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

- Read all the questions before starting.
- Submit your code as **one diff file**. Not a tarball including the whole exp4a directory.
- diffs can be generated by command "diff -r --new-file <old_dir> <new_dir>", or "git-diff".
- **Warning!** Do not wait until the last minute to learn diff and git-diff, which may surprise you.
- The syllabus contains some information about diff and git-diff

(50) Upload a standalone diff file named as [ComputingID].diff. The code should address all the design questions below.

Adding the WAIT state to tasks

(10) What can a task WAIT for? IO interrupts (i.e. "IO events"). At this moment, timer is the only IO device that can generate interrupts. So implement a kernel function "**void sleep(int X)**" that puts the current task in WAIT state until X seconds have elapsed. Briefly describe how you would implement sleep().

A task can wait for IO interrupts such keyboard/mouse input and letting other tasks finish. In the sleep() function I set the state of the current task to be TASK_WAIT and set the *wait* field to the argument that is specified when the function is invoked. I would invoke the schedule function in order to reschedule the tasks since the schedule needs update.

I would modify the handler for the timer interrupt to keep track of the shortest wait time, *t*, among all the tasks that are in the TASK_WAIT state. Then wait for that amount of time. I would update each of the wait times for the tasks in the TASK_WAITING state by subtracting their wait times from the *t*. When one of the waiting tasks has no more wait time left, then the code will change that task to TASK_RUNNING. This process will perpetually loop.

If needed, feel free to add the function prototype to a header file of your choice, and the function body to a C file of your choice. We do NOT mandate which file(s) you should modify.

Note: you may know that some C libraries provide sleep() already. It is different. Such a library function relies on an OS and its syscalls underneath. If you remember from previous lectures, why we must implement our own printf to print messages, it's the same idea here. The syscalls that support these library functions are not implemented in our baremetal kernel. What the experiment asks you to do is the implementation itself.

(5) Briefly describe in one paragraph: How would you keep track of one or more tasks that may be in the WAIT state?

I created a new field in the *task_struct* called *wait* where *wait* holds the number of seconds left before the task wakes up. In the timer interrupt handler, I looped through all the tasks and found which task had lowest wait time, *t*. In addition, I changed the state of the task to TASK_RUNNING if that particular task is in TASK_WAIT and its *wait* field is 0. The code waits for *t* seconds and then updates each of the tasks' *wait* field in the TASK_WAIT by subtracting each task in the TASK_WAIT state by *t*. This entire process perpetually loops.

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

(5) Briefly describe in one paragraph: How would you demonstrate that sleep() and WAIT work properly. For instance, you may create multiple tasks that sleep for different intervals, observe their execution frequencies, and talk about if the observation matches your expectation.

Theoretically, I would test sleep() by having the first process sleep for 3 seconds and the second process sleep for 5 seconds. The system should print the first character of the first process, print the second character of the second process and not do anything for 3 seconds. Afterwards, it should resume the first process for two seconds. Afterwards, the kernel should resume alternating between the first and second processes as it did before.

I did not get the observations that I was hoping for unfortunately and I think my implementation needs more improvement.

The idle task

Modify the kernel to implement an idle task. The kernel schedules the idle task only when all other tasks are in WAIT. The idle task only does one thing when it is scheduled to run – putting the CPU to power saving mode using WFI (more on that below).

How to create such an idle task? You can repurpose the existing "init" task. The given kernel code has an init task (PID 0, executing kernel_main()). At the end of kernel_main(), it calls schedule() in an infinite loop. You can modify kernel_main() and schedule() so that the init task *serves* as the idle task.

(10) We have introduced the aarch64 WFI instruction in a lecture. Read about WFI. How should you use WFI in implementing the idle task?

D1.18.2 Wait For Interrupt

Software can use the *Wait for Interrupt* (WFI) instruction to cause the PE to enter a low-power state. The PE then remains in that low-power state until it receives a *WFI wake-up event*, or until some other IMPLEMENTATION DEFINED reason causes it to leave the low-power state. The architecture permits a PE to leave the low-power state for any reason, but requires that it must leave the low-power state on receipt of any architected WFI wake-up event.

WFI should be called when ALL tasks are in the TASK_WAIT state. In other words, I should call WFI in the while loop after the *schedule()* function because only when all tasks are in TASK_WAIT will the *schedule()* function will exit its execution.

(10) The purpose of an idle task in your own words?

The purpose of the idle task is to keep the kernel busy and prepare it for when a task wakes up.

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

(10) Describe briefly how you determine that the idle task works properly. Attach screenshots if needed.

Theoretically, I would test if the idle task works properly by using the same method as described for testing the sleep() function. The result I would expect to see is that the message saying that wfi is being entered is printed for two seconds. Afterwards, it waits for 3 seconds and then does the process as described in one of the previous questions.

Unfortunately, I was not able to test the WFI function because I don't think I was able to get the sleep function completely functional.

(Optional) WAIT for UART

Right now our kernel busy waits for incoming UART characters. This is naïve. Turn on the UART interrupt so each incoming character generates one interrupt. Add a kernel function "int getc()" which will put the current task in WAIT until the next UART interrupt happens. getc() should return the character read from the UART.

(**No credits**) Describe briefly how you determine that getc() works properly. Attach screenshots if needed.

Changelog:
Jan 2023 – updated. Advise to use init task as idle task.