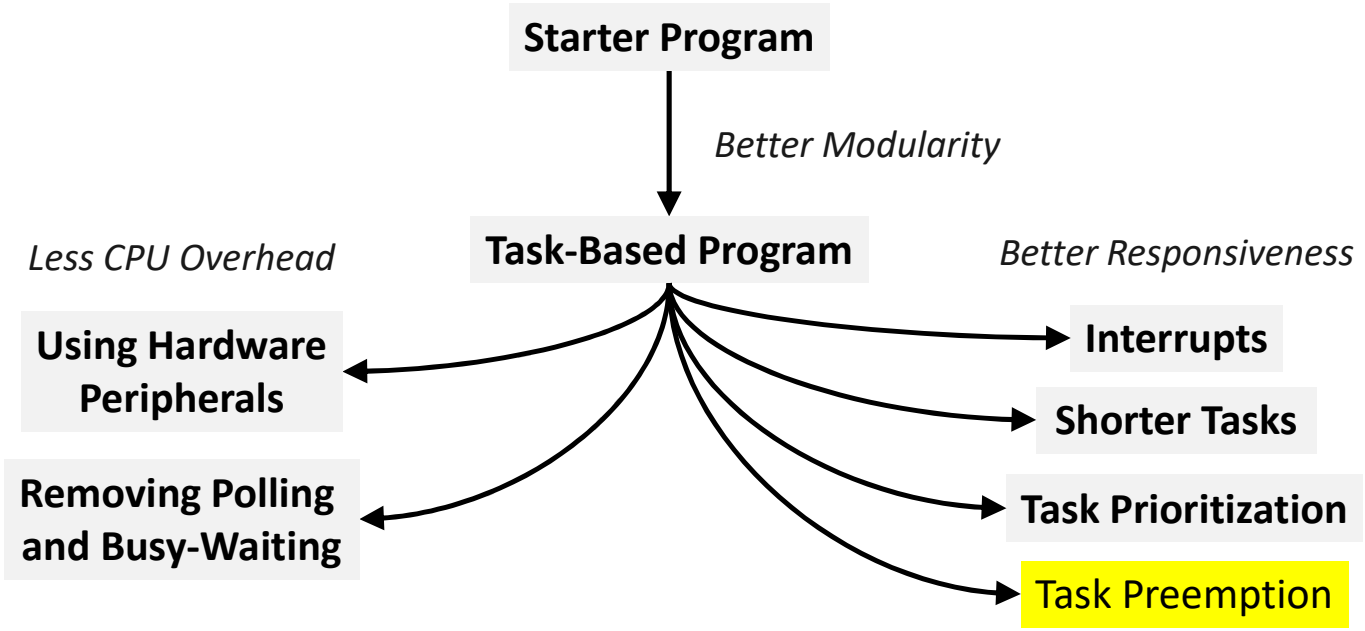
21: Preempting and Resuming Tasks

v2

Overview

- How does preemption improve responsiveness?
- How do we preempt and resume tasks?
- What states can a task be in, given a preemptive scheduler?



