PROGRAMMERING AV INBYGGDA SYSTEM

Real-time operating systems

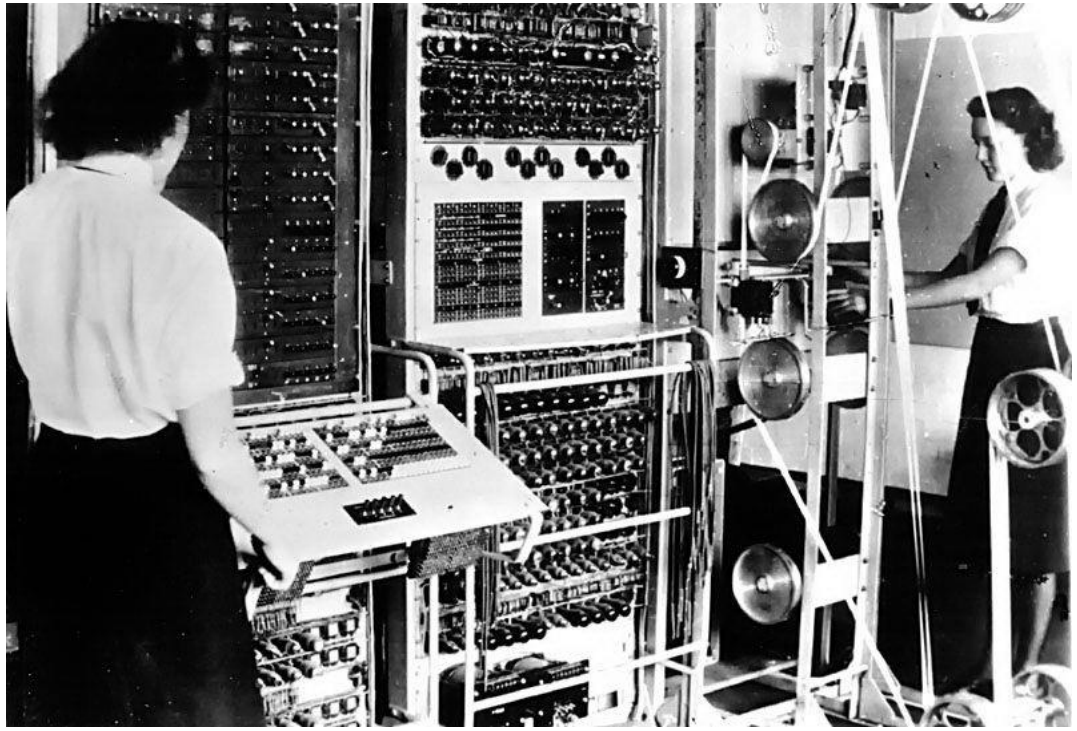
Dario Salvi, 2025

Materials

- ESP32 Technical Reference Manual:
- Making embedded system: chapter 5 “Task management”
- Real-time fixed and dynamic priority driven scheduling algorithms: theory and experience
- Online resources:
  - https://www.freertos.org/
  - http://dev.ti.com/tirex/explore/node?node=AIHYsnK2VMe6vGRWXhb3uQ__eCfARaV__LATEST
  - https://en.wikipedia.org/wiki/Context_switch
  - https://csperkins.org/teaching/2015-2016/adv-os/
  - https://en.wikipedia.org/wiki/Worst-case_execution_time
  - https://en.wikipedia.org/wiki/Semaphore_(programming)
  - https://en.wikipedia.org/wiki/Priority_inversion
  - https://docs.espressif.com/projects/esp-idf/en/stable/api-reference/system/freertos.html
  - https://www.freertos.org/RTOS-software-timer.html
  - https://docs.espressif.com/projects/esp-idf/en/stable/api-reference/system/wdts.html
  - https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html
  - https://www.freertos.org/Embedded-RTOS-Queues.html
  - https://www.freertos.org/RTOS-task-notifications.html

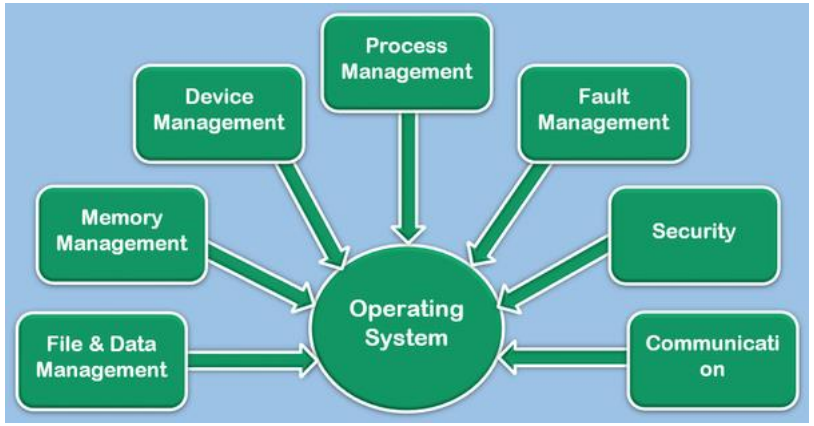
Before Operating systems

- The first computers did not have an OS.
- They loaded data to be processed in memory.
  - Started execution
  - Printed the result
- All the code was machine-specific.



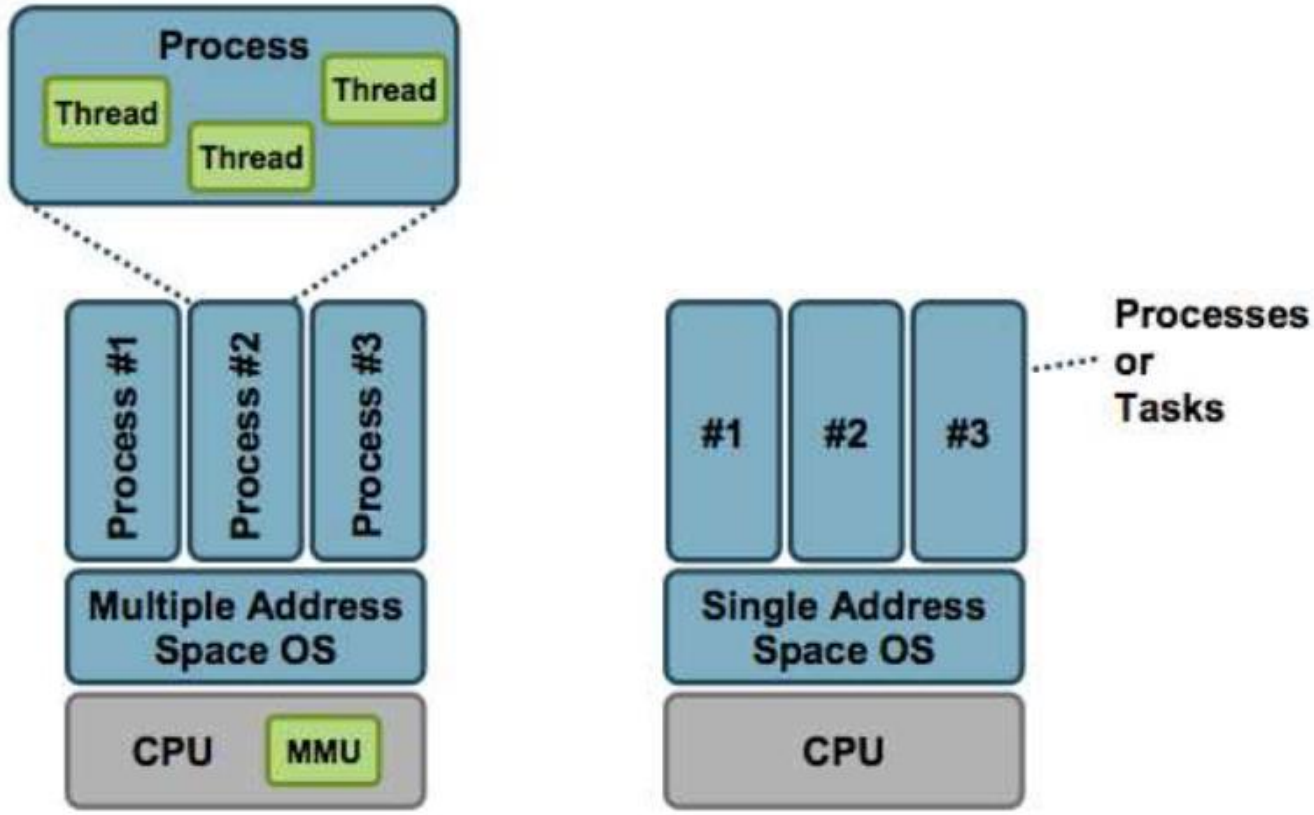
Operating system

- An OS is the first software that is loaded by a computer.
- It loads other programs from different sources (HD, CD, USB).
- It abstracts hardware through a common programming interface.
- It offers “shared services” to the running programs.
- Isolates running programs and allows communication only under strict rules.



