Embedded Operating System

Amir Mahdi Hosseini Monazzah

Room 332,
School of Computer Engineering,
Iran University of Science and Technology,
Tehran, Iran.

monazzah@iust.ac.ir

Fall 2024

Outline

- Embedded (real-time) operating systems
  - Introduction
  - Definition
  - Features and architectures
- Classes of operating systems
  - Class 1: Fast proprietary kernels
  - Class 2: Extensions to standard OSes
  - Class 3: Research systems
- Case study
  - FreeRTOS

Towards embedded operating system

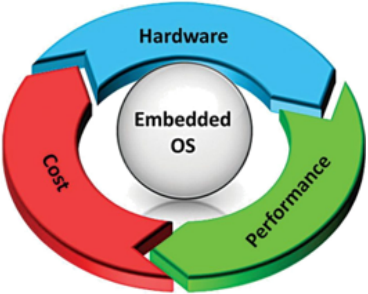
- Why do we need an embedded operating system (OS)?
  - Same reasons why we need one for a traditional computer
  - Not every devices needs all services
- In embedded systems we find a large variety of requirements and environments
  - Critical applications with high functionality (medical applications, space shuttle, process automation, . . .)
  - Critical applications with small functionality (ABS, pace maker, . . .)
  - Not very critical applications with broad range of functionality (smart phone, . . .)

Towards embedded operating system

- Why is a desktop OS not suited?
  - The monolithic kernel of a desktop OS offers too many features that take space in memory and consume time
  - Monolithic kernels are often not modular, fault-tolerant, configurable
  - Requires too much memory space and is often too resource hungry in terms of computation time
  - Not designed for mission-critical applications
  - The timing uncertainty may be too large for some applications

Towards embedded operating system

- Essential characteristics of an embedded OS: Configurability
  - No single operating system will fit all needs, but often no overhead for unused functions/data is tolerated. Therefore, configurability is needed!
    - For example, there are many embedded systems without external memory, a keyboard, a screen or a mouse

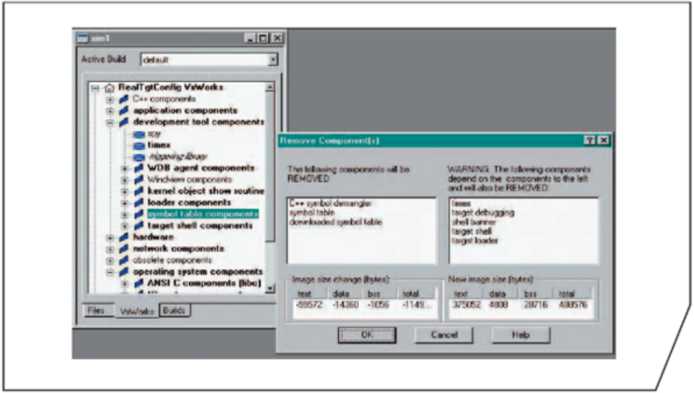


Towards embedded operating system

- Configurability examples:
  - Remove unused functions/libraries (for example by the linker)
  - Use conditional compilation (using #if and #ifdef commands in C, for example)
  - But deriving a consistent configuration is a potential problem of systems with a large number of derived operating systems
    - There is the danger of missing relevant components

Towards embedded operating system

- Example: Configuration of VxWorks
  - A real-time OS by Wind River Systems



http://www.windriver.com/products/development_tools/ide/tornado2/tornado_2_ds.pdf

Automatic dependency analysis and size calculations allow users to quickly custom-tailor the VxWORKS operating system.

© Windriver

What is real-time operating system?

- A real-time operating system is an operating system that supports the construction of real-time systems
- Key requirements
  - The timing behavior of the OS must be predictable
    - For all services of the OS, an upper bound on the execution time is necessary
      - For example, for every service upper bounds on blocking times need to be available, i.e. for times during which interrupts are disabled
      - Moreover, almost all processor activities should be controlled by a real-time scheduler

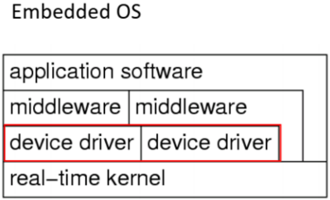
What is real-time operating system?

