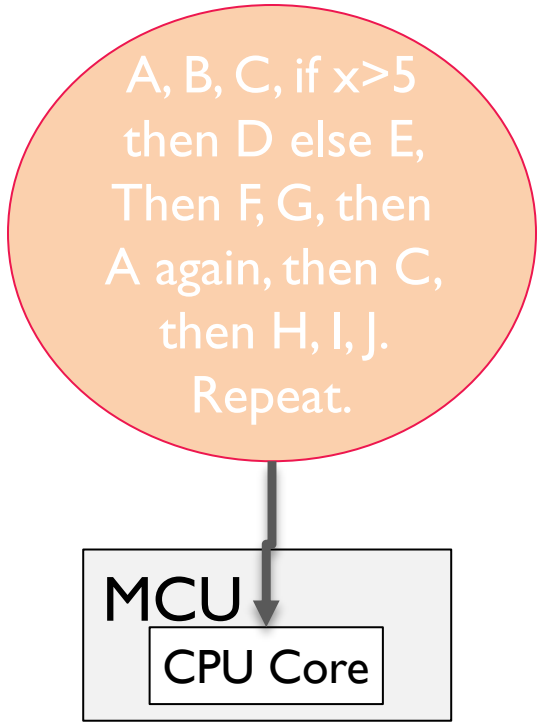
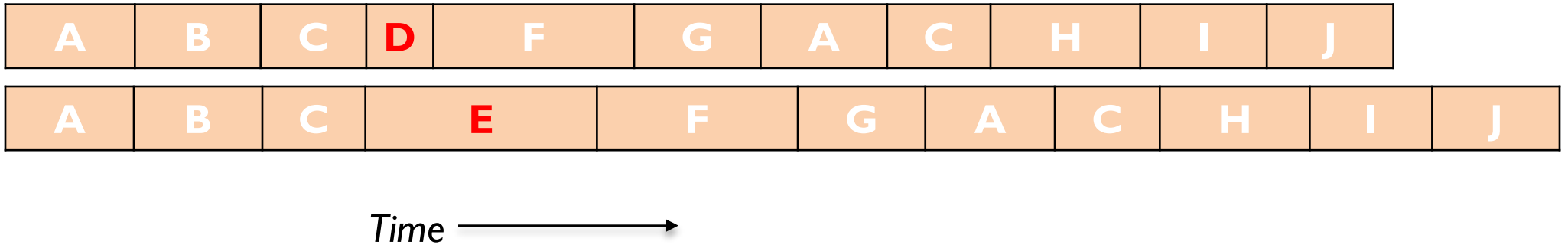
Scheduling: Making Software Share the CPU

Cyclic Executive: Infinite Loop in Main Thread



- What code does CPU run?
  - Main thread starts when system comes out of reset
  - Infinite loop in main thread
- Put everything in the loop
  - Easy! (at first)
- Modularity scales up badly
  - No separation between different parts of the program…
  - …unless you do it explicitly (separate subroutines, source files, etc.)
- Often need to design system to meet response time requirements

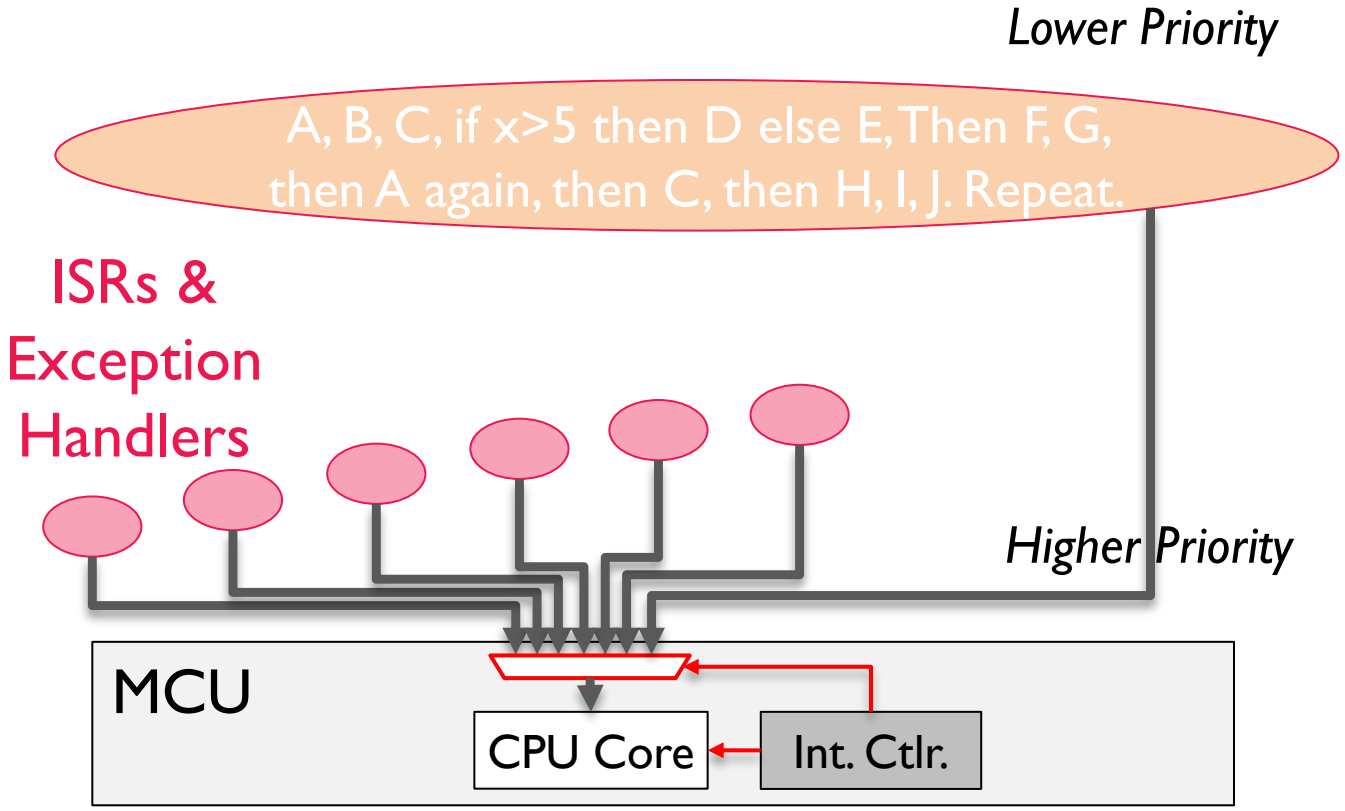
Cyclic Executive: Infinite Loop in Main Thread



