Bring ideas to life
VIA University College

FreeRTOS

Semaphore, Mutex

RTOS Concepts

Concurrent and/or Parallel execution
- Dependent on the number or processors
- The ATMEGA 2560 (your Arduino board MCU) does not support parallel execution (has only one processor)

Processes/Programs, Threads/Tasks and Co-routines
- Dependent on the OS
- FreeRTOS only supports one process but both Tasks (threads) and Co-routines
- Static or Dynamic task creation
- FreeRTOS has both static and dynamic task creation

RTOS Concepts

Static or Dynamic task priority

- FreeRTOS supports dynamic task priorities

Flat or nested task levels

- FreeRTOS is flat (tasks can create other tasks but they are all declared and visible at the top level and there is no dependence between them)

Co-routines

- Like functions
- Not pre-emptive

Task Summary

- Simple.
- No restrictions on use.
- Supports full preemption.
- Fully prioritised.
- Each task maintains its own stack resulting in higher RAM usage.
- Re-entrancy must be carefully considered if using preemption.

Co-Routine Summary

- Sharing a stack between co-routines results in much lower RAM usage.
- Cooperative operation makes re-entrancy less of an issue.
- Very portable across architectures.
- Fully prioritised relative to other co-routines, but can always be preempted by tasks if the two are mixed.
- Lack of stack requires special consideration.
- Restrictions on where API calls can be made.
- Co-operative operation only amongst co-routines themselves.

Why is Synchronization and protection of resources necessary?

In order to ensure efficiency the kernel will preform context switches while low priority tasks are running and higher priority task is ready to run

- Stopping and saving one task and letting another task run

In most systems tasks can be pre-empted

- Force task stopped by the kernel to allow another thread to run

Functions and pre-emptive kernels

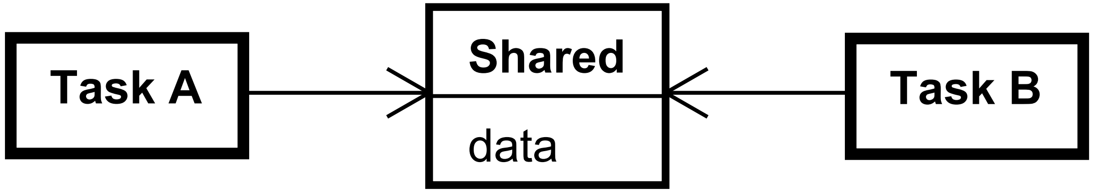
Thus it would be nice if the application only contain **re-entrant** functions

- Functions that can be stopped at an arbitrary point and resumed later whiteout fear of malfunctioning or corruption of data if another task has called the same function in the mean time

But most often this is not the case

- Data or peripherals need to be shared between tasks

Shared Resources



Resources

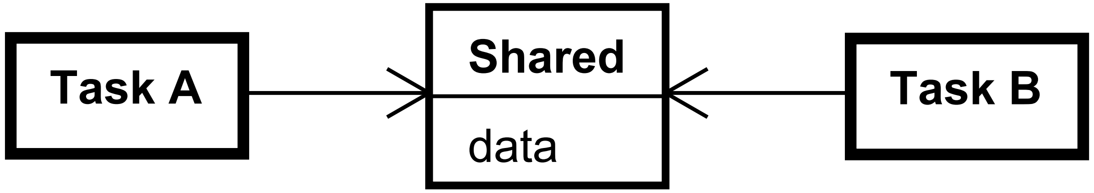
- Shared data
- Shared functions
- Shared hardware interfaces/drivers
  - The classic example is two task sending to a printer and get their messages mix up

How-to Protect Resources
Disabling Interrupts

Task will not be pre-empted – no context switching

No external interrupts can be handled

- and will sometime be lost



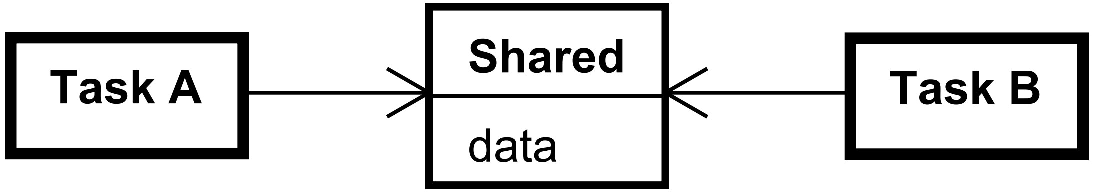
Be careful

- It can be OK for very small pieces of code
- Efficient compared to other protection means
- It is normal/necessary to do in drivers and in OSs