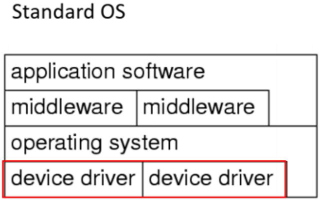
- A real-time operating system is an operating system that supports the construction of real-time systems
- Key requirements
  - The timing behavior of the OS must be predictable
    - OS has to be aware of deadlines and should have mechanism to take them into account in the scheduling
    - OS must provide precise time services with a high resolution

How do the things manage in embedded operating system?

- Device drivers are typically handled directly by tasks
  - Instead of drivers that are managed by the operating system
- This architecture improves timing predictability as access to devices is also handled by the scheduler
  - If several tasks use the same external device and the associated driver, then the access must be carefully managed (shared critical resource, ensure fairness of access)

How do the things manage in embedded operating system?





How do the things manage in embedded operating system?

- Every task can perform an interrupt
  - For standard OS, this would be serious source of unreliability
    - But embedded programs are typically programmed in a controlled environment
  - It is possible to let interrupts directly start or stop tasks (by storing the tasks start address in the interrupt table)
    - This approach is more efficient and predictable than going through the operating system's interfaces and services

How do the things manage in embedded operating system?

- Protection mechanisms are not always necessary in embedded operating systems
  - Embedded systems are typically designed for a single purpose, untested programs are rarely loaded
    - Software can be considered to be reliable
  - However, protection mechanisms may be needed for safety and security reasons

RTOS-kernels main functionality: task management

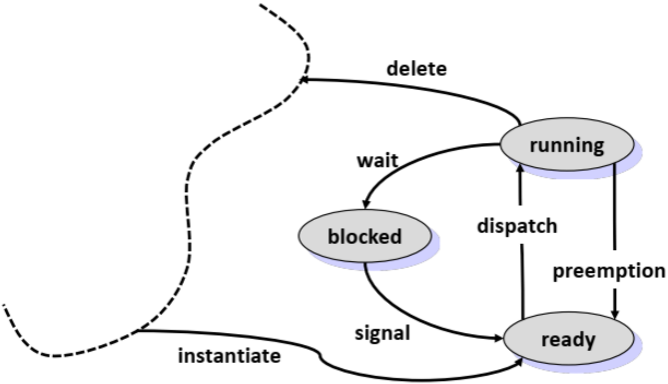
- Execution of quasi-parallel tasks on a processor using processes or threads (lightweight process) by maintaining
  - Process states
  - Process queuing allowing for preemptive tasks (fast context switching) and quick interrupt handling
- CPU scheduling (guaranteeing deadlines, minimizing process waiting times, fairness in granting resources such as computing power)
- Inter-task communication (buffering)

RTOS-kernels main functionality: task management

- Support of real-time clocks
- Task synchronization (critical sections, semaphores, monitors, mutual exclusion)
  - In classical operating systems, synchronization and mutual exclusion is performed via semaphores and monitors
  - In real-time OS, special semaphores and a deep integration of them into scheduling is necessary (for example priority inheritance protocols)

Task states (from here: session 26)

- Minimal set of task states:



Task states (up to here: session 25)

- Running:
  - A task enters this state when it starts executing on the processor. There is as most one task with this state in the system
- Ready:
  - State of those tasks that are ready to execute but cannot be run because the processor is assigned to another task, i.e. another task has the state "running"

Blocked:

- Blocked:
  - A task enters the blocked state when it executes a synchronization primitive to wait for an event, e.g. a wait primitive on a semaphore or timer. In this case, the task is inserted in a queue associated with this semaphore. The task at the head is resumed when the semaphore is unlocked by an event

Threads

- A thread is the smallest sequence of programmed instructions that can be managed independently by a scheduler
  - E.g., a thread is a basic unit of CPU utilization
- Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources
  - Typically shared by threads: Memory
  - Typically owned by threads: Registers and stack

