Content of this Module

- Tasks in RTX, Task States, Creation and Deletion
- Advanced Real Time Task Scheduling
  - Periodic Task scheduling
  - Earliest Dead Line First
  - Rate Monotonic
  - System performance and Assumptions
  - Prioritized Scheduling
  - Priority Inversion
  - Priority Inheritance
  - Dead Lock
  - Priority Ceiling

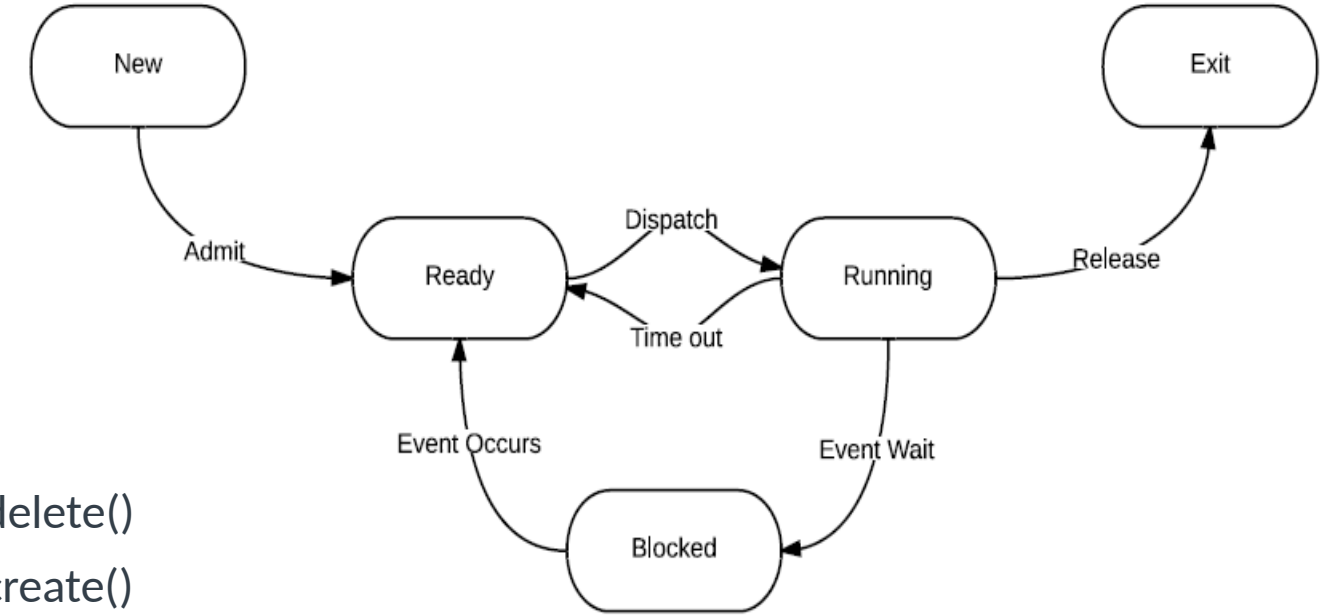
- Priority Scheme in RTX, Options
- Simple Time Management API
- RTX control functions

Tasks in RTX

- Task is the term used to refer to a process in RTX
- C function with keyword __task preceding the function return type

- Some basic API functions:
  - OS_TID
  - os_tsk_create(task, priority);
  - os_tsk_create_user(task, priority, &stack, sizeof(stack));
  - os_tsk_create_ex(task, priority, parameter);
  - os_sys_init(task);
  - os_tsk_delete(taskID);
  - os_tsk_delete_self();

Task State



- P_TCB->state
  - rt_TypeDef.h

- #define INACTIVE 0; - Related function: os_tsk_delete()
- #define READY 1; - Related function: os_tsk_create()
- #define RUNNING 2; - Related function: os_tsk_self() – only one running task, this returns its TID
- #define WAIT_DLY 3; - Related function: os_dly_wait() – blocked for a certain time
- #define WAIT_ITV 4; - Related function: os_itv_set() and os_itv_wait() – repeat the task periodically
- #define WAIT_OR 5; - Related function: os_evt_wait_or() – wait for or-event
- #define WAIT_AND 6; - Related function: os_evt_wait_and() – wait for and-event
- #define WAIT_SEM 7; - Related function: os_sem_wait() – wait for sempahore
- #define WAIT_MBX 8; - Related function: os_mbx_wait() – wait for mailbox message
- #define WAIT_MUT 9; - Related function: os_mut_wait() – wait for mutex
  - rt_Task.h