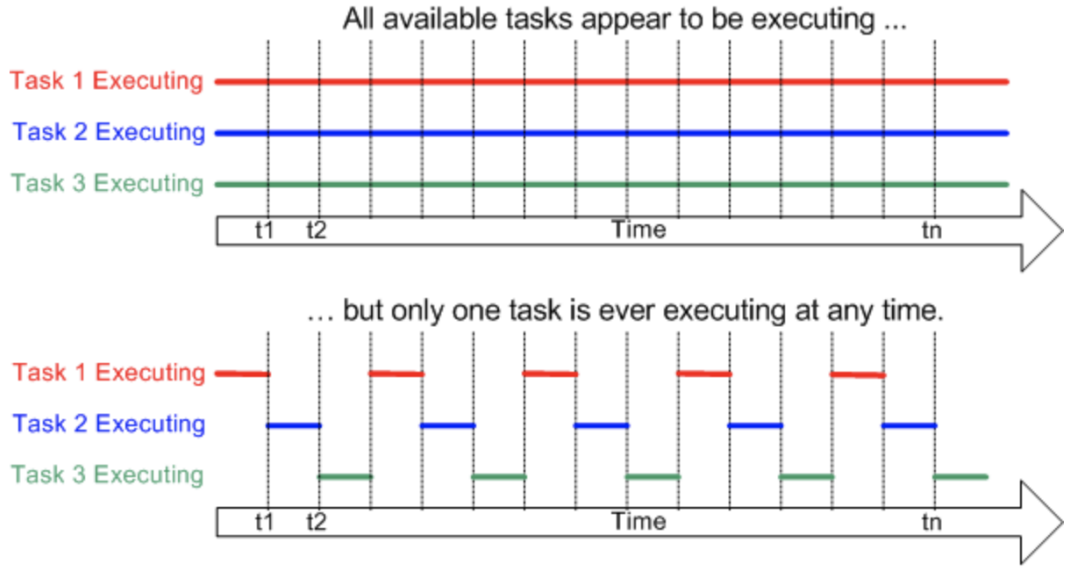
Processes, threads and tasks

- A *task* is something the processor does.
- A *thread* is a task plus some overhead such as memory.
- A *process* is usually a complete unit of execution with its own memory space, usually executed separately from other processes.



Parallel execution

- Tasks (or threads or processes) appear to be executed at the same time.
- This can be partially true on multi-core CPUs.
- In reality tasks are split into smaller chunks and executed one by one.
- The operating system has a *scheduler* that does the switching between tasks, allowing each to run in its proper turn.

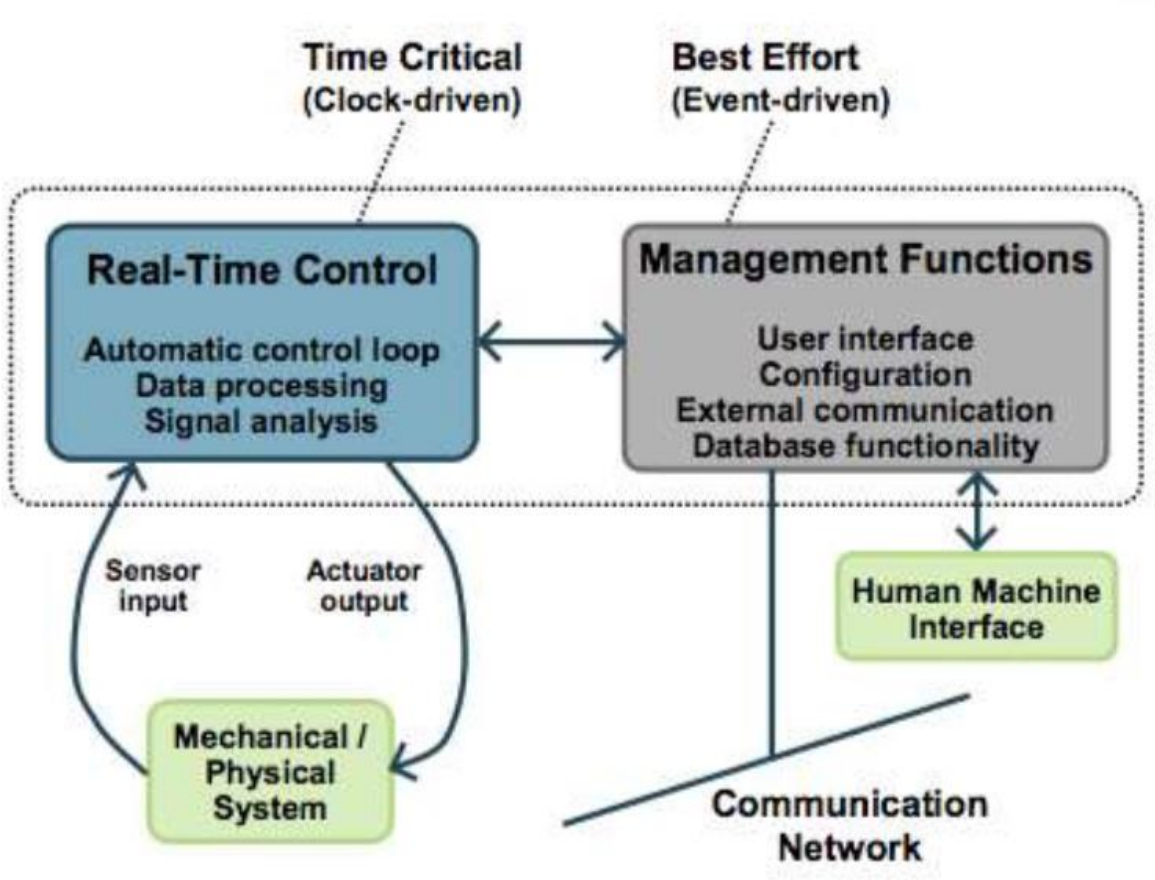


Real Time Operating System (RTOS)

- An RTOS may be needed when several different, independent or almost **independent programs** (processes, tasks, threads) need to be run simultaneously.
- RTOS takes care of the **prioritization** between the processes and determines access to the processor and other resources.
- "**Real time**" means that time-critical tasks are executed within some timing constraints.
- The OS takes care of **scheduling tasks**, not the programmer!

Real time constraints

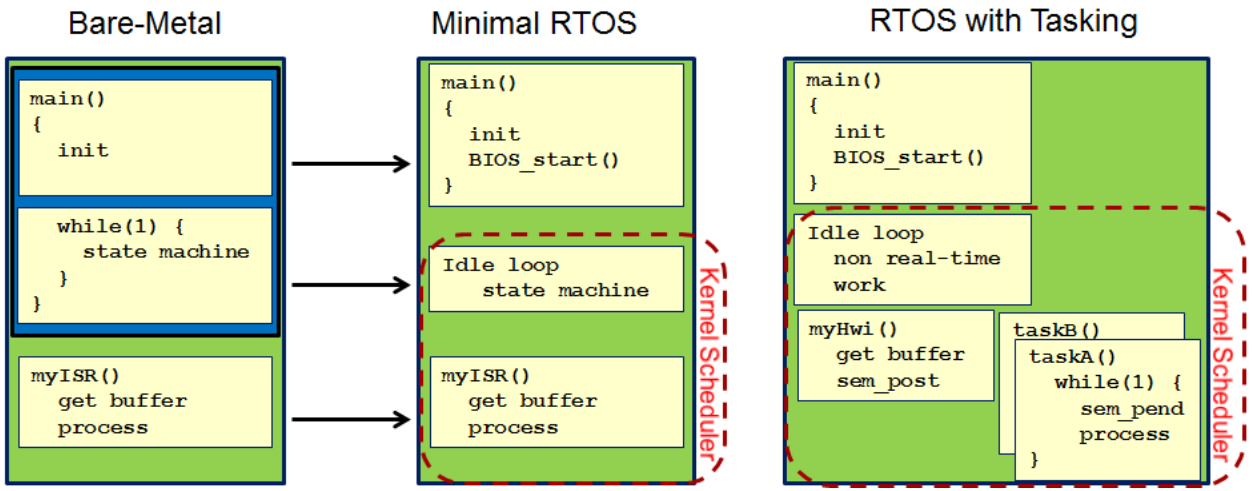
- **Hard** real time: timing constraints are mandatory.
  - Examples:
    - Automatic control (car engine)
    - Signal processing (musical instrument)
- **Soft** real time: should be fulfilled "on average".
  - Examples:
    - User interfaces (a thermostat)
    - Access to resources (reading files)
- Most systems have few hard deadlines and a number of soft deadlines.



Who defines time requirements?

- Time requirements can be:
  - **Explicitly** defined by the system design.
  - **Implicitly** defined by other requirements.
- Examples of **explicit requirements**:
  - A production line must be able to make two units per second.
  - The image for a traffic controller must be updated with a maximum of two seconds delay end-to-end (transmitted radar signal to monitor).
- Examples of **implicit requirements**:
  - Accuracy: track an aircraft's position with a maximum of ten meters of error.
  - Reliability: reaction to sensors changes must be quick so that no one gets injured.
  - Usability: sufficiently quick response at the push of a button (like 200 ms).

