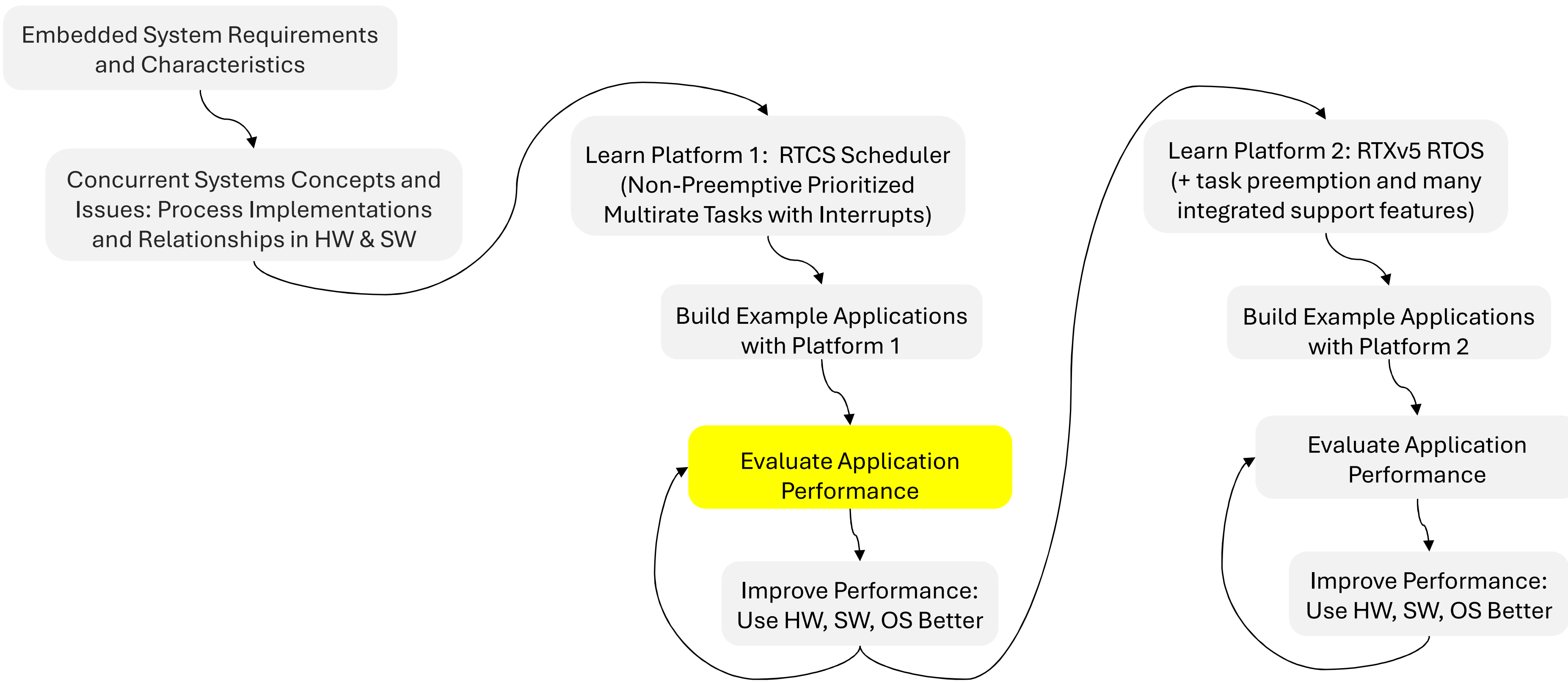
RESPONSE TIME ANALYSIS CONCEPTS FOR SOFTWARE

V3
9/30/2025

NC STATE UNIVERSITY

Where are we in the class?



NC STATE UNIVERSITY

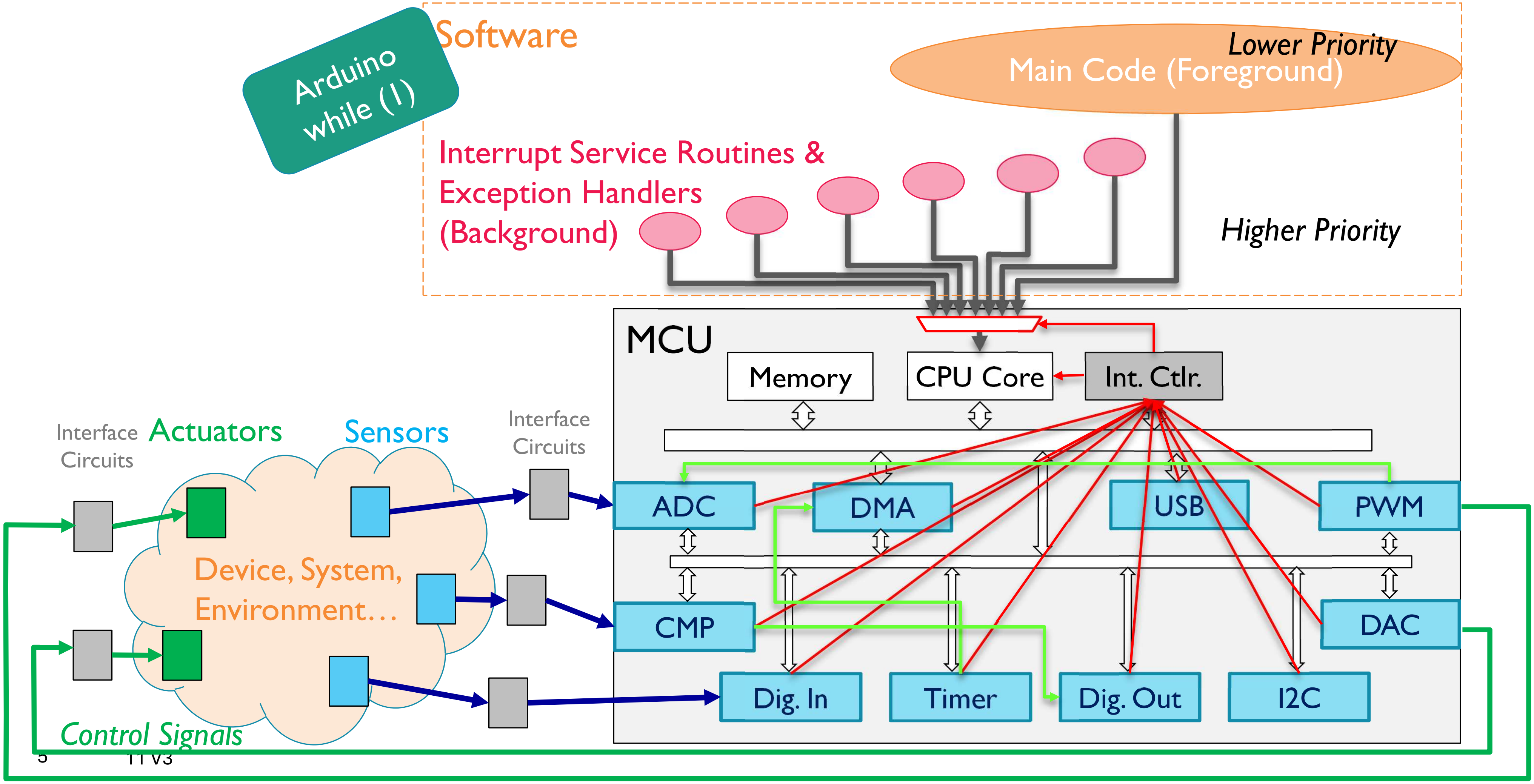
LN11 – Concepts for Response Time Analysis