Blocked

Task Creation and Deletion

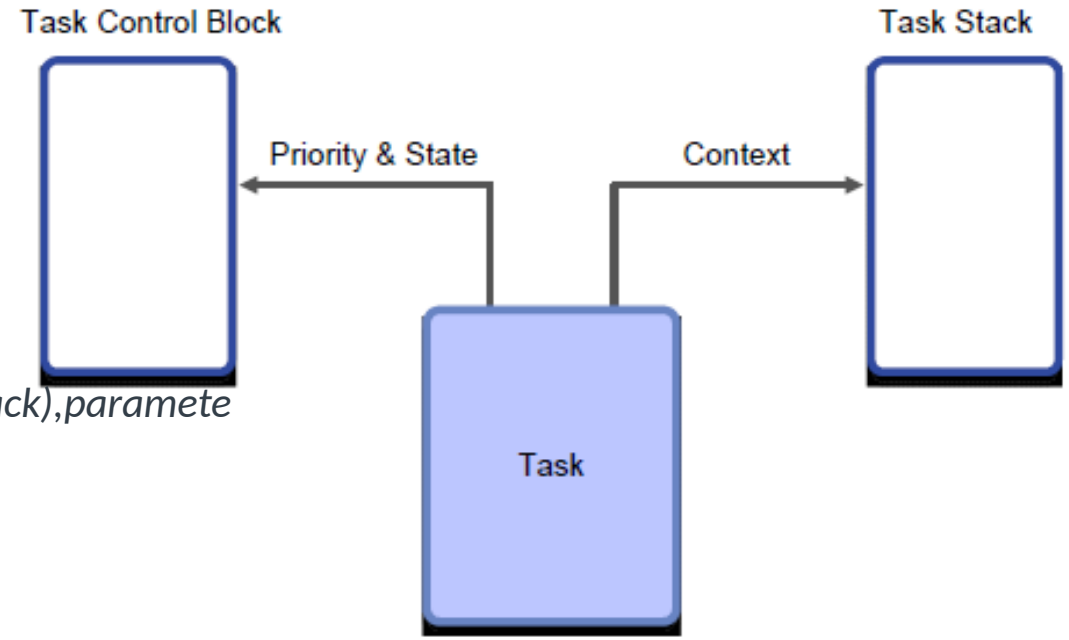
- Related functions

  Task creation:

  OS_TID - taskID;
  __task void some_task() {...}
  taskID = os_tsk_create (some_task, priority);
  taskID =
  os_tsk_create_user_ex(some_task,priority,&stack,sizeof(stack),parameter)

  Task deletion:

  os_tsk_delete (some_task);



- Priority
  - U8 member in the TCB in rt_TypeDef.h
  - Larger number = higher priority
  - ‘0’ for an ideal task, ‘1’ for a round-robin task
- User Stack (_user)
  - You can set the default size in the configuration wizard

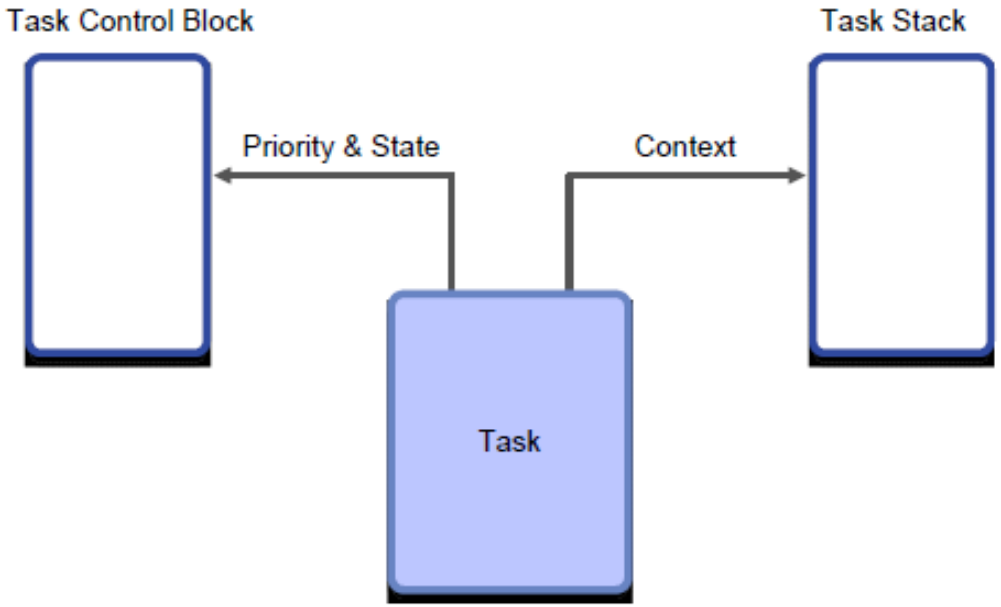
Task Creation and Deletion

- Parameters (_ex)
  - RTX allows you to have multiple instances of the same task
  - Passing parameters to a task at creation, for example, to specify GPIO pins

    taskID = os_tsk_create_ex(led_on_off, 0, led1)
    taskID = os_tsk_create_ex(led_on_off, 0, led2)
- Tasks can be created and deleted by tasks
  - Tasks can delete other tasks and themselves

    os_tsk_delete(some_task);
  - Tasks can create other tasks (but cannot create themselves)
  - The first task is created by:

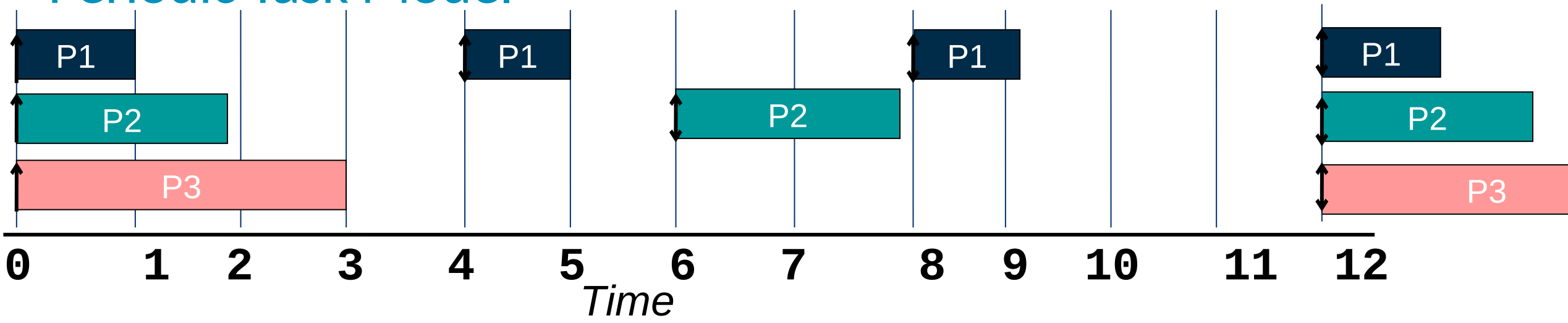
    os_sys_init(first_task);
    - The first task is commonly used for
      - Initialization of hardware
      - Creation of all other tasks
      - Then deletes itself



Advanced Real-time Scheduling

- Recall the aperiodic real-time scheduling policies introduced previously
  - EDD, EDF and LL
  - Same priority
  - No inter-task dependency
  - Aperiodic
- Before introducing the RTX scheduler, we will discuss two more real-time scheduling scenarios
  - **Periodic**
  - **Prioritized**

Periodic Task Model



- Describes real-time computational requirements
- Constant process execution time C_i
- Tasks are periodic with period T_i
- Release time
  - Task is released every T_i time units and is ready to run
- Deadline D_i .
  - Might be related to period T_i making analysis easier
  - Can also be earlier than the end of a period!

Task	Exec. Time C_i	Period T_i	Deadline D_i
P1	1	4	4
P2	2	6	6
P3	3	12	12

arm

Periodic Scheduling

- Goals
  - Meet all task deadlines!
    - Not always **schedulable** to meet all deadlines!
  - Maximize processor utilization *(U)*
    - The fraction of time CPU performs useful work
    - Limit scheduling overhead
    - Limit context switching overhead
- Which ready task to run?
  - Based only on task importance?
    - Would starve consistently low priority task unnecessarily
- Some common approaches
  - Earliest deadline first (EDF)
  - Rate monotonic (RM)
  - Deadline monotonic (DM) ($D_i \leq T_i$)

$$U = \sum_{i=1}^{m} \frac{C_i}{T_i}$$

Earliest Deadline First

- Run the job with the earliest deadline first
- Utilization-based schedulability test depends on deadline constraints
  - Di = Ti : Schedulable if utilization ≤ 100%
  - Di > Ti : Schedulable if utilization ≤ 100%
  - Di < Ti : Have to use a more complicated test
- Hard to implement such dynamic policy
  - Run-time scheduler has to keep track of all deadlines, which also depends on tasks' release time
  - Sorting by deadline, time complexity: *O(n)*
- Periodic tasks only? Will repeat every hyper-period, the least common multiple of all task periods

Rate Monotonic

- Shortest period first, preemptive
- Optimal for workload where deadline = the end of period, i.e., $D_i=T_i$.
- Assume $D_i=T_i$, the utilization bound for RM: U_{max} is*

$$U_{Max} = n\left(2^{1/n} - 1\right)$$

- U_{max} approaches ln2 (around 0.693) as n approaches positive infinity.
- If $U<U_{max}$, 100% schedulable, otherwise, maybe schedulable. An example:

Task	Execution Time C	Period T	RM Dispatch
P1	1	8	First
P2	1	9	Second
P3	9	12	Third

- A modified approach: harmonic RM – task period can be evenly divide every longer task period for example, periods of 10, 20, 40, 80… Bound is 100%

*Originally proved by CL Liu and JW Layland in 1973. "Scheduling algorithms for multiprogramming in a hard-real-time environment."

System Performance During Transient Overload

- RM, DM – Each task has fixed priority. *So?*
  - This priority determines that tasks will be scheduled consistently
    - Task A will always preempt task B if needed
    - Task B will be forced to miss its deadline to help task A meet its deadline

- EDF – Each task has varying priority. *So?*
  - This priority depends upon when the task's deadline is, and hence when the task becomes ready to run (*release time*)
    - Task B may have higher priority than A depending on release times
    - To determine whether task A or B will miss its deadline we need to know their release times
    - Can theoretically harness up to 100% of the CPU performance