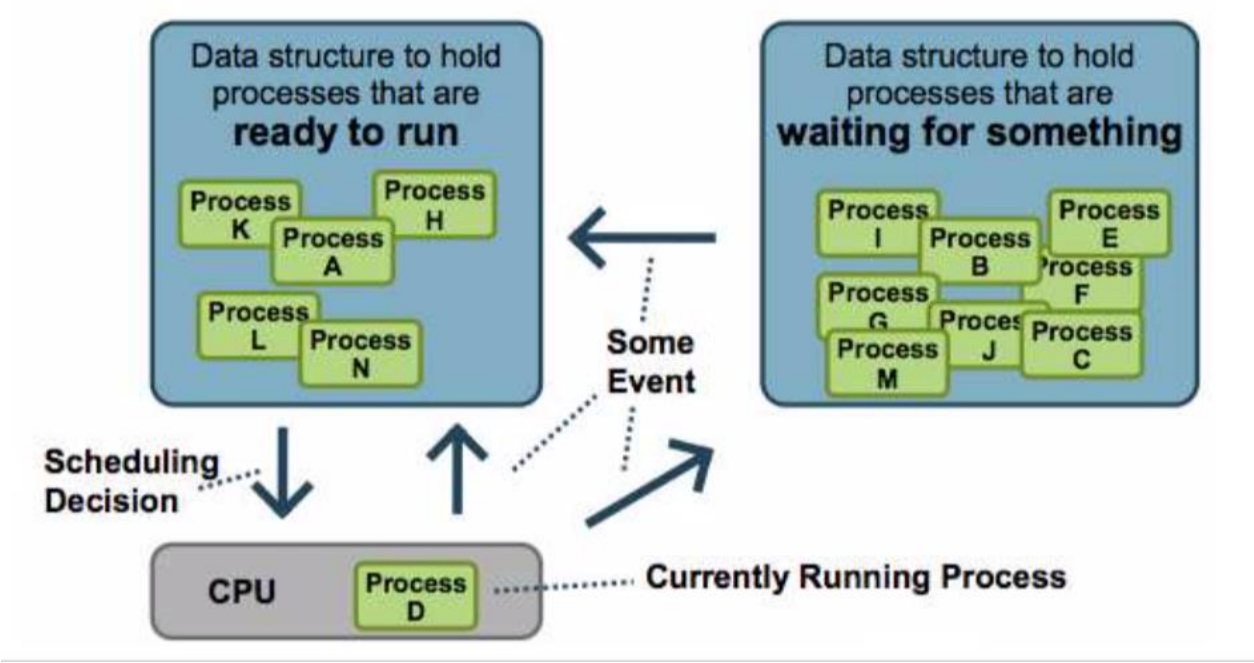
From single programs to OS tasks



- In single-programme software, synchronisation is managed by the developer through a (fragile) main loop and ISRs.
- In a simple RTOS, real time operations (ISRs) and non real-time operations are clearly split and managed by a scheduler, but there is a shared state machine and no separation of tasks.
- In a complex RTOS, operations can be split into tasks, each one can be blocking.

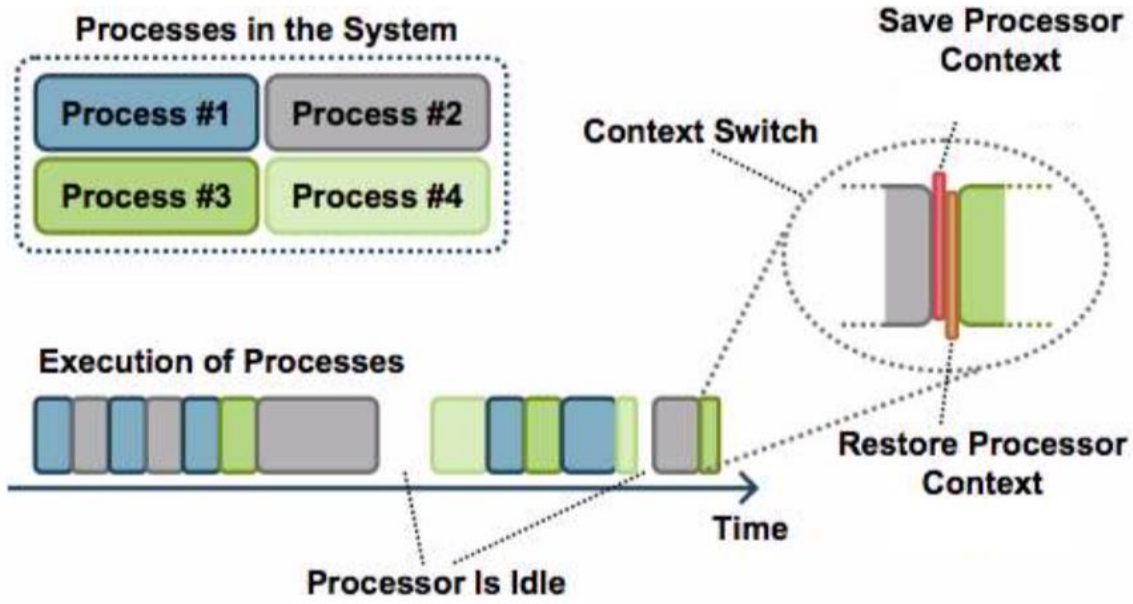
Scheduling

- The OS is responsible for "choosing" the process to run.
- Some processes may be waiting for an event (e.g. user interaction, sensor).
- Other processes may be ready to run.



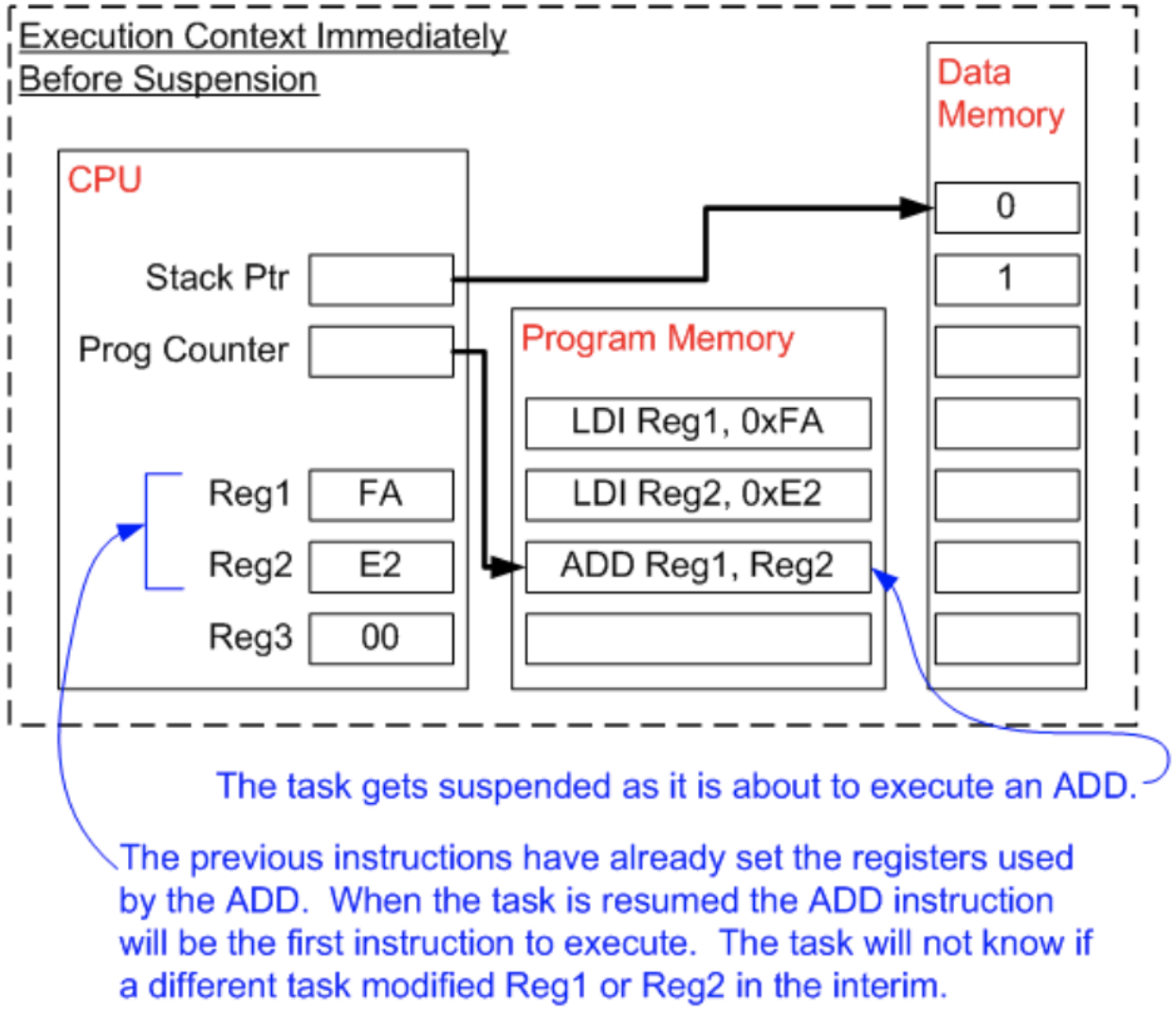
Context switch

- Processes are executed in a given order.
- Each time a process is stopped to give space to another process there is a “context switch”.
- In a context switch all registers, stack etc. of a process need to be saved and the ones of the coming process need to be restored.
- This is the same as it happens with interrupts, but it’s managed in software.
- Context switches are costly.



