Assignment 1: Code Review

1)What happens to a thread when it exits (i.e., calls thread_exit())? What about when it sleeps?

WHERE: kern/thread/thread.c Lines 789 - 822, Lines 1036 - 1042

- It clears virtual file system (VFS) [Lines 796 - 799], erases its virtual physical memory [Lines 801 - 813], puts itself in S_ZOMB state [Line 820], decreases the counter of vnode it may be pointing at, tells us if stack is overflowed [Line 816], disables all the interrupts by setting priority level to high [Line 819].

When it sleeps, it makes sure it's not in an interrupt handler [Line 1039], yields control to the next thread, puts itself in S_SLEEP state [Line 1041].

2)What function(s) handle(s) a context switch?

WHERE: kern/thread/thread.c Lines 550 - 725

- thread_switch. It is high level machine independent context switch code.

3)How many thread states are there? What are they?

WHERE: kern/include/thread.h Lines 61 - 66

- Four:

1. S_RUN (thread is running)

2. S_READY (thread is ready to run)

3. S_SLEEP (sleeping)

4. S_ZOMB (zombie)

4)What does it mean to turn interrupts off? How is this accomplished? Why is it important to turn off interrupts in the thread subsystem code?

WHERE: kern/thread/spl.c Lines 132 - 155, kern/thread/thread.c Lines 550 - 725

- If interrupt is turned off, then even if interrupt is signaled the handler won't execute.

Interrupt is turned off using the function splhigh (set priority level to high), and back on again using spl0 (set priority level zero). The priority levels can also be set to intermediate levels using splx function [Lines 131 - 155]

Turning off interrupts for thread operations is necessary to ensure that these operations complete successfully and aren't broken mid-execution. This happens in thread_switch. [Line 561]

5)What happens when a thread wakes up another thread? How does a sleeping thread get to run again?

WHERE: kern/thread/thread.c Lines 1044 - 1067

- It removes the sleeping thread from the queue, and calls the make-runnable on the thread [Line 1066], which currently adds it to the end of runqueue. e thread gets to run again when an mi_switch is called, and that thread is returned by the scheduler.

6)Semaphores are implemented using spinlocks. Where are spinlocks implemented? And what does it mean to get the data for a spinlock?

- Spinlocks are implemented in spinlock.c. To get the data of spinlock simply means to get the information about if the lock is held or not.

7) Are OS/161 semaphores "strong" or "weak"?
OS/161 semaphores are "weak" because the processes are not subject to an absolute order of removal from the semaphore because FIFO is not enforced in src/kern/thread/synch.c.

8) Why does the lock API in OS/161 provide `lock_do_i_hold()`, but not `lock_get_holder()`?

Because unless the thread is holding the lock and tries to acquire it additionally, we only need lock_do_i_hold() and not lock_get_holder().

9) What are the ELF magic numbers?
Found in src/kern/include/elf.h (0x7f, 'E', 'L', 'F'), they are the preceding bits of a file that are used to check and validate the type of the file.

10)What is the difference between UIO_USERISPACE and UIO_USERSPACE? When shouldone use UIO_SYSSPACE instead?
WHERE: kern/include/uio.h Lines 48 - 53, kern/syscall/loadelf.c Line 101

- Found in uio.h, UIO_USERISSPACE contains user process code and UIO_USERSPACE contains user process data. UIO_USERISSPACE is used when process is happening at the user space, and executable. If it's not executable, UIO_USERSPACE is used [Line 101 in loadefl.c]

UIO_SYSSPACE is used whenever data transfer are in kernal space i.e. we are handling the kernal data.

11) Why can the `struct uio` that is used to read in a segment be allocated on the stack in `load_segment()` (i.e., where does the memory read actually go)?

It can be allocated on the stack in load_segment because the memory read is put into a virtual address, which is pointed to by the struct uio's pointer variable. Look at src/kern/syscall/loadelf.c and src/kern/include/uio.h.

12)In runprogram(), why is it important to call vfs_close() before going to usermode?

- The file must be closed, so that if another thread or process wants to execute the same program, it can open the file. (???? We need to end process in kernal space before moving on to user space)

13)What function forces the processor to switch into usermode? Is this function machine dependent?

WHERE: kern/arch/mips/locore/trap.c Lines 349 - 416, Lines 405 - 431

- mips_usermode() function for entering user mode. Called in enter_new_process(). The function uses machine independed interface to diable all interrupts.

14)The mips_trap function handles all exceptions. Is it possible to determine whether the exception being handled was triggered during user execution or kernel execution?

WHERE:kern/arch/mips/locore/trap.c

- Thread can only check whether or not it is in the kernal execution.[Line 140]