HW7-1
On ARM, when a function is called (with BL), **lr (link register)** holds the **return address** – the address of the next instruction that should run after the function finishes.
Here, Save_context() is called during a context switch, and it does:
__asm__("PUSH {lr}");

That value of lr is later treated as pc in KernelTaskContext_t.

So the content of lr is "where this task should continue running after the context switch" — i.e., the saved program counter (return address) of the task.

HW7-2
We store the content of lr and restore it to pc. Please explain the reason and why this is necessary.
In Save_context() we push lr onto the task's stack, and in Restore_context() we finally do:
__asm__("POP {pc}");
Because we saved lr first, that popped value becomes the **new pc**.
Reason / why necessary:

- When we switch away from a task, lr holds the address of the point **right after** the call to Kernel_yield() / context switch.
- We save that address in the task's stack (as pc in the context struct).
- When the scheduler later chooses this task again, Restore_context() restores registers and finally loads that saved address into pc.
- The CPU then **jumps back to exactly where the task left off**, as if nothing happened in between.

Without saving lr and restoring it into pc, the task would **resume at a wrong address** (e.g., re-entering the context-switch code or some random place), causing crashes or infinite loops.

HW7-3
Please show what will happen if we miss calling Kernel_yield() and explain the reason.
From the lab, our RTOS is **non-preemptive**: the scheduler only runs when a task explicitly calls Kernel_yield().
If we remove Kernel_yield() from the user tasks:

```
void User_task0(void)
{
  uint32_t local = 0;
  while (1) {
    debug_printf("User Task #0 SP=0x%x\n", &local);
    delay(100);
    // Kernel_yield();  <-- removed
  }
}
```

What happens:

- User_task0() runs after Kernel_start().
- Because it **never calls Kernel_yield()**, the scheduler is never invoked.
- Context switching never occurs, so User_task1() and User_task2() are **never run at all**.
- On the console you will only see User Task #0 ... printed repeatedly; tasks 1 and 2 are "starved".