Context switch: example

- A task is suspended before it was going to execute an addition of two registers.
- When resumed, the registers must contain the values to be added!



Scheduling strategies

- How does a RTOS know when to switch to another task?
- **Cooperative** scheduling
  - Each task decides to give up control with an explicit call to a function (e.g. sleep()).
  - Pros: easy to implement, very quick.
  - Cons: if a task hangs, the whole system hangs.
- **Time-slice** scheduling:
  - The OS takes control on a timer tick and guarantees that each task is given a slot to execute.
  - Pros: a blocked task does not block the others.
  - Cons: not feasible for real time operations.
- **Pre-emptive** scheduling:
  - A running task continues until:
    - It finishes.
    - A *higher priority* task becomes ready. In this case the higher priority task *preempts* the lower priority task.
    - The task gives up the processor.
  - Pros: tasks are given CPU time according to their priority.
  - Cons: complex.

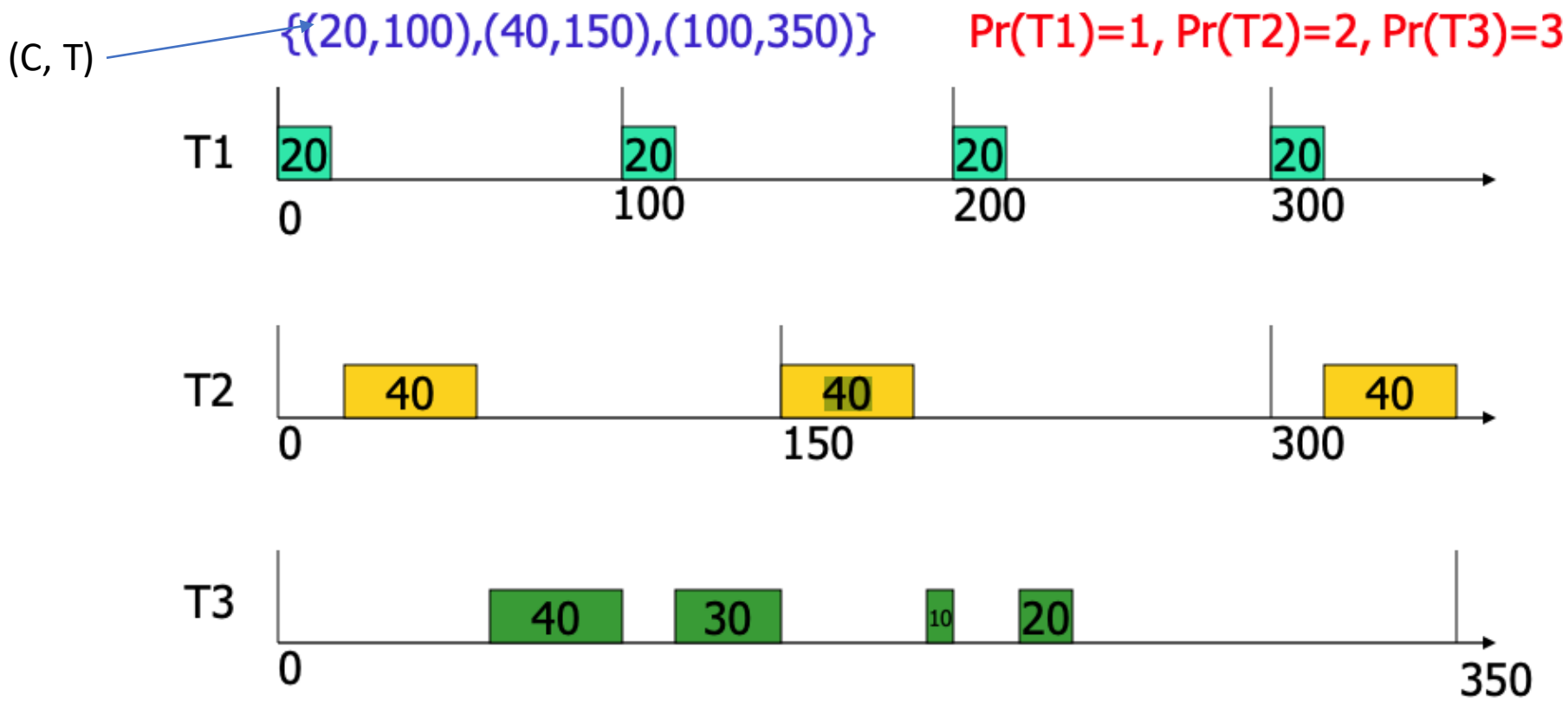
Scheduling with priorities

- **Static Cyclic Scheduling**:
  - The programmer builds a fixed timetable based on time constraints (e.g. sampling frequency, execution time).
  - The execution is repeated cyclically according to the time table.
- **Fixed Priority Scheduling**:
  - Assigns fixed priority to all the single executions (jobs) of each task.
  - Priorities can be assigned based on periods or deadlines.
- **Dynamic Priority Scheduling**:
  - Assigns different priorities to the individual jobs of each task.
  - Priorities can change, for example if deadlines are not met.
- Mixed approaches.

Rate-monotonic Scheduling (RMS)

- It's a Fixed-Priority Algorithm.
- It assigns priorities to tasks based on their periods: the shorter the period (higher frequency), the higher the priority.
- Suppose that we have a group of **periodic** tasks with: period time T_i and *maximum* execution time C_i.
- The deadline for each task is the same as the period time T_i.
- **Each task has a fixed priority: the shorter the time period, the higher the priority.**
- Pre-emption: A higher priority task can interrupt a task with lower priority.
- **Processor utilization factor**: $U = C_1 / T_1 + C_2 / T_2 + ... + C_n / T_n = \sum_{i=0}^{n} \frac{C_i}{T_i}$
- All tasks will meet their deadlines if: $U \leq 1$
  - Proof: given an amount of time T, task i will be executed T/T_i times occupying C_i (T / T_i) computing time. So, in total, all tasks will occupy $\sum_{i=0}^{n} T\frac{Cl}{Ti} = UT$, which should be < T

RMS: working example



RMS: non-working example

- Task set: T1=(2,5), T2=(4,7)
- U = 2/5 + 4/7= 34/35 ~ 0.97 (schedulable?)
- RMS priority assignment: Pr(T1)=1, Pr(T2)=2

Utilization bound test

- U <1 doesn’t imply ’schedulable’ with RMS (but the opposite is true)
- Utilization bound test:
  - Given a task set S, find X(S) such that U<= X(S) if and only if S is schedulable by RMS (necessary and sufficient test).
  - Result: if $U <= n*(2^{1/n}-1)$, then S is schedulable by RMS.

For proof, see liu-layland.pdf

- Previous example: $2*(2^{1/2}-1) = 0.83$, U = 0.97

Context switching overhead

- Let's consider the worst case: two activities per task, when one starts and when one ends.
- C_s : the longest time for a complete context switch.
- The time that needs to be calculated for each task becomes then: Ci + (2 * C_S)
- U becomes $\sum_{i=0}^{n} \frac{C_i+2Cs}{T_i}$

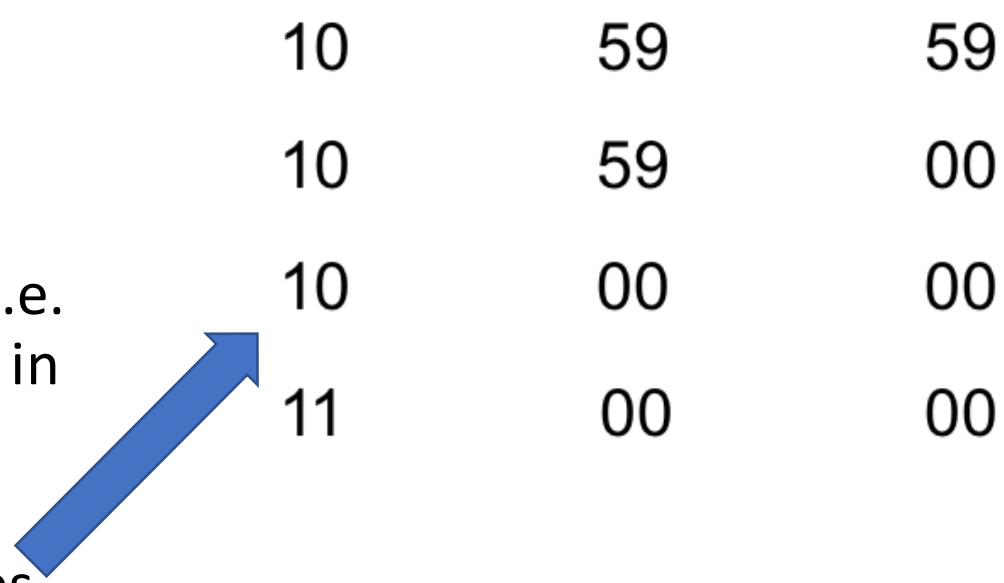
- 2 * C_S is negligible if it is smaller than the smallest T_i (the period of time of the most frequent task in the system).
- Often stated in number of clock cycles.