How-to Protect Resources

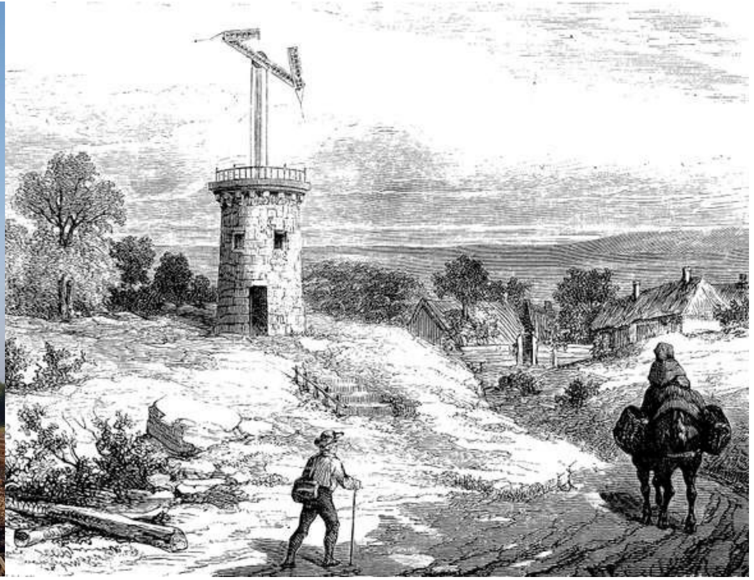
Disabling Interrupts – Be careful

```c
void TaskA( void * pvParameters )
{
    int tmp;
    for( ;; )
    {
        ....

        /* Call taskENTER_CRITICAL() to create a critical section. */
        taskENTER_CRITICAL();
        /* Execute the code that requires the critical section here. */
        tmp = shared.data;
        taskEXIT_CRITICAL();
        ....
        /* Use tmp here */
        ....
    }
}
```



Task B must do the same when accessing `shared.data`

How-to Protect Resources *Semaphores*

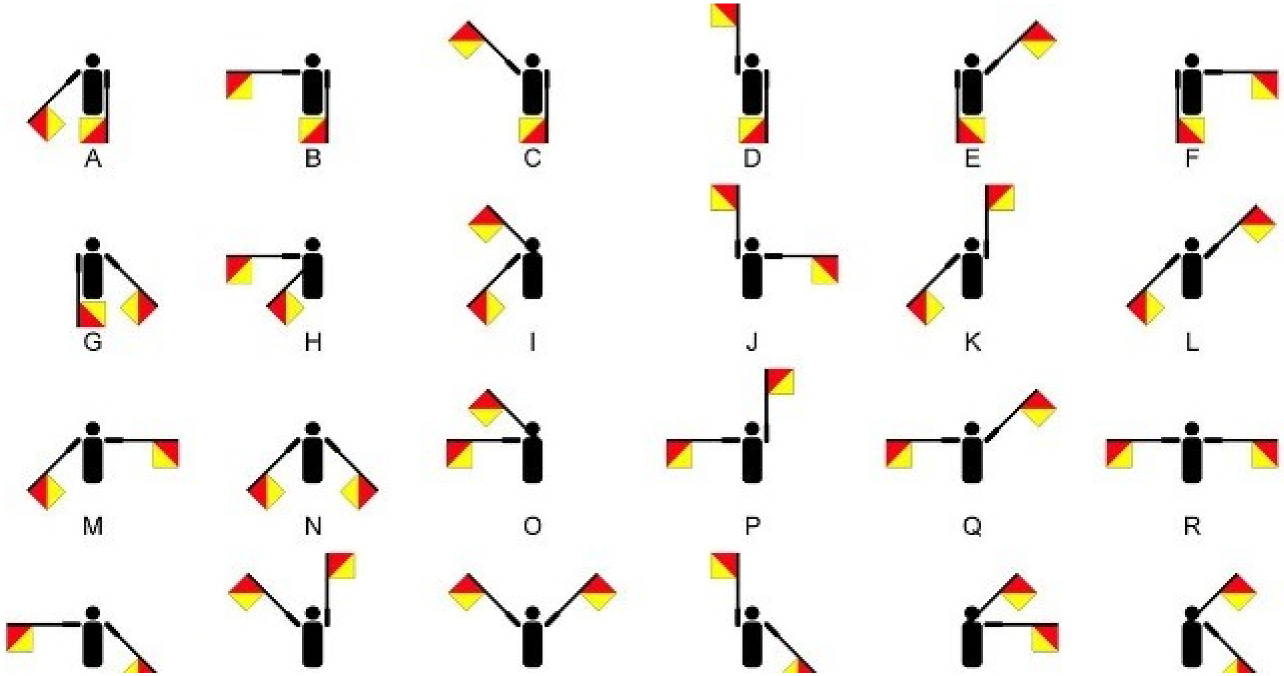


Used in centuries

Is a signal/sign

In OS

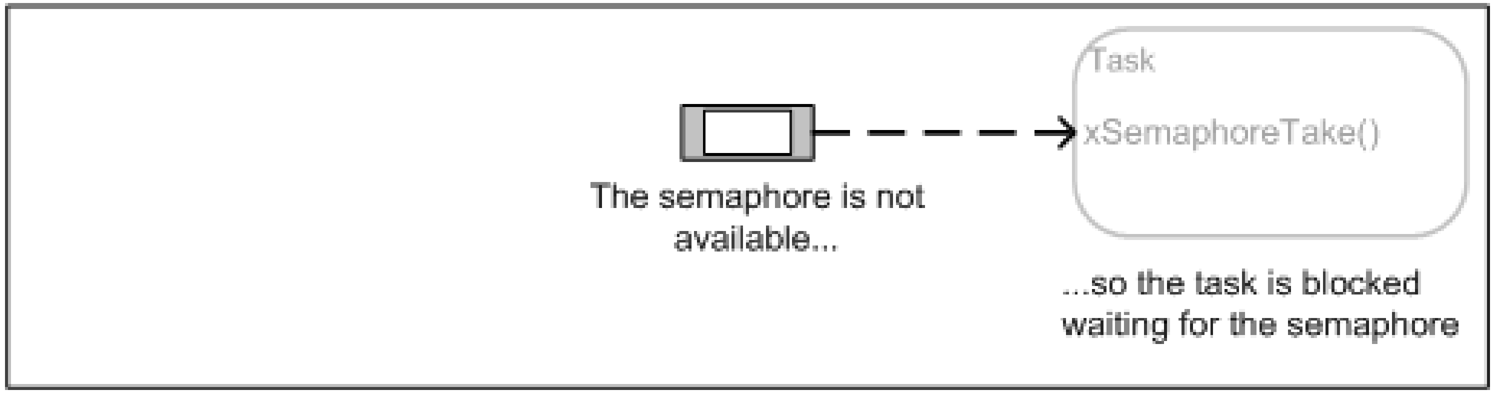
- Binary semaphores
- Counting semaphores