- Key Concepts
  - Process PA needs to run some of its code in response to event EvA
  - Other software in system can delay that running
    - Blocking by another currently-running process
    - Delays from other processes based on process scheduling
      - Non-prioritized: all other processes
      - Prioritized: higher-priority processes
    - Preemption by other processes
      - Interupt/Exception handlers (ISRs)
      - Preemptive process scheduling: Preemption by higher-priority processes
  - Estimate worst-case response time for task i (R_i)
    - First estimate
      - Start assuming all events happen simultaneously
      - R_i is sum of time needed to process all events
    - Determine if any more events happened in that time R_i
      - If so, update R_i to include additional event processing
      - Repeat until done
  - Process is more vulnerable to timing interference if scheduler supports task preemption
- Contents
  - Schedulers help software share the CPU's time better
  - Task run behavior: Run-to-completion vs. yielding and preemption
  - Scheduler

NC STATE UNIVERSITY

RESPONSE TIME ANALYSIS

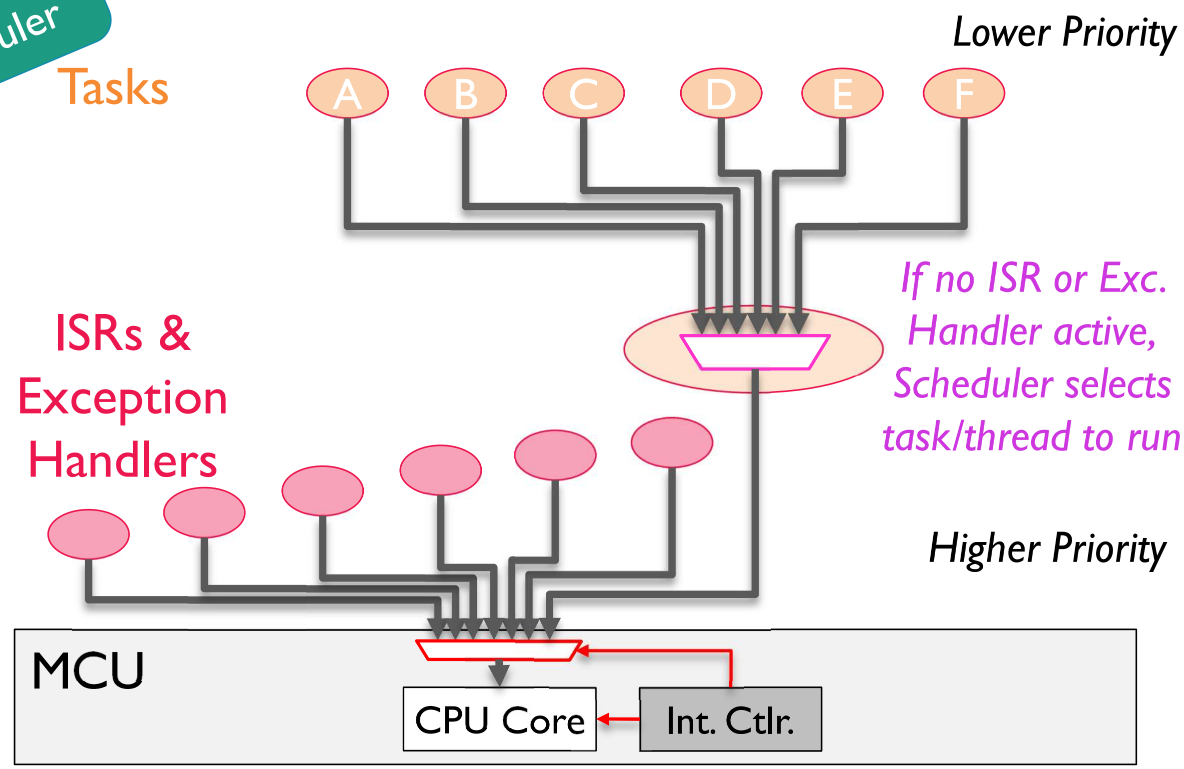
Parallel Hardware … but Serialized Software



