# Read the project description (https://fxlin.github.io/p1-kernel/) before proceeding

# Programming is required, which your answers should depend on.

# No code submission is needed.

# Saving PC/SP when a task was interrupted.

**This set of questions reinforce what you have learnt in exp4b**

(10) In your words, explain the difference between the values of PC/SP at the interrupt time and the values of PC/SP at the context switch time.

To further clarify:

* Interrupt time: when the timer fires, and the kernel is about to enter the interrupt handler, kernel\_entry
* context switch time: when the cpu\_switch\_to is called, i.e., inside the handler and about to execute the next task

At the interrupt time, the SP value refers to the bottom of the current task’s stack where the current task’s saved registers and the irq contexts are stored and the PC points to the next instruction that the current task would have executed if left uninterrupted. At the context switch time, the SP value refers to the memory at the top of the new task’s stack and the PC points to the first instruction that will be executed once the context switch finishes.

(10) When a task is scheduled for the first time, where does its task\_struct.cpu\_context.[pc|sp] point to? Why?

When the task is scheduled for the first time, the **task\_struct.cpu\_context.pc** points to the **ret\_to\_fork** function because the kernel needs to know to grab the function from the *x19* and the argument from the *x20* register in order to actually execute the task. The **task\_struct.cpu\_context.sp** points to the very top of the task’s page so that it can use the task’s stack and prepare it for task switch.

(10) Is the “save\_regs” region on the very top of a task’s stack (i.e. the location where the task starts to grow)? If not, what is the stack content above the “save\_regs” region?

Not necessarily because a task may be interrupted in the middle of a process which may involve the use of that task’s particular stack for intermediary values such as variables and caller return addresses.

# The significance of exit()

In **exp4b**, this is the main function for a task:

void **process**(char \*array)

{

    while (1) {

        for (int i = 0; i < 5; i++){

**uart\_send**(array[i]);

**delay**(5000000);

        }

    }

}

It is an infinite loop and will never return. What will happen if the function returns? For instance, instead of while(1), the function just iterates 5 times and returns.

(10) Describe your observation. QEMU users may attach screenshots additionally.

(20) Explain your observation.

In the given code of **exp5/kernel.c,** this is the function of a **user task**

void **user\_process1**(char \*array)

{

    char buf[2] = {0};

    while (1){

        for (int i = 0; i < 5; i++){

            buf[0] = array[i];

**call\_sys\_write**(buf);

**delay**(**DELAYS**);

        }

    }

}

Same as above, what will happen if the function returns?

(10) Describe your observation. QEMU users may attach screenshots additionally.

(20) Explain your observation.

(10) Is the kernel still functioning properly after the function returns? For instance, is the task calling exit() terminated properly? Are other tasks running as expected?

**The question below is optional.**

In **exp5/kernel.c,** this is the function of kernel\_process():

void **kernel\_process**(){

**printf**("Kernel process started. EL %d\r\n", **get\_el**());

    int err = **move\_to\_user\_mode**((unsigned long)&**user\_process**);

    if (err < 0) {

**printf**("Error while moving process to user mode\n\r");

    }

}

(**20, bonus**) Is it a bug that it does not call exit()? Will returning from this function crash the kernel? Explain your answer.

# Validate the user/kernel separation

The following questions refer to the code of **exp5/**.

Design a small experiment. From a user task, access some system registers inaccessible at EL0. Confirm a synchronous exception is generated. Handle this exception, use ESR\_EL1 to distinguish the exception from a system call.

To handle the exception, any method that keeps the kernel and other tasks operating (i.e. not crashing them) is allowed.

(10) What system register did you access?

(10) Attach a screenshot showing that an exception was triggered.

(10) How did you handle this exception?

*Changelog*

*Feb 2024. Clarification on no code submission.*

*Jan 2024. Clarification.*