Binary Semaphores

A binary semaphore has two states
- Taken
- Given

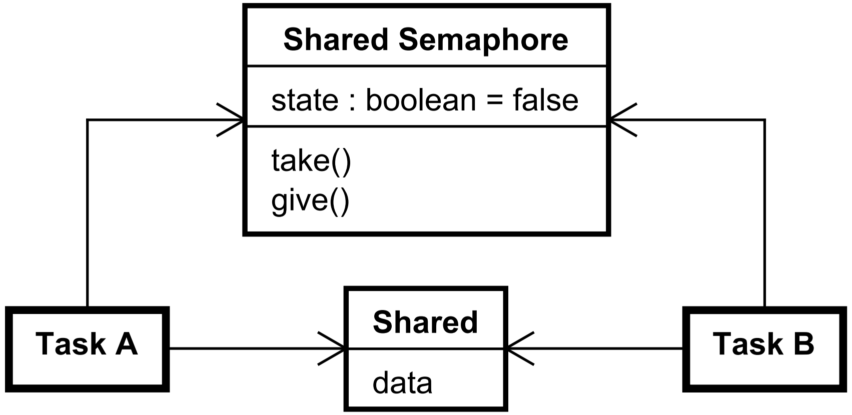
Used for
- Signalling between tasks
- Synchronisation of tasks
- Protecting of resources



Read more here: https://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html

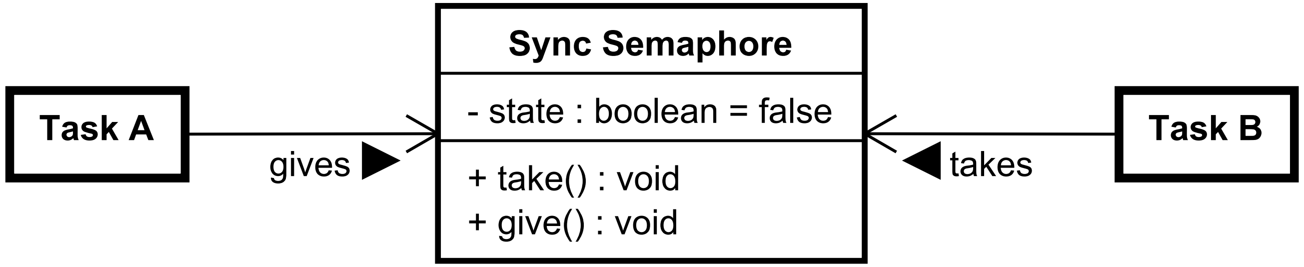
Binary Semaphores
– here Protection

```
SemaphoreHandle_t  sharedSemaphore = xSemaphoreCreateBinary();
xSemaphoreGive(sharedSemaphore);

void TaskA( void * pvParameters )
{
    int tmp;

    for( ;; )
    {
      ....
      if( xSemaphoreTake( sharedSemaphore, pdMS_TO_TICKS(200) ) == pdTRUE ) // Wait maximum 200 ms
      {
         /* Execute the code that uses the shared data here. */
         tmp = shared.data;
         xSemaphoreGive( sharedSemaphore );
      }
      else
      {
         /* We timed out and could not obtain the semaphore and can
         therefore not access the shared resource safely. */
      }
      ....
    }
}
```



Binary Semaphores
– here Synchronisation

```c
SemaphoreHandle_t  syncSemaphore = xSemaphoreCreateBinary();

void TaskA( void * pvParameters )
{
     for( ;; )
     {
        ....
        /* Tell we are done */
        xSemaphoreGive(syncSemaphore);
        ....
     }
}


void TaskB( void * pvParameters )
{
     for( ;; )
     {
          xSemaphoreTake(syncSemaphore, portMAX_DELAY);
          ....
          ....
     }
}
```



Semaphore Exercises

1. Task B must do an operation opB() only after Task A has done operation opA()
   - How can you guarantee this using semaphores?

2. Consider the tree following Tasks

Task A	Task B	Task C
Prints "R" Prints "OK"	Prints "I" Prints "OK"	Prints "O" Prints "OK"

a) Add operations on semaphores such that:
   - The result printed is *R I O OK OK OK*
   - The final value of the semaphores must be identical to their initial value

Semaphore Exercises

3. Consider the following tasks

Task A	Task B

// initialisation code
int x = 0;
x = y + z;
// other code…
	// initialisation code
	int y = 0, z = 0
	y = 1;
	z = 2;
	// other code…

a) What are the possible final values for x?
b) Is it possible, using semaphore, to have only two values for x?