Threads

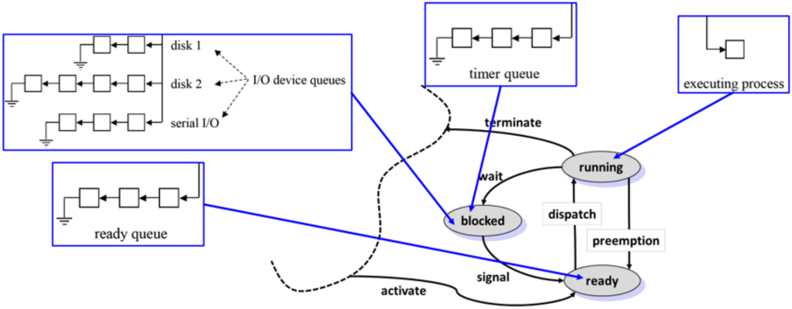
- Thread advantages and characteristics
  - Faster to switch between threads; switching between user-level threads requires no major intervention by the operating system
  - Typically, an application will have a separate thread for each distinct activity
  - Thread Control Block (TCB) stores information needed to manage and schedule a threads

Threads

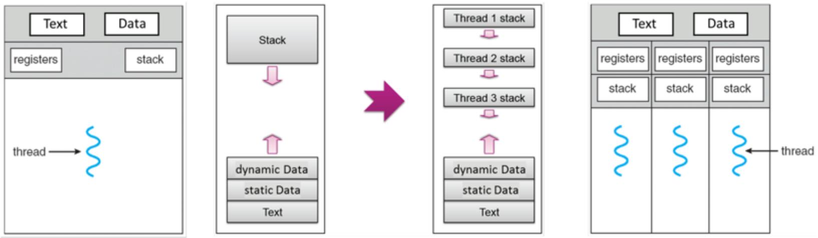
- The operating system maintains for each thread
  - A data structure (TCB – Thread Control Block)
    - Contains its current status such as
      - Program counter
      - Priority
      - State
      - Scheduling information
      - Thread name

Threads

- The TCBs are administered in linked lists



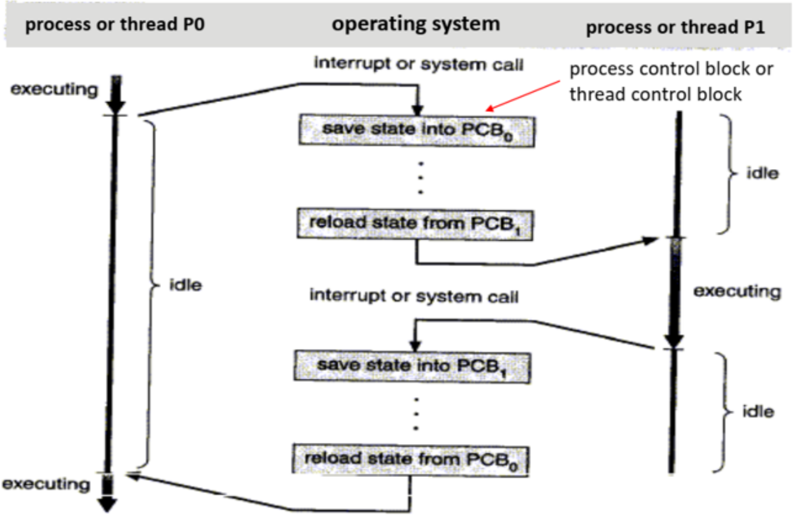
Multiple threads within a process



process with a single thread

process with several threads

Context switch: processes or threads

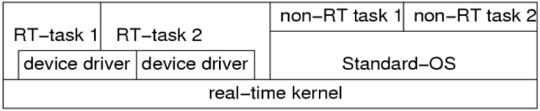


When performance is of decisive importance!