Worst case execution time (WCET)

- How do you compute C_i ?
- Simplifying:
  - 1 line of C code <-> 10 machine instructions.
  - 1 machine instruction takes 1 clock cycle.
- So a task with 100 lines of C code takes 1000 clock cycles-> with a 84 MHz clock frequency this gives C = 0.011 ms.
- Notable exceptions:
  - Floating point calculations.
  - Wait for hardware.
- This rule of thumb does not apply to safety-critical systems, in those cases you have to measure!

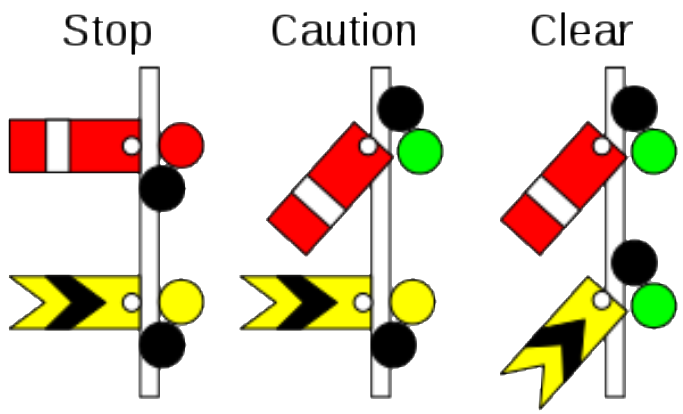
Synchronisation and race conditions

- As with interrupts, **race conditions** are possible between tasks.
- Example: global clock
  - Assume that a task "manages" the clock i.e. ensures that the current time is available in three global variables: hour, minutes, seconds
  - What happens if a task reads the variables while they are being changed?



Semaphores

- Each RTOS has a number of synchronisation primitives that can be used to solve these types of problems.
- The most important and most common type found in all RTOS: **semaphore**



Semaphores

- A semaphore has a value that is an integer greater than or equal to zero i.e. 0, 1, 2, ...
- A semaphore can be initiated to a specified value.
- A semaphore has two operations, in addition to initiation:
  - take () (also called wait ())
  - give () (also called signal ())

Semaphores

- Take / wait ()
  - If the value of the semaphore is greater than zero, the value is reduced by one and the program goes on (doesn’t wait).
  - Otherwise, the task called on take () is placed in a wait state.
- give / signal ()
  - The value of the semaphore is increased by one.
  - If one or more tasks are waiting for the semaphore, they are given the OK to go.

Semaphores

- Let’s consider the clock example:
- A semaphore called time_mutex is initiated to the value 1. 1

Task that updates the clock:

.

.

take (time_mutex); 0

code that counts up the time

give (time_mutex); 1

.

.

Task that reads the clock:

.

.

take (time_mutex); 0

read the clock

give (time_mutex); 1

.

.

Mutex and deadlock

- This use is called mutual exclusion and is used to protect common resources. A semaphore used in this way is often called a **Mutex**.
- When tasks get stuck trying to access resources we have a **deadlock**.

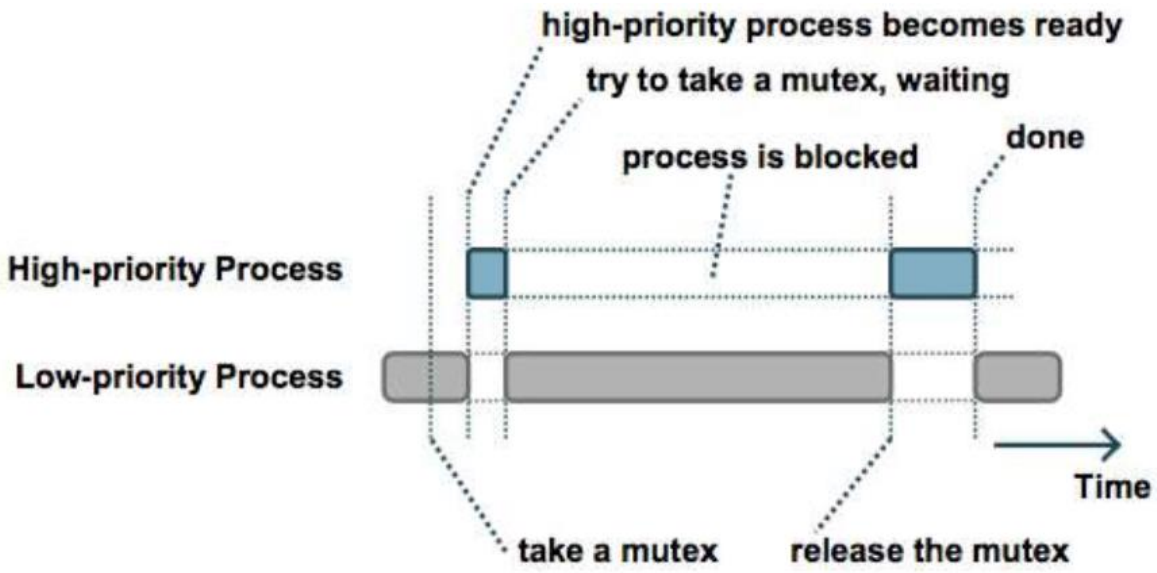
Process 1:
.
take (S1);
take (S2);
Do something
give (S1);
give (S2);
.

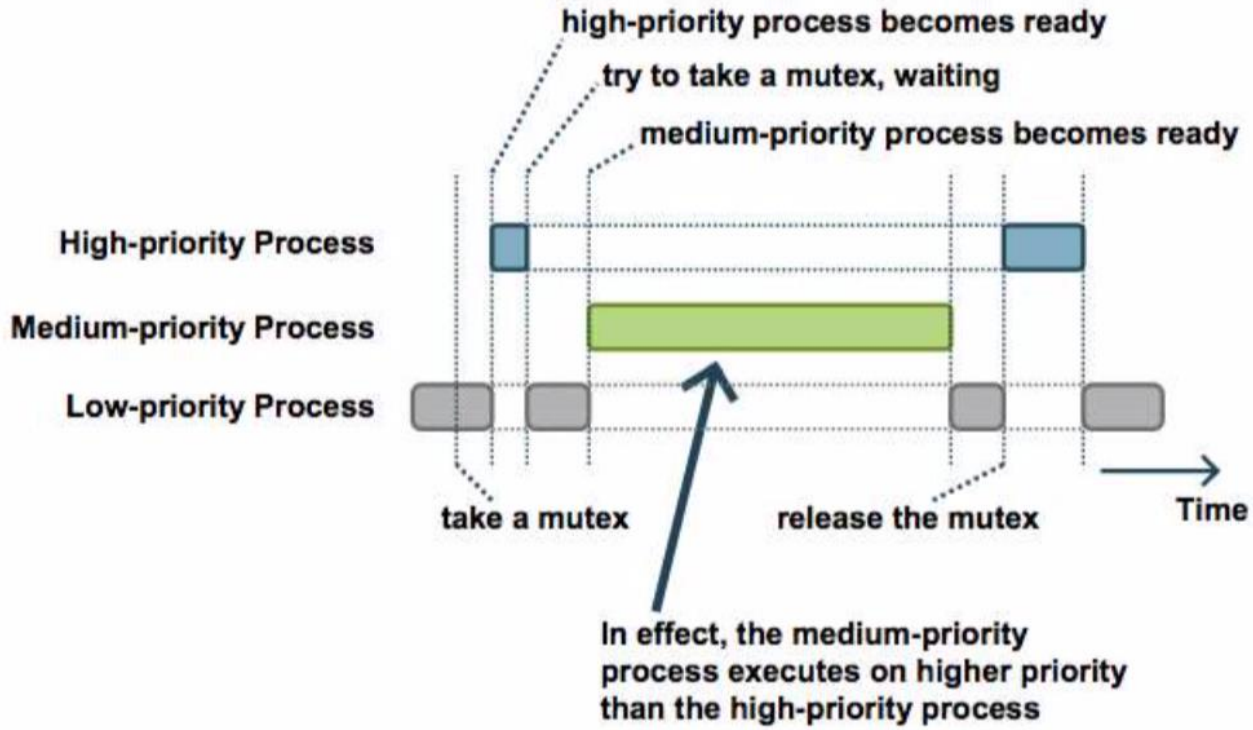
Process 2:
.
take (S2);
take (S1);
Do something
give (S2);
give (S1);
.