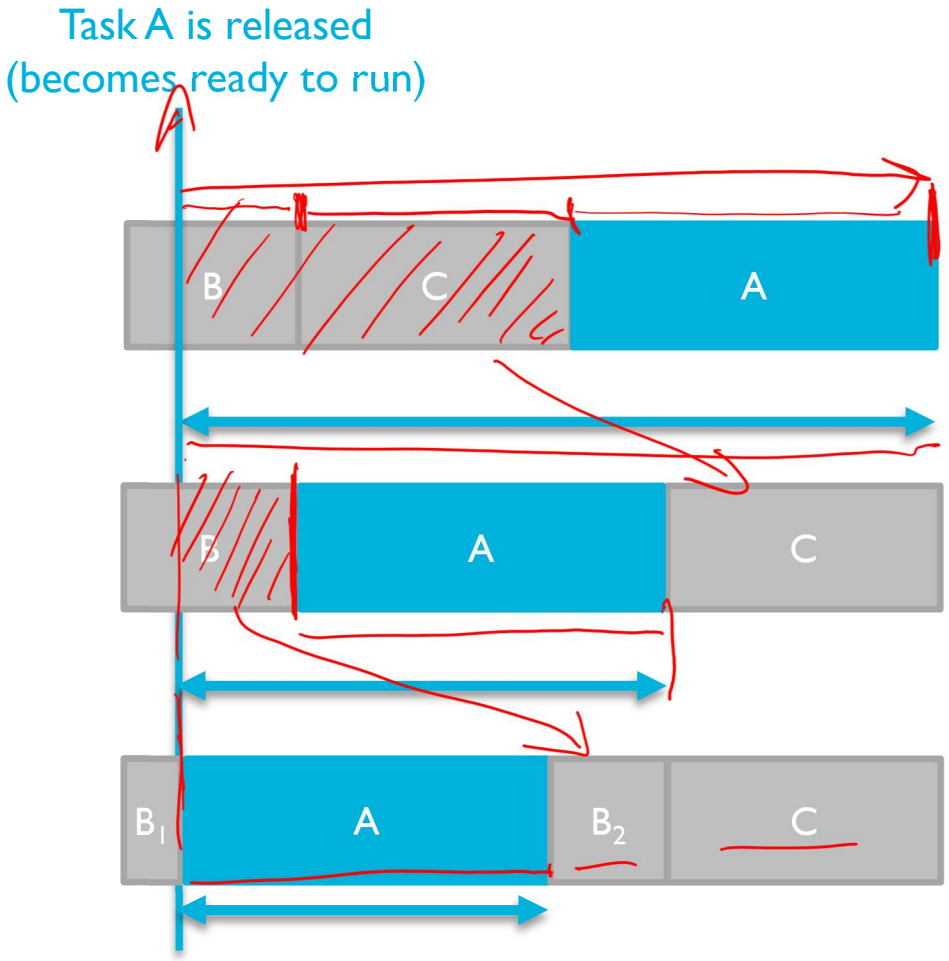
Tasks, Threads and Processes

- Two types of software processes:
  - Tasks, threads: Arm CPU executes in thread mode
    - Both terms used interchangeably here
  - Interrupt/Exception handlers: Arm CPU executes in handler mode
- Many embedded systems use single address space
  - All software processes can access the entire address space (unless restricted with privileges or other partitioning)
  - Simple, inexpensive, but vulnerable
- More complex computer systems provide separate memory spaces
  - Goal: Protect processes from each other. Process can access only its memory space (and no other)
  - Example Implementation: memory address translation with virtual memory system
  - Refining the terms
    - A process contains one (main) or more threads
    - All threads in a process can access its address space (and no other)

Improving Task A's Responsiveness

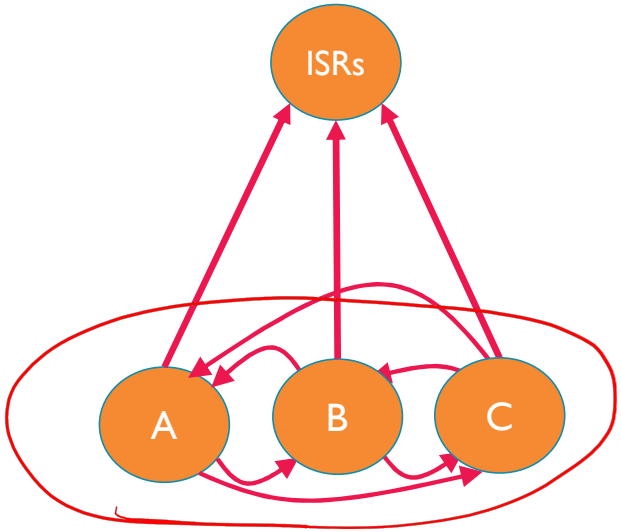
From Task Release to Task Completion

- *Note:* ***independent*** *tasks*
- Non-prioritized
- Prioritized (A>B>C)
- Prioritized and Preemptive