Schedulers: Helping Software Share the CPU Better

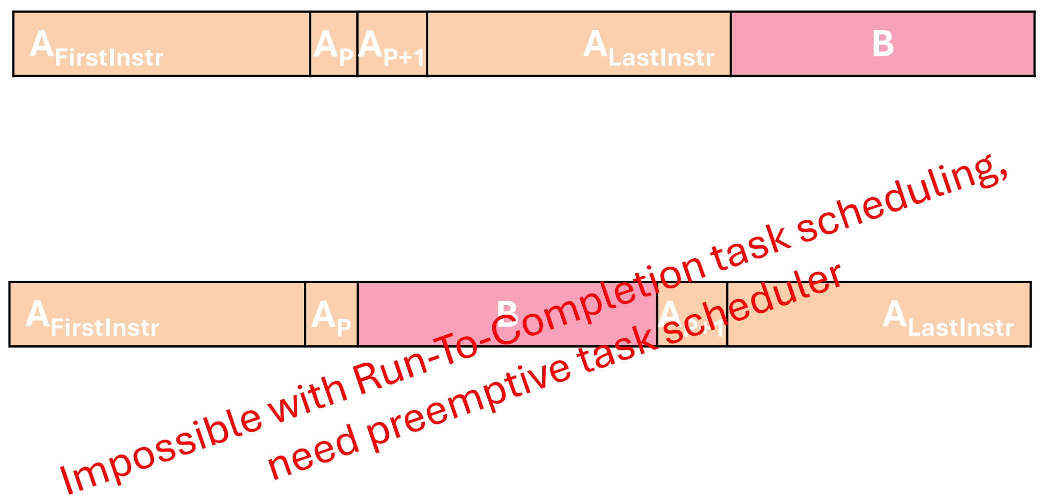
Arduino Scheduler

- Build modular program
  - Separate tasks/threads and ISRs, each running (mostly) independently
  - Easier to develop, maintain, debug
- What code does CPU run?
  - Normally CPU executes next instruction in program,
  - But **interrupt controller** can force CPU to execute handler code for interrupt or exception request
  - **Task scheduler can** decide which task/thread to run next

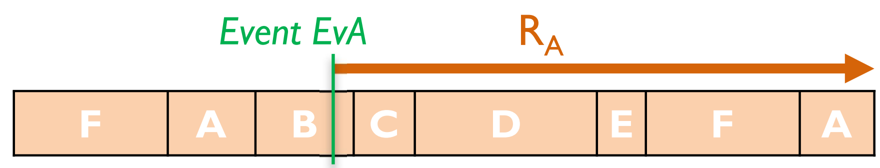


Run-To-Completion Tasks and Task Preemption

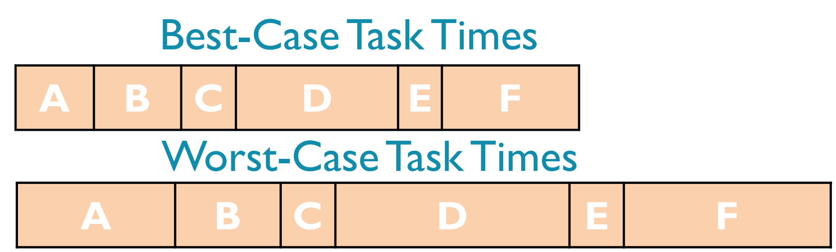
- With run-to-completion task scheduling…
  - Scheduler must wait for current task to **complete** before running another task
  - Tasks cannot pause (or be preempted by other tasks) partway through, and later resume at that point within the task.
  - If scheduler is running task A, it cannot
    - Pause A partway through (after instruction A_P),
    - Switch in task B and run it,
    - Resume task A partway through (at instruction A_{P+1})
- Preemptive task scheduler will support such task switching …
  - Letting task B preempt task A
  - Letting task A yield the CPU and later pick up where it left off



Responsiveness of While (1) Loop Scheduler



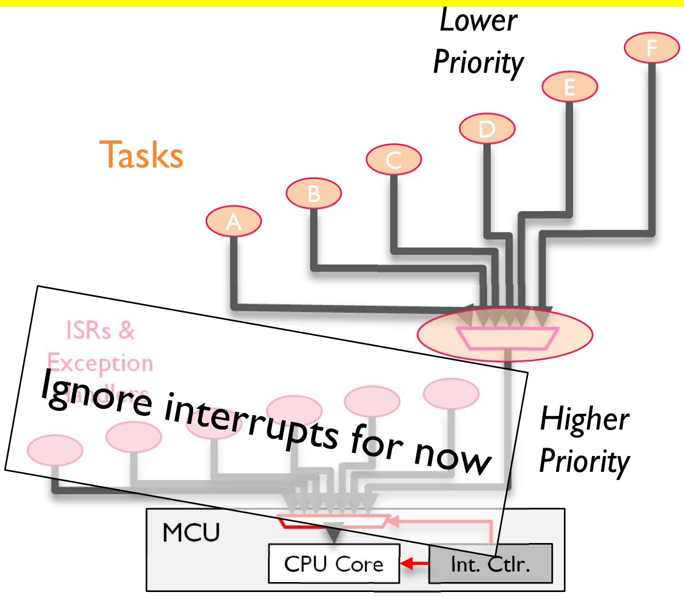
- Task **A** must run to service (handle) its event **EvA**
  - EvA makes scheduler ***release*** task A
- Task A's **response time** (R_A): How long from event EvA until task A **finishes** servicing it?
- Scheduler is While (1) loop
  - Tasks run to completion.
  - Fixed schedule: same task order every time
  - Round-robin: each task gets same number of chances to run
- Scheduler behavior:
  - EvA happened? Release A, run A until done.
  - EvB happened? Release B, run B until done.
  - EvC happened? Release C, run C until done.
  - Continue for all events/tasks, then repeat with EvA