- Fast proprietary kernels
  - For hard real-time systems, these kernels are questionable, because they are designed to be fast, rather than to be predictable in every respect
- Examples include
  - FreeRTOS, QNX, eCOS, RT-LINUX, VxWORKS, LynxOS

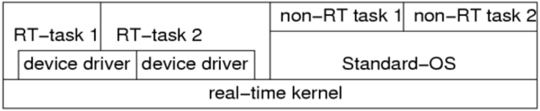
Real-time extensions to standard OS

- Attempt to exploit existing and comfortable main stream operating systems
- A real-time kernel runs all real-time tasks
- The standard-OS is executed as one task



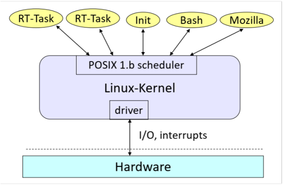
Real-time extensions to standard OS

- Pros
  - Crash of standard-OS does not affect RT-tasks
- Cons
  - RT-tasks cannot use standard-OS services
    - less comfortable than expected
- Revival of the concept: Hypervisor



Example: Posix 1.b RT-extensions to Linux

- The standard scheduler of a general purpose operating system can be replaced by a scheduler that exhibits (soft) real-time properties

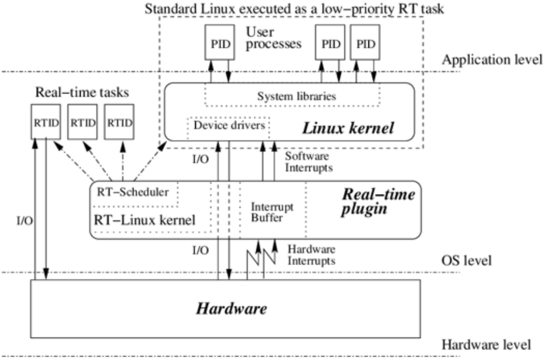


Special calls for real-time as well as standard operating system calls available.

Simplifies programming, but no guarantees for meeting deadlines are provided

Example: RT Linux

- RT-tasks cannot use standard OS calls
  - Commercially available from fsmlabs and WindRiver (www.fsmlabs.com)



What is going on in research fields?

- Research systems try to avoid limitations of existing real-time and embedded operating systems
  - Examples include L4, seL4, NICTA, ERIKA, SHARK
- Typical research questions:
  - Low overhead memory protection
  - Temporal protection of computing resources
  - RTOS for on-chip multiprocessors
  - Quality of service (QoS) control (besides real-time constraints)
  - Formally verified kernel properties

What is FreeRTOS (from here: session 27)

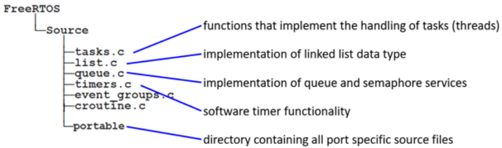
- FreeRTOS (http://www.freertos.org/) is a typical embedded operating system
  - It is available for many hardware platforms, open source and widely used in industry
- FreeRTOS is a real-time kernel (or real-time scheduler)
- Applications are organized as a collection of independent threads of execution

What is FreeRTOS

- Characteristics:
  - Pre-emptive or co-operative operation
  - Queues
  - Binary semaphores
  - Counting semaphores
  - Mutexes (mutual exclusion)
  - Software timers
  - Stack overflow checking
  - Trace recording
  - Etc.

How does FreeRTOS works

- Typical directory structure (excerpts)



- FreeRTOS is configured by a header file called FreeRTOSConfig.h that determines almost all configurations (co-operative scheduling vs. preemptive, time-slicing, heap size, mutex, semaphores, priority levels, timers, . . .)

Task management

- Tasks are implemented as threads
- The functionality of a thread is implemented in form of a function
  - Prototype:

```
void ATaskFunction( void *pvParameters );
```

some name of task function pointer to task arguments

  - Task functions are not allowed to return! They can be “killed” by a specific call to a FreeRTOS function, but usually run forever in an infinite loop
  - Task functions can instantiate other tasks
    - Each created task is a separate execution instance, with its own stack

Task management

- Example

```
void vTask1( void *pvParameters ) {
   volatile uint32_t ul; /* volatile to ensure ul is implemented. */
   for( ;; ) {
     ... /* do something repeatedly */
     for( ul = 0; ul < 10000; ul++ ) {  /* delay by busy loop */ }
   }
 }
```

Task management

- Thread instantiation

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

a pointer to the function that implements the task

a descriptive name for the task

each task has its own unique stack that is allocated by the kernel to the task when the task is created; the usStackDepth value determines the size of the stack (in words)

returns pdPASS or pdFAIL depending on the success of the thread creation

the priority at which the task will execute; priority 0 is the lowest priority

pxCreatedTask can be used to pass out a handle to the task being created.

task functions accept a parameter of type pointer to void; the value assigned to pvParameters is the value passed into the task.