Counting Semaphores

A counting semaphore can be **given** and **taken** a number of times

Used for
- Signalling between tasks
- Synchronisation of tasks

Counting Semaphore
- value : int {0..maxCount}
+ take() : boolean + give() : void + create(maxCount : int, initialCount : int) : void

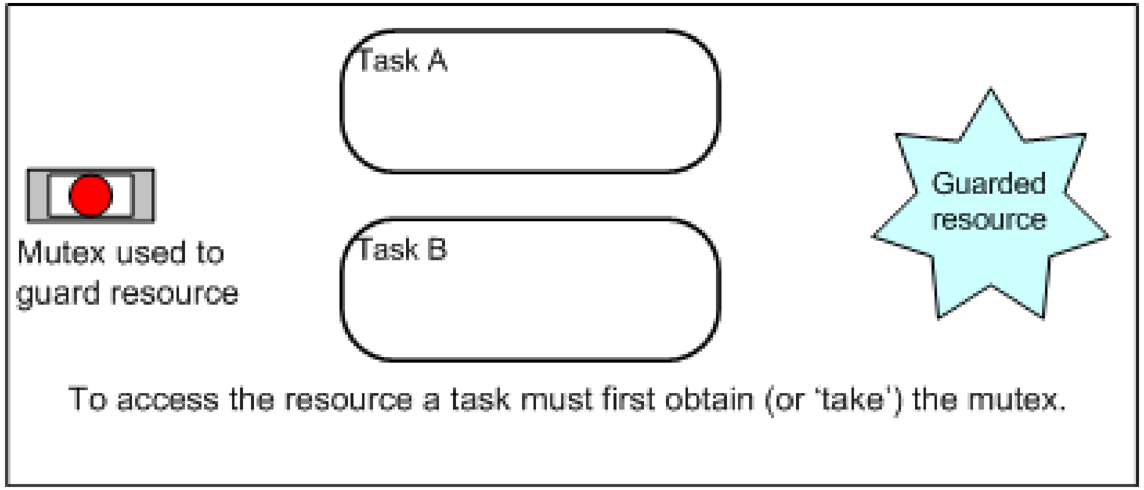
Read more here https://www.freertos.org/Real-time-embedded-RTOS-Counting-Semaphores.html

Mutex's

Mutexes are binary semaphores that include a **priority inheritance** mechanism

Whereas binary semaphores are the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing simple mutual exclusion (hence '**MUT**'ual '**EX**'clusion)

