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

# Read the “implementation notes” that is downloadable from the assignment page

|  |
| --- |
| * **You may want to read “p1 exp4b notes”** * Submit your code as **one diff file**. Not a tarball including the whole lesson directory. * diffs can be generated by command “diff -r <old\_dir> <new\_dir>”, or “git-diff”. * **Warning!** Do NOT wait until the last minute to learn diff and git-diff. Many students missed deadlines because of this. * 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.

# Kernel tracing

(5) Create two more tasks (total: four, in addition to init task). The point here is to test scheduling multiple tasks and observe their dynamics. In one or two sentences, describe what changes you have made.

I have simply created two new process functions *void process3(char \*array)* and *void process4(char \*array)* and I called *int copy\_process(fn, arg)* two additional times in **kernel.c**. The first new *int copy\_process* was assigned to *void process3(char \*array)* with an arg of “67890” and the second *int copy\_process* was assigned to *void process4(char \*array)* with an arg of “fghij” in **kernel.c**.

(5) Add a function “int getpid(void)”. The function returns the PID of the current task. In one or two sentences, describe what changes you have made. Note that a task’s PID is the index of the task’s task\_struct pointer in the “task” array (sched.c).

In the **sched.c**, I created a new global variable called *int current\_pid = 0* and a new function *int getpid(void)* which simply returns *current\_pid*. In order to fetch the current task’s pid, I set *current\_pid* to be equal to *next* right before *switch\_to(task[next])* is invoked in the *void \_schedule(void)* function.

(5) Set the interval of rescheduling to be roughly 100 ms. In one or two sentences, describe what changes you made.

In **timer.c**, I changed the function *void handle\_generic\_timer\_irq( void )* such that the function sets *int interval* to 1 \* 1000 \* 100.

(10) Add a function “unsigned long get\_time\_ms(void)”. The function returns the elapsed time since boot, in milliseconds. In one or two sentences, describe what changes you made. Your approach can be empirical; the returned time does not have to be precise. In doing so, avoid floating point number computation (which you do not need and is not supported within kernel).

In **timer.c**, I created a new function called *unsigned long get\_time\_ms(void)* which gets the lower 32 and upper 32 bits of the value in the timer register using *get32(TIMER\_CLO)* and *get32(TIMER\_CHI)*, respectively. Finally, I returned the sum of the upper 32 bits shifted left by 32 and the lower 32 bits and divided that quantity by 1000.

**Tracing context switch**. Create a data structure in memory. Whenever a context switch happens, record (1) the current timestamp in milliseconds; (2) the IDs of tasks that are switching in and out; (3) the PCs of tasks that are switching in and out; (4) the SPs of tasks that are switching in and out.

(10) Why is it a bad idea to print out the information of a context switch to UART **as it happens**? i.e. why you need to store the information as context switch happens and print the information afterwards?

It is a bad idea to print out the information of the context switch to UART as it happens because the context switch happens significantly faster than printing the message because printing requires writing to UART. This will distort the whole kernel execution timing making the timing very slow.

(10) Describe your choice of data structure. Justify your choice.

I chose to use an array of structs of size 50 as my choice for the data structure. Each struct stores the timestamp, the pid’s, pc’s, and sp’s of the incoming and outgoing task. I chose this particular data structure because after 50 iterations of context switches, I can simply loop through the array and dereference each struct element’s fields for printing. In addition, I can simply change each field’s value at anytime during the context switch.

After 50 context switches, print out a list of switch records to UART. Suggested format: each line is a record starting with a timestamp, e.g.

1234 from task1 (PC 0x81000 SP 0x83F00) to task2 (PC 0x82000 SP 0x85F00)

* The trace should include task1/2/3/4, as well as the init task (task0) although it performs no useful work.
* (**5; bonus**) If your trace includes the context switches when tasks are scheduled for their first times (i.e. in addition to their subsequent context switches)

(10) Attach a screenshot of the printout.

(0; optional) How would you assess the overhead of tracing added to the kernel?

*Changelog*

*Jan 2024. Clarify task0 must be traced; ask for tracing 1st switches*