1
        mov     (sp),r0
        /* DELETED: mov  sp,(r0)+ */
        mov     r5,(r0)+
        bic     $340,PS
        jmp     (r1)
```

```
/* the new retu: */
_retu:
    bis     $340,PS
    mov     (sp)+,r1
    mov     (sp),KISA6
    mov     $_u,r0
1:
    /* DELETED: mov (r0)+,sp */
    mov     (r0)+,r5
    mov     r5, sp  /* ADDED */
    sub     $6, sp  /* ADDED */
    bic     $340,PS
    jmp     (r1)
```

The `sub $6, sp` preserves the three registers saved by the `csv` at the entry to the function that called `savu`. They must be preserved since they are callee-saved registers which will be needed when we return to the function that called the function that called `savu`.

II Address spaces

6. [10 points]: A process calls `exec()` to run an executable whose first four 16-bit values (in octal) are 410, 1010, 200, 100. In decimal: 264, 520, 128, 64. Write down the content of the prototype segmentation registers after the call to **estabur()** on line 3152 has completed. Please write the addresses and lengths in decimal.

	length	bits		address
PDR 0	**8**	**RO**	PAR 0	**0**
PDR 1	**2**	**RW**	PAR 1	**16 (USIZE)**
PDR 2	**0**	**0**	PAR 2	**0**
PDR 3	**0**	**0**	PAR 3	**0**
PDR 4	**0**	**0**	PAR 4	**0**
PDR 5	**0**	**0**	PAR 5	**0**
PDR 6	**0**	**0**	PAR 6	**0**
PDR 7	**108**	**RW|ED**	PAR 7	**16+3+20−128 = −89**

7. [5 points]: What is the main benefit of keeping the text area separate from the user and data area? (Explain briefly)

It allows the kernel to keep the text section read-only, which may catch some accidental stores to incorrect addresses. It also allows different processes executing the same program to share the physical memory that holds the program's instructions.

8. [5 points]: Why is it convenient for the user stack's physical memory to start just above the user data? What does the kernel have to do if the user program wishes to use more data memory? (Explain briefly)

The kernel can describe a process's physical memory with just two numbers, the start address and the length.

If a program needs to grow its stack or data, in general the kernel will need to copy the user area, data, and stack to a new contiguous region of physical memory that's large enough to hold the sum of the new sizes.

III JOS

The staff solution for the user-level `pgfault` in `lib/fork.c` starts as follows:

```
static void
pgfault(struct UTrapframe *utf)
{
   int r;
   void *addr = (void*)utf->utf_fault_va;
   uint32_t err = utf->utf_err;

   if (debug)
       cprintf("fault %08x %08x %d from %08x\n", addr,
                     %&vpt[VPN(addr)], err & 7, (&addr)[4]);

   if ((vpt[VPN(addr)] & (PTE_P|PTE_U|PTE_W|PTE_COW)) != (PTE_P|PTE_U|PTE_COW))
          panic("fault at %x with pte %x from %08x, not copy-on-write",
             addr, vpt[PPN(addr)], (&addr)[4]);

        // rest of the pgfault
        .....
}
```

9. [10 points]: What is `vpt`? How did the kernel set up an environment's virtual address space to make this code fragment work correctly? (Explain briefly)

`vpt` refers to a region of virtual address space into which the kernel has mapped the current process's page table. The kernel sets the page directory entry for the address `vpt` to point to the physical address of the page directory itself.

The staff solution for kernel-level `page_fault_handler` in `kern/trap.c` contains the following fragment:

```
page_fault_handler(struct Trapframe *tf)
{
    uint32_t fault_va;
    struct UTrapframe *utf;

    // Read processor's CR2 register to find the faulting address
    fault_va = rcr2();

    // some code for earlier labs
    ....

    // lab 4 code
    if (curenv->env_pgfault_upcall == 0) {
        cprintf("[%08x] user fault va %08x ip %08x\n",
            curenv->env_id, fault_va, tf->tf_eip);
        print_trapframe(tf);
        env_destroy(curenv);
    }

    // Second if statement:
    if (tf->tf_esp >= UXSTACKTOP - PGSIZE &&
        tf->tf_esp < UXSTACKTOP) {
        utf = (struct UTrapframe*)(tf->tf_esp
                    - sizeof(struct UTrapframe)
                    - 4);
    } else {
        utf = (struct UTrapframe*)(UXSTACKTOP
                    - sizeof(struct UTrapframe));
    }
    utf->utf_fault_va = fault_va;
    utf->utf_err = tf->tf_err;
    utf->utf_regs = tf->tf_regs;
    utf->utf_eip = tf->tf_eip;
    utf->utf_eflags = tf->tf_eflags;
    utf->utf_esp = tf->tf_esp;

    tf->tf_esp = (uintptr_t) utf;
    tf->tf_eip = (uintptr_t) curenv->env_pgfault_upcall;
    page_fault_mode = PFM_NONE;