- Important timing questions
  - **How long** does it take for code A to run?
  - **How often** does code B get to run?
    - Affects input-to-output response time
    - Depends on how long it takes **all other code** to run
- Timing variability
  - Time taken to run code A may depend on input data, current state information
  - Some code may run only if needed (D vs. E)
- Timing scales up badly
  - Hard to manage timing as more code is added
  - Example: Had to insert 2nd calls to A and C in loop to reduce response time

CPU Scheduling Foundation: Interrupt System

- What code does CPU run?
  - **Interrupt controller** decides code to run next.
    - Checks interrupt/exception requests **before** starting each instruction
  - Interrupt/exception requested and handler not completed?
    - Yes: Force CPU to run its handler
    - No: keep running main thread (started when system came out of reset)
- Foreground/Background scheduler
  - Foreground: interrupt/exception handlers
  - Background: main thread

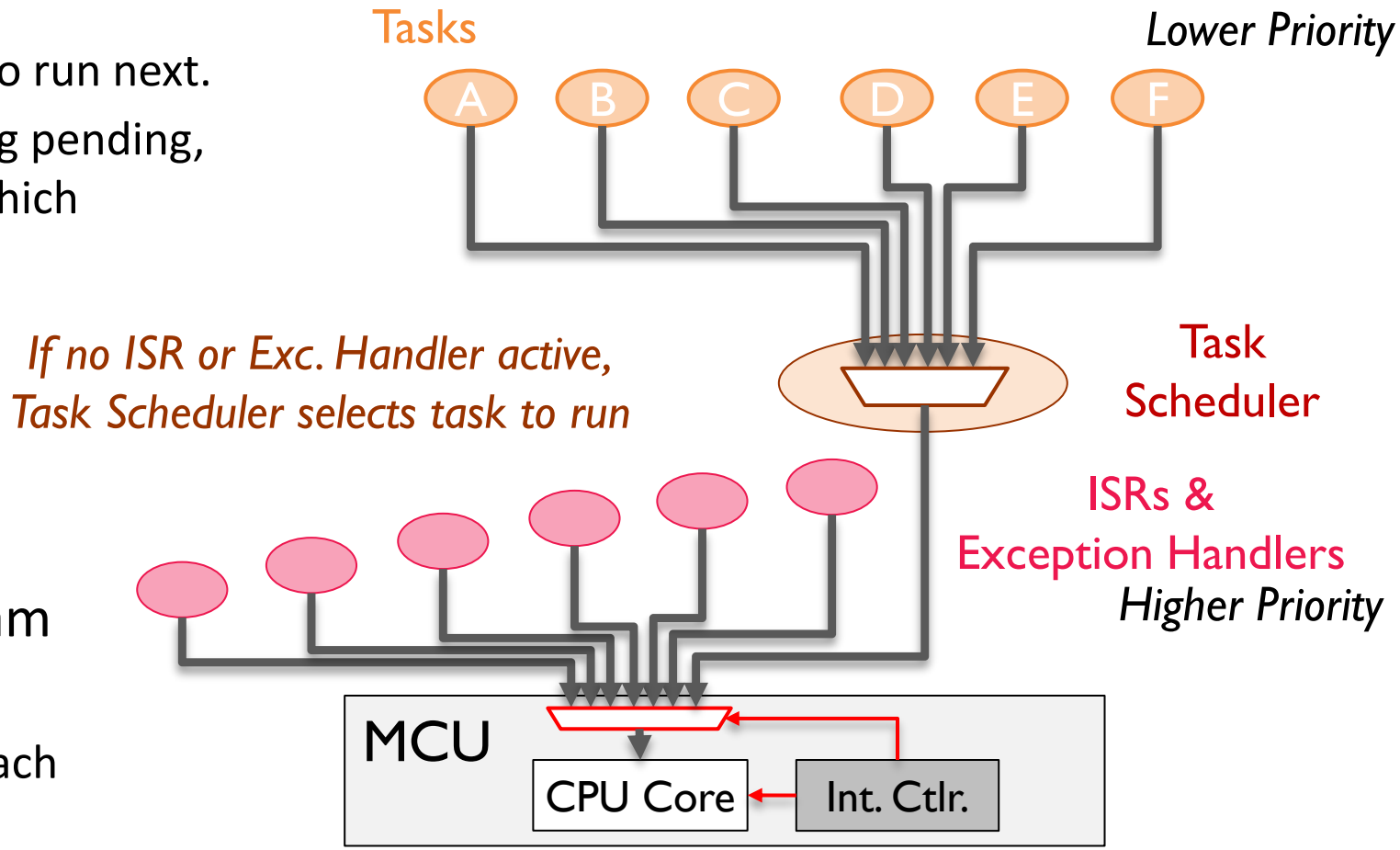


Task Schedulers: Helping Software Share the CPU Better

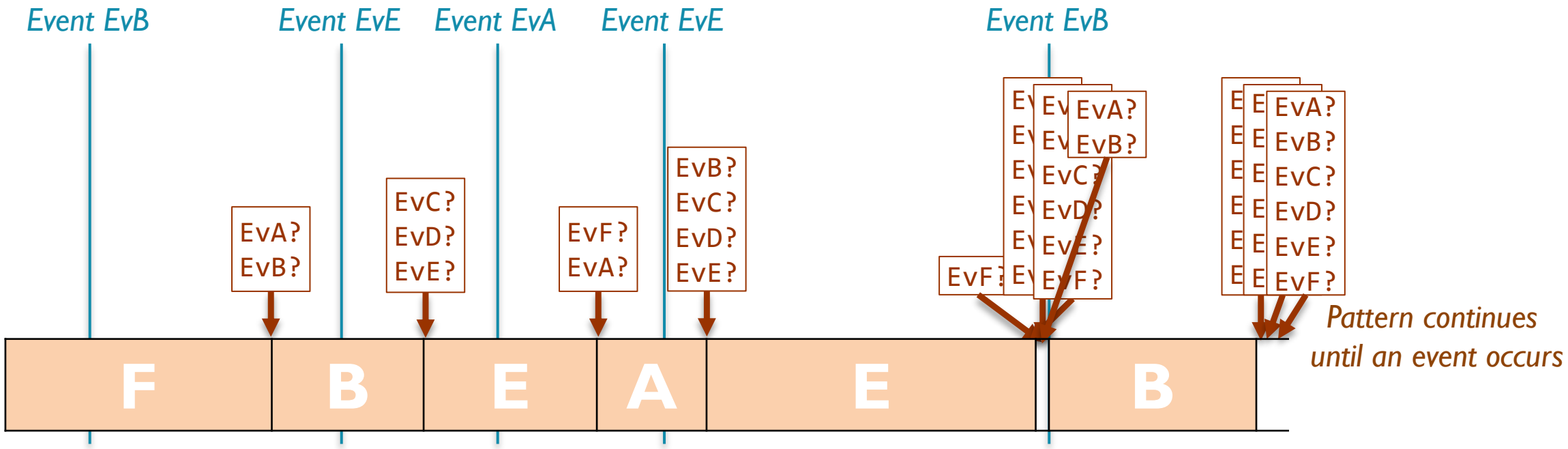