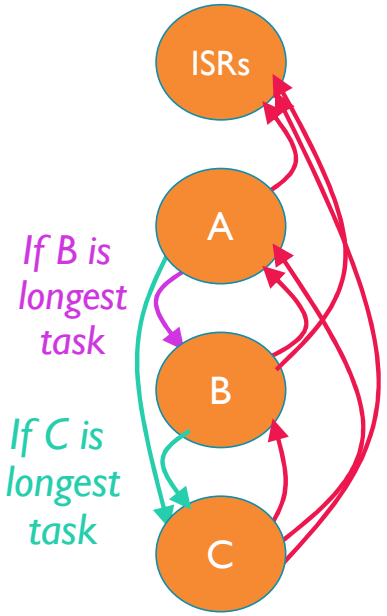
Response Time and Task Prioritization

Non-Preemptive Static Scheduling



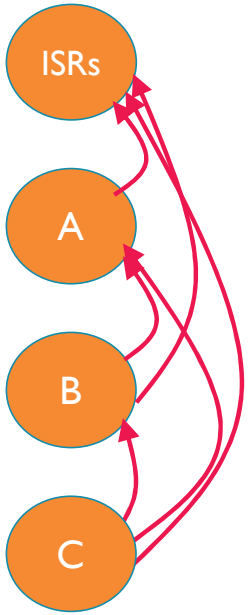
Each task depends on ***all other tasks and ISRs***

Non-Preemptive Prioritized Scheduling



Each task depends on ***higher priority tasks, the longest other task (regardless of priority) and ISRs***