Our Assumptions

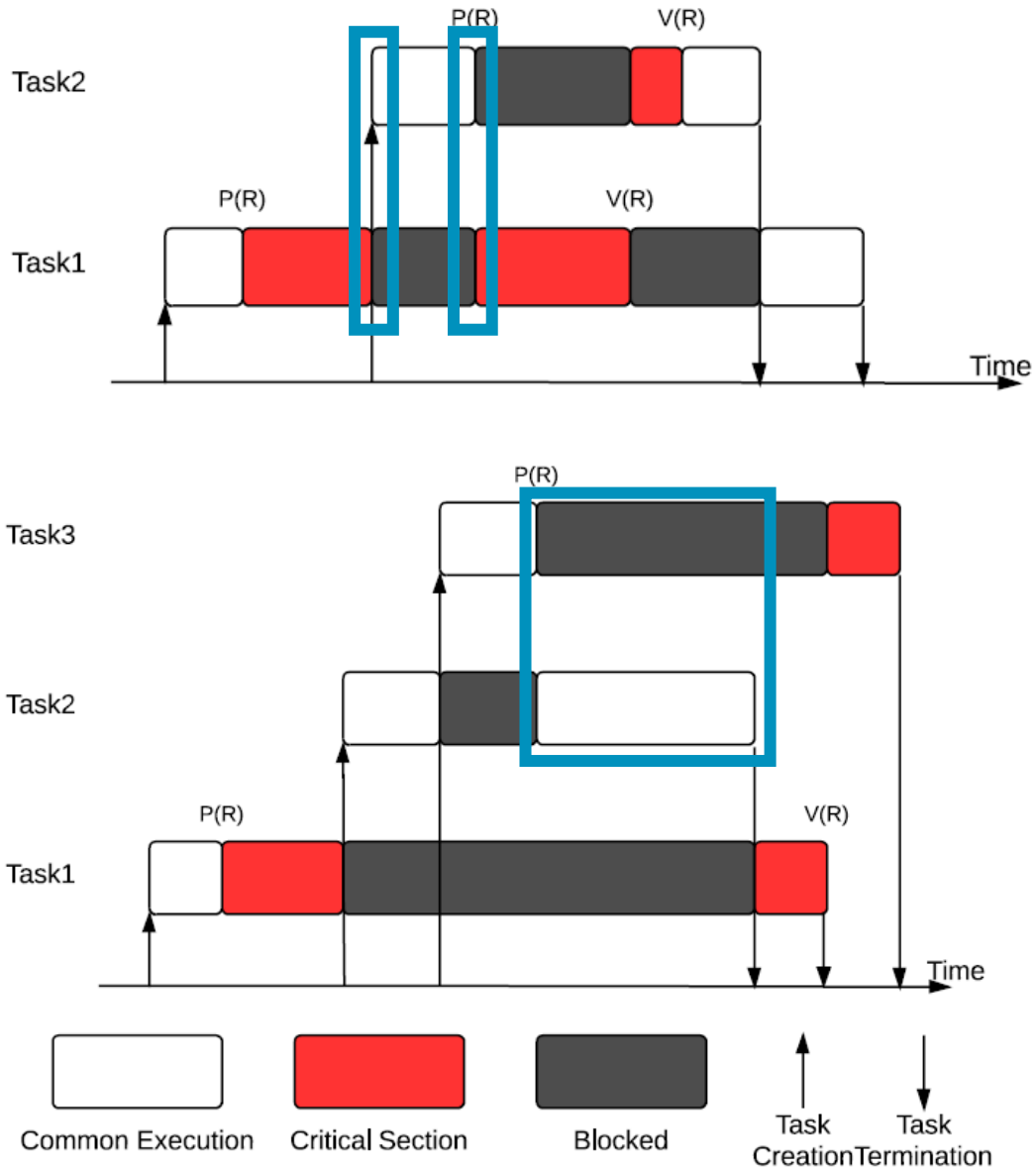
- Tasks are periodic with period T_i
  - Extension 1: Support aperiodic tasks by setting $T_i = min(T_{Interarrival\ i})$
- Single CPU
  - OK for some systems, but not all
- $C_{ContextSwitch} = 0$
  - Scheduler speed. EDF requires ordered list of ready tasks.
  - Task state information needs to be swapped
- Constant process execution time C_i
  - What if the worst-case time is much slower than average?
    - Then need a processor fast enough to handle the worst case
    - Most of the time the processor will be wasting resources
- No data dependencies between tasks
  - Not all tasks are ready to run simultaneously
  - Some tasks block on other tasks

Prioritized Scheduling

- *taskID = os_tsk_create(some_task, priority);*
- Prioritized Scheduling means that task with higher priority should be dispatched and can preempt lower priority tasks
- The problem is dependency. For instance, can a task in its critical section be preempted?
  - If no, any problem?
  - If yes, any problem?
  - The point of prioritized scheduling is based on the emergency level of the task, instead of whether it's in a critical section. But you have to pay extra attention to critical sections as otherwise they may cause conflict between tasks.
- **Priority inversion**