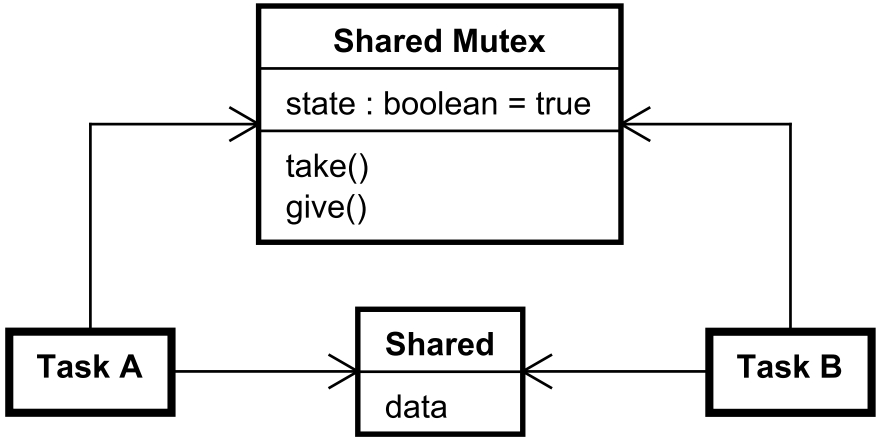
For protection of a resource



Read more here: https://www.freertos.org/Real-time-embedded-RTOS-mutexes.html

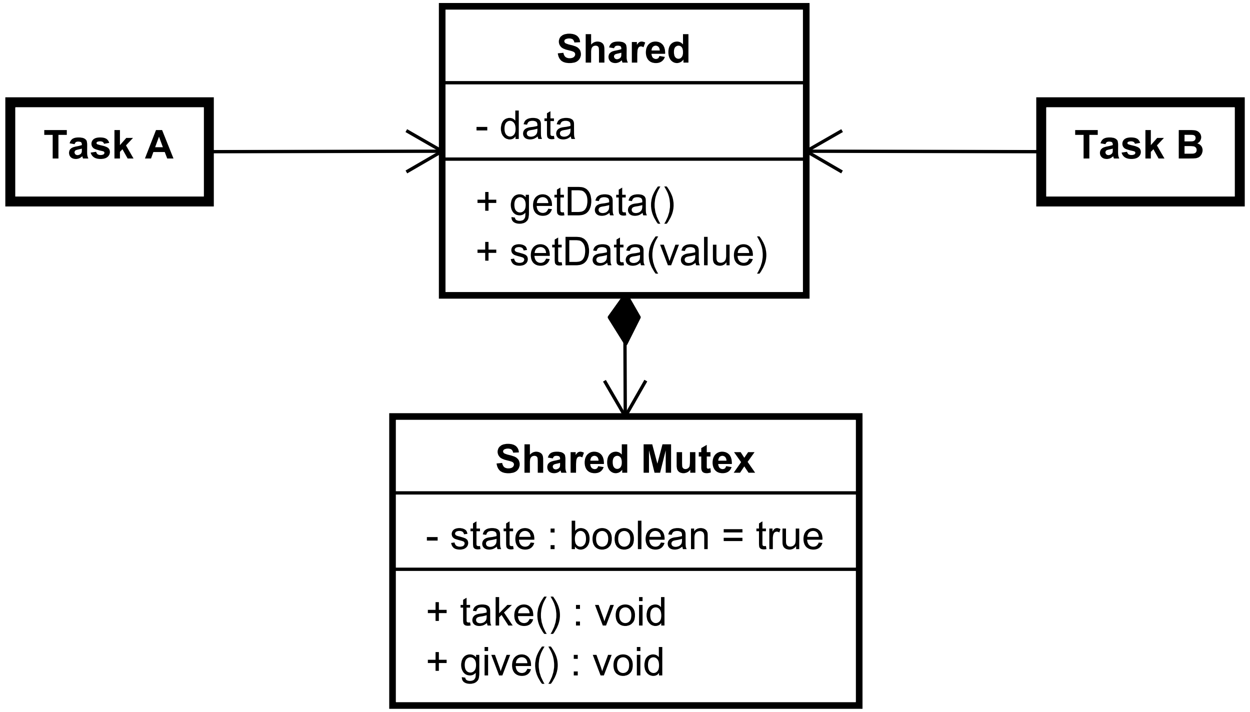
Mutex

```c
SemaphoreHandle_t  sharedMutex = xSemaphoreCreateMutex();
// xSemaphoreGive(sharedMutex); A mutex is given when it is created

void TaskA( void * pvParameters )
{
    int tmp;

    for( ;; )
    {
      ....
      if( xSemaphoreTake( sharedMutex, pdMS_TO_TICKS(200) ) == pdTRUE ) // Wait maximum 200 ms
      {
        /* Execute the code that uses the shared data here. */
        tmp = shared.data;
        xSemaphoreGive( sharedMutex );
      }
      else
      {
        /* We timed out and could not obtain the mutex and can
        therefore not access the shared resource safely. */
      }
      ....
    }
}
```



Better use of Mutex

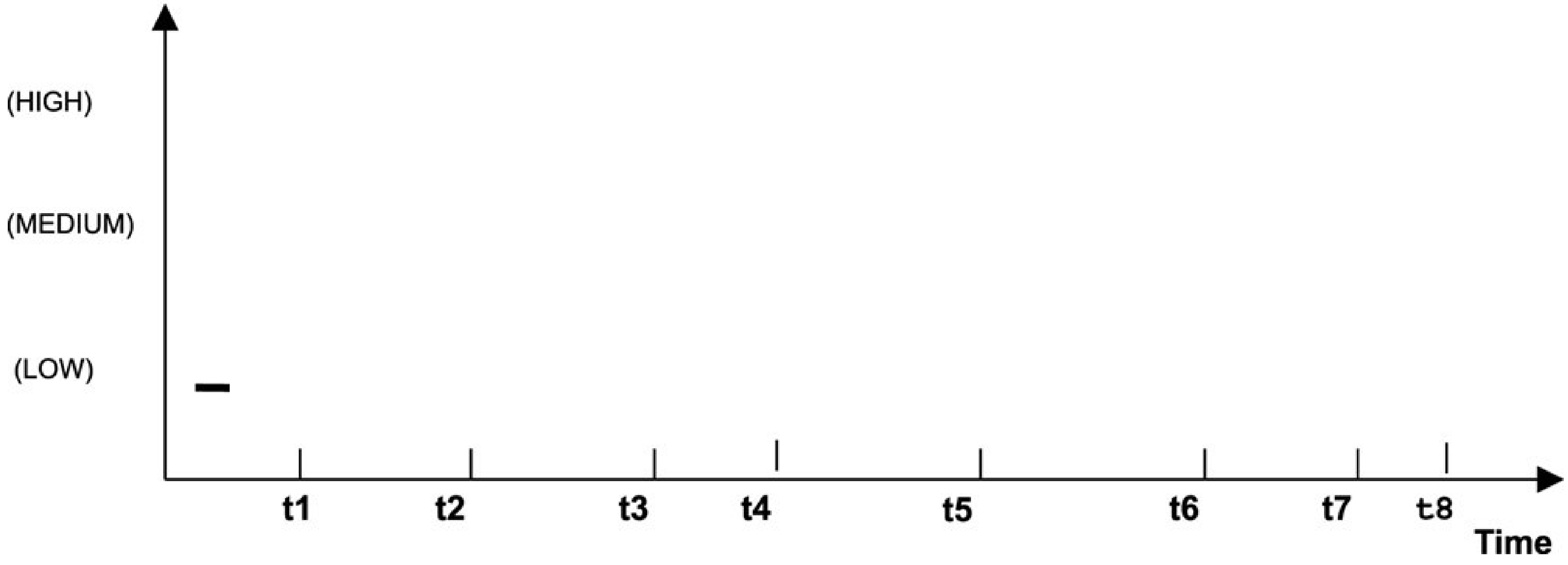
Task A and B do not handle the Mutex
The protected data (*Shared*) handles the Mutex in
- *getData()*
- *setData()*