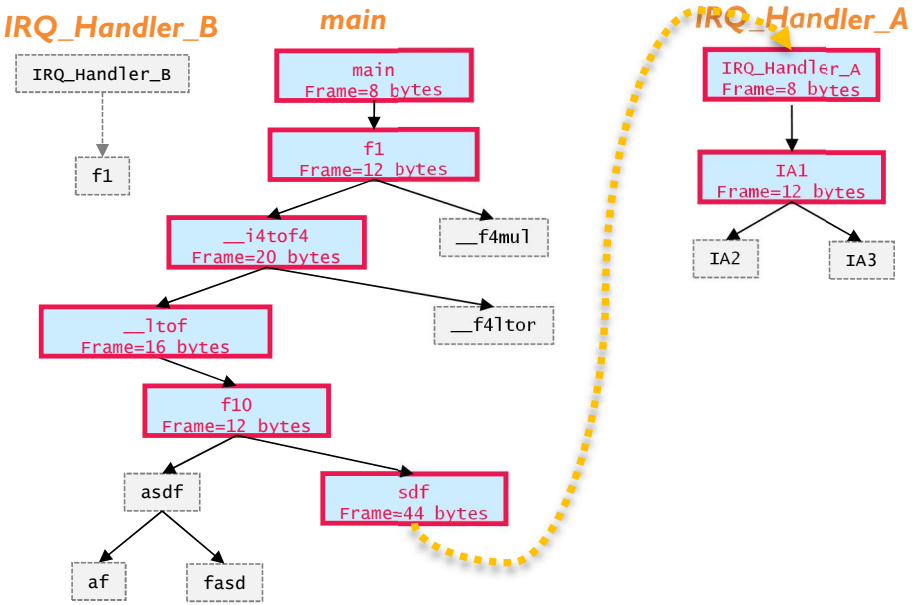
Preemptive Dynamic Scheduling



Each task depends on ***higher priority tasks and ISRs*** and nothing else

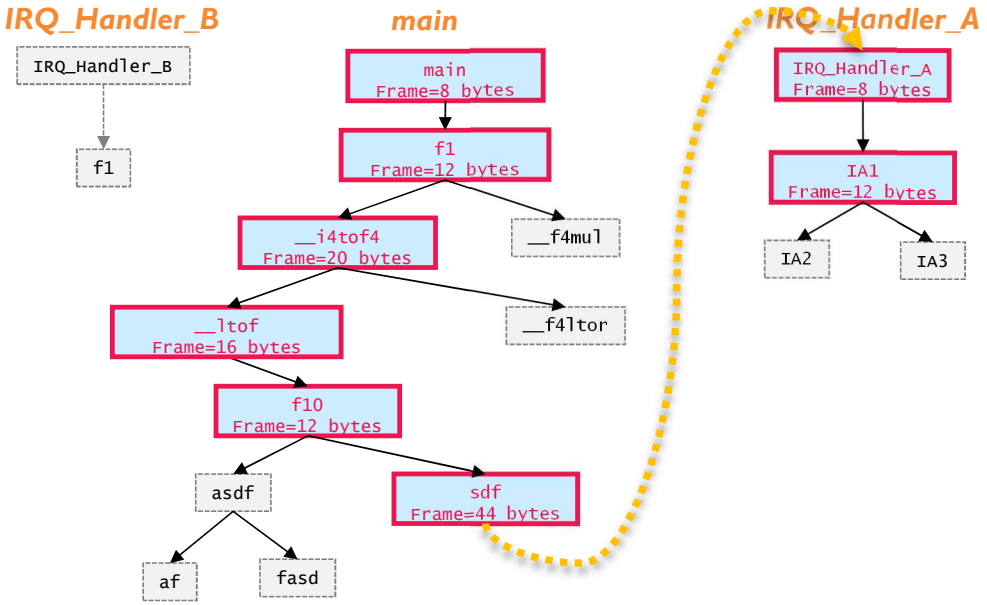
How to Preempt and Resume Tasks

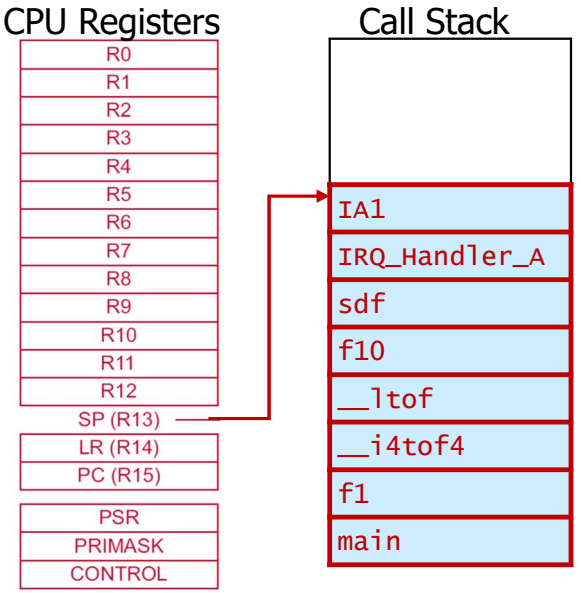
Thread State Information



- ***Context***: snapshot of the thread's *current execution state*
- Context information is needed for active functions
  - **Active** function has started (activated) but has not finished (deactivated)
  - Active functions include main() and any active subroutine calls or interrupt/exception handlers
- Each active function may have:
  - Next instruction to execute (program ctr., instr. ptr.)
  - Function arguments
  - Local variables
  - Doesn't include variables shared with other threads (e.g. globals), since used for communication
  - Where to go after function ends (return address)
    - Subroutine: Where do I go in the function that called me?
    - Handler: Which do I go in the function was running when I started?

