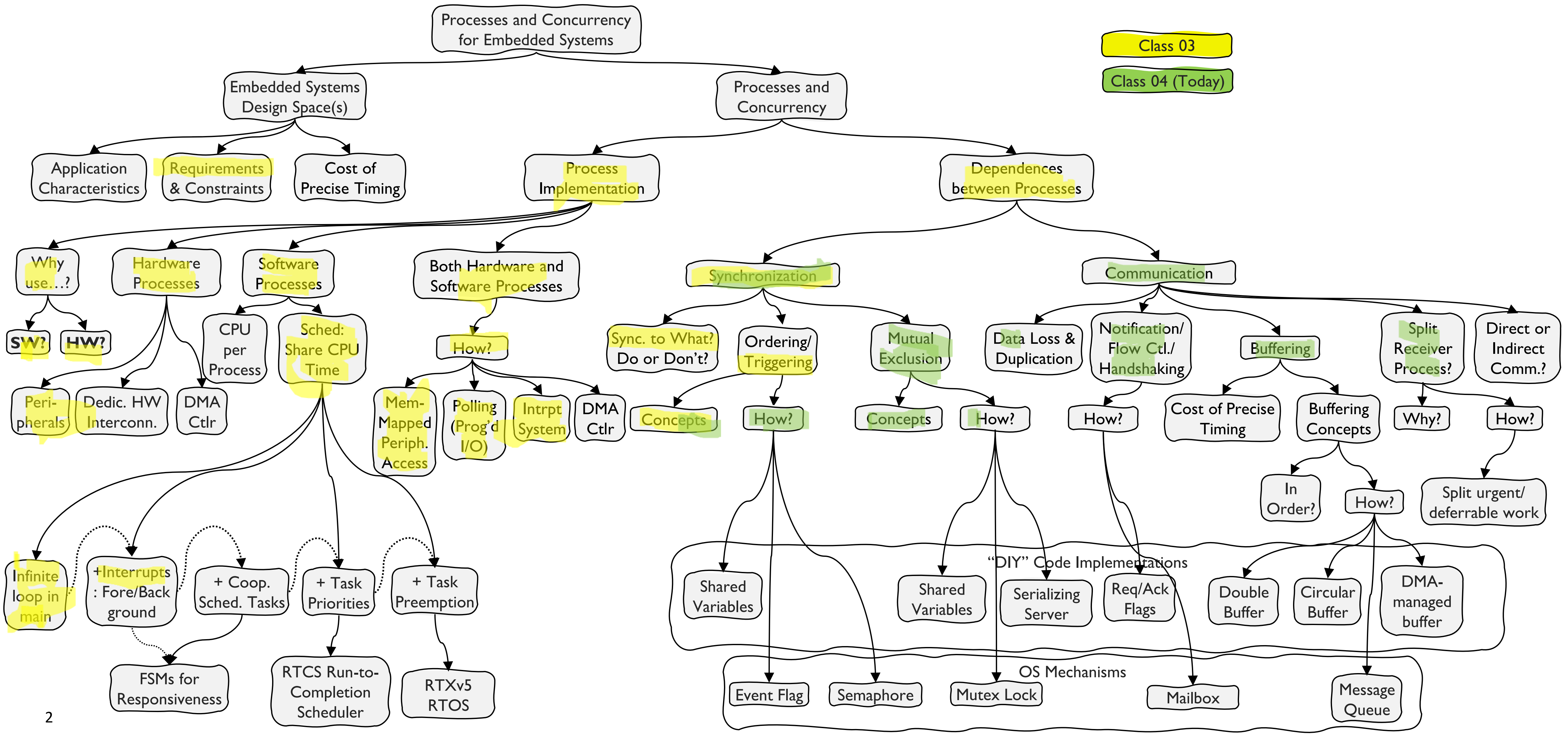
More Synchronization: Synchronization, Communication and Mutual Exclusion