- What code does CPU run?
  - **Interrupt controller** decides code to run next.
  - If no interrupt/exception processing pending, **task scheduler runs and** decides which task/thread to run next

```
while (1) {
    if (EvA>0) A();
    if (EvB>0) B();
    …
    if (EvF>0) F();
}
```

- Enforces a more modular program structure
  - Separate tasks/threads and ISRs, each running (mostly) independently
  - Easier to develop, maintain, debug



Example of Execution Sequence

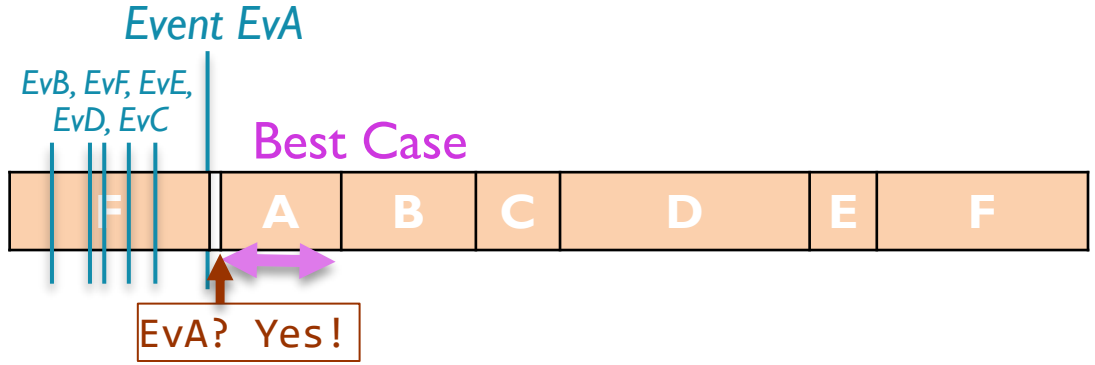


- Scheduler loop behavior:
  - EvA happened? Run A until done.
  - EvB happened? Run B until done.
  - …
  - EvF happened? Run F until done.
- Terms: No task preemption, round-robin task ordering (each task gets a turn to run)

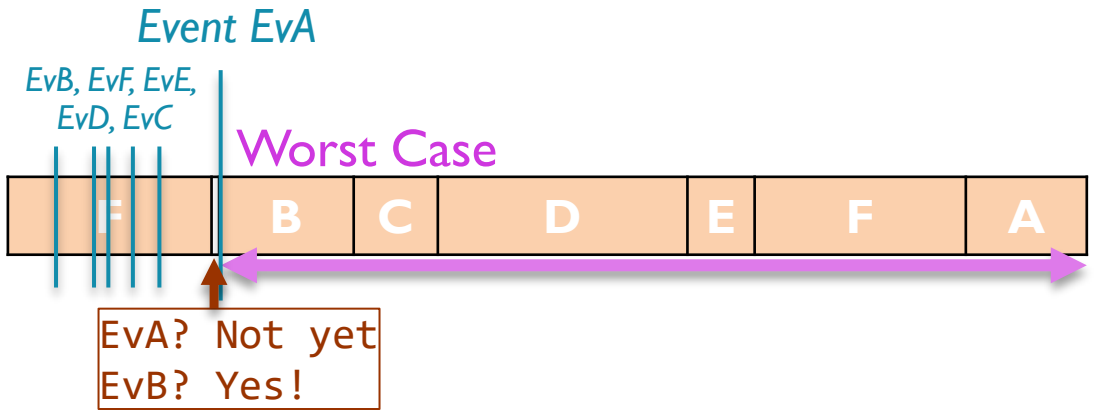
- Assume scheduler is much faster than a task
  - For example: 0.2 µs for scheduler to check an event, vs. tens of µs for a task to do its work
- Implications
  - Usually scheduler's time can be ignored: 0.2 µs << 10 µs
  - If no tasks are ready to run, scheduler time matters:
    - 0.2 µs * check each of 6 tasks (none ready)* 100,000 checks = 120,000 µs = 0.12 seconds