Context Locations

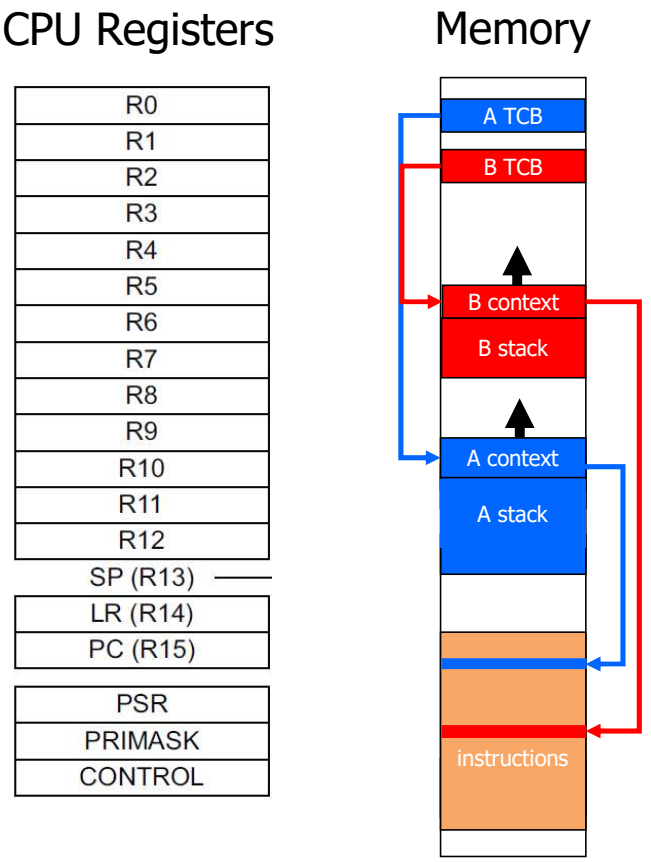




- Context is in registers and memory
  - Based on program, ISA (register set, instructions, addressing), compiler optimizations
- Program Call Stack
  - Holds stack frame/activation record for each active function
  - Stack organized by nesting sequence of currently active functions and handlers
- May have two stacks
  - One for thread's function calls
  - One for interrupt/exception handlers
- Simple example here: one stack holds both calls and handlers
- If we save a thread's context, can resume it later
  - .. and share the processor among multiple threads

Managing Multiple Threads

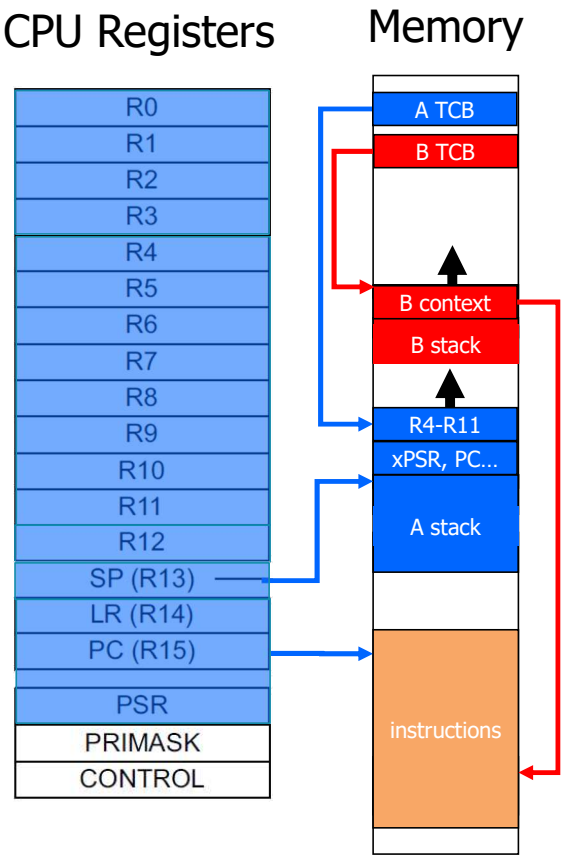
- Each thread in a preemptive system needs its own call stack
- Must store the context of each non-running thread
  - Stored on thread's stack – we'll see why soon
- Also need a thread control block (TCB) for each thread, stored in static data section (fixed locations)
- Kernel swaps information between CPU registers and thread's context storage (e.g. on top of stack) to suspend or resume a thread



Context Switching from Thread A to B

Save Context		Scheduler	Restore Context	
HW	SW		SW	HW

- Starts with exception handler
  - Hardware pushes some of CPU context onto stack: xPSR, PC, LR, R12, R3-R0
  - Software saves future value of A's SP into its TCB
    - Offset of 32 bytes for upcoming push
  - Software pushes remaining CPU registers onto A's stack: R4-R11
    - Future value saved in TCB is now correct
- Scheduler decides what to run
- SW loads stack pointer with B's saved stack pointer value and restore B's context
  - Software pops values from B's stack into CPU registers R4-R11
  - CPU exits handler by loading PC with EXC_RETURN, causing popping of R0-R3, R12, LR, PC, xPSR
- B resumes executing…
  - Because PC was loaded with B's PC value