INTERVIEWER: EXPLAIN DEADLOCK AND I'LL HIRE YOU
©thecrazyprogrammer
PROGRAMMER: HIRE ME AND I'LL EXPLAIN IT TO YOU

Priority inversion

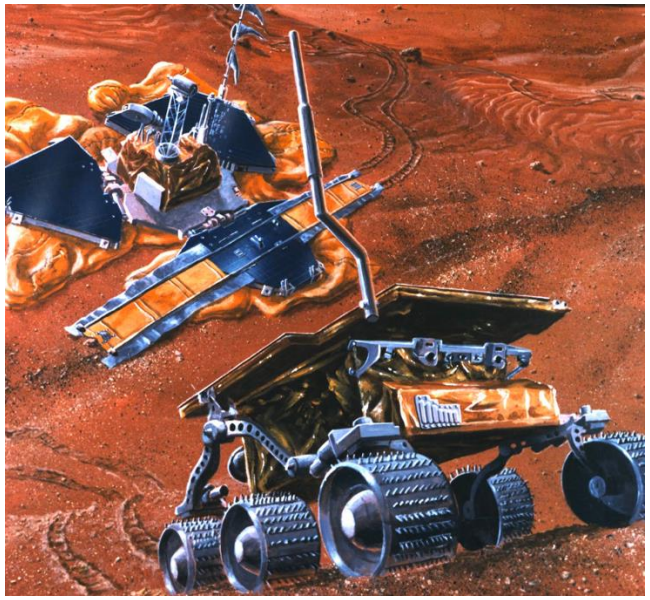
- Using semaphores can affect priority scheduling!





Solutions

- Disable all interrupts.
- **Priority ceiling**: a task's priority is immediately raised when it locks a resource. The priority is set to the maximum priority of the resource, thus no task that may lock the resource is able to get scheduled.
- **Priority inheritance**: increases the priority of a process (A) to the maximum priority of any other process waiting for any resource on which A has a resource lock.
- **Random boosting**: processes holding locks are randomly boosted in priority and allowed to run long enough to exit the critical section.



https://www.rapitasystems.com/blog/what-really-happened-to-the-software-on-the-mars-pathfinder-spacecraft

FreeRTOS

- https://www.freertos.org/tutorial/index.html
- Real-time operating system kernel for embedded devices.
- Ported to 35 microcontroller platforms.
- Distributed under the MIT License.
- Small and simple: the kernel itself consists of only three C files.
- Methods for multiple threads or tasks, mutexes, semaphores and software timers.
- No device drivers, memory management, users accounts, networking

Tasks state machine

- **Blocked State**: A task that is waiting for an event.
  - Temporal (time-related) events.
  - Synchronization events: originate from another task or interrupt.
- It is possible for a task to block on a synchronization event with a timeout.
- **Suspended state**: tasks are not available to the scheduler.
- The only way into the Suspended state is through vTaskSuspend(), the only way out is through vTaskResume() or xTaskResumeFromISR().
- Tasks that are in the Not Running state but are not Blocked or Suspended are said to be in the **Ready state**.

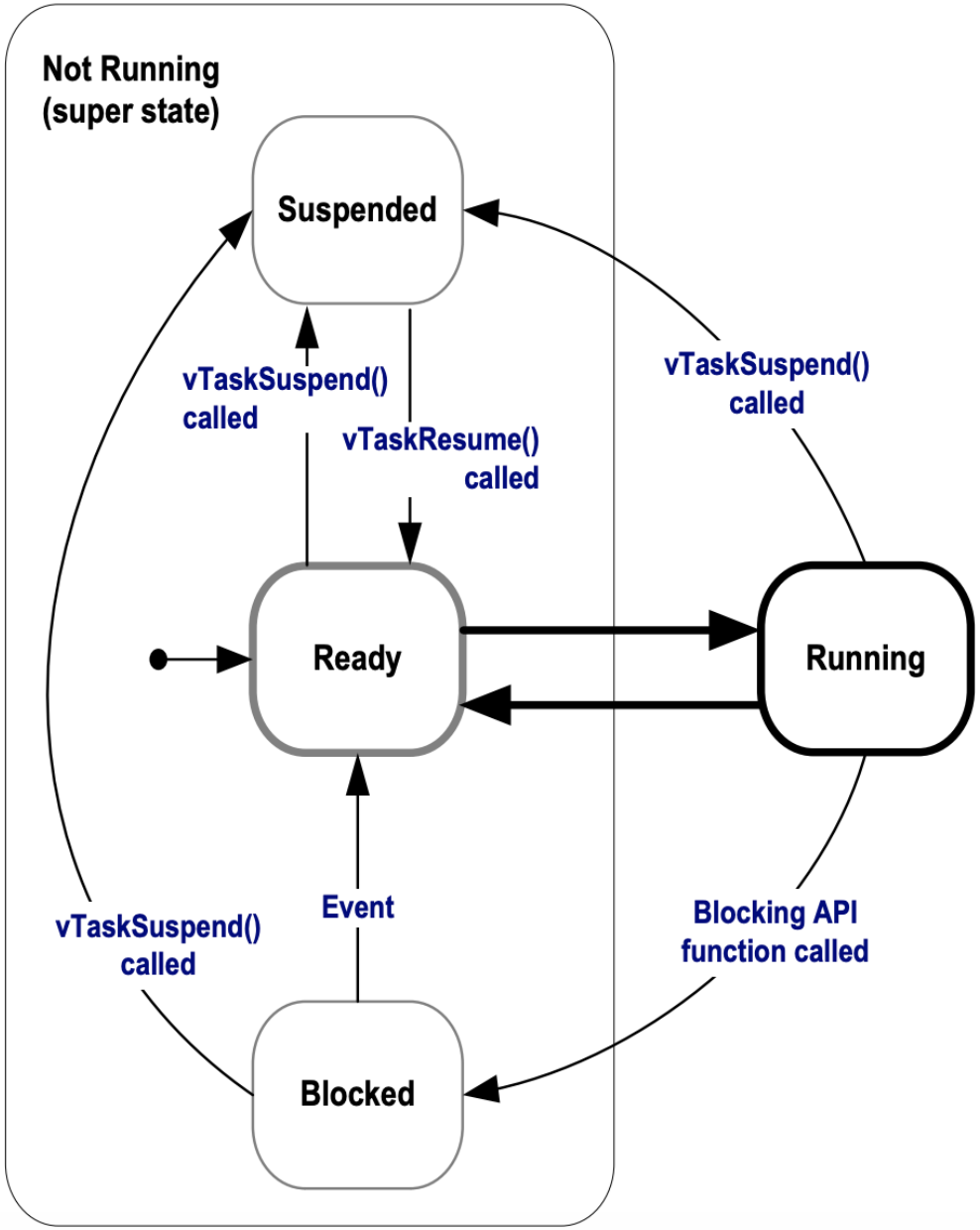


Figure 15. Full task state machine

ESP-IDF and FreeRTOS

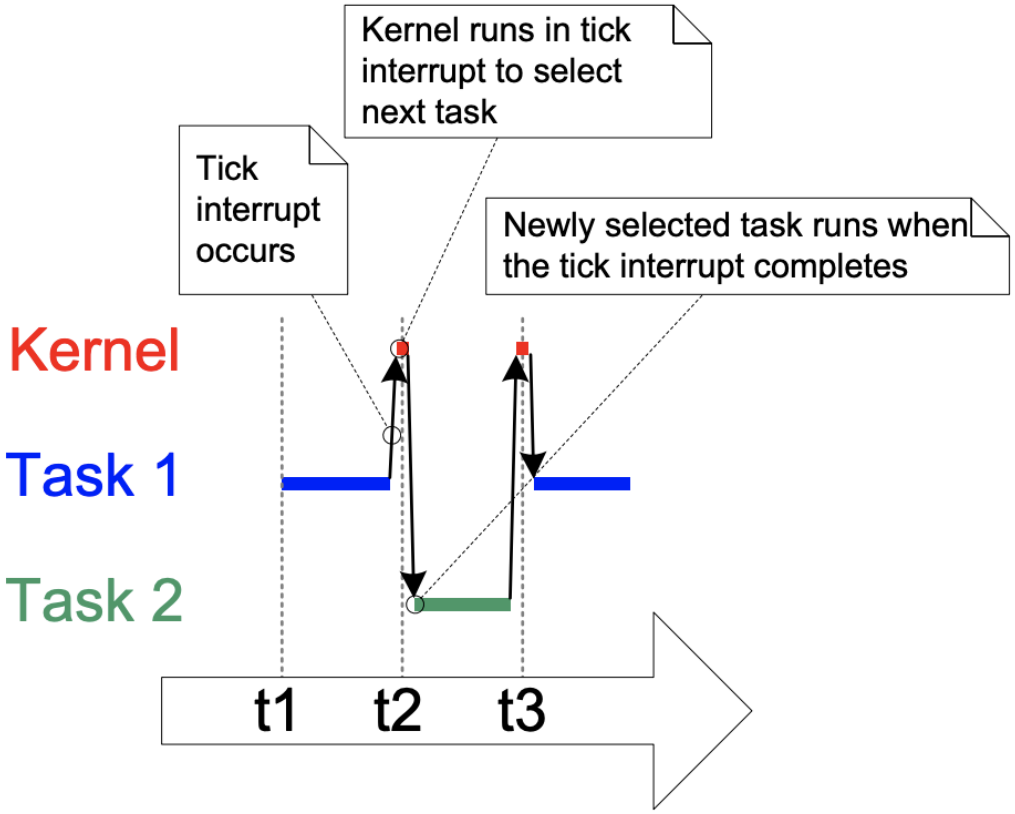
- The ESP-IDF is **based on FreeRTOS!**
  - but It includes some additions and changes.
- https://docs.espressif.com/projects/esp-idf/en/stable/api-reference/system/freertos.html
- Remember that the ESP32 has 2 cores!
  - You can “pin” a task to a specific core or let FreeRTOS decide.
- Important functions:
  - **xTaskCreate**() create a task. There’s a version called xTaskCreateStatic where memory must be explicitly allocated for each task.
  - **xTaskCreatePinnedToCore**() create a task with a given *affinity* (means on which core to execute it).
  - **vTaskDelete**() deletes a task.
  - **vTaskDelay**() delays a task for some time.
  - **vTaskDelayUntil**() delays a task until a given time is reached (stricter than vTaskDelay)
  - **vTaskSuspend**() and **vTaskResume**() suspend and resume tasks.

