Lecture 06 Notes – (Synchronization and) Scheduling and Dispatch, Scheduler Wish List

I. **Overview**
   A. **Review**
      1. Where does this fit into the big picture? Sync/Comm/Sched
      2. Refining event detection & processing chain
         a. Sample, Quantize, Analyze, Decide
         b. Basic HW and SW options
   B. **Today**
      1. Continue refining event detection & processing chain
         a. Schedule, Dispatch, Handle
         b. Basic response time analysis
         c. Observations
         d. OS Code characteristics
      2. Scheduler Wish List

II. **Closer look at Sync/Comm/Sched**
   A. **Where does this fit into the big picture?**
   B. **Processing chain options:**
      1. Various options available for each step/slot. Can implement in software and/or hardware
   C. **Processing Chain Refinement**
      1. Problem and Adjustment
      2. **Understanding Synchronization with SQ and AD**. Should processing take place (eventually)?
         a. **Detect** is split into
            i. **Sample & Quantize** (Digitize) input (**SQ**): Must be hardware for input signals
            ii. **Analyze & Decide (AD)** if event happened: Software and/or Hardware
               - **Analyze** signal (past and present values, may depend on system state)
                  - **Software** Rising Edge detection:
                     - if cur_val different from prev_val (previous cur_val) then diff = 1 else diff = 0 (simple logic version: diff = cur_val XOR prev_val)
                     - prev_val = cur_val
                  - **Hardware** Rising Edge detection:

        - PORT module compares current SQ value with previous one. Set port interrupt status flag (ISF) to 1 in PORT's pin control register (PCR), else leave ISF unchanged
- **Decide** if event happened which affects handler code being able to run, save that decision for scheduling.
    - **Software** Rising Edge detection:
        - if diff ==1 and cur_val == 1, then then tell the scheduler that the process is allowed to run
    - **Hardware + Software** Rising Edge detection: Still needs software.
        - PORT interrupt is disabled, so code reads PORT's pin control register, tests the interrupt status flag (ISF). If 1 (event detected), then tell the scheduler that the process is allowed to run
    - **Hardware** Rising Edge detection:
        - PORT interrupt is enabled, so sends interrupt request (IRQ) to interrupt controller (NVIC)

3. **Scheduling** and Dispatching: Decide what to do next, and start it
    a. Behavior
        i. Low-level behavior: Decision tree, control flow
            - Is this process allowed to run?
                - Yes: Dispatch and run it
                - No:
                    - Blocking: Repeat software to analyze and decide (loop back to it)
                    - Non-blocking: Advance to next process (via scheduler or program structure) and repeat decision tree analysis
        ii. Resulting system behavior timelines
            - Non-blocking detection
            - Blocking detection

D. **Basic Response Time Analysis**

1. For **which** process?
   a. This particular process. Local view.
   b. All the other processes. How does this process affect timing for the rest of the system?
2. Basic Idea
   a. **What can run** between when a process A's input event happens t_{A_event} and response processing completes t_{A_done}?
      i. Which process? BTW, process may be an interrupt handler.
         - Timing for this process usually depends on other processes.
         - One scheduling goal for urgent processes is to reduce this dependence/timing vulnerability
      ii. How long does it take process A to do its work in response to the event?
         - Term is C_A – amount of computation needed.
         - Want max value (or upper bound) to determine worst case response time
         - **Blocking** vs. **Non-blocking** scheduling tests and computation time requirements
            - Blocking? If event never happens, process will take infinite time.
            - Non-blocking? Finite (short!) maximum time to pass test.
            - May also have a bound on blocking time (if not passed within 10 tests, try again later)

   b. **Note: we may ignore some terms** to simplify math, giving a less accurate timing model

   c. How soon does the scheduler start running process A?
      i. Is anything already running that will delay when the scheduler gets to run?
      ii. What else can run before A starts running? Causes timing interference

d. Can anything else preempt process A after it starts running? Causes more timing interference
    i. Interrupt/Exception handlers (service routines) for threads
    ii. Higher-priority Interrupt/Exception handlers (service routines) for handlers
    iii. Higher-priority process, if scheduler allows a process to preempt another process
e. Might process A have to wait because of synchronization or communication with another other process?
    i. Using shared resource, waiting for message from other process, etc.
    ii. May lead to priority inversion… more later
f. How much time does each interfering process take?
    i. Remember, may contain blocking or non-blocking operations
g. How many times does scheduler/OS need to run over this time, and how long does each run take? (Starting point: What does it need to do?)

h. **Add them all up**
   i. Sum of [number of times this activity runs * duration of this activity] for all activities (i=0, 1, 2 ...)
   ii. N_i = number of times activity runs
   iii. C_i = duration for activity

i. Is the response time acceptable?

3. Apply to non-blocking infinite loop, with simplifications

E. **Observations:**
   1. Sync/comm/sched/dispatch operations are often **interdependent**, so
   2. If scheduler/OS can see all these operations, it can make better decisions and offer more features
      a. Example: If process event test will **block**, then pause process execution and automatically switch in another process. Reclaims Idle Time.

III. Scheduler/OS Wish List

A. **Allow better responsiveness where needed**
   1. Dynamic task execution order
   2. Allow Process Prioritization to drive scheduling. Static priority? Dynamic?
   3. Allow Process Preemption by higher priority processes

B. **Generalize/standardize code structure for modular code**
   1. Move many/most scheduling decisions from user code to scheduler code

C. **Features to simplify programming and reclaim idle time**
   1. Provide mechanisms for synchronization: Let software processes trigger each other with event flags, semaphores, etc. Provide mutexes, etc.

D. **What kind** and **where is** the code to do detect/sync/sched/dispatch (part of **"OS code"**), and **how do we get it to run?**

1. What **kind of software process** does the OS work?
    a. Can be in threads and/or interrupt/exception handlers, depending on other factors
2. Which **files** have the OS code? Two possible locations, may have some in both.
    a. **Application program** functions have OS code **integrated** within work/handlers of processes (may be implicit)
        i. Code for OS operations is integrated into program source code, executes from main thread (and maybe interrupt/exception handlers).
    b. A **separate module** with OS code
        i. Code for OS is in different files/modules from application program
3. How do we make OS code execute?
    a. Application Program:
        i. OS code runs when function gets to those integrated operations.
    b. Separate OS module:
        i. OS runs when **application program** or **OS function** invokes OS through ...
            - macro,
            - subroutine call or
            - software interrupt (e.g. SVC)
        ii. We will see OS often uses interrupt and exception handlers for key features (e.g. timer tick)