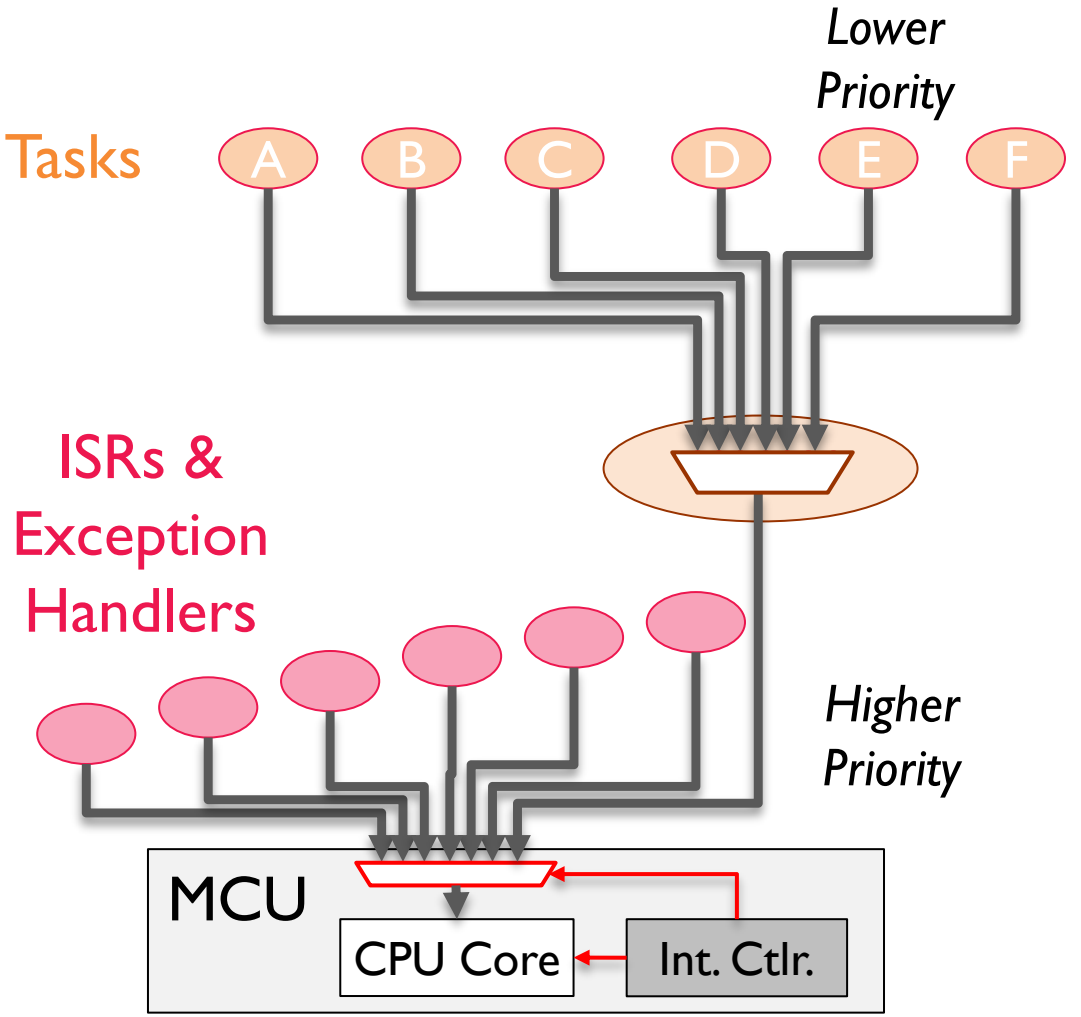
Examining Responsiveness

- How long from event EvA until task A finishes servicing it?
  - Best case: EvA happens **just before** scheduler checks it



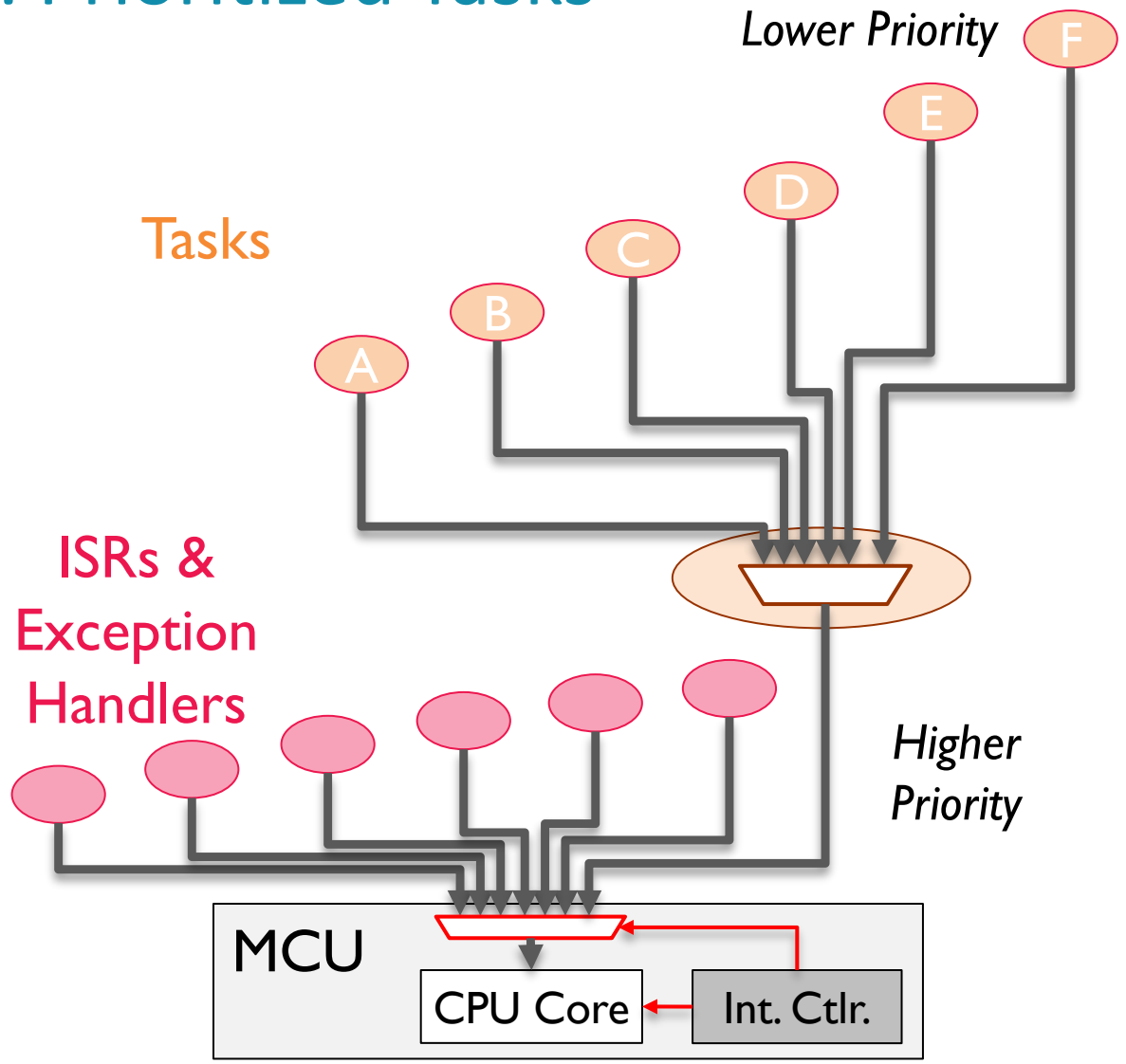
  - Worst case: EvA happens **just after** scheduler checks it
    - Every other event (EvB – EvF) happens, scheduler checks EvA (hasn't happened yet), and then EvA happens