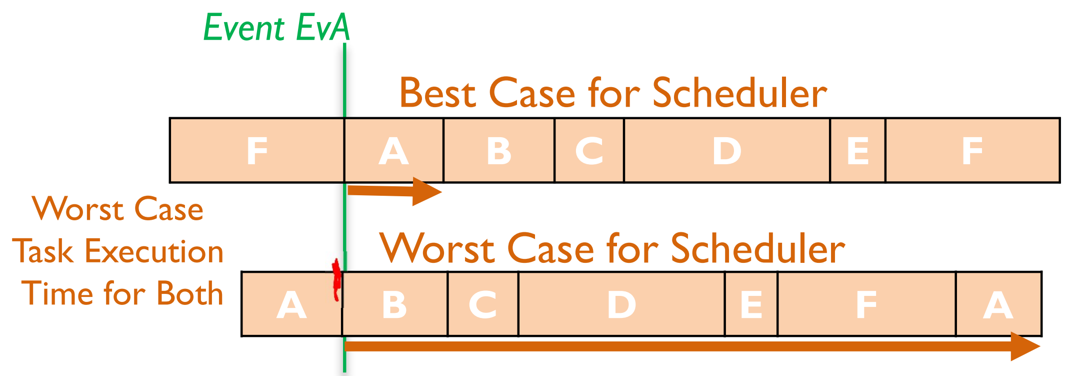


- Note
  - We assumed each task takes a constant amount of time to execute
  - Task i probably has range of possible execution times, between $C_{i,Min}$ and $C_{i,Max}$
- Simplify timing model by making some worst-case performance assumptions
  - Design for worst case, so assume task i always takes $C_i = C_{i,Max}$
- Model will likely overestimate response time, but will never underestimate it – so it will be **safe**.

NC STATE UNIVERSITY

Responsiveness of While (1) Loop Scheduler

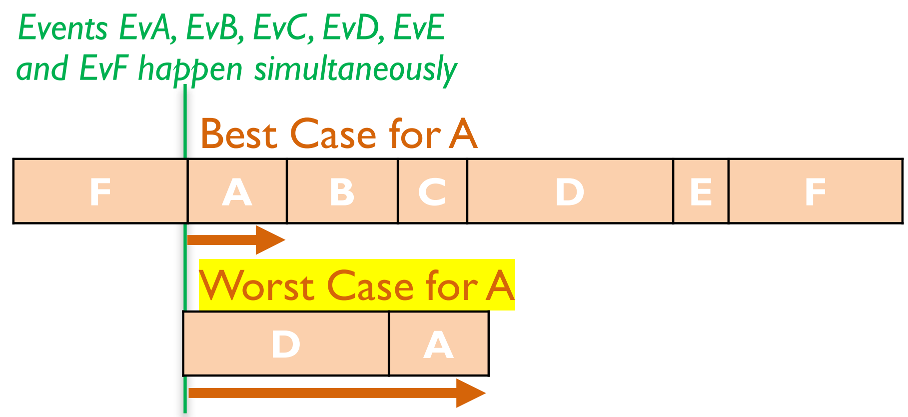




- Task A's **worst-case** response time: What is the **longest possible time** from event EvA until task A finishes servicing it?
  - Depends on what code runs: Interrupt Controller decides on ISRs, task scheduler decides on tasks

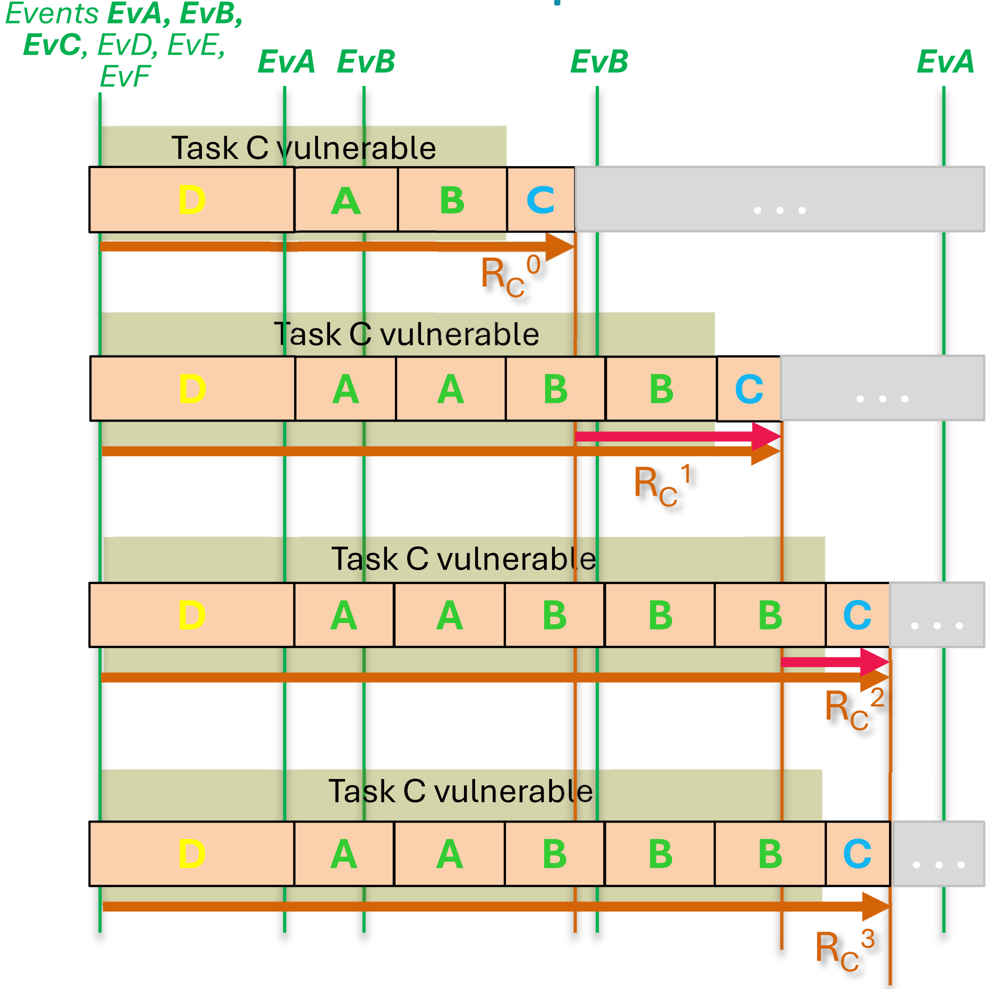
- Simplify: Initially ignore time taken by scheduler, interrupt system and interrupt handlers.
  - Best case: EvA happens just before scheduler checks it. $R_A = C_A$
  - Worst case: Every other event (EvB – EvF) happens before scheduler checks EvA, and EvA happens just after that check: $R_A = C_B + C_C + C_D + C_E + C_F + C_A$

