Concurrency Issues
- All aspects of scheduling apply to tasks, threads, and interrupts
    - We preempt execution
- Must maintain deadline required in the face of all conflicts
    - The conflicts are caused by:
        - Aperiodic events disrupt execution flow
        - Deadlock & infinite loops
            - Managed (in hardware) with a watchdog timer
            - On overflow, reset the microcontroller
            - Code must reset timer to avoid this

Resource Contention:
- 1 task/interrupt locks a resources needed by another to complete
    - Meet deadline

  Locks:
    - disable all interrupt until done (i.e. SIGNAL)
    - CPU priority upped to limit which interrupts can run(i.e. ISR)
- If we allow higher priority tasks to override lower priority ones, we can have a potential problem:
    - If a lower priority task has a lock on a resource we need, we are stuck
        - Priority inversion(name/technical term of this problem)

Solutions:

a) Priority inheritance
    - Lower priority task gets a higher priority temporary so it can finish
        - Returns to lower priority when it's done
    - Computationally expensive
        - OS constantly working to detect

b) Priority ceiling
    - Define the priority required for a task to ensure that it's non-blocking
    - task/interrupt automatically gets this priority
        - When it locks the resource
    - Priority based on highest priority task that can lock it

Priority inversion could occur in A3, need to analyze and implement appropriate priority ceiling

---

How can we transfer data out of an interrupt without locking?
- Read and write memory without priority problems

4-slot algorithm:
- Want concurrency read/write of consistent data
    - Not necessarily “latest”

- Maintain 2 pairs of data
    - Each pair has 2 slots so we can write to 1 while reading the other
    - IMG 81
    - Read:
        - Read from the lastWritten pair
            - After read we set lastRead to point to this pair
        - Use currentSlot for that pair
    - Write:
        - Write to the opposite pair from lastRead
        - Use the opposite slot for that pair
        - After write, update currentSlot and lastWritten to where we just wrote
    - If we have concurrent read & write, we will access the same pair but different slots

Scheduling

- We have a number of variables to consider:
    - Preemptive vs non-preemptive system
    - Static vs dynamic scheduling
    - Aperiodic tasks
    - Load on CPU
- Preemption
    - Do we allow 1 task to stop the execution of another?
        - How does that affect the deadline?
            - Should we preempt a task with a hard deadline?
    - Note: non-preemptive scheduling is deterministic
- Static vs dynamic:
    - Can tasks be introduced on the fly?
        - Do we fix the task and their execution order?
            - Most common solution
    - Note: we can have static with preemption
- Aperiodic tasks:
    - Having just periodic tasks is “much” easier
    - When an aperiodic task arrives, how do we handle it?
        1. Insert as another task with regular scheduling
            - Exit after 1 run
        - Note: a scheduler is just iterating over a table of function pointers
            - Then, aperiodic task is just another call
        2. Allow aperiodic task to run when no periodic task need to run.
            - Known as slack stealing
        3. Make them solely interrupt driven
            - Good for hard deadlines

Loading:

- As tasks are added (or take more CPU time), how do we respond?

        - Must prioritize hard deadlines (soft deadlines could be missed compare to hard deadlines)
            - As load increases, preemption becomes important (when multiple hard deadline competing)
                - To the point of using fail safe or “limp home” mode
- We design for load ahead of time
    - Instead of a scheduler with preemption, build a schedule that has load guarantees