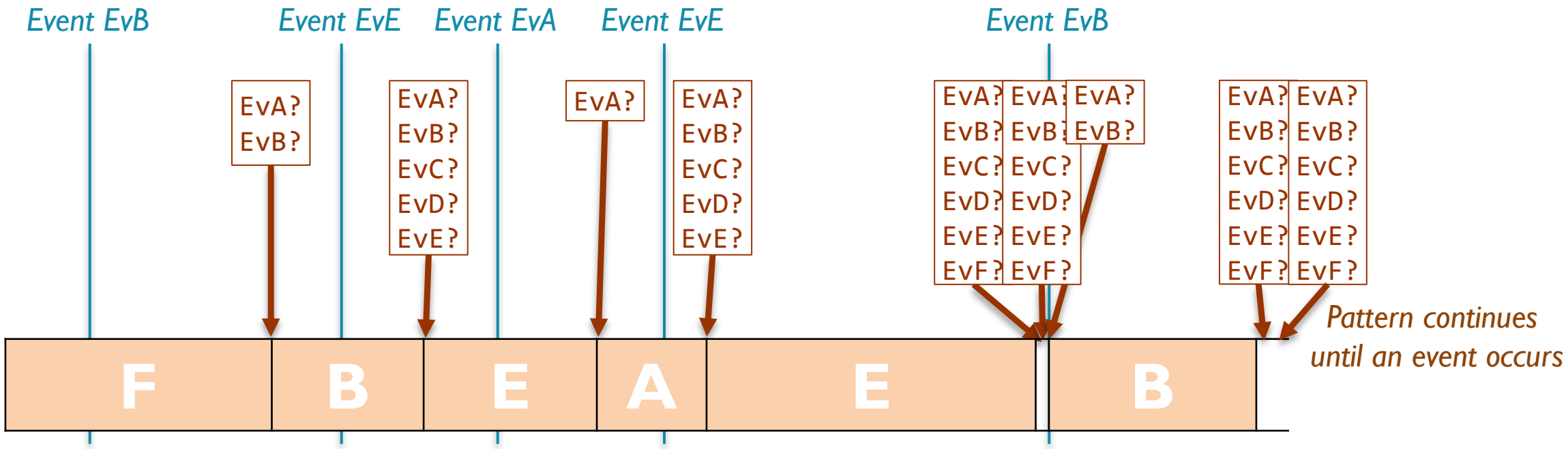
Improvement for Responsiveness: Prioritized Tasks

```
while (1) {
  if (EvA>0)
    A();
  else if (EvB>0)
    B();
  …
  else if (EvF>0)
    F();
}
```



- Change scheduler to prioritize A > B > C etc.
- New behavior:
  - If EvA happened, run A until done.
    - Else if EvB happened, run B until done.
      - Et cetera
- Best case: Same as before
- Worst case: Delayed only by longest other task (F)

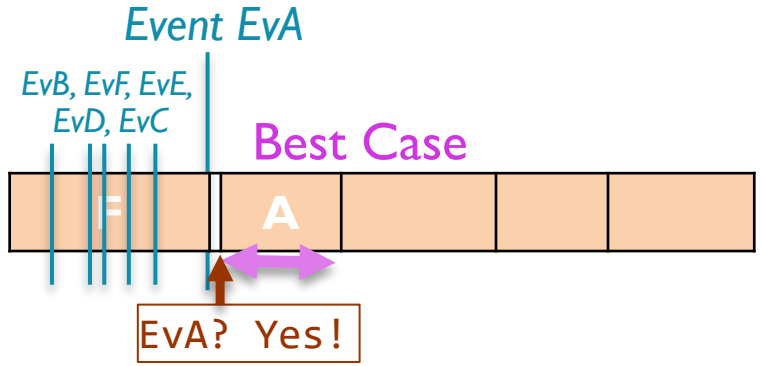
Example of Execution Sequence



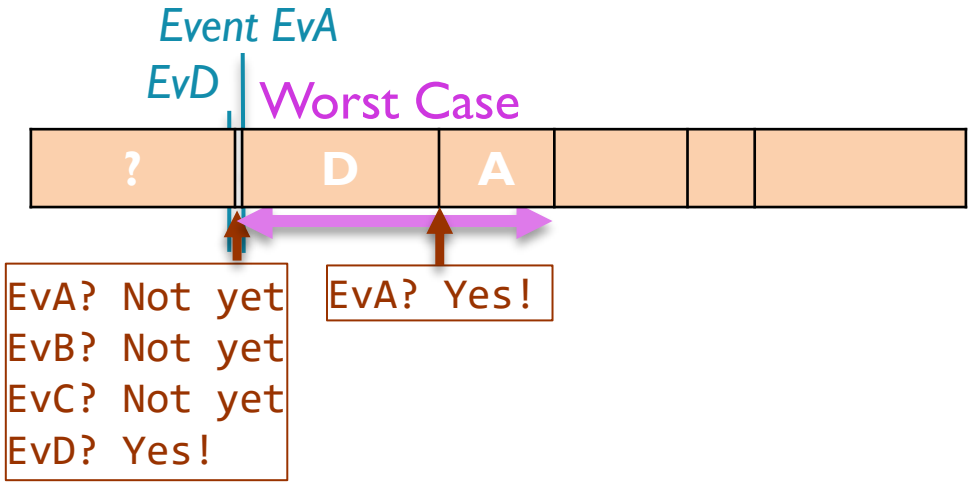
- Scheduler loop behavior:
  - Always test events starting with highest-priority task
- Terms: No task preemption, Prioritized task ordering

