The (weights) of questions are relative and may not necessarily add up to 100.

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 top of the current task's stack where the kernel will attempt to save the registers and the irq contexts. At the interrupt time, the PC points to the some instruction in schedule to prepare for switching. 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 somewhere in the **switch_to** function where the kernel is getting ready to start the next task.

(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) {
```

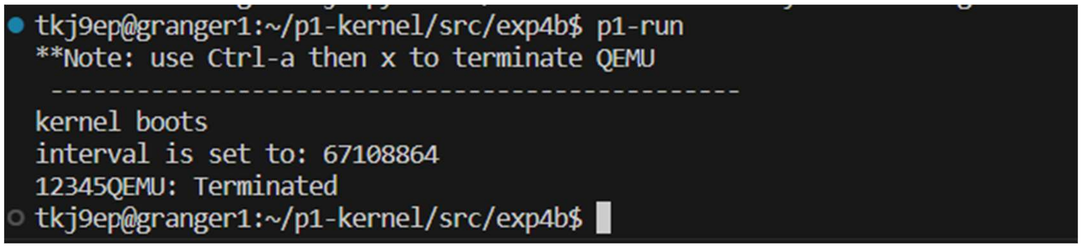
The (weights) of questions are relative and may not necessarily add up to 100.

```
        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.

The kernel simply just prints the argument once and waits for a certain period of time before proceeding to the next task or argument.



```
tkj9ep@granger1:~/p1-kernel/src/exp4b$ p1-run
**Note: use Ctrl-a then x to terminate QEMU
 ---------------------------------------------
kernel boots
interval is set to: 67108864
12345QEMU: Terminated
tkj9ep@granger1:~/p1-kernel/src/exp4b$
```

(20) Explain your observation.

The reason why the kernel finishes the process, it points to the **err_hang** assembly function which will cause the machine to hang forever.

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

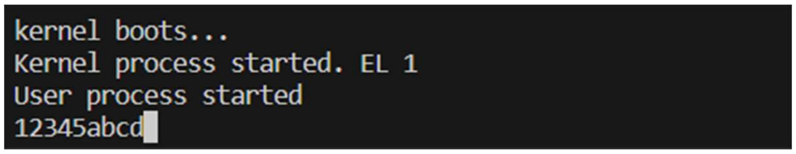
```
void user_process1(char *array)
{
    char buf[2] = {0};
    while (1){
        for (int i = 0; i < 5; i++){
            buf[0] = array[i];
            call_sys_write(buf);
            delay(DELAYS);
        }
    }
}
```

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

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

What I observed is that the kernel simply just points “12345abcd” and hangs forever.

The (weights) of questions are relative and may not necessarily add up to 100.



```
kernel boots...
Kernel process started. EL 1
User process started
12345abcd
```

(20) Explain your observation.

The kernel is looping forever between **switch_to** and **_schedule** because in the **switch_to** function, the kernel is constantly returning and not actually switching since the argument for the *current* and *next* tasks are basically the same. It seems as if the scheduling system is struggling to actually schedule the next task instead of scheduling the current task over and over again. This blocks the kernel from calling **ret_from_fork** since **cpu_switch_to** is not being called. This is most likely because the task is in the *zombie* state which means that the scheduler will not schedule it.

(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 kernel seems to be functioning properly since it is calling **exit_process()** after a process is done. Other tasks are running as expected and the tasks that were already executed are turned into ZOMBIE tasks which means that the scheduler will not schedule them.

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.

It is not a bug that this function does not call **sys_exit()** because the processes are done in user mode and user mode will call **sys_exit()** anyways. In addition, the kernel process does not have any memory or resources to manage, yet. The kernel will crash lingering in **err_hang**.

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.

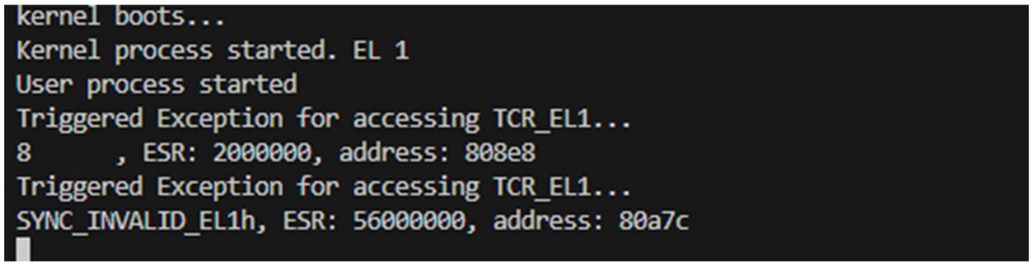
The (weights) of questions are relative and may not necessarily add up to 100.

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?

I tried to access the **TCR_EL1** register which is the translation control register.

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



```
kernel boots...
Kernel process started. EL 1
User process started
Triggered Exception for accessing TCR_EL1...
8      , ESR: 2000000, address: 808e8
Triggered Exception for accessing TCR_EL1...
SYNC_INVALID_EL1h, ESR: 56000000, address: 80a7c
```

(10) How did you handle this exception?

In the **show_invalid_entry_message**, I simply called **exit_process()** in the **show_invalid_entry_message** which will immediately cause the kernel to kill the process and set the state to a zombie once an exception is triggered.

Changelog

Feb 2024. Clarification on no code submission.

Jan 2024. Clarification.