Example: RTX5 Code to Save and Restore Context

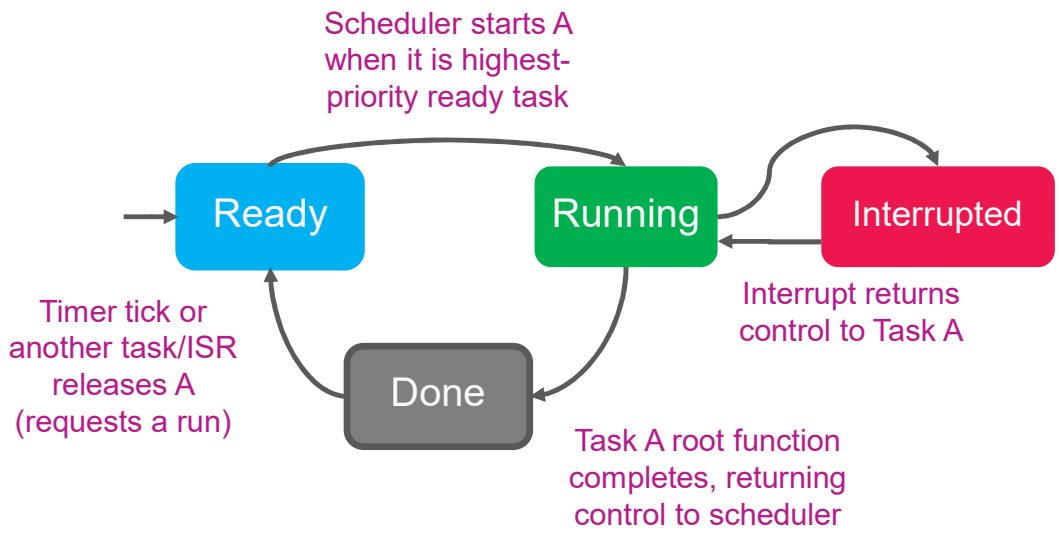
- Discussion of using PendSV and SVCall exception handlers in OS
  - Advanced topic, not covered here
  - https://developer.arm.com/documentation/107706/0100/System-exceptions/Pended-SVC---PendSV?lang=en

```
SVC_ContextSave
                MRS     R0,PSP                  ; Get PSP
                SUBS    R0,R0,#32               ; Adjust address
                STR     R0,[R1,#TCB_SP_OFS]     ; Store SP
PUSH            STMIA   R0!,{R4-R7}             ; Save R4..R7
                MOV     R4,R8
                MOV     R5,R9
                MOV     R6,R10
                MOV     R7,R11
PUSH            STMIA   R0!,{R4-R7}             ; Save R8..R11

SVC_ContextRestore
                LDR     R0,[R2,#TCB_SP_OFS]     ; Load SP
                ADDS    R0,R0,#16               ; Adjust address
POP             LDMIA   R0!,{R4-R7}             ; Restore R8..R11
                MOV     R8,R4
                MOV     R9,R5
                MOV     R10,R6
                MOV     R11,R7
                MSR     PSP,R0                  ; Set PSP
                SUBS    R0,R0,#32               ; Adjust address
POP             LDMIA   R0!,{R4-R7}             ; Restore R4..R7

                MOVS    R0,#~0xFFFFFFFD
                MVNS    R0,R0                   ; Set EXC_RETURN value
                BX      R0                      ; Exit from handler
```