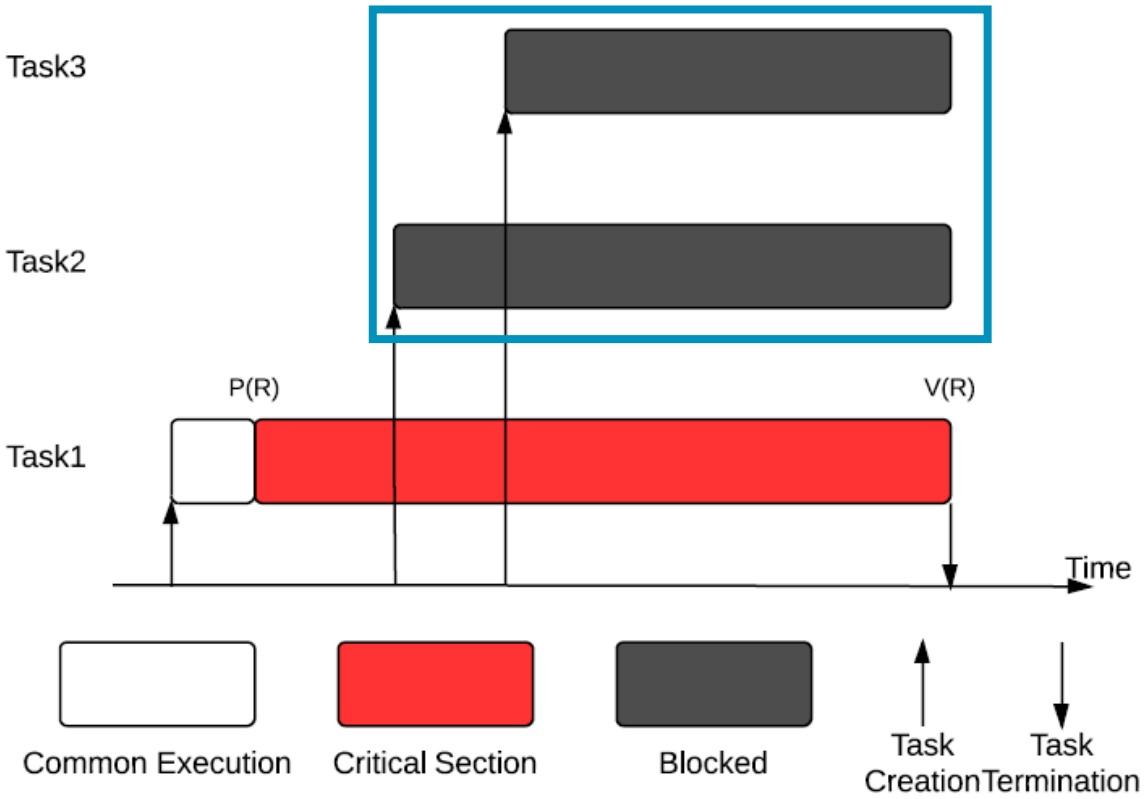
Preemptive Critical Section

- Priority(T_N)>Priority(T_M),if N>M
- Seems okay?
- But what if a higher priority task would like to enter the same critical section? P(R) blocks task 2 if task 1 has the mutex (recall the requirements for mutex). Still acceptable as contaminating CS may cause even worse disaster, better avoid that and let task 1 finish first.
- Blocking time seems to be bounded by the maximum length of critical section of lower priority tasks. But no. See the second example.
- Priority inversion: even though task 2 is not requesting R, it blocks task 3!



Non-Preemptive Critical Section

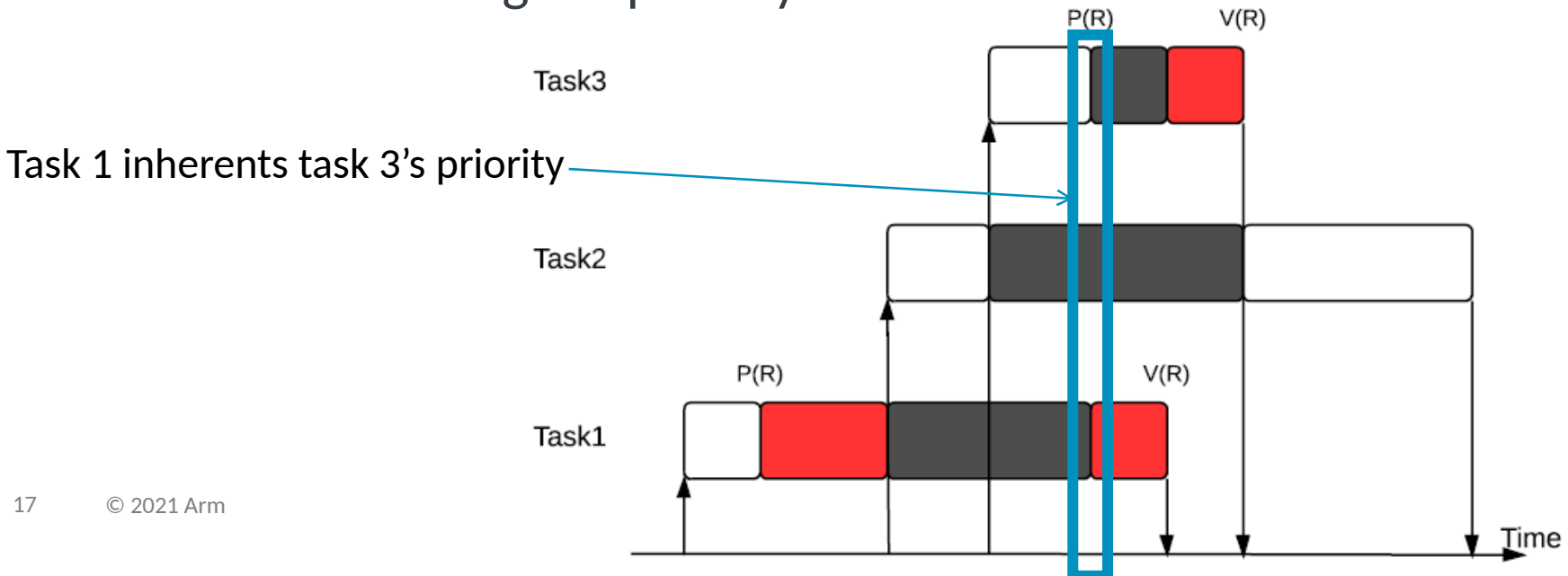
- Plausible? Can solve previous problem. But…
- Task 3 and 2 are on fire!