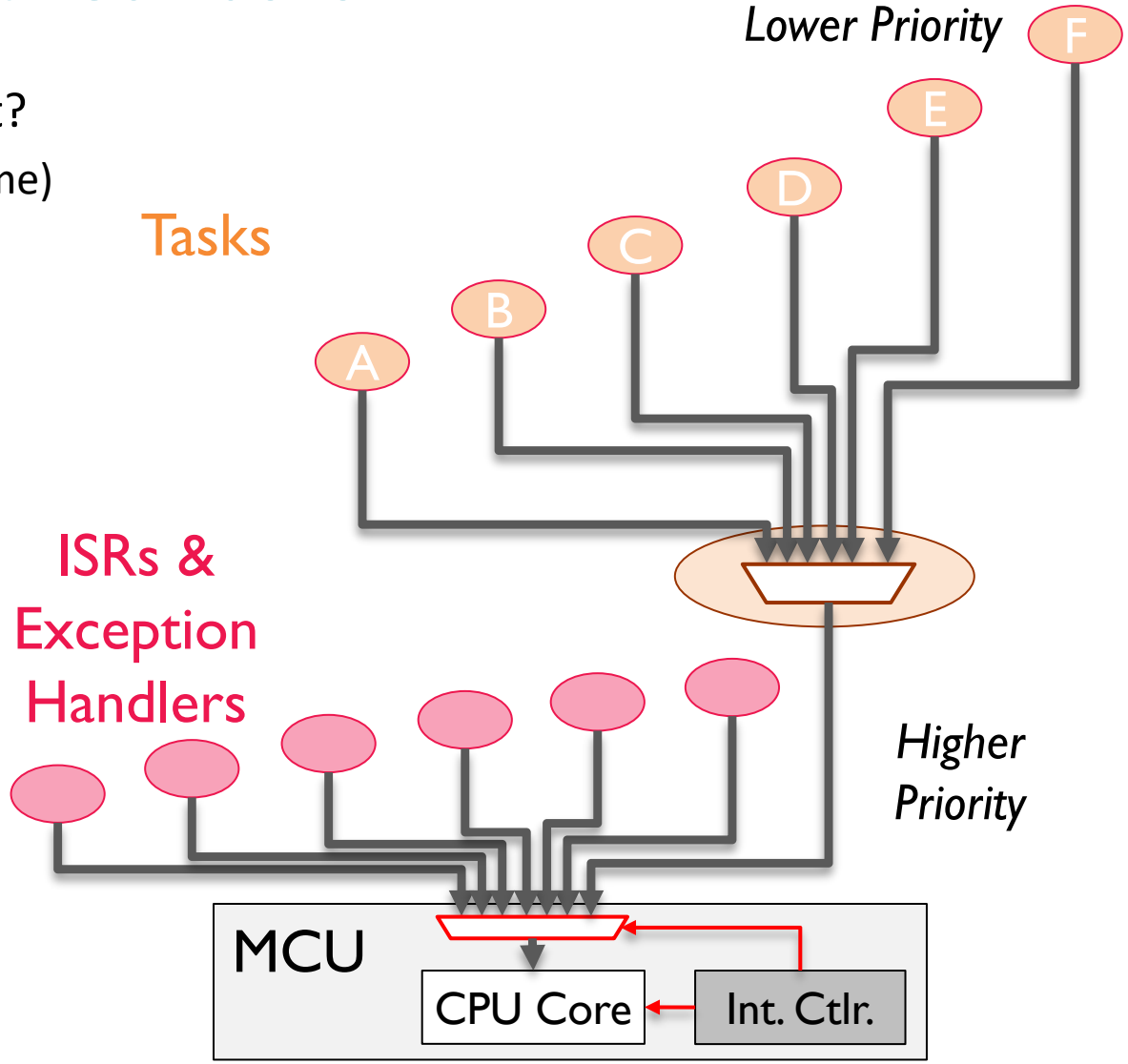
Examining Responsiveness: Prioritized Tasks

- How long from event EvA until task A finishes servicing it?
  - Best case: EvA happens just before scheduler checks it (same)