Task States

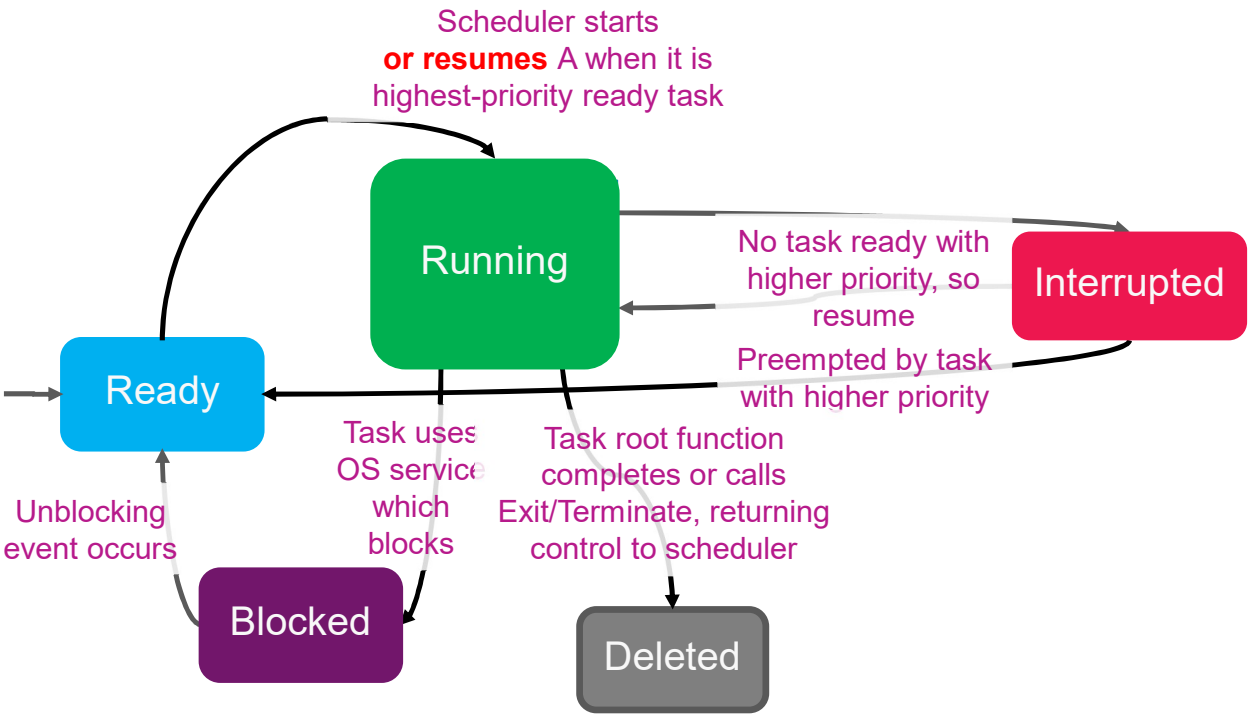
Task State Behavior with Non-Premptive Scheduler & Timer Tick

- Tasks run to completion and then block until scheduler restarts them
- Task States
  - Ready
    - Ready to run (released), but CPU is running other code
  - Running
    - Executing on the CPU
    - Only one exit: Run until end of task (completion)
  - Interrupted/Exception Handler
    - Task was preempted by exception or interrupt
  - Done
    - Task has completed and is waiting to be released (triggered) by timer tick ISR



Task State Behavior With Preemptive Scheduler

- Slight rule change:
  - Repeating tasks **never** complete, but instead call a function to block until needed again
- New State:
  - **Blocked:** Let task wait for something (event or time delay). Enables task to **yield** CPU **before** end, and later **resume** there.



More Scheduler Rules

1. All tasks start in ready state
2. Scheduler picks the highest-priority ready task and starts it running on CPU
   - Does this at every *scheduling point*
     - *W*hen RTOS starts running (with osKernelStart)
     - Timer tick, OS call,
     - ISR
     - Et cetera
   - If no tasks are ready, run *idle task*
3. If a ready task X has higher priority than the running task Y, move task Y to the ready state and run task X instead
   - Task X preempts task Y
4. A running task may call a function which makes it block (e.g. wait for event)
   - Scheduler moves that task to blocked state, then starts highest-priority ready task running (2.)
5. Scheduler moves task from blocked to ready when unblocking event happens