Live coding

- Example of tasks in ESP32.

System “tick”

- FreeRTOS uses a hardware timer configured to generate an interrupt every 1ms (by default).
- This timer is used for scheduling in a RTOS.
- We refer to this interrupt as the “tick”.
- At each tick, the scheduler decides what task to run.



Priorities

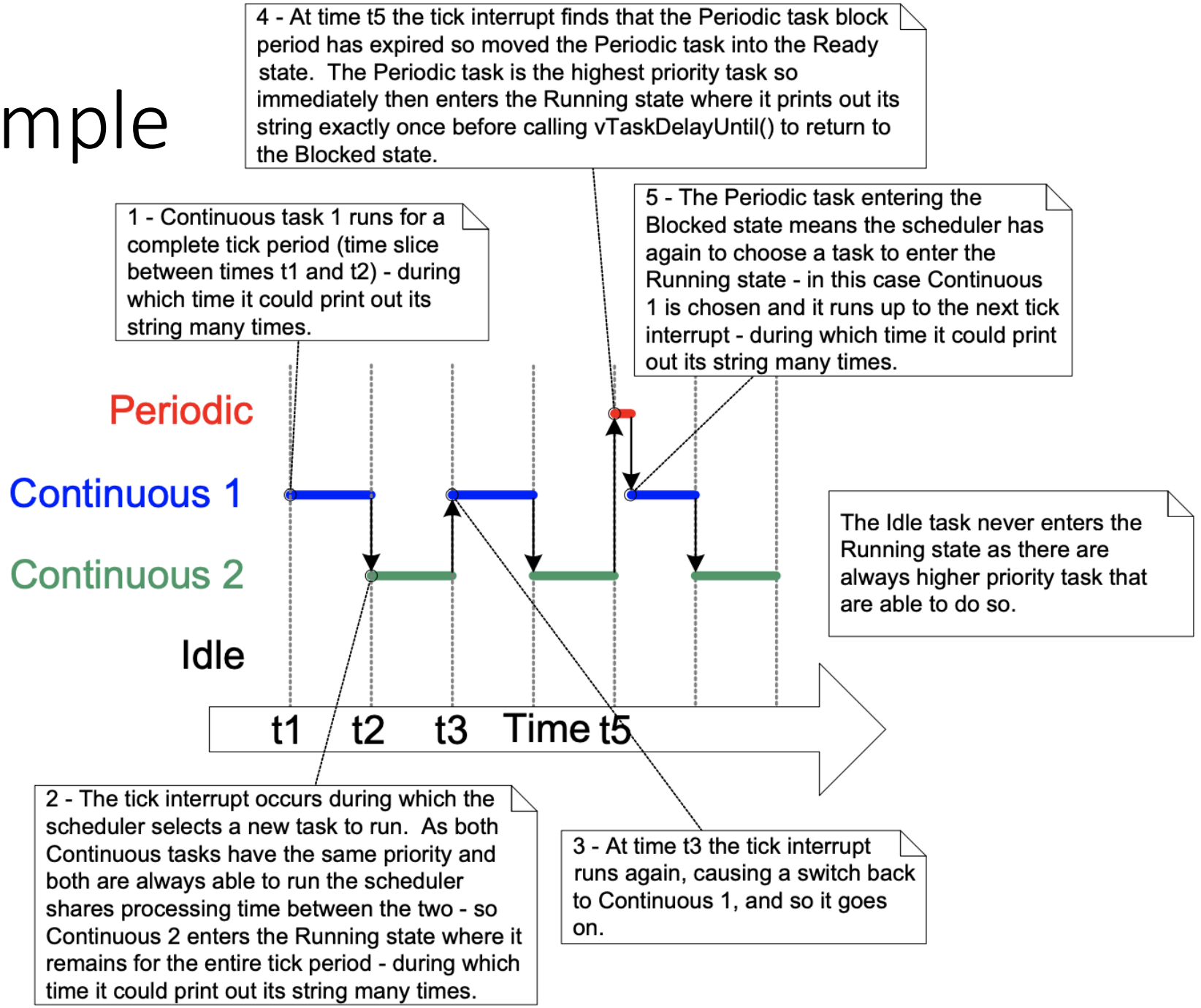
- When tasks are created, the RTOS priority is set.
- Higher figure indicates higher priority.
- Priority 0 is used by the Idle task, so do not use 0 except in special cases, see the manual.
- The highest possible priority is stated in FreeRTOSConfig.h.

Idle task

- There must always be at least one task that can enter the Running state.
- To ensure this is the case, an Idle task is automatically created by the scheduler when it is started.
- The idle task does very little more than sit in a loop.
- The idle task has the lowest possible priority (priority zero), to ensure it never prevents a higher priority application task from entering the Running state.
- It is possible to add application specific functionality directly into the idle task through the use of an **idle hook** (or idle callback) function—a function that is called automatically by the idle task once per iteration of the idle task loop.
- Used for very low priority processing.

Scheduling example

- Continuous1 and Continuous2 have the same priority and are continuously running.
- Periodic is periodic and has high priority.



Periodic tasks

1. Use vTaskDelay()
   - Stops a task for a given amount of ticks.
   - Does not consider execution time of the task nor other tasks being executed in the middle.
2. Use vTaskDelayUntil()
   - Stops a task until a number of ticks are reached.
3. Use software timers
   - A software timer allows a callback function to be executed at a set time in the future.
   - The FreeRTOS implementation does not execute timer callback functions from an interrupt context (no issues with IRAM and all APIs are available!).
   - Callbacks should never block the task (e.g. by calling vTaskDelay).

Live coding

- Example of periodic tasks.

Watchdog

- The interrupt watchdog makes sure the FreeRTOS task switching interrupt isn’t blocked for a long time.
- The **Task Watchdog Timer** (TWDT) is responsible for detecting instances of tasks running for a prolonged period of time without yielding.
  - By default the TWDT will watch the Idle Tasks of each CPU, however any task can elect to be watched by the TWDT.
  - Each watched task must ‘reset’ the TWDT periodically to indicate that they have been allocated CPU time.
  - Use **esp_task_wdt_init**() to initialise the watchdog, **esp_task_wdt_add**() to add a task to the watchdog timer, and **esp_task_wdt_reset**() inside the task to “feed” the dog.

Live coding

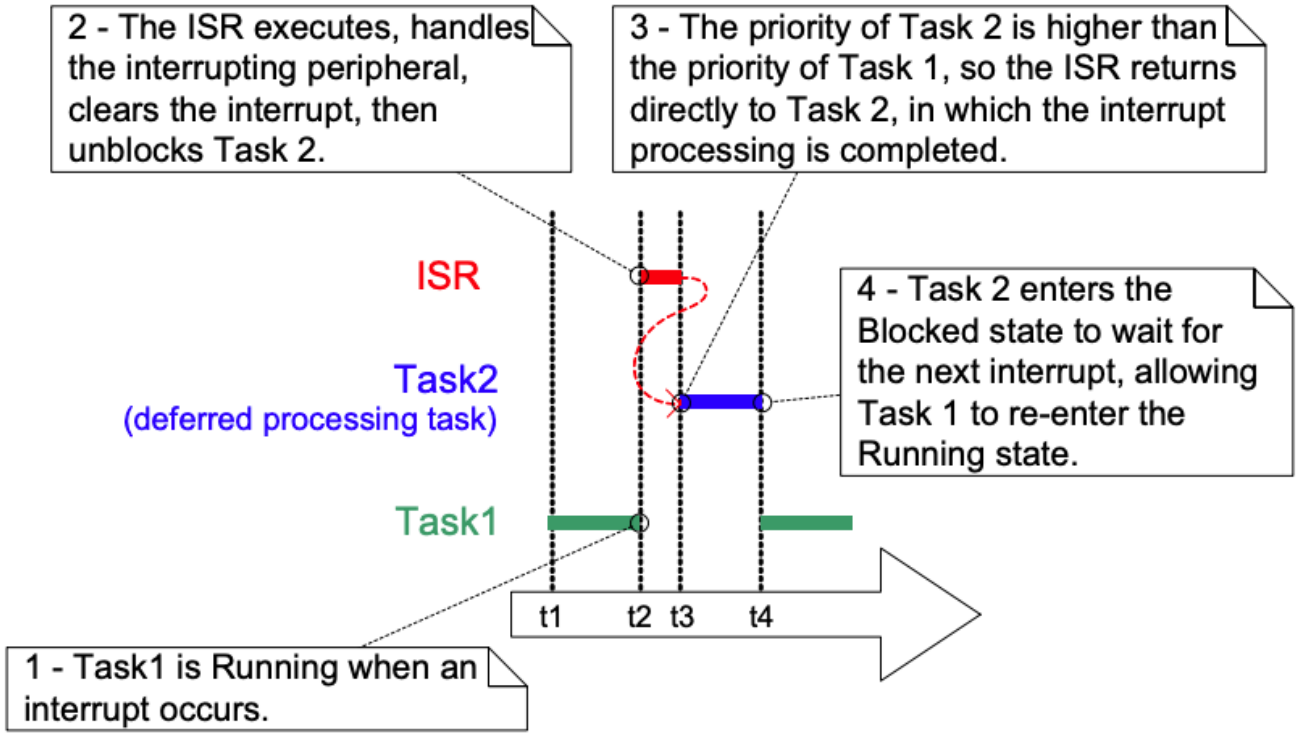
- Example of tasks with watchdog feeding.