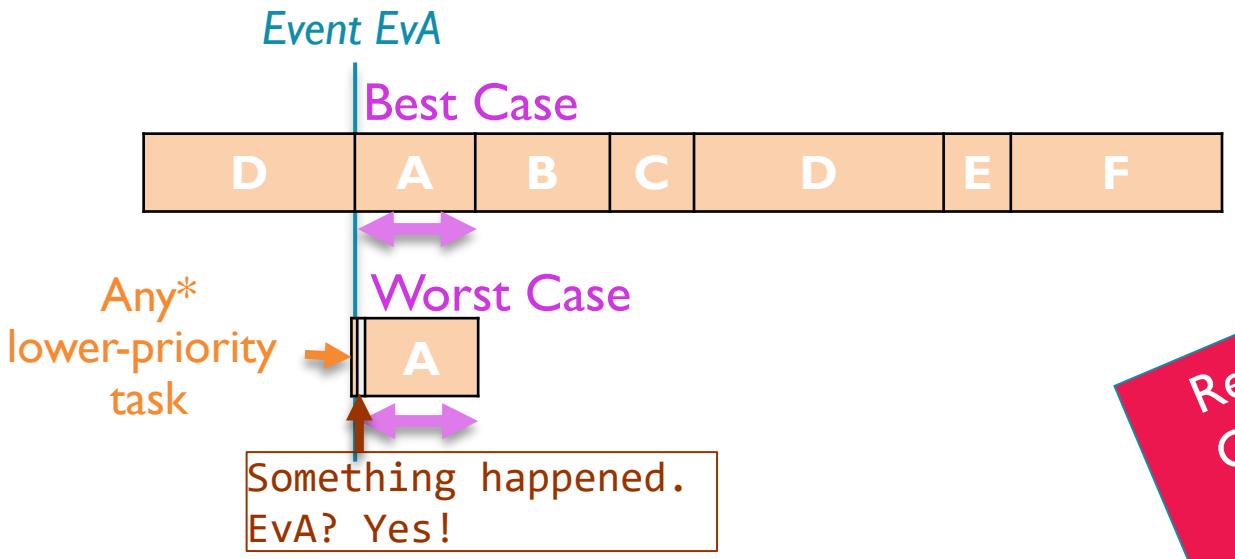


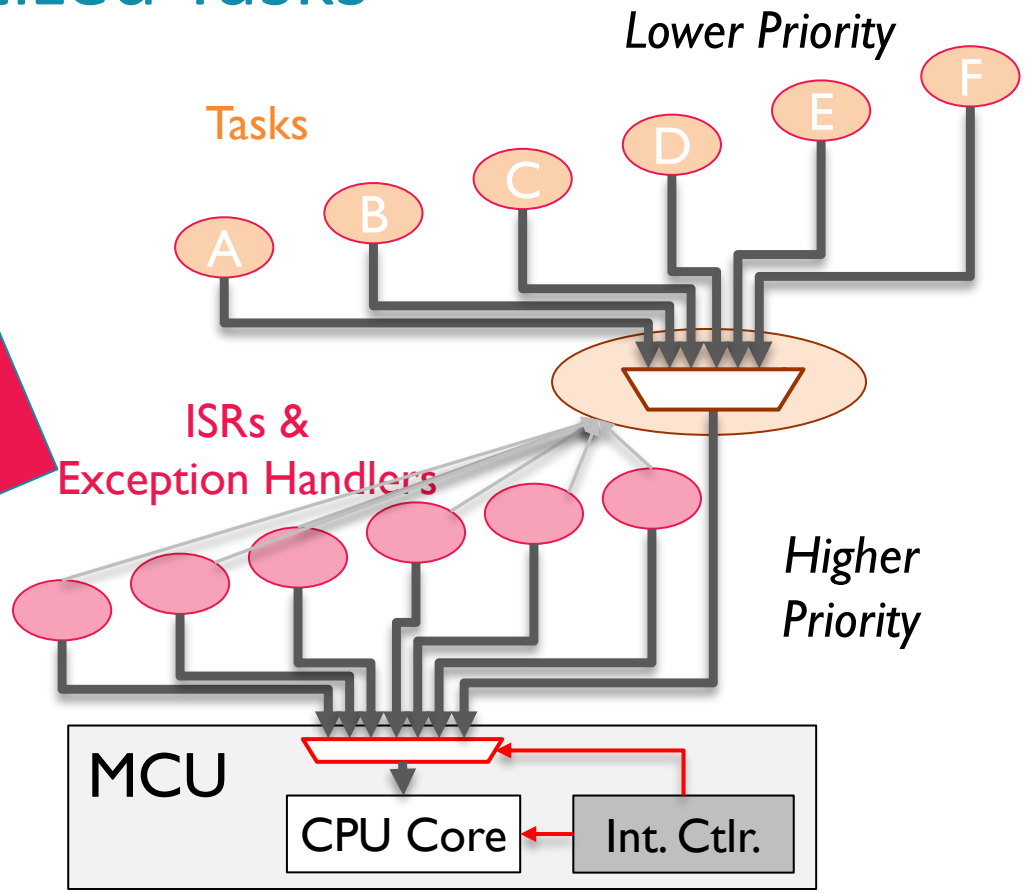
  - Worst case: Delayed only by longest other task (e.g. D)