Improvement: Prioritized Tasks



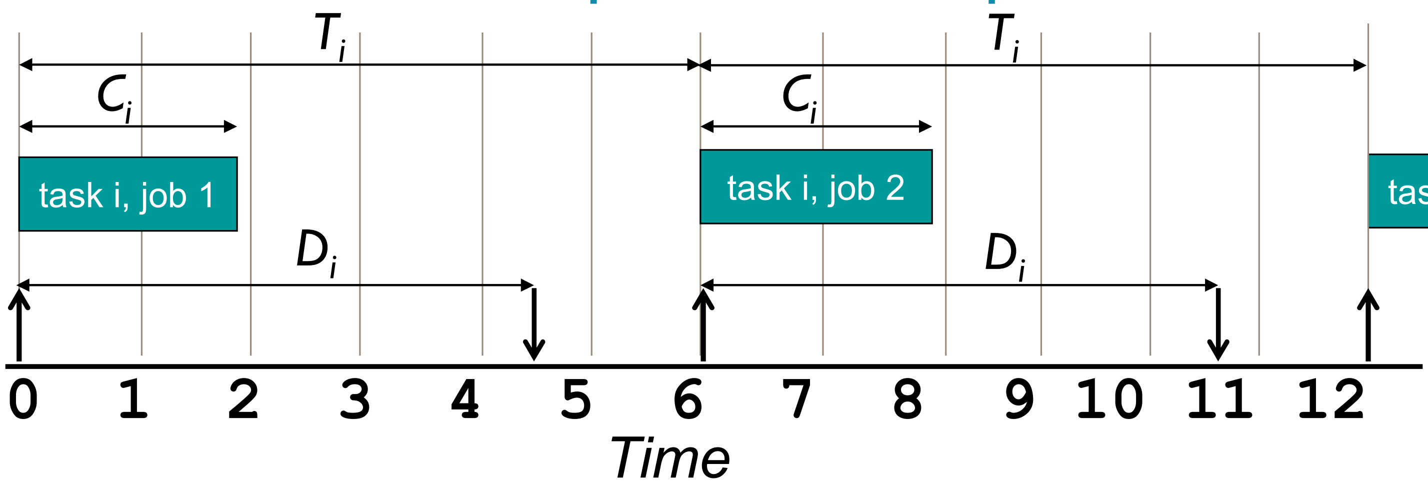
- Change scheduler to prioritize A > B > C etc.
- New behavior:
  - If EvA happened, run A, then check for EvA again.
    - Else if EvB happened, run B, then check for EvA again.
      - Else if EvC happened, run C, then check for EvA again.
        - Et cetera
- Implications
  - Not round-robin. Now have **dynamic** (not static) schedule of task orders, since higher priority tasks get chance to run before lower priority tasks.
  - Higher priority task may run multiple times before lower priority task gets to run once.
  - There may be more events (and task releases) further delaying the start of a task.
- Best case for Task A: Same as before. $R_A = C_A$
- Worst case for Task A (highest priority)?
  - Delayed by longest task (D). $R_A = C_A + Max(C_A, C_B, C_C, C_D, C_E, C_F)$
- Worst case for lower-priority tasks (B, C, D, E, F)?
  - Also may be delayed by higher-priority tasks. Details on next slide.

What about Response Time for Lower Priority Tasks?