This way the protection can not be violated by mistake

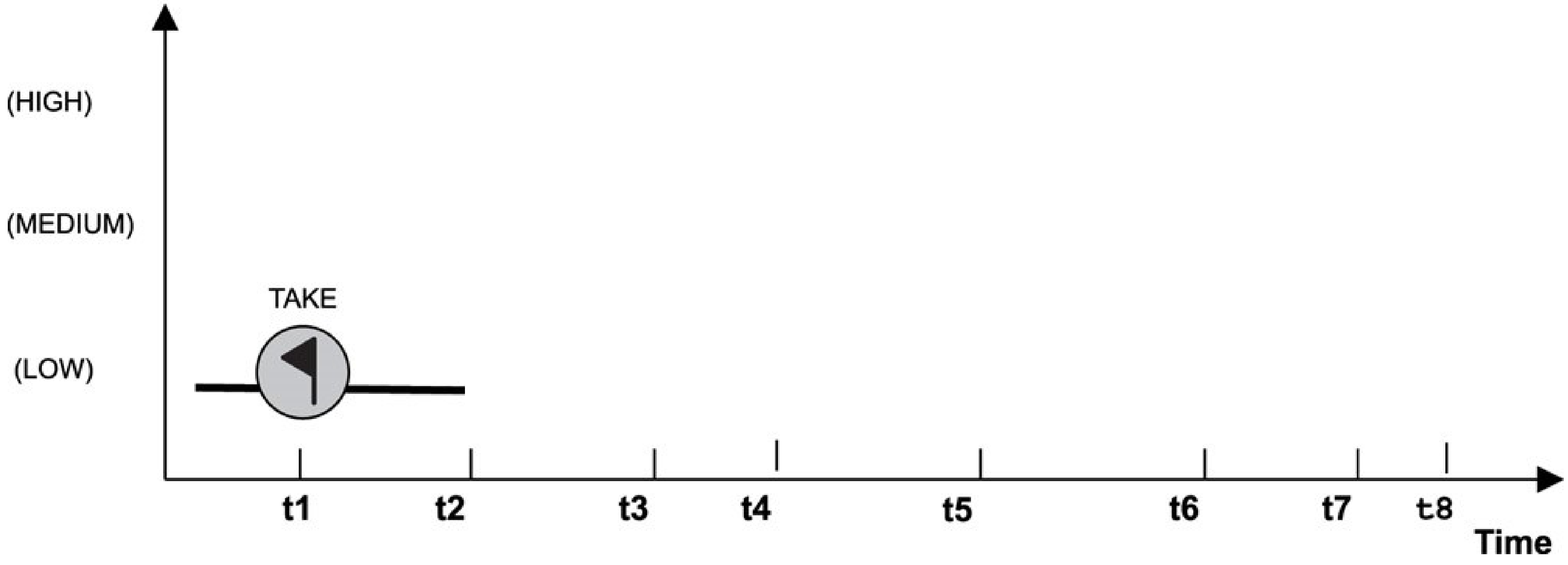
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



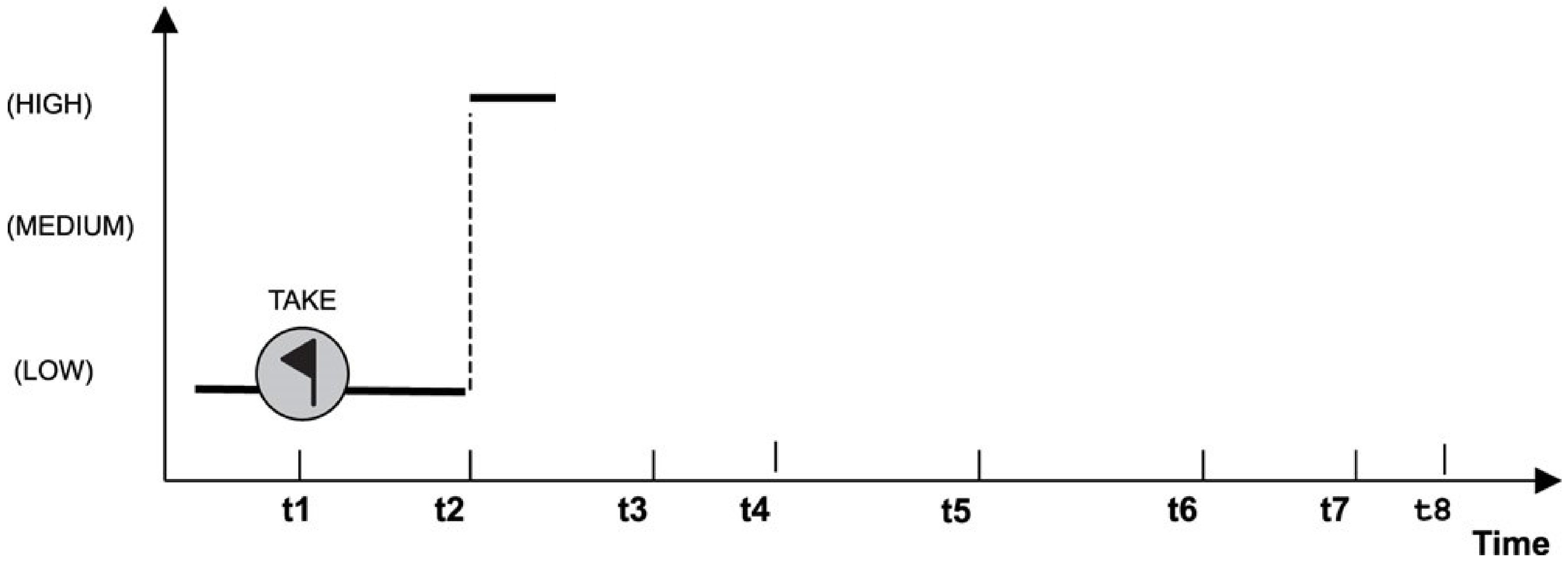
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