v.4 9/2/25

Extending the Topic Map



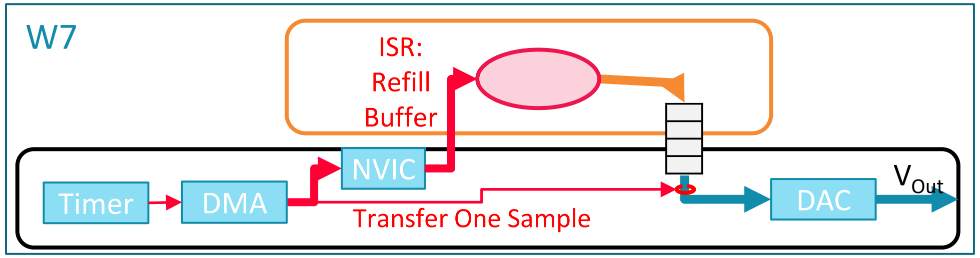
Blocking vs. Non-Blocking Tests

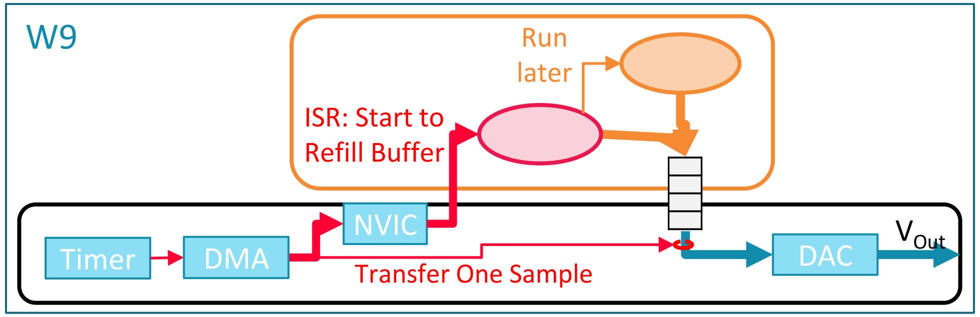
```
prev_A = read signal A from port
while (1) {
  // Quad Decoder - Blocking
  do {
    cur_A = read signal A from port
    detected = (prev_A==0) && (cur_A==1);
    prev_A = cur_A;
  } while (!detected);
  cur_B = read signal B from port
  if (cur_B==0)
    pos++;
  else
    pos--;
  // Other work X
  …
  // More other work Y
  …
}
```

```
prev_A = read signal A from port
while (1) {
  // Quad Decoder – Non-blocking
  cur_A = read signal A from port
  detected = (prev_A==0) && (cur_A==1);
  prev_A = cur_A;
  if (detected) {
    cur_B = read signal B from port
    if (cur_B==0)
        pos++;
    else
        pos--;
  }
  // Other work X
  …
  // More other work Y
  …
}
```

Waveform Generator: ISR to Refill Buffer

- Key Processes
  - Software: DMA ISR calculates new waveform samples, stores them in buffer
  - Hardware: DMA transfers
- DMA transfers a sample from memory buffer





Sync and Don’t: Mutual Exclusion

Data Race Corruption and Mutual Exclusion

Vulnerability	Overlapped Critical sections: memory object updates, access to shared peripheral	**All** memory data object updates are critical sections
Fault Trigger	*Preemption: Overlapping critical sections from different threads for same object*	*Load/Store Architecture: makes updates of **all** memory objects into critical sections*
Solution	*Ensure mutually exclusive execution of a resource's critical sections by using protection algorithm in code, or prevent preemption of critical sections (interrupt/scheduler locking, mutex, …)*	

Different Threads Accessing Same Variable, Peripheral, …

```
volatile int32_t counter=3;
```

```
Thread_1{
    counter = counter + 1;
    // T1.I1 load r0 from memory
    // T1.I2 add 1 to r0
    // T1.I3 store r0 to memory
}
```

```
Thread_2{
    counter = counter + 1;
    // T2.I1 load r3 from memory
    // T2.I2 add 1 to r3
    // T2.I3 store r3 to memory
}
```

- Example: two threads increment shared variable **counter**
  - "Threads" includes software processes, interrupt handlers, etc.
- Compiler translates C (source code) to Arm machine language instructions (object code)
- Arm architecture is a **Load/Store** architecture
  - Variables must be in CPU registers for instr. to process
  - Arm architecture CPU core has 13 32-bit registers for this processing: r0 through r12
  - Need to transfer data between memory and register
- For Arm code, any modification of a memory-resident variable uses at least 3 instructions
  - Read: load memory value into register
  - Modify value in register
  - Write: store register value into memory
- Compiler tries to eliminate extra work
  - In a function, eliminates reads and writes that they don't seem to do anything
  - Volatile: tells compiler the variable may be changed by something else (software thread, hardware), so keep all of its reads and writes.
- For the increment operation, the compiler generates a series of machine language instructions:
  - T1.I1, T1.I2, T1.I3
  - T2.I1, T2.I2, T2.I3