arm

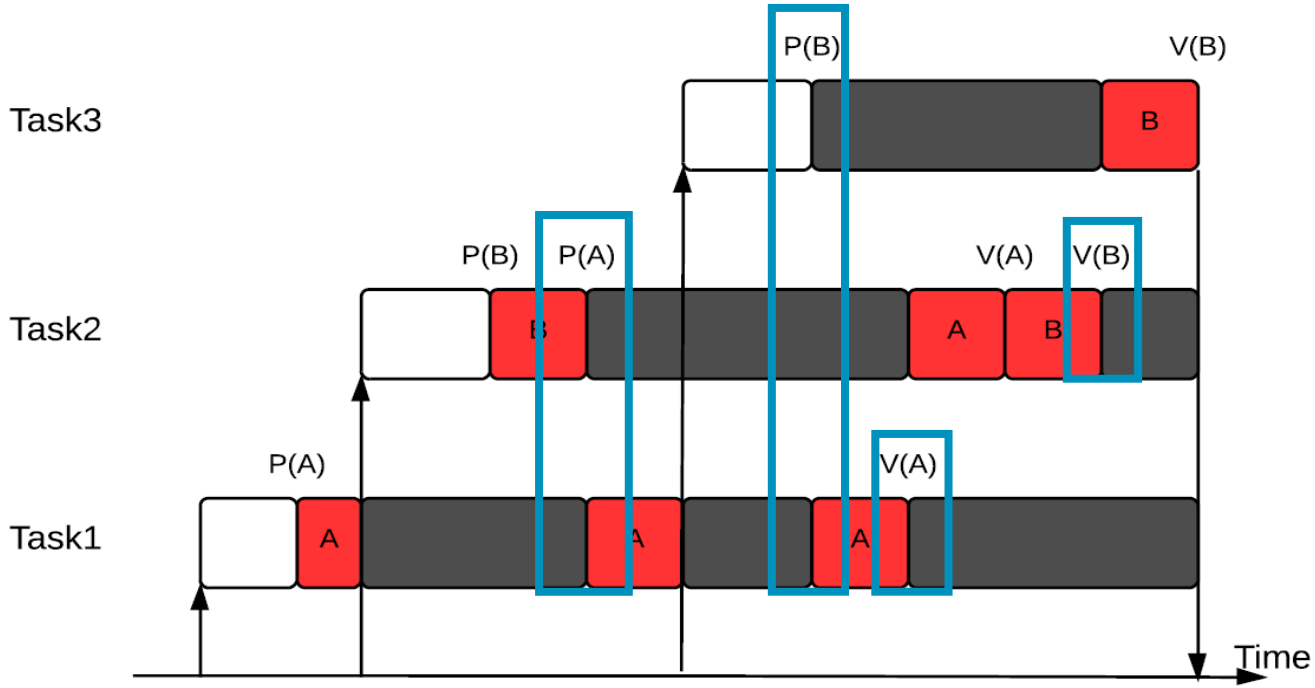
Priority Inheritance

- Applied in many RTOS including RTX.
- Blocking time is now bounded by the maximum (sum) length of critical section of lower priority tasks.
- The idea is to elevate the priority of low priority task (if it blocks high priority task) to the highest priority of tasks blocked by it.
- And resume its original priority when it exits the critical section.