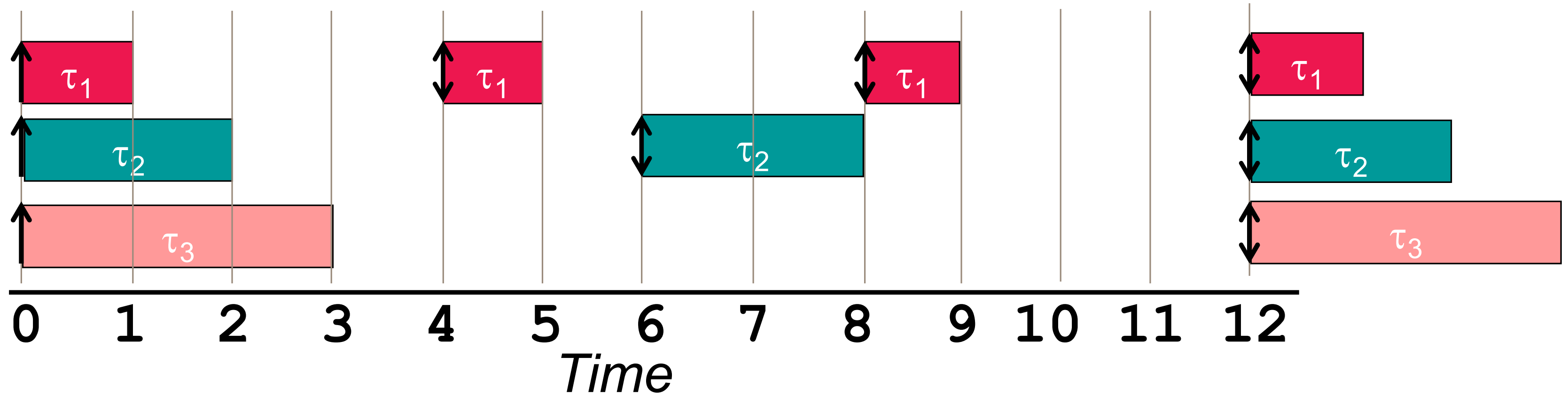
- First estimate ($R_C{}^0$) of response time R_C
  - Task C's finish may be …
    - delayed by **blocking** once by longest task if it is already running: $Max(C_A, C_B, C_C, C_D, C_E, C_F)$
    - delayed at least once by each higher priority task (C_A, C_B)
  - Equations
    - $R_C{}^0 = Max(C_A, C_B, C_C, C_D, C_E, C_F) + C_A + C_B + C_C$
    - Here: $R_C{}^0 = C_D + C_A + C_B + C_C$
- Second estimate ($R_C{}^1$)
  - **More** events for higher-priority tasks may happen before C starts (during vulnerable time), so more releases delay C starting
    - $R_C{}^1 = Max(C_A, C_B, C_C, C_D, C_E, C_F) + 2{*}C_A + 2{*}C_B + C_C$
    - Here: $R_C{}^1 = C_D + 2{*}C_A + 2{*}C_B + C_C$
- What if C ***still hasn't started*** and A or B is released again?
  - ***Repeat*** until no new releases, or $R_C{}^N$ is too large (past deadline)
    - $R_C{}^2 = Max(C_A, C_B, C_C, C_D, C_E, C_F) + 2{*}C_A + 3{*}C_B + C_C$
    - Here: $R_C{}^2 = C_D + 2{*}C_A + 3{*}C_B + C_C$
- Observations
  - Task C is vulnerable to timing interference from blocking and higher-priority tasks from release (EvC) until C starts running
  - Number of additional task releases depends on minimum time between events (EvA to EvA, EvB to EvB) in the worst case (burst)

Periodic Task Model of Computational Requirements



- Periodic Task Model describes characteristics for each task τ_i
  - Job = a specific instance of that task running
  - Task releases job so scheduler can run it
- ↑ A periodic task *i* releases a job every T_i time units
- ↓ Job may have an absolute deadline D_i after its release
- Job takes a constant time C_i to execute
- Simplifying assumptions include
  - No time needed for scheduler, task switching, ISR response/return

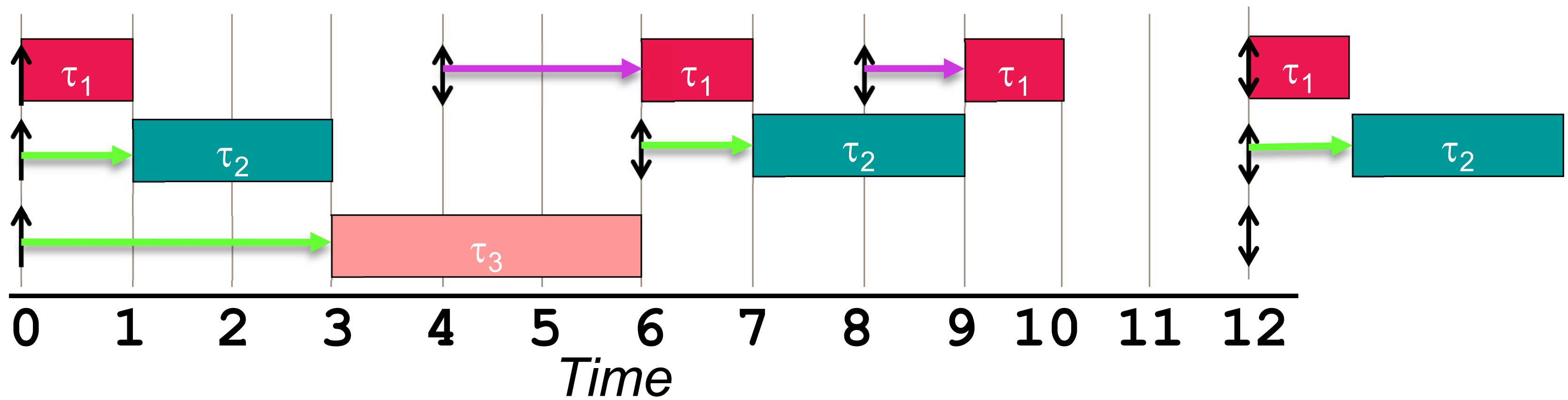
Example Workload: What We Ask For



- Set of tasks with real-time requirements
- What gets executed when?
  - Depends on scheduler and task priorities