Concurrency Bug: Race Condition

```
volatile int32_t counter=3;
```

```
Thread_1{
    counter = counter + 1;
    // T1.I1 load r0 from memory
    // T1.I2 add 1 to r0
    // T1.I3 store r0 to memory
}
```

```
Thread_2{
    counter = counter + 1;
    // T2.I1 load r3 from memory
    // T2.I2 add 1 to r3
    // T2.I3 store r3 to memory
}
```

- Machine code is vulnerable to generating wrong result for counter increment if instructions are overlapped in certain ways
- Counter starts at 3
- Incrementing counter twice should end with counter at 5
- Some execution sequences trigger race condition in code (type of concurrency bug)
    - One Example: T1.I1, T1.I2, T2.I1, T2.I2, T2.I3, T1.I3
    - Incrementing 3 twice results in 4, not 5

Instruction	counter (in memory)	T1 r0	T2 r3
	3	?	?
T1.I1	3	3	
T1.I2	3	4	
T2.I1		4	3
T2.I2		4	4
T2.I3	4	4	
T1.I3	4	4	

Analysis of Race Condition

```
volatile int32_t counter=3;
```

```
Thread_1{
    counter = counter + 1;
    // T1.I1 load r0 from memory
    // T1.I2 add 1 to r0
    // T1.I3 store r0 to memory
}
```

```
Thread_2{
    counter = counter + 1;
    // T2.I1 load r3 from memory
    // T2.I2 add 1 to r3
    // T2.I3 store r3 to memory
}
```

- Vulnerable if T2.I1,2,3 execute after T1.I1 starts but before T1.I3 finishes.
  - Called “race condition”
  - Other execution sequences cause race conditions
- Problem
  - Code in T1 and T2 to modify counter is sequence of instructions which may be overlapped in execution
  - This creates *critical sections in T1 and T2 for **counter** from each load instruction to the corresponding store instruction* (inclusive)
- Observations
  - Each shared variable is accessed by its critical sections of code in different threads.
  - A variable’s critical sections must be executed without overlapping: no critical section **accessing that variable** starts if another **has started but not yet finished**.

Definitions

- Race condition
  - An ordering of instructions in multiple threads (or ISRs) on one or more cores which causes code to behave anomalously.
  - Is there a way to jump back and forth between two threads and get the wrong answer?
- Critical section
  - A section of code which creates a possible race condition.
    - *Any access to a shared object in a system with preemption is a critical section of code*
  - Only one critical section per shared object can be executed at a time.
    - OK to execute multiple critical sections concurrently if they access different shared objects.
  - Some synchronization mechanism is required at the entry and exit of each critical section to ensure exclusive use.

Generalizations

- Shared variable is example of a **shared resource**. Peripherals, other items may be shared as well.
  - LCD Controller accepts series of bytes: Command, data, data, data, …..
- Threads sharing resources are vulnerable to *race conditions*
- How to protect the critical sections? We'll see later.
  - Shared resource accesses must be executed atomically without overlapping.
- For code compiled for a load-store arch. (e.g. Arm), even single-word variables are vulnerable.
  - Turns some operations into non-atomic instruction sequences.
- May take a deeper look later
  - Do any instructions give atomic memory access, and how are they used?
  - What about multi-core CPUs?
  - What about an increment-in-memory instruction?
  - What about a multi-core CPUs with an increment-in-memory instruction?

Preemption and Data/Operation Corruption

Vulnerability

Overlapped Critical sections:
memory object updates,
access to shared peripheral

All memory data object updates are critical sections

Fault Trigger

Preemption: Overlapping critical sections from different threads for same object

*Load/Store Architecture: makes updates of **all** memory objects into critical sections*

Solution

Use protection algorithm in code, or prevent preemption of critical sections (interrupt/scheduler locking, mutex, …)