    env_run(curenv);
}
```

10. [10 points]: What is the purpose of the second `if` statement (labeled "second if statement" in the code)? (Explain briefly)

The second `if` detects a page fault in the user fault handler. In that case, the kernel pushes the new `UTrapframe` onto the user exception stack, rather than at `UXSTACKTOP`.

IV File systems

11. [5 points]: The file system can be viewed as graph with i-nodes as nodes and directory entries as links. Using `link` can a program create cycles in the v6 i-node graph? If so, show a sequence of commands that creates a cycle. If not, how does v6 prevent cycles? (Explain briefly)

V6 prevents cycles by disallowing links to directories, except for the superuser. The superuser can create a cycle with the following commands:

```
# mkdir foo
# chdir foo
# link ../foo bar
```

12. [5 points]: Suppose that because of kernel bug a v6 process were able to get a file descriptor that referred to a free i-node (that is, `i_mode` is zero). Explain briefly why this would be undesirable, giving an explicit example of a bad outcome.

Suppose process A has such a file descriptor. If process B then creates a new file and is allocated that i-node, then A will be able to read B's data. That would be bad. If both processes wrote to the file, that would also be bad.

13. [10 points]: Explain how a v6 process can acquire a file descriptor that references a free i-node. Your explanation is not allowed to involve crashes, reboots, or any other external source of file system corruption. It must only involve ordinary file-oriented system calls, such as `creat`, `open`, `close`, and `unlink`. This question is difficult. Look at lines 7663 through 7664.

Two processes are required to exploit the bug. Suppose one process calls `open("a", 0)` for a file that exists at the time of the call. `open()` calls `namei()`, and `namei()` calls `iget()` on line 7664 to fetch `a`'s i-node from the disk. If `iget()` blocks waiting for the disk, then the other process could run and call `unlink("a")`. Then `iget()` in the first process will fetch a free i-node; since there's no check for that error, `open()` would then return a file descriptor to that i-node.

The following code provides evidence of the bug.

```
/*
 * exercise bug at line 7663/7664 of v6 namei.
 * Robert Morris <rtm@csail.mit.edu>
 * http://pdos.csail.mit.edu/6.828/2005/inode-thing.c
 *
 * first, mkdir d
 *
 * one process keeps creating d/a over and over, writing "d/a" into it.
 * the other process keeps deleting d/a and creating a different file,
 * containing "d/XXXX" (for some number XXXX).
 * so if the first process ever reads d/a but sees anything other than
 * "d/a" in the file, the bug occured.
 *
 * I think this output is evidence of the problem:
 * # cc x.c
 * # ./a.out
 * starting
 * read d/a failed, ret 0, errno 0
 * done
 */

extern errno;
char junk[20*512];

cr(s)
     char *s;
{
  int fd, n;

  fd = creat(s, 0666);
  if(fd < 0){
    printf("creat %s failed\n", s);
    exit(1);
  }
  if(write(fd, s, strlen(s)) != strlen(s)){
    printf("small write %s failed\n", s);
    exit(1);
  }
  close(fd);
}

main()
{
  int pid, fd, n, i;
  char name[10];

  for(i = 0; i < 1000; i++){
```

```
    name[0] = 'd';
    name[1] = '/';
    name[2] = '0' + (i / 1000) % 10;
    name[3] = '0' + (i / 100) % 10;
    name[4] = '0' + (i / 10) % 10;
    name[5] = '0' + (i / 1) % 10;
    name[6] = '\0';
    unlink(name);
  }

  printf("starting\n");

  pid = fork();
  if(pid < 0){
    perror("fork");
    exit(1);
  }

  if(pid == 0){
    while(1){
      cr("d/a");

      fd = open("d/a", 2);
      if(fd >= 0){
        errno = 0;
        n = read(fd, junk, sizeof(junk));
        if(n < 3){
          printf("read d/a failed, ret %d, errno %d\n",
                 n, errno);
          sleep(2);
          errno = 0;
          if((n = write(fd, "deadbeef\n", 9)) < 0){
            printf("write haha to bad d/a, ret %d, errno %d\n",
                   n, errno);
          }
          exit(1);
        }
        if(junk[2] != 'a'){
          printf("read %d, did not start with a\n", n);
          exit(1);
        }
        close(fd);
      }
    }
  } else {
    for(i = 0; i < 1000; i++){
      unlink("d/a");
    name[0] = 'd';
    name[1] = '/';
    name[2] = '0' + (i / 1000) % 10;
    name[3] = '0' + (i / 100) % 10;
    name[4] = '0' + (i / 10) % 10;
    name[5] = '0' + (i / 1) % 10;
    name[6] = '\0';
      cr(name);
    }
    printf("done\n");
    kill(pid, 9);
  }
}
```