Task	Exec. Time C_i	Period T_i	Deadline D_i
τ_1	1	4	4
τ_2	2	6	6
τ_3	3	12	12

Scheduled Workload: What We Get



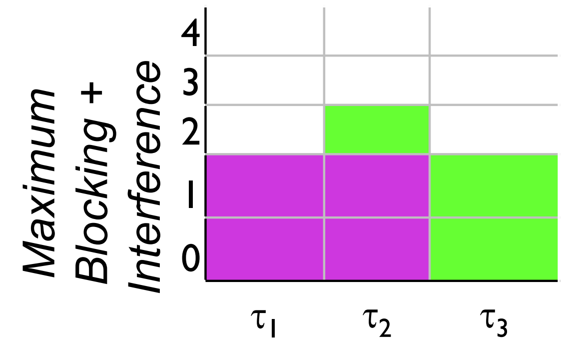
- Example: Scheduler and task fixed priorities
  - Assign priorities as shown
  - Use a non-preemptive scheduler
- What can delay a task?
  - I: Interference caused by higher priority tasks
  - B: Blocking caused by lower priority tasks
- Response time = Computation + Blocking + Interference

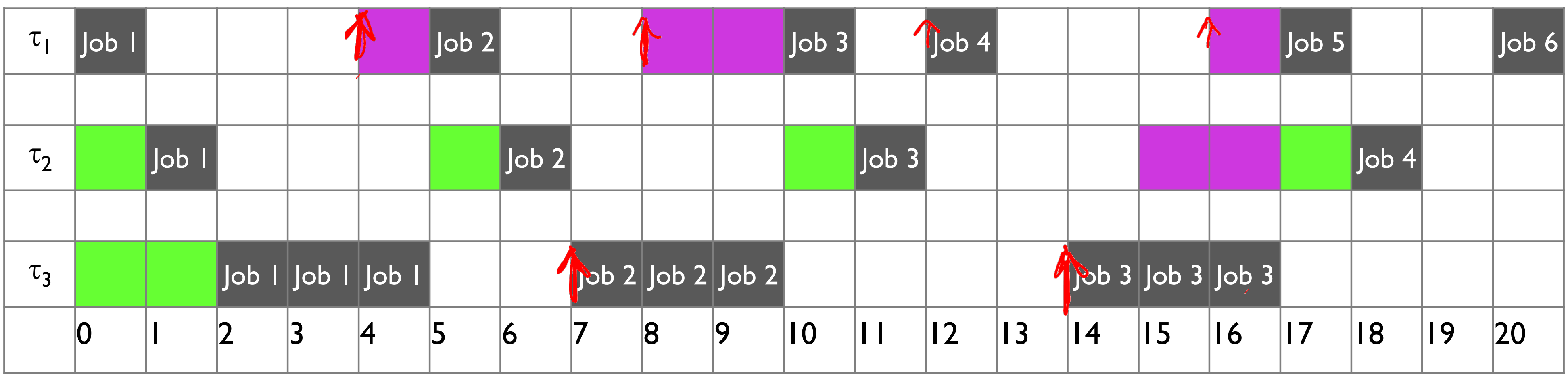
Task	Exec. Time C_i	Period T_i	Deadline D_i	Priority
τ_1	1	4	4	*High*
τ_2	2	6	6	*Medium*
τ_3	3	12	12	*Low*

$$R_i = C_i + B_i + I_i$$

Non-Preemptive Scheduling

Task	Exec. Time C_i	Period T_i	Priority
τ_1	1	4	*High*
τ_2	1	5	*Medium*
τ_3	3	7	*Low*





NUMERICAL RESPONSE TIME ANALYSIS

Numerical Response Time Analysis, Step 1

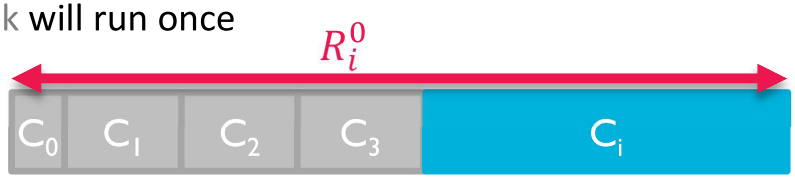
- How long could it take for task *i* to complete? What is its *response time* R_i?
- Initial estimate based on *critical instant*: every task is released simultaneously

  R_i^0 = computation time for task *i* + computation time for all other tasks*

  $$R_i^0 = \boxed{C_i} + \boxed{\sum_{j \neq i} C_j}$$

  - Non-prioritized scheduling: Every task will run once

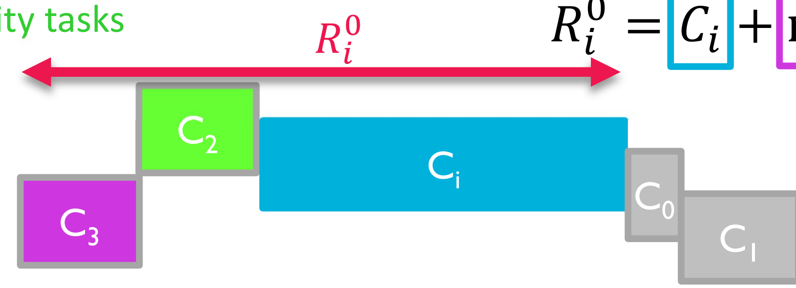
```
while (1) {
  for (j=0; j<NUM_TASKS; j++) {
    if (Tasks[j].RP > 0) {
      Tasks[j].RP--;
      Tasks[j].Task();
} } }
```



