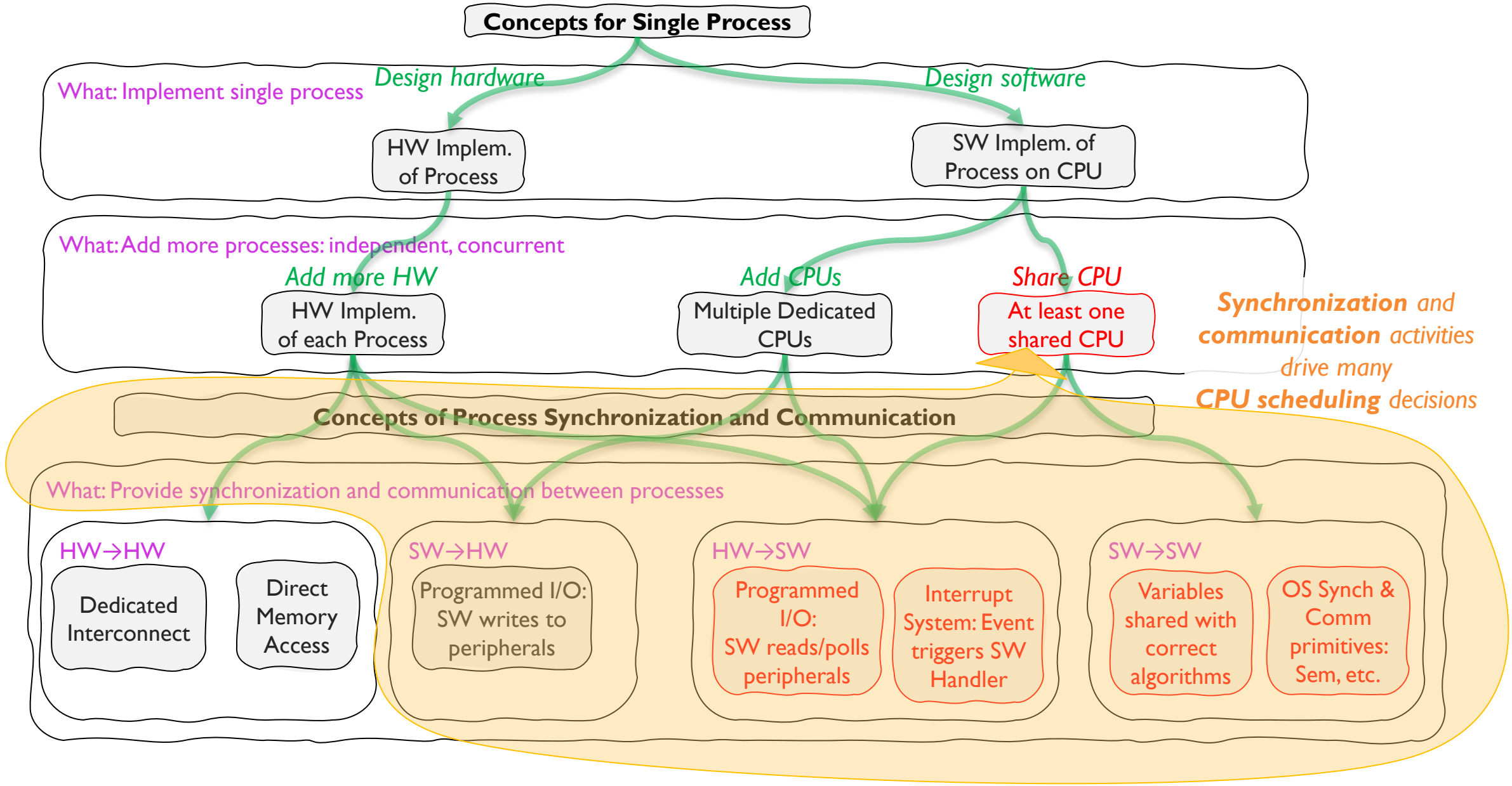
06: Scheduling and Dispatching, Response Time Analysis and OS Wish List