Task management

- Examples for changing properties of tasks
  - Changing the priority of a task. In case of preemptive scheduling policy, the ready task with the highest priority is automatically assigned to the "running" state

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

handle of the task whose priority is being modified

new priority (0 is lowest priority)

  - A task can delete itself or any other task. Deleted tasks no longer exist and cannot enter the "running" state again

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

handle of the task who will be deleted; if NULL, then the caller will be deleted

Timers (up to here: session 26)

- The operating system also provides interfaces to timers of the processor
- As an example, we use the FreeRTOS timer interface to replace the busy loop by a delay. In this case, the task is put into the “blocked” state instead of continuously running

```
void vTaskDelay( TickType_t xTicksToDelay );
```

time is measured in “tick” units that are defined in the configuration of FreeRTOS (`FreeRTOSConfig.h`). The function `pdMS_TO_TICKS()` converts ms to “ticks”.

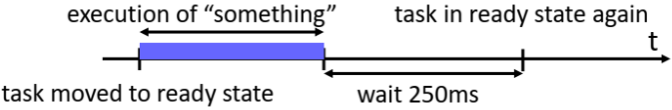
Timers

- Example

```
void vTask1( void *pvParameters ) {
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelay(pdMS_TO_TICKS(250));  /* delay by 250 ms */
   }
 }
```

Timers

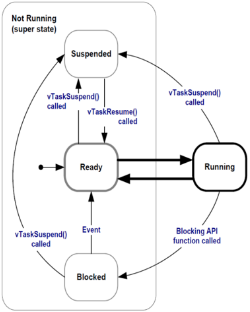
- Problem: The task does not execute strictly periodically



- The parameters to vTaskDelayUntil() specify the exact tick count value at which the calling task should be moved from the "blocked" state into the "ready" state
  - Therefore, the task is put into the "ready" state periodically.

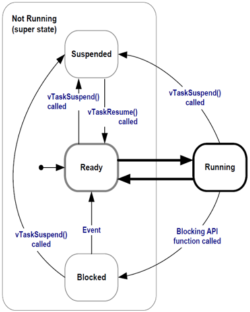
Task states

- What are the task states in FreeRTOS and the corresponding transitions?
- A task that is waiting for an event is said to be in the "Blocked" state, which is a sub-state of the "Not Running" state



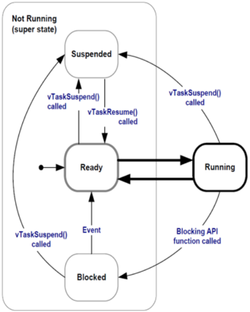
Task states

- Tasks can enter the "Blocked" state to wait for two different types of event
  - Temporal (time-related) events
    - The event being either a delay period expiring, or an absolute time being reached



Task states

- Tasks can enter the "Blocked" state to wait for two different types of event
  - Synchronization events
    - Where the events originate from another task or interrupt
      - For example, queues, semaphores, and mutexes, can be used to create synchronization events



Task states

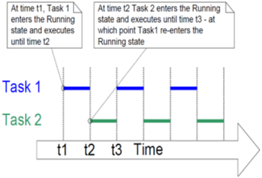
- Example 1: Two threads with equal priority

```
void vTask1( void *pvParameters ) {
   volatile uint32_t ul;
   for( ;; ) {
     ... /* do something repeatedly */
     for( ul = 0; ul < 10000; ul++ ) { }
   }
}
```

```
void vTask2( void *pvParameters ) {
   volatile uint32_t u2;
   for( ;; ) {
     ... /* do something repeatedly */
     for( u2 = 0; u2 < 10000; u2++ ) { }
   }
}
```

```
int main( void ) {
   xTaskCreate(vTask1, "Task 1", 1000, NULL, 1, NULL);
   xTaskCreate(vTask2, "Task 2", 1000, NULL, 1, NULL);
   vTaskStartScheduler();
   for( ;; );
}
```

Both tasks have priority 1. In this case, FreeRTOS uses time slicing, i.e., every task is put into “running” state in turn.