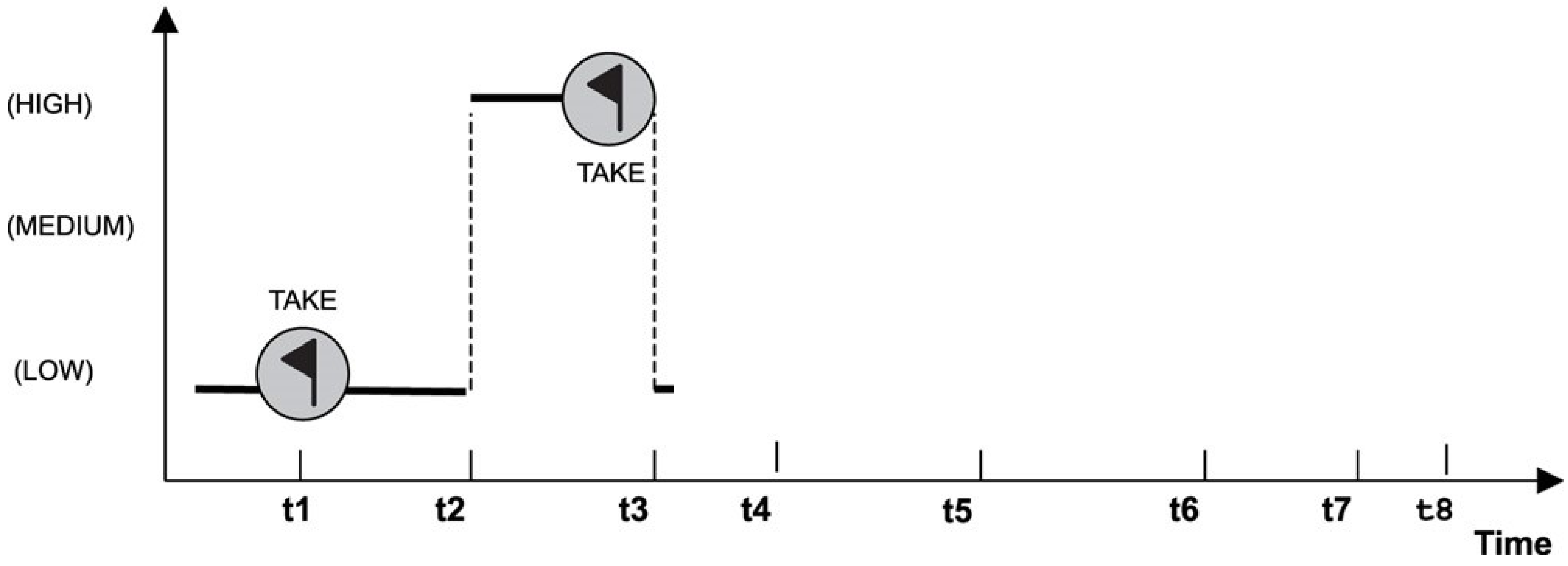
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