  - Prioritized, non-preemptive scheduling: computation time for task *i* + longest of all tasks + All higher-priority tasks

  $$R_i^0 = \boxed{C_i} + \boxed{\max_{\forall j}(C_j)} + \boxed{\sum_{j \in hp(i)} C_j}$$

```
while (1) {
  for (j=0; j<NUM_TASKS; j++) {
    if (Tasks[j].RP > 0) {
      Tasks[j].RP--;
      Tasks[j].Task();
      break;
} } }
```

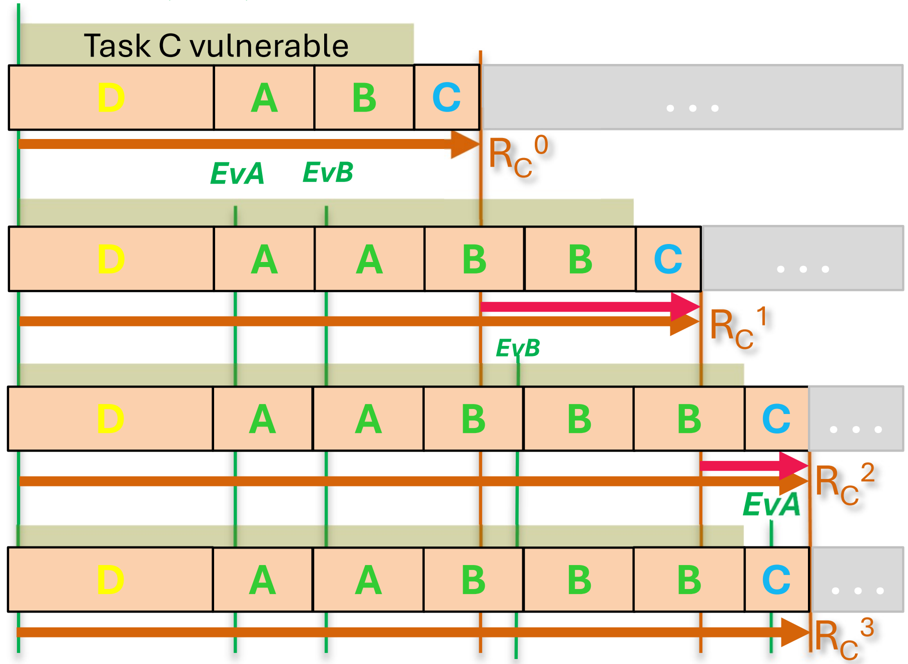


- * Pesky detail: Could previous job of task *i* still be running when all tasks are released?
  - Depends on other assumptions. If not, changes equations slightly

Additional Timing Interference? Steps 2, 3, 4 …

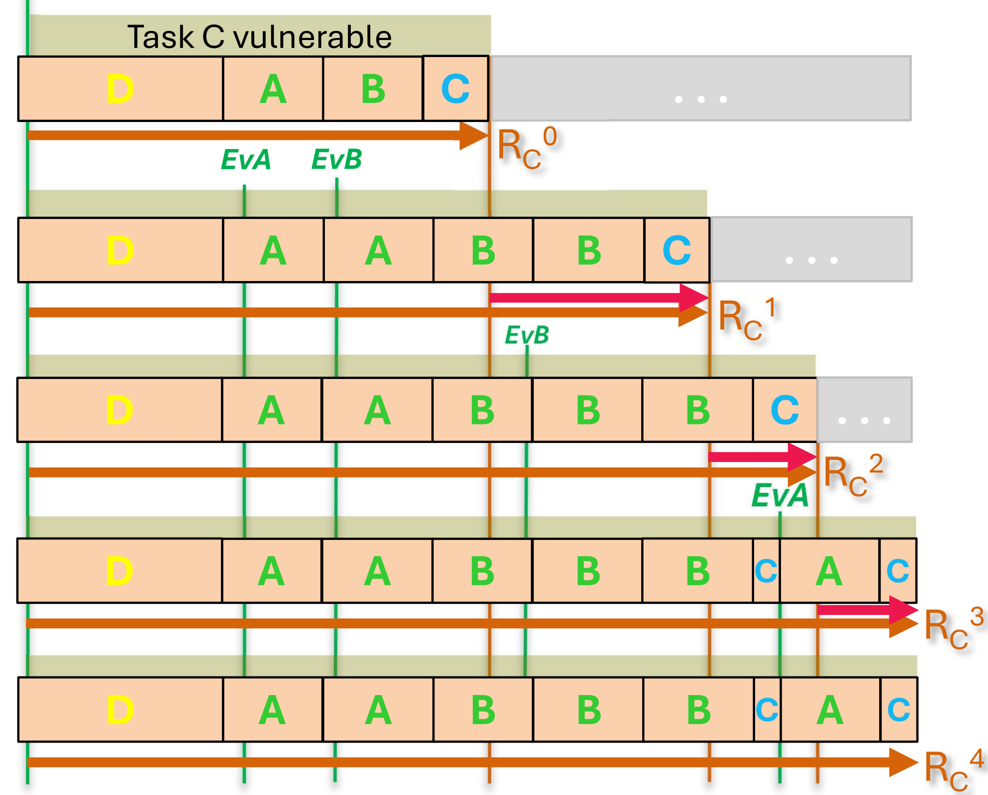
Non-preemptive: *Vulnerable from 0 to $R_i^n - C_i$ since task i can't be preempted after it starts*

*Events **EvA, EvB, EvC,** EvD, EvE…*



Preemptive: *Vulnerable from 0 to R_i^n since higher-priority task can preempt task i*

*Events **EvA, EvB, EvC,** EvD, EvE…*



- Task *i* may be delayed by *new job releases* during ***vulnerable time***
- Consider new releases to update completion time estimate R_i^{n+1}
- Repeat until no new releases, or too late (e.g. deadline missed)