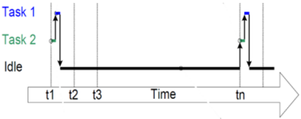
Task states

- Example 2: Two threads with delay timer

```
void vTask1( void *pvParameters ) {
   TickType_t xLastWakeTime = xTaskGetTickCount();
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelayUntil(&xLastWakeTime,pdMS_TO_TICKS(250));
   }
}
```

```
int main( void ) {
  xTaskCreate(vTask1,"Task 1",1000,NULL,1,NULL);
  xTaskCreate(vTask2,"Task 2",1000,NULL,2,NULL);
  vTaskStartScheduler();
  for( ;; );
}
```

```
void vTask2( void *pvParameters ) {
   TickType_t xLastWakeTime = xTaskGetTickCount();
   for( ;; ) {
     ... /* do something repeatedly */
     vTaskDelayUntil(&xLastWakeTime,pdMS_TO_TICKS(250));
   }
}
```



If no user-defined task is in the running state, FreeRTOS chooses a built-in Idle task with priority 0. One can associate a function to this task, e.g., in order to go to low power processor state.

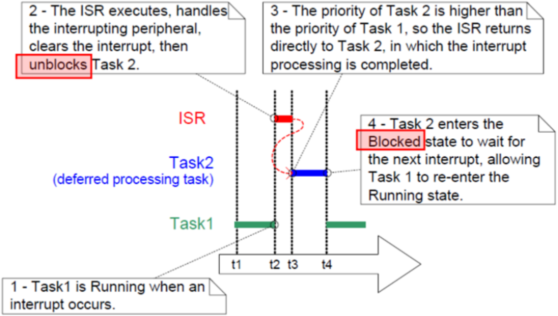
Interrupts (from here: session 28)

- How are tasks (threads) and hardware interrupts scheduled jointly?
  - Although written in software, an interrupt service routine (ISR) is a hardware feature
    - Because the hardware controls which interrupt service routine will run, and when it will run
  - Tasks will only run when there are no ISRs running
    - So the lowest priority interrupt will interrupt the highest priority task,
      - There is no way for a task to preempt an ISR
      - In other words, ISRs have always a higher priority than any other task

Interrupts (up to here: session 27)

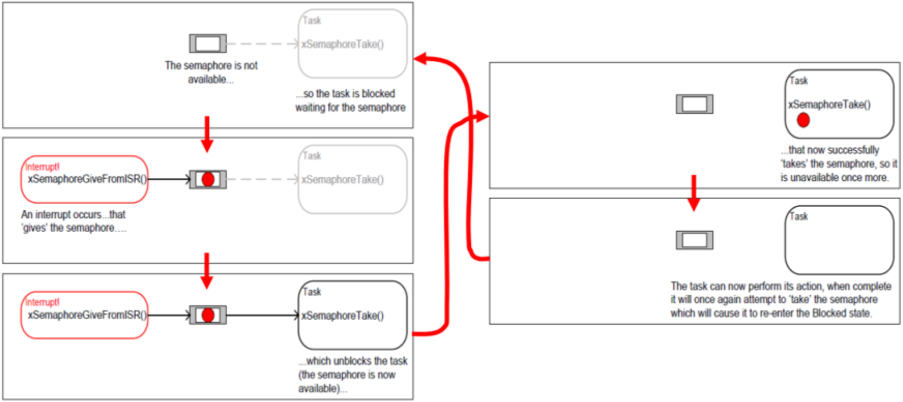
- Usual pattern: ISRs are usually very short
  - They find out the reason for the interrupt, clear the interrupt flag and determine what to do in order to handle the interrupt
  - Then, they unblock a regular task (thread) that performs the necessary processing related to the interrupt
    - For blocking and unblocking, usually semaphores are used

Interrupts



blocking and unblocking is typically implemented via semaphores

Interrupts


Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...
Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.
Task
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

The End