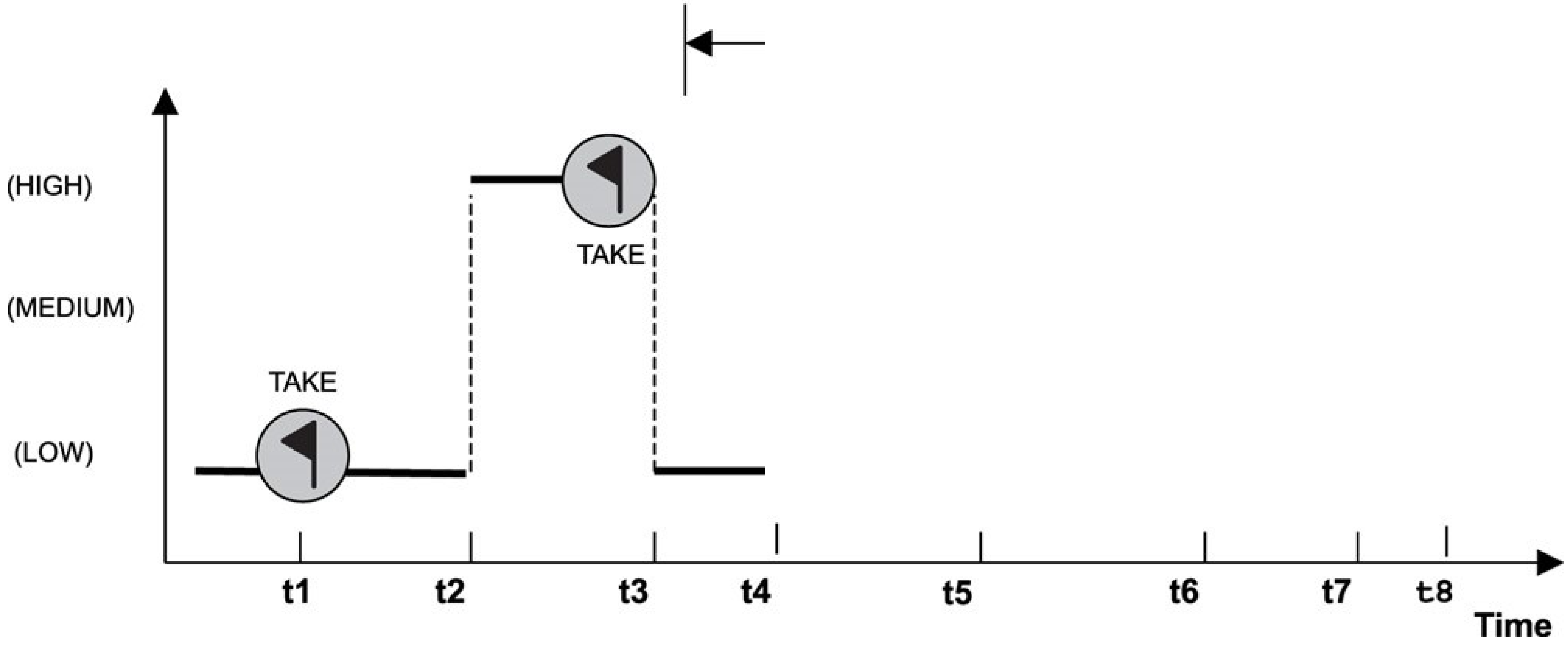
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



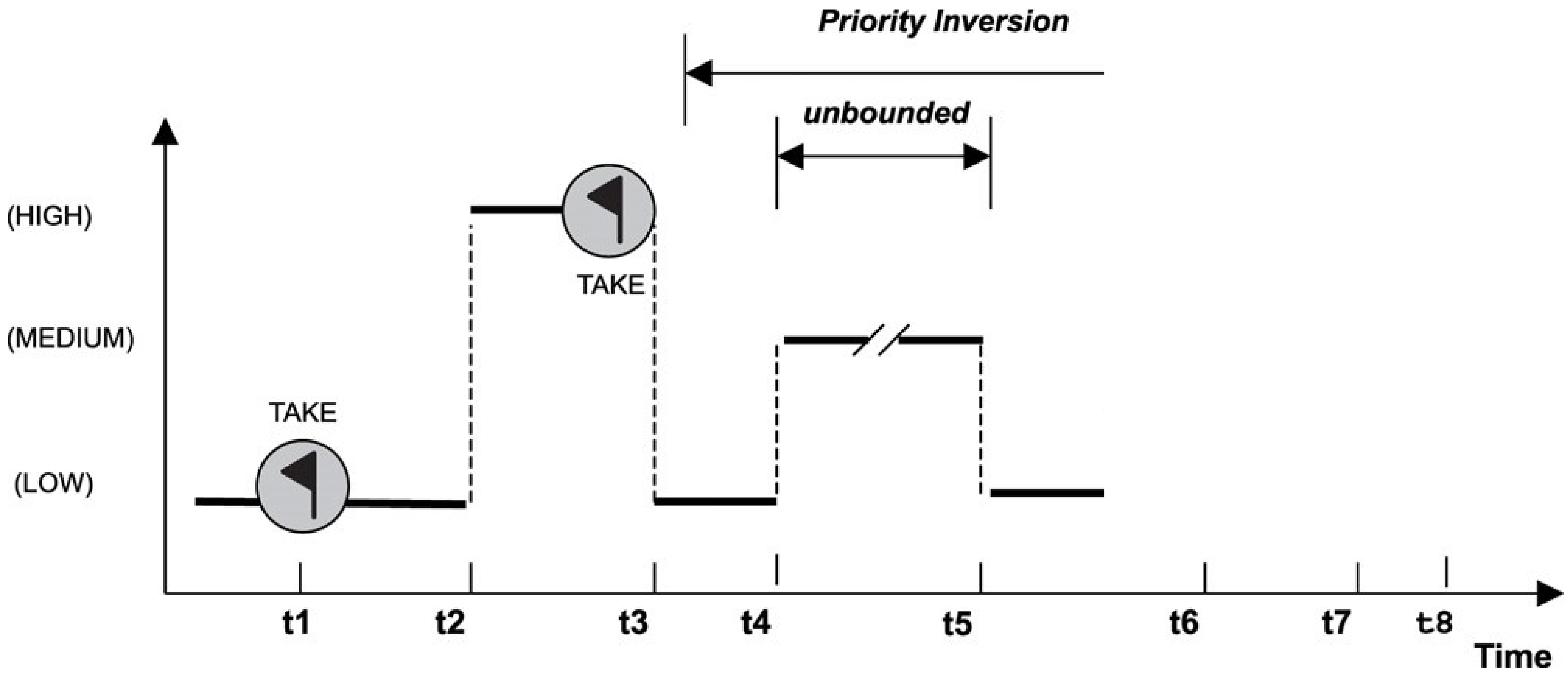
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