Priority Inheritance

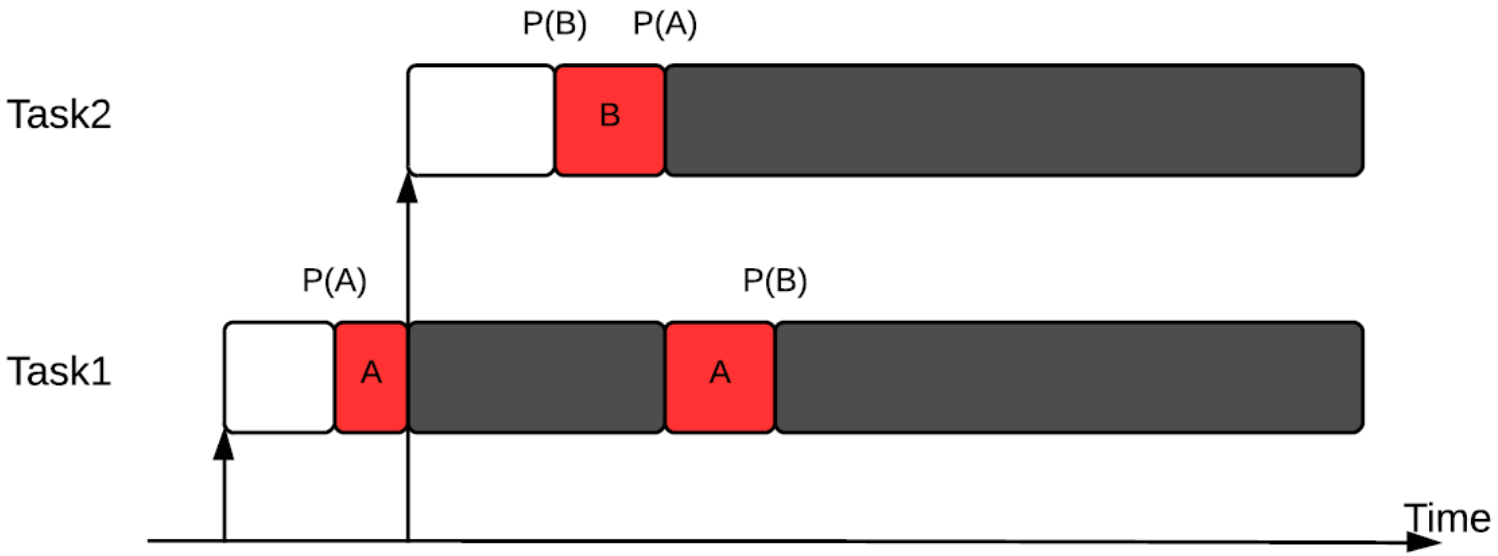
- If same priority, then FIFO
- Another example:



- Task 2 blocked by task 1, hence Priority(T_1) = Priority(T_2)
- Then, task 3 is blocked by task 1, hence Priority(T1) = Priority(T_3), also Priority(T_1)=Priority(T_2)=Priority(T_3) as task 2 is blocked by task 1. So Task 1 continues(FIFO)
- After task 1 releases A, its priority will be resumed. Task 2 continues.
- Similarly, after task 2 releases B, its priority will be downgraded and task 3 continues.

Deadlock Caused by Priority Inheritance

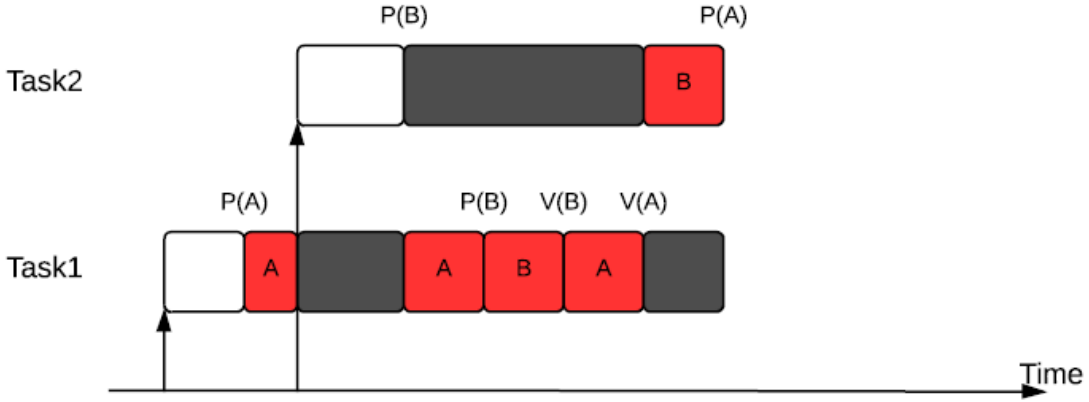
- Unfortunately, priority inheritance is not perfect. Consider the following example:



- Recall Coffman conditions for deadlock. Deadlock here!