9/8/2025

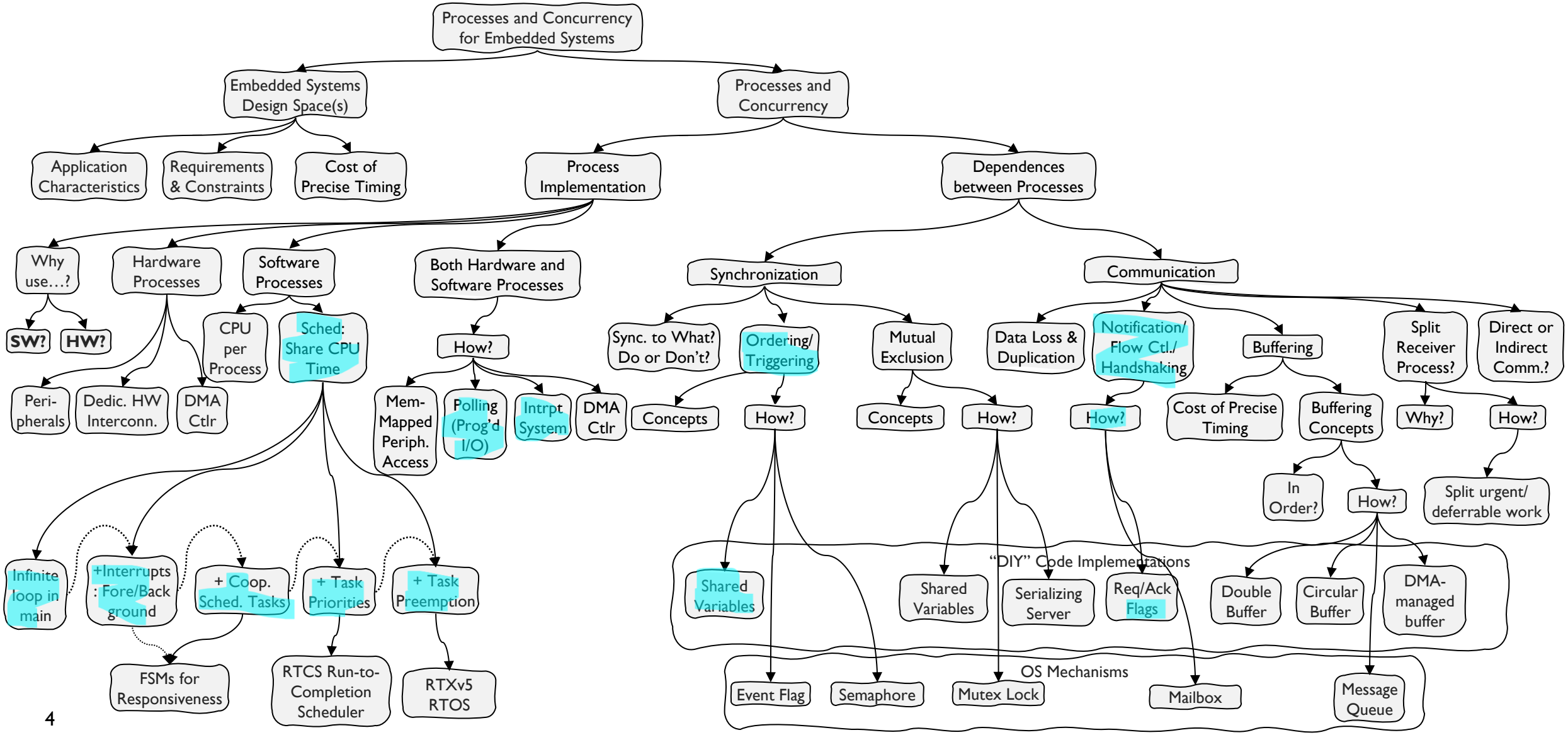
Overview

- Where are we?
- Examining the Processing Chain
  - Scheduling and Dispatching: Where are they done?
- Response Time Analysis
- OS Wish List

SW Processes: CPU Scheduling, Synchronization & Communication



Extended Topic Map: Class 06

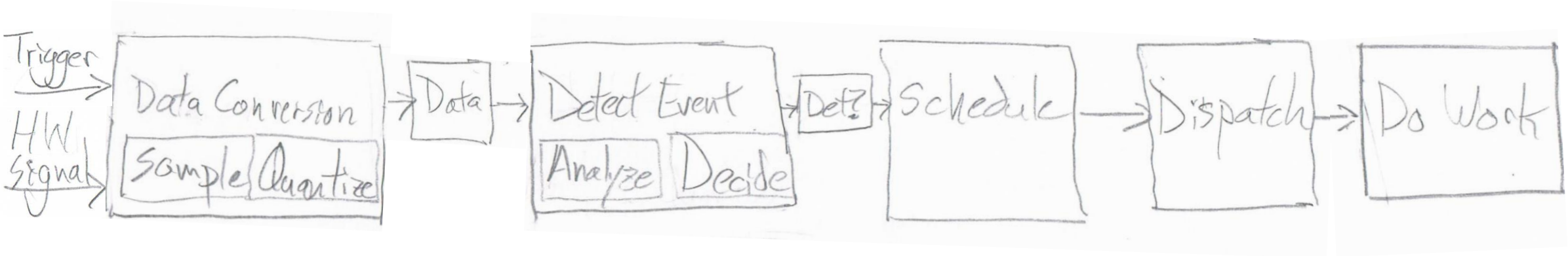


Software Processes and Arm CPU Modes

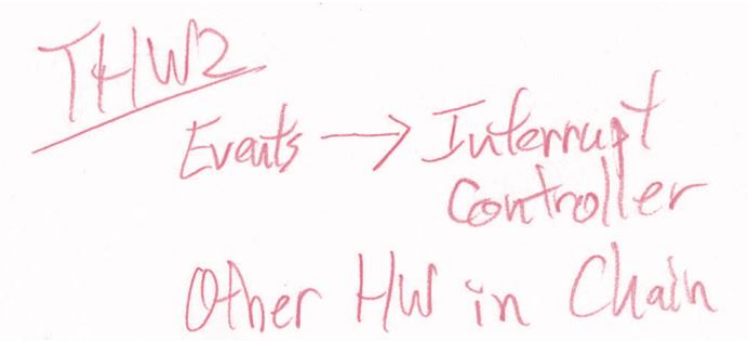
- Arm CPU may run a SW process in…
  - Thread Mode
  - Handler Mode
- Main differences: *These don't matter yet*
  - Which stack pointers (SP) are available
    - Main SP, Process SP
  - Which access privilege levels are available
    - Unprivileged level prevents using certain instructions and accessing certain peripherals, control registers and memory regions
    - Privileged level has no restrictions
- Transitions
  - Thread mode -> handler mode: When starting to respond to an interrupt or exception request
  - Handler mode -> thread mode: after finishing handling last nested interrupt or exception request

Processing Chain: Schedule and Dispatch Stages

Processing Chain Refinement



- Is data visible to software?
  - Yes: variable in memory
  - No: Need to convert data so it is visible
    - Trigger data conversion
    - Convert Data: Sample and quantize signal into digital data
- Detect Event: analyze data, decide if event happened
  - Save that decision for scheduler
- Schedule Process: Use detector's decision to pick which processing to do next (e.g. do process work)
- Dispatch Process: Start it running (or resume it)
- Do Work: Perform processing work to handle event