Latency

- How to get quick response to external events (small latency)?
- Simple option: short distance between ticks and task with higher priority.
  - Disadvantage of too short time between ticks: the time for the context switch becomes a large proportion of the execution time, i.e. inefficient.
- Better idea: **interrupts**.
  - Interrupts work independently from the tasks!

ISRs in FreeRTOS

- A regular task waiting for an input ”takes” a semaphore.
- An ISR related to that input has as little code as possible to manage the input and "gives" the semaphore to the task.



Interrupts and priorities

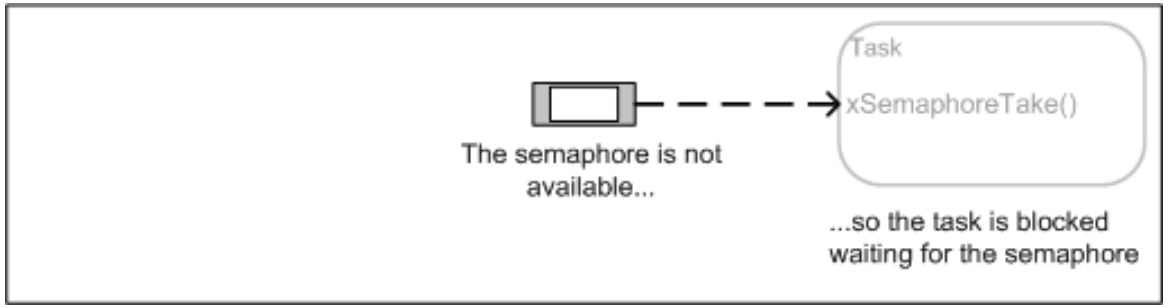
- NOTE: Distinguish carefully between **task priority and interrupt priority**!
- RTOS tick interrupt is usually set for the lowest available interrupt priority.

FreeRTOS Interrupt safe API

- Often it is necessary to use the functionality provided by a FreeRTOS API function from an interrupt service routine (ISR).
- Many FreeRTOS API functions perform actions that are not valid inside an ISR.
  - Example: placing the task that called the API function into the Blocked state: the ISR is not in any task!
- Two versions of some API functions:
  - one version for use from tasks.
  - one version for use from ISRs.
- Functions intended for use from ISRs have "FromISR" appended to their name.
- Note: **Never call a FreeRTOS API function that does not have "FromISR" in its name from an ISR**.

Semaphores in FreeRTOS

- Binary semaphores are used for both mutual exclusion and synchronisation purposes.
- **xSemaphoreCreateBinary**(): creates a binary semaphore.
- **xSemaphoreCreateCounting**(): creates a counting semaphore.
- **xSemaphoreGive**(): “gives” the semaphore (xSemaphoreGiveFromISR if from an ISR).
- **xSemaphoreTake**(): “takes” the semaphore (xSemaphoreTakeFromISR if from an ISR).
- **vSemaphoreDelete**(): deletes the semaphore.

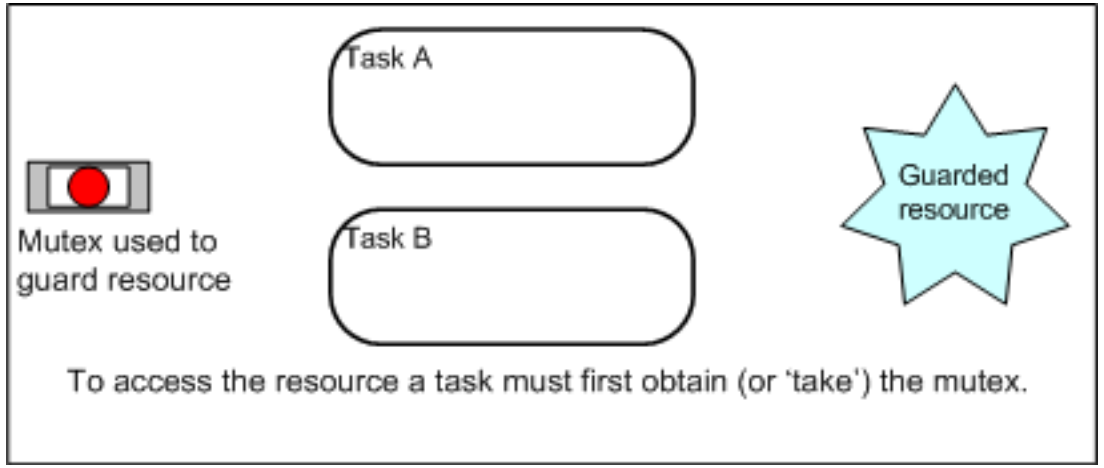


Live coding

- Example of ISR – task synchronisation using a semaphore.

Mutexes

- Mutexes are binary semaphores that include a priority inheritance mechanism.
  - Better choice for implementing simple mutual exclusion
- xSemaphoreCreateMutex(): creates a mutex
- Take and Give are the same as semaphores.
- Recursive mutexes: a mutex used recursively can be ‘taken’ repeatedly by the owner.

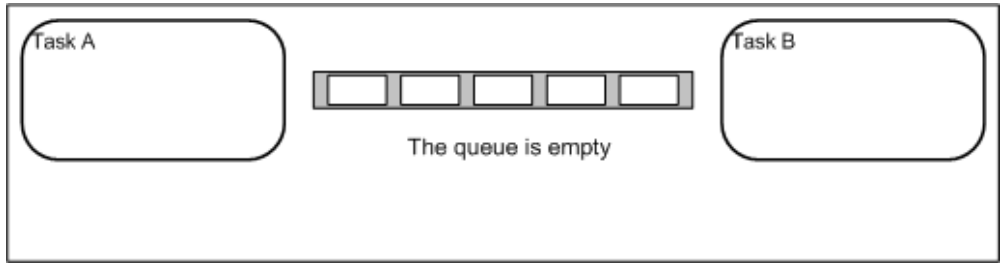


Tasks notifications

- RTOS task has a 32-bit notification value.
- An RTOS **task notification** is an event sent directly to a task that can unblock the receiving task, and optionally update the receiving task's notification value.
- Basic primitives for stopping / resuming a task on notifications:
  - xTaskNotifyWait(): puts a task on hold
  - xTaskNotify(): (or xTaskNotifyFromISR if called from an ISR) notifies and unblocks a task
- **Faster than the other methods**!

Queues

- Queues can be used to send messages between tasks, and between interrupts and tasks.
- In most cases they are used as **thread safe FIFO** (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.
- xQueueCreate(): creates a new queue.
- xQueueSend(), xQueueSendToFront() and xQueueSendToBack(): send an item to the queue.
- xQueuePeek(): receive an item from a queue without removing it.
- xQueueReceive(): receives an item from a queue.



Example exam questions

- Explain the difference between a general purpose OS (like Windows) and a Real-Time Operating system (like FreeRTOS).
- What is the difference between a hard real time and a soft real time constraints? Provide examples of use cases.
- In what sense is a context switch between executing tasks costly?
- Describe the difference between cooperative, time-slice and pre-emptive scheduling approaches.
- Provide 6 examples of hard and soft deadlines to be met by an embedded system.
- Define the Processor Utilization Factor, U. What condition(s) must be met about U to ensure that tasks will be schedulable?
- What is the formula of the Processor Utilization Factor when we consider context switching into it?
- I have 1 task that needs to be scheduled every 2 seconds and takes 2 seconds to be executed and another task that needs to be executed every 3 seconds and takes 2 seconds to be executed. Would these tasks be schedulable with an RMS approach?
- A task is developed with 200 lines of code. Given an approximate Worst Case Execution Time estimation for that tasks supposing a 80MhZ clock.
- What's priority inversion and how can it be fixed?
- Describe the FreeRTOS task state machine.
- Name and briefly 3 examples of task synchronisation primitives available in FreeRTOS.
- You are developing a little computer for a bike. The device includes a sensor that detects rotations of the wheel. At each rotation an interrupt is raised on a pin. When a user presses a button (which can trigger another interrupt), the device shows current distance (based on rotations) for 1 minute and then the screen is turned off to save battery. How would you structure the code to manage this device into FreeRTOS tasks? What would each task do? How do synchronise tasks and ISRs?