Priority Ceiling

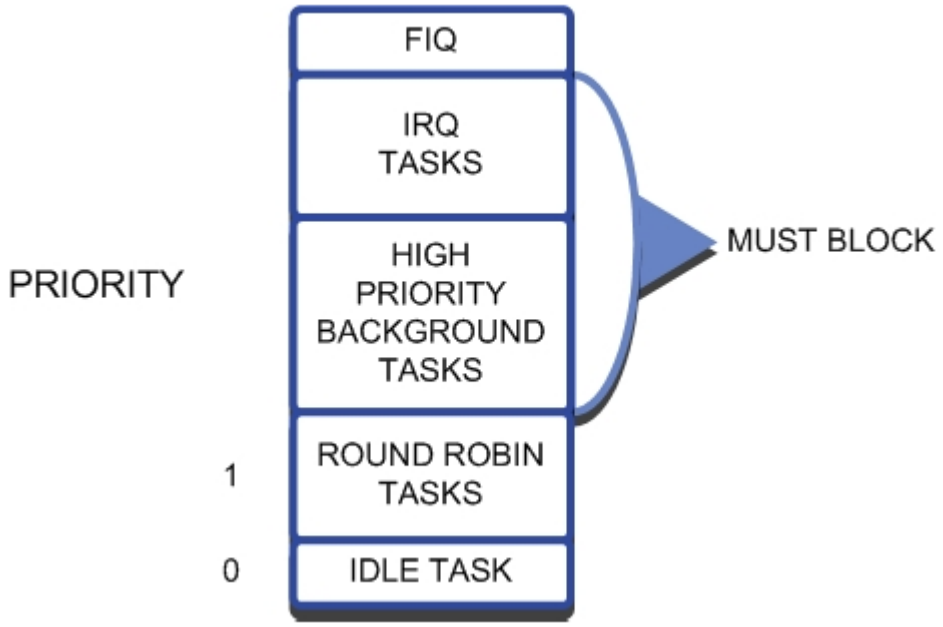
- Improvement on priority inheritance. But **not implemented in the RTX**.
- Define the priority ceiling for a resource: PC(R) are the highest priority processes that may request mutual exclusive access to R
- Priority ceiling accepts requests to enter any critical section only if task A's priority is **larger** than PC (all resources accessed by tasks other than A). This solves deadlocks by preventing circular wait conditions
- If task A is blocked due to this, then the priority of task B which is blocking task A, will be elevated to A's priority (if A is the highest) as the usual priority inheritance protocol.
- Another improvement is that it reduces the blocking time to at most the length of one critical section of a low priority task.



Task Priority Scheme in RTX

- RTX supports priority scheduling and applies priority inheritance policy with mutex.
- Possible to alter the priority of a task after creation.

  OS_RESULT os_tsk_prio(taskID, priority);
  OS_RESULT os_tsk_prio_self(priority);
- Higher number, higher priority, from 0 to 255
  - A FIFO queue for each priority
  - Fast interrupt has the highest priority (handled by CM core instead of RTX)
  - Then interrupts
  - Then tasks with a priority larger than 1. RTX will not preempt tasks with priority larger than 1 so they must be self-blocking!
  - Tasks in queue 0 will be put into queue 1 when there are no tasks running



RTX Scheduling Options

- Pre-emptive scheduling
  - Each task has a **different priority** and will run until it is pre-empted or has reached a blocking OS call.
- Round-Robin scheduling
  - Each task has the **same priority** and will run for a fixed period, or time slice, or until it has reached a blocking OS call.
  - Quantum is determined by the **RTX_Conf_CM.c** file: Round-Robin Timeout [ticks] × Timer tick value. For example, by default, the time quantum is 10000µs ×5 = 50000µs=50ms.
  - If quantum expired, state will be changed to READY.
- Co-operative multi-tasking
  - Each task has the **same priority** and the Round-Robin is disabled. Each task will run until it reached a blocking OS call or uses the *os_tsk_pass()* call.

- The default scheduling option for RTX is **Round-Robin Pre-emptive**. Prioritized RR.
- RTX does not provide aperiodic or periodic deadline scheduling.

Simple Time Management APIs

- Besides the scheduling options, RTX can also manage time through the following APIs:
- Get current time
  - *U32 os_time_get (void);*
  - Returns current OS time measured in ticks
- Simple delay
  - *void os_dly_wait (unsigned short delay_time)*
  - Turn the state of calling task into WAIT_DLY
  - Actual delay time = *delay_time* × timer tick value
  - After the delay time, turn the state of the task back to READY
  - Make sure you have the correct timer clock value
- Periodic task execution
  - Set the interval within the task by calling *void os_itv_set (unsigned short interval_time)* first
  - Actual delay time = *interval_time* × timer tick value
  - Then change the to WAIT_ITV by calling *void os_itv_wait (void)* inside the task

Simple Time Management APIs

- Cooperate function
  - *void os_tsk_pass ();*
  - Task calling this function will voluntarily give up its current possession of CPU and allow next ready task with same priority to run.
  - No task switch if the next ready task has a lower priority than the calling task.

- User timer function
  - Set the number of available user timer first.
  - Set a single shot timer by calling *OS_ID os_tmr_create (unsigned short tcnt, unsigned short info)*
  - Actual countdown time = *tcnt* × timer tick value
  - After it reaches zero, RTX runs the callback function *void os_tmr_call (U16 info)*. The prototype function can be found in RTX_Conf_CM.c
  - *Info* is the parameter passed to the callback function

Other RTX Control Functions

- Scheduler lock – stops task switching, though ISRs can still execute
  - If you want your code to run as a contiguous block, (critical section for example), it is a good idea to temporally stop the scheduler.
  - *void os_tsk_lock();*
    - Disables kernel timer interrupts, stopping task switching
  - *void os_tsk_unlock();*
    - Enables kernel timer interrupts, allowing task switching
- Power management - used in low power RTX
  - Used with *os_dly_wait* or *os_tmr_create.* Hibernate the CPU until next activity.
  - *U32 os_suspend();*
    - Calculates for how many ticks the system can sleep
    - Locks task scheduler
    - Use this in idle_task to determine when to set wake-up timer before sleeping (wfi instruction)
  - *void os_resume(U32 sleep_time);*
    - Updates system timer with sleep duration (number of ticks)
    - Unlocks task scheduler, allowing normal scheduling
    - Use this in idle_task to resume

Next

- Lab
- Sharing data in RTX