Next Improvement: Preemptive, Prioritized Tasks



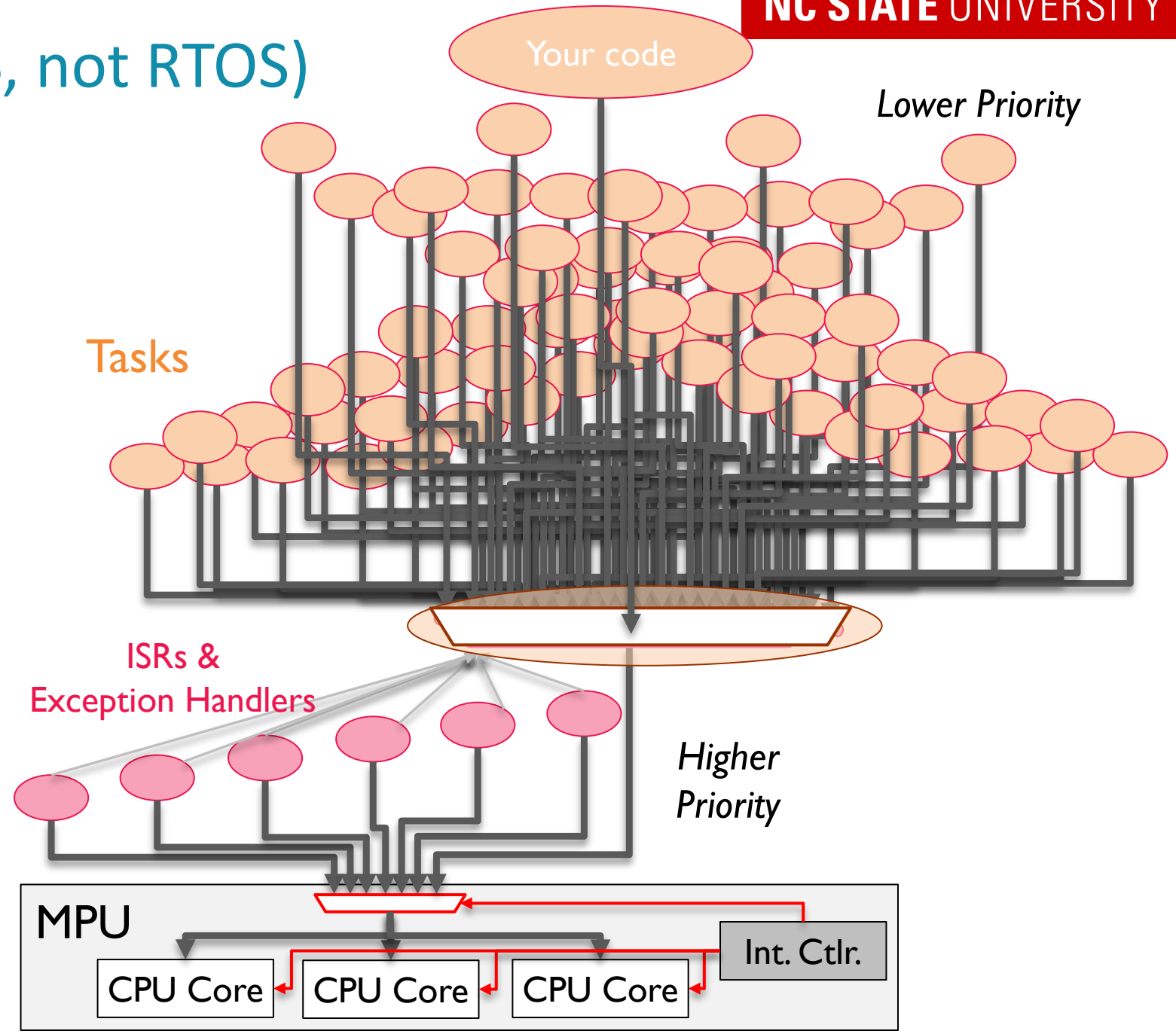


- Improvements
  - Tell scheduler about scheduling-related events ASAP
    - Interrupt handlers (e.g. button press, timer tick)
    - OS operations in other tasks
  - Have scheduler preempt lower-priority task to run higher-priority task which is now ready
- Delay from EvA -> A completion doesn't* depend on other tasks now, just A**
  - * only if A is not dependent on other tasks
  - ** and ignores scheduler time

- Delay from **any event** to the **completion** of its task or handler depends only* ** on **how long** and **how often** higher priority tasks run
  - * assumes task not dependent on any other tasks
  - ** and ignores scheduler time

Now Add Linux (is an **OS**, not RTOS)

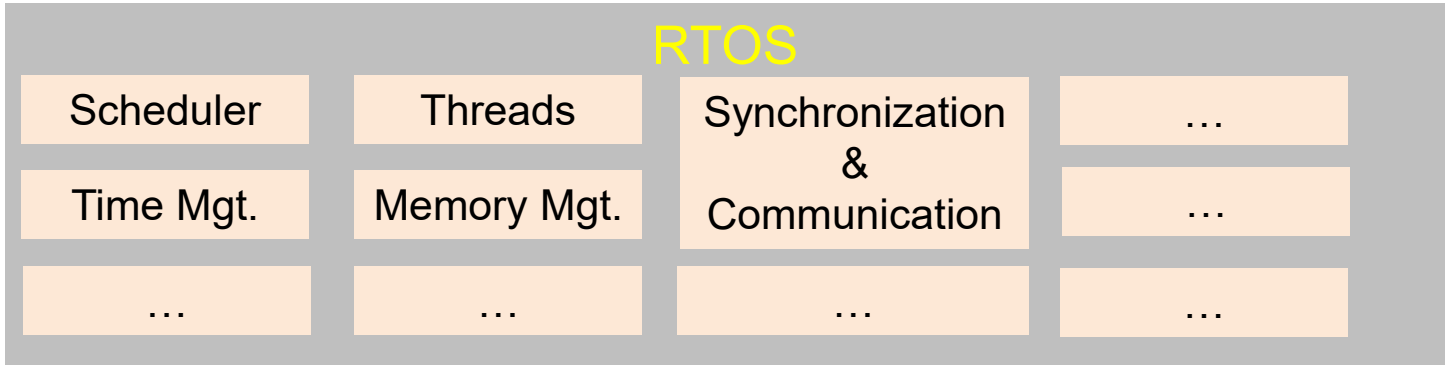
- Lots of great features!
- And the CPU is going at over 2 GHz!
  - Running the CPU at that speed makes us add deep CPU pipelines, branch predictors, caches and virtual memory…
  - These features make timing much more variable and unpredictable
  - Timing matters for responsiveness
- Ok, but what about multicore?
  - Do you get the core to yourself, or do you have to share it?



OS Synchronization and Communication Primitives

	Receiver Thread	Information Provided	Can Accumulate Multiple Pending Events?	Handshake?
Event Flag	Any thread	"The event has occurred"	No	No
Thread Flag	Specified thread	"The event has occurred"	No	No
Semaphore	Any thread	"The event has occurred"	Yes (counting semaphore), No (binary semaphore)	Yes
Message Queue	Any thread	"An event described by this message has occurred"	Yes, up to number of available queue elements	Yes
Mutex Lock	Any thread	"This object is available"	No	Yes

RTOS: What and Why



- Real-Time Operating System:
  - An OS designed to operate with **deterministic** (repeatable) timing
    - Typically uses a preemptive scheduler
    - Timing: Deterministic, predictable, bounded
- Why use one? RTOS vs. OS
  - Easier to build a system with deterministic timing
    - Developer can more easily manage the response times of urgent processing through prioritization
    - Don't need to restructure code repeatedly or re-invent the wheel (*hopefully correctly*)
  - Cutting response time reduces processor & memory speed requirements (and HW $$)
- Improve software modularity
  - Improve software reliability by isolating threads
  - Simplify maintenance and upgrades
- Leverage built-in OS/RTOS services
  - Interprocess communication and synchronization (safe data sharing)
  - Time management
  - I/O abstractions
  - Memory management
  - File system
  - GUI
  - Networking support

Summary

Summary: Responsiveness is Key

- Software is “funneled” to the CPU by the interrupt system and scheduler (if any)
- The more software in the system, the more can get in the way, increasing response time
- Scheduler can help by providing task prioritization and task preemption