Dispatching a Software Process

Comes after scheduling, but let's get it out of the way early

- Dispatch = cause SW process to start/resume executing
- Different methods available
  - Implicit: next instruction in code is part of the process

```
if (ev_A_det) {
   // implicit dispatcher
   a_work_1 …;
   a_work_2 …;
   a_work_3 …;
}
```

  - Subroutine call to with process' root (overall) function – for better modularity

```
if (ev_A_det) {
   A_Work(); // Subroutine call is dispatcher
}
```

- Interrupt Controller forces CPU to execute ISR containing code or call to process root function

```
// Interrupt System is dispatcher
ISR_Peripheral {
   a_work_1 …;
   a_work_2 …;
   a_work_3 …;
}
```

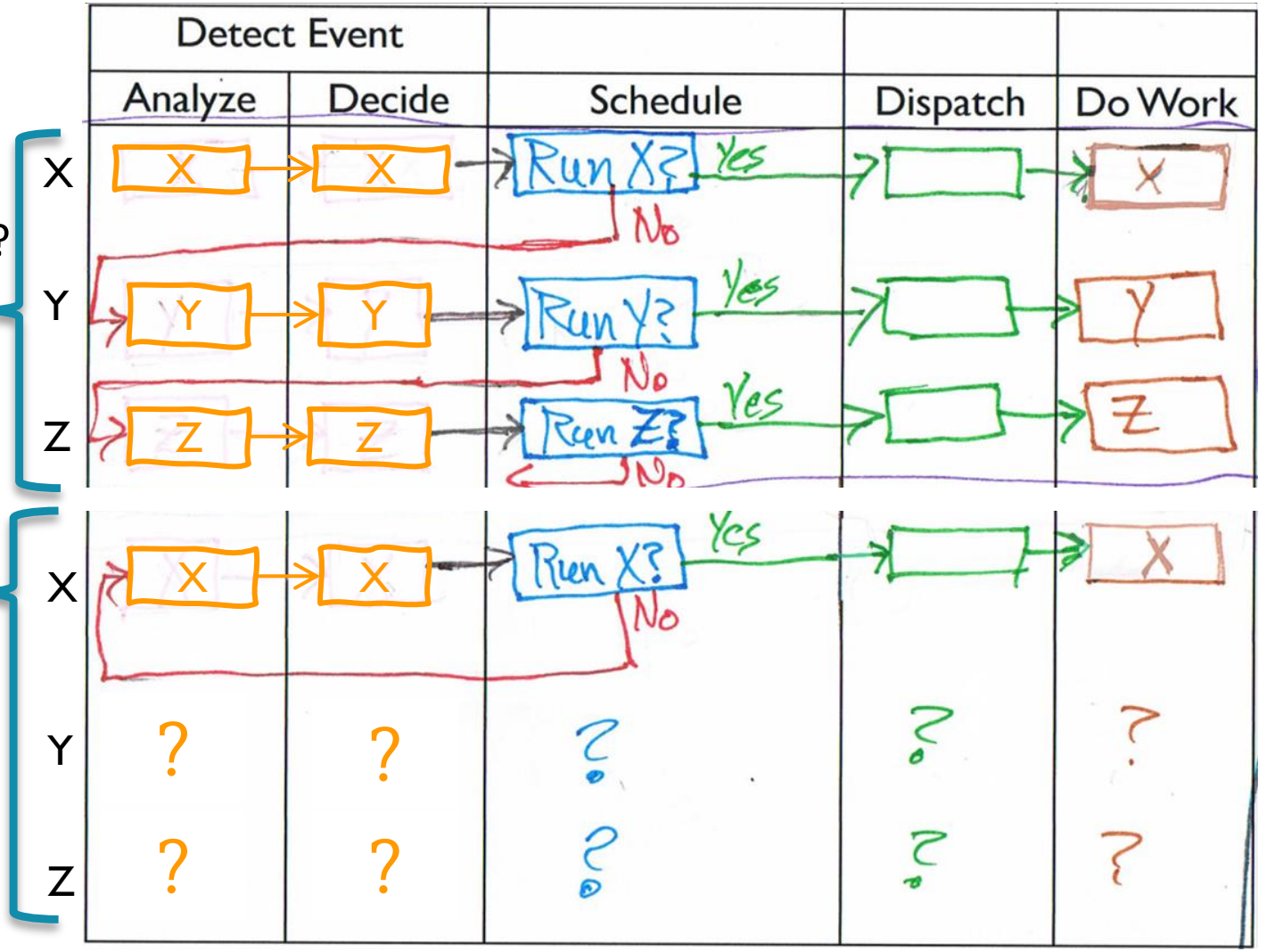
- Process or interrupt handler ask OS to do something, which may cause OS to run its scheduler and dispatch a process with context switching.

- Interrupt Controller forces CPU to execute handler which does work. It may also ask OS to do something, which may cause OS to run its scheduler and dispatch a process with context switching.

```
// Interrupt System is dispatcher
ISR_Peripheral {
   a_work_1 …;
   a_work_2 …;
   a_work_3 …;
   OS_reschedule();
}
```

Scheduling & Dispatching: Decide what to do next, and start it

- Behavior depends on two decisions
- 1. Is this process **allowed to run**?
  - Yes: Dispatch and run it
  - No: 2. What kind of event detection **test**?
    - Non-blocking: Advance to detect stage for next process (via scheduler or program structure) and continue
    - Blocking: Repeat software starting with detect (analyze and decide) by looping back to it



System Timelines for Non-Blocking vs. Blocking Detection

- Resulting system timelines
  - Non-blocking detection: round-robin
  - Blocking detection: greedy

Processing Chain Variations 1: Where to Detect & Schedule?

```
main() {
  while (1) {
    // Detect Event for A
    ev_A_det = ….
    if (ev_A_det) {
        ev_A_det = 0;
        A_Work();
    }
    // Detect Event for B
    ev_B_det = ….
    if (ev_B_det) {
        ev_B_det = 0;
        B_Work();
}}
```

- Main Thread Loop: Scheduling loop in main thread
  - Main polling event detection code

```
A_Det_Sched_Work( ) {
  ev_det = …
  if (ev_det) A_Work();
}
B_Det_Sched_Work( ) {
  do { // blocking
    ev_det = …
  } while (!ev_det);
  B_Work();
}
main() {
  while (1) {
    A_Det_Sched_Work();
    B_Det_Sched_Work();
}}
```

- Process: in Do Work
  - May have built-in Detect/Schedule/Do
  - Schedule may be single non-blocking test or looping blocking test

```
ISR_A( ) {
  A_AllWork();
}

main() {
  while (1) {
    …
}}
```

- Interrupt System: Hardware
  - Peripheral detects, interrupt controller schedules & dispatches, interrupt handler (ISR) does all the work

Processing Chain Variations 2: Where to Detect & Schedule?

- Interrupt System & (Main or Process): Multiple locations
  - Combines interrupt approach with another
    - Allows splitting of work between ISR and thread for better responsiveness
    - Needs synchronization between processes
    - Results in foreground/background system
  - Operation
    - Peripheral detects, interrupt controller schedules, handler does some work and requests more processing (ev_A_det)
    - Main loop detects request, schedules process, process does requested processing work
      - Could instead use A_Det_Sched_FinishWork()
- Note:
  - Operating system will give us a scheduling point (reschedule and dispatch) every time our process uses the OS

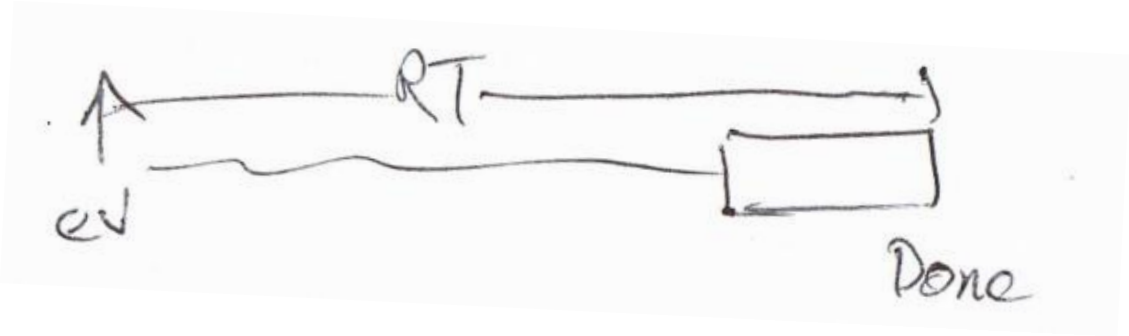
```
volatile int ev_A_det = 0;

ISR_A( ) {
   A_StartWork();
   ev_A_det = 1;
}
A_FinishWork( ) {
   …
}
B_Det_Sched_Work( ) {
   do {
      ev_det = …
   } while (!ev_det);
   B_Work();
}
```

```
main() {
   while (1) {
      if (ev_A_det > 0) {
           ev_A_det = 0;
           A_FinishWork();
      }
      B_Det_Sched_Work();
   }
}
```

Response Time Analysis

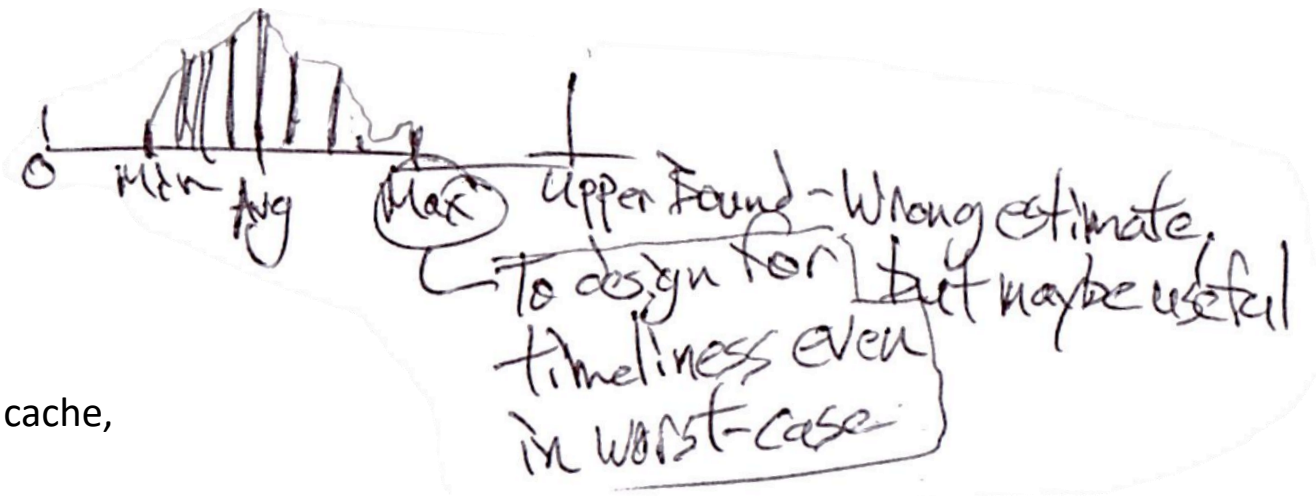
Starting Point for Response Time Analysis



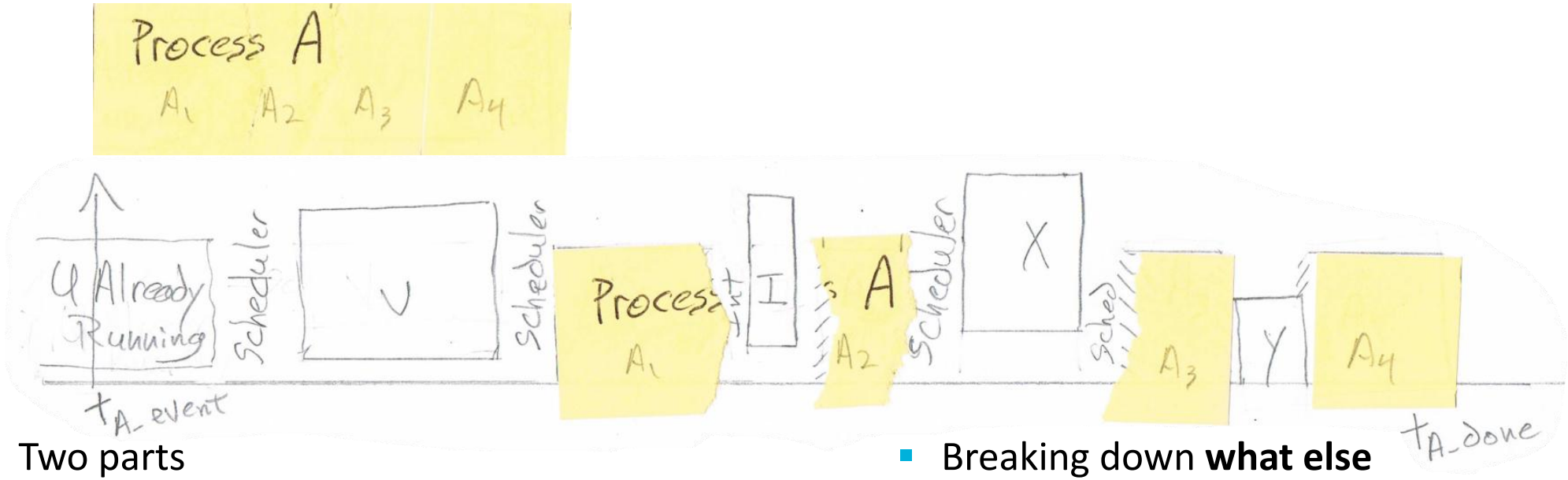
- Response time = time between **event** and **completion of response processing**
- RTA for **which** process?
  - This process?
  - Other processes in the system? How does this process affect/disrupt timing for the other processes in the system?
- Managing variations in processing chain structure
  - Standardize to simplify timing analysis:
    - Assume detection (analyze, decide) and simple mini-scheduler (if) for process is performed in its Do Work stage
      - Still have outer scheduler deciding which process to run next
    - Process Do Work stage is short if event not detected, longer if event is detected
  - Can link multiple processing chains together if the processes synchronize with each other

Overview of Basic Approach to RTA

- Two parts
  - How much **processing time** is needed for **software process A**'s instructions?
  - **What else can run** between input event at t_{A_event} and response processing completion at t_{A_done}? **How long does it take**?
    - Includes other processes and the scheduler/operating system process(es)
- Part 1: How much **processing time** is needed for **software process A**'s instructions?
  - May have **range** of possible times. Variations comes from:
    - **Which instructions** in A are executed. Dependence on variations in
      - Input data, event timing & sequences, …
    - **How long the hardware takes** to execute those instructions. Dependence on variations in
      - Instruction execution time from pipeline, multiple function units, out-of-order, branch prediction, etc.
      - Memory system access time from flash accelerator, cache, virtual memory system…



Overview of Basic Approach to RTA



- Two parts
  - How much **processing time** is needed for **software process A**'s instructions?
  - **What else can run** between input event at t_{A_event} and response processing completion at t_{A_done}? **How long does it take**?
    - Includes other processes and the scheduler/operating system process(es)
- Breaking down **what else**
  - **How soon** does the scheduler start **running A**?
    - Is a **process already running** that will delay when the scheduler gets to run? **U**
    - Does the scheduler have **other processes to run before** starting A? **V**
  - Can anything **delay A after it has started**?
    - Can anything **preempt** A after it starts running? **I, X**
    - Could A have to **wait for another process** for synchronization? **Y**

RTA Examples with Four Schedulers

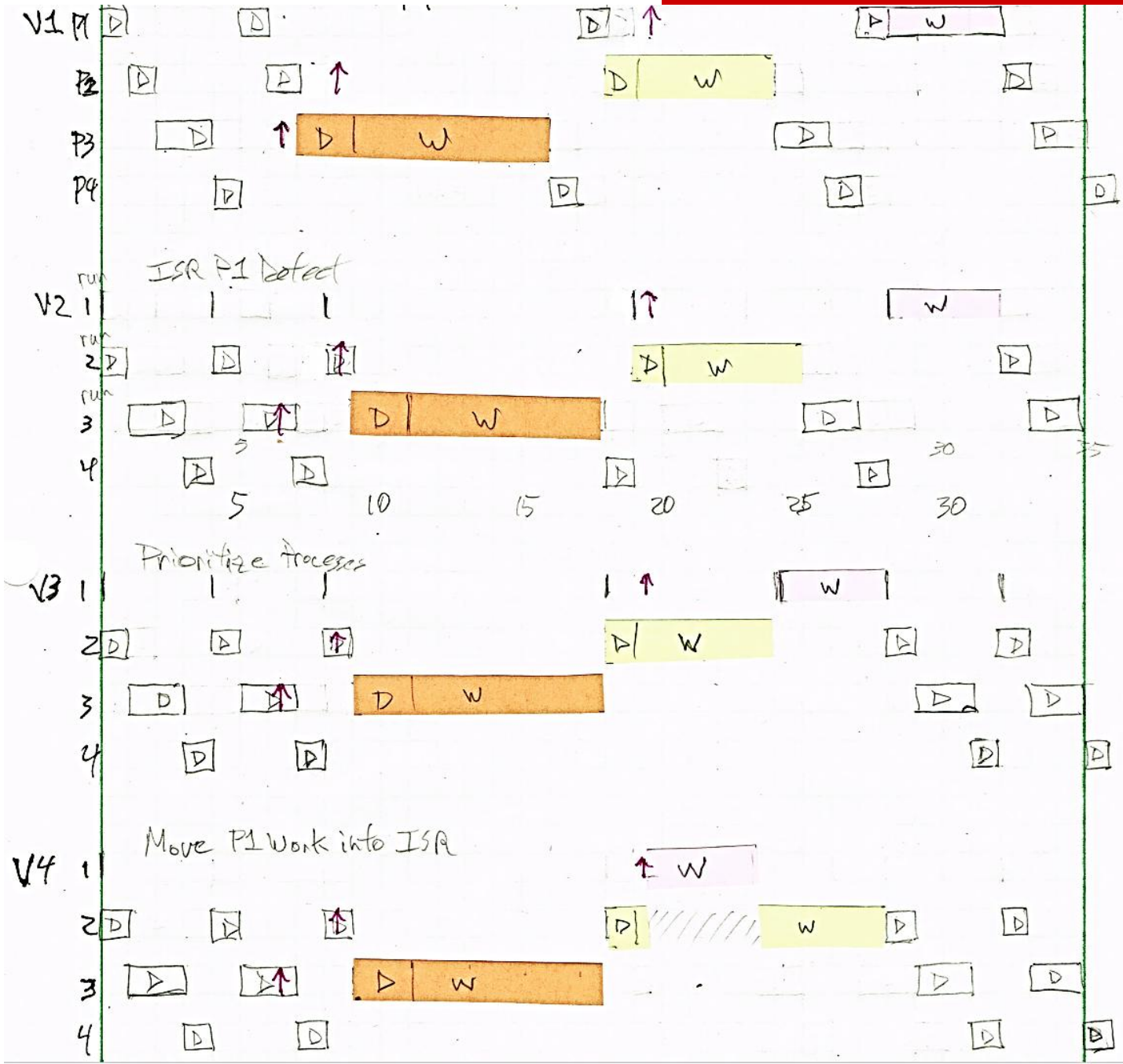
- Workload
  - Processes: P1, P2, P3, P4
    - Structure for each process
      - Run Detect code first
      - If detected, run Work code
  - Triggering Events (or conditions): E1, E2, E3, E4

	C_{Detect}	C_{Work}
P1	1	4
P2	1	5
P3	2	7
P4	1	1

- Scheduler Versions
  - V1. Fixed order round-robin. P1, P2, P3, P4, repeat.
  - V2. And Interrupt detects event E1, Interrupt handler sets flag requesting a run of P1
  - V3. And Prioritize processes. P1 > P2 > P3 > P4
  - V4. And Move P1 work into ISR

Execution: Event Timing R1

	Event Time (R1)
E1	19.5
E2	8.5
E3	6.5
E4	n/a



Execution Schedules With Event Timing R2

	Event Time (R1)
E1	
E2	
E3	
E4	n/a

Execution Schedules With Event Timing R3

	Event Time (R1)
E1	
E2	
E3	
E4	n/a

Response Times for Schedulers

	Release Schedule R1	Release Schedule R2	Release Schedule R3
P1			
P2			
P3			
P4			

Observations

Observations

- Limitations of run-to-completion process model.
  - Thread duration vs. responsiveness
  - Thread preemption only by interrupts, complicating design
  - Shortening threads with finite state machines

- Sync/comm/sched/dispatch operations are often interdependent
  - If scheduler/OS can see all these operations, it can make better decisions and offer more features
  - Example: If process event test will block, then pause process execution and automatically switch in another process. Reclaims idle time.

Observations

- Scheduling model variations: Where are detect and schedule performed?
  - A. Main sched thread (Det, Sched) only: in main loop with polling detection
  - B. Interrupt only: peripheral detects, interrupt controller schedules, handler does
  - C. Main & Interrupt system:
    - Peripheral detects, interrupt controller schedules, handler does some work and requests more processing
    - Main loop detects by polling request, scheduling process, process does requested work
  - D. Do Work/Handler
    - Do Work/Handlerportion of process may also contain Get/Detect/Schedule/Do built in, where Schedule step may be single non-blocking test or looping blocking test

OS Wish List

Wish List - Better control of responsiveness

- Methods to provide ("allocate") responsiveness to processes (or parts within) as needed
  - Don't waste responsiveness on processes which don't need as much
  - Reduce vulnerability of responsiveness for urgent processing by isolating it from less urgent processing
- Improvements
  - Improve task execution order (not round-robin A B C D A B C D)
  - Add process priorities, use to drive scheduling.
    - Static priority?
    - Dynamic? Based on slack time?
    - Both?
  - Improve run-to-completion processes (non-preemptive) with yield and resume features: Finite state machines (FSMs), other methods. Cooperative multitasking
  - Provide preemption of processes by higher priority processes

Generalize/standardize code structure for modular code

- Standardize data format for scheduler
  - Essential data: process is ready (has permission to run since event was detected). Count to 1 or higher?
- Provide protected interface for scheduler data. E.g. request another run.
- Support scheduling decisions more locations: scheduler code, user code

Features to simplify programming

- Support time-based process scheduling (e.g. with periodic timer tick)
  - Run this process every N ticks, etc.
- Features for synchronization between processes:
  - Signaling event has occurred, counting pending un-serviced events. E.g. for triggering processing: ISR->thread, etc.
  - Protecting critical sections with mutually exclusive execution.
- Features for communication between processes
  - Send a message: data and provide sync support for receiver (and sender too!)
  - Send a message, allowing multiple pending messages (FIFO/queue)

Features to reclaim idle time

- Take advantage of OS knowledge of system state
  - Switch processes when blocking

- Leverage preemption

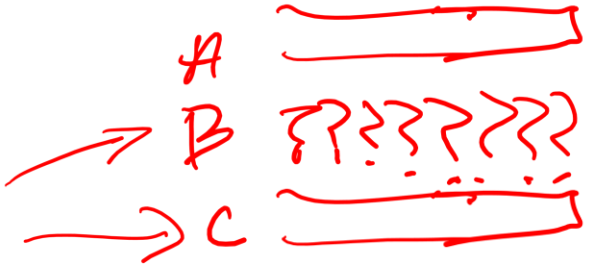
Unused

Connections between Synchronization and Scheduling

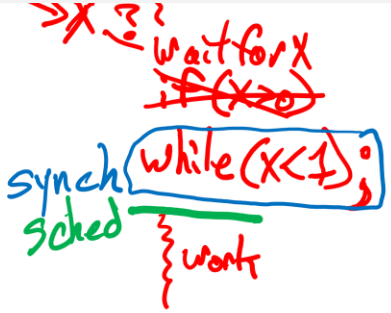
```
Process A
…
x=1;
…
```

```
Process B
…
// Detector
while (x<1)
  ;
x=0;
// Scheduler
task_ready = 1;
// Dispatcher
if (task_ready) {
  // Handler process
  do_the_work();
}
```

- Synchronization **must** be able to make a process **wait** until an eve
  - Spinlock example
  - BTW, communication often includes synchronization (“Wait for message”)
- Consequences of a process waiting depend on implementation
  - Dedicated processor (HW, SW on own CPU)? No problem, not delaying anything else.
  - Shared processor (SW on shared CPU)? Has issues
    - While waiting, process B is not doing useful work
    - Waiting process might block all other processes A, C… using that CPU for simple schedulers (e.g. main loop)
    - **Relies on other support** to make it practical. Time slicing, interrupt triggering handler to set X, modifying B to share CPU, some other kind of scheduling, etc.



- Observation and Opportunity
  - Synch and Comm operations are good places to make scheduling decisions.
  - CPU could use that time to work on another process, sharing CPU time better.
    - What does CPU do if sync condition test fails? Must choose from available methods
    - When does CPU do next sync condition test?



Synchronization and Scheduling Interdependence

```
…
x=1;
…
```

```
…
while (x<1)
    ;
x=0;
task_ready = 1;
if
(task_ready)P
do the work
```

- Synchronization **must** be able to make a process **wait** until an event occurs
  - Spinlock example
  - BTW, communication often includes synchronization (“Wait for message”)
- Consequences of a process waiting depend on implementation
  - Dedicated processor (HW, SW on own CPU)? No problem, not delaying anything else.
  - Shared processor (SW on shared CPU)? Has issues
    - While waiting, process B is not doing useful work
    - Waiting process might block all other processes A, C… using that CPU for simple schedulers (e.g. main loop)
    - **Relies on other support** to make it practical. Time slicing, interrupt triggering handler to set X, modifying B to share CPU, some other kind of scheduling, etc.
- Observation and Opportunity
  - Synch and Comm operations are good places to make scheduling decisions.
  - CPU could use that time to work on another process, sharing CPU time better.
    - What does CPU do if sync condition test fails? Must choose from available methods
    - When does CPU do next sync condition test?

Synchronization and Scheduling Interdependence

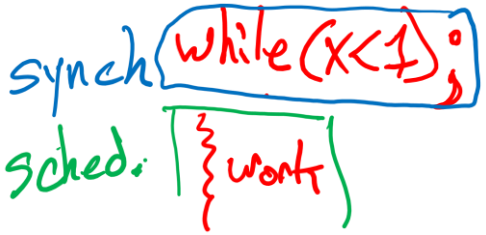
```
…
x=1;
…
```

```
…
if (x>0)
   do work
else
   do what?
```

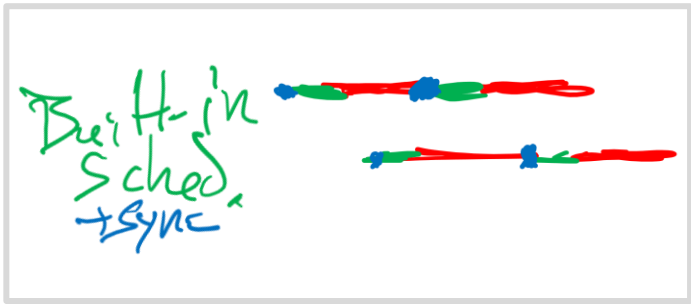
- Synchronization **must** be able to make a process **wait** (not proceed) until an condition becomes true occurs
- Synchronization by software polling
- Version 1:
  - Test the condition
    - If true, can continue doing the work
    - If not true, then do what?
- Busy-wait polling example
  - Synchronization test done in software
  - If test succeeds, After test

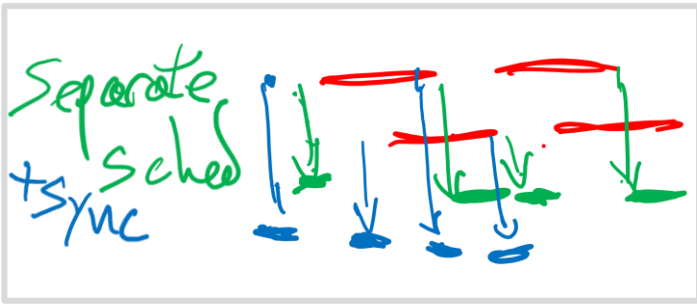
```
…
while (x<1)
   ;
x=0;
work
```

Mechanisms for Sharing/Scheduling CPU's Time



- Implicit, defined by instruction order in process code
- CPU's interrupt system
- Explicit software scheduler
  - Where?
    - Integrated into program? FSM, etc.
    - In separate modular support software?
  - Key Feature: Cooperative or preemptive process(task) scheduling?
    - Preemption simplifies design of software, improves responsiveness (usually)

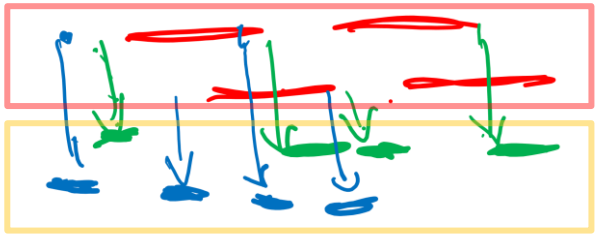


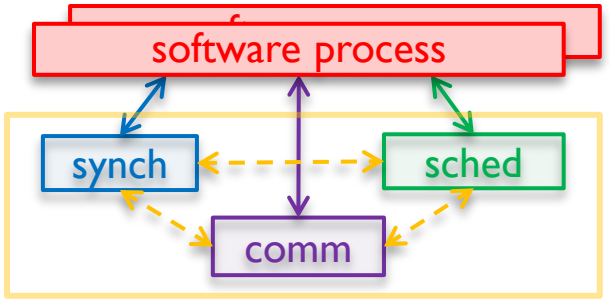


- Bottom line
  - Choice of scheduling approach (and the underlying support it relies upon) affects feasibility of different synchronization & communication options

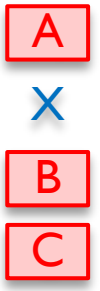
Modular Software for Sched/Sync/Comm

- Design software process to use standardized software component to schedule CPU time
  - Simplifies design, makes it easier to *get it right*
  - Also use for synchronization and communication
- Benefits of Sched/Sync/Comm interactions
  - Components can cooperate to provide useful (and more complex) behaviors, offloading implementation from process (easier!)
- Example: Process B awaits synchronization condition X
  - If sync condition X is true, then
    - if B is highest priority ready process, then
      - schedule B to resume running
    - else B is not highest priority ready process, so
      - schedule the highest priority ready process to resume running
  - else sync condition X is not true, so
    - make a note that B is waiting for X, revisit issue when X becomes true
    - schedule the highest priority ready process to resume running
- Package these up: foundation of an operating system

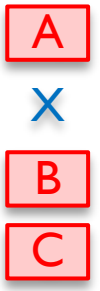




X is True



X is False

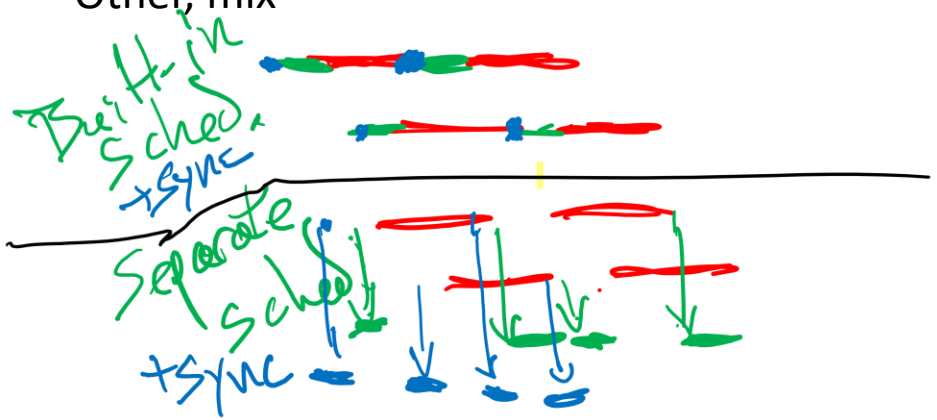


CPU Scheduling of Software

- How do we want the CPU's time to be shared among software processes?
  - Fairly?
    - Equal **chances** to run?
    - Equal **time** to run? Time-slice.
    - Something else, some combination, etc.
  - Priorities?
    - Based on what? Urgency? Importance to application?
    - Static (fixed) or Dynamic (changing)?
    - Affected by communication and synchronization?
    - Something else, some combination, etc.
  - Many other aspects possible to consider. We'll see some useful ones later
- Implementation and resource requirements
  - How hard will it be to implement this in a scheduler?
  - How will it affect our design process?

(Original Version: Consequences on CPU Scheduling of Software)

- How do we want the CPU's time to be shared? scheduling
  - Fairly?
    - Equal chances to run?
    - Equal time to run? Time-slice.
    - Other, mix
  - Priorities?
    - Based on what?
    - Static or Dynamic?
    - Other, mix



while(x<1)
;

FSM
+Simple loop
sched.

- What mechanism shares the CPU's time?
  - Interrupt system
  - Implicit software scheduler in program
  - Explicit software scheduler
    - In program or support software?
    - Cooperative or preemptive

- Bottom line: Choice of scheduling approach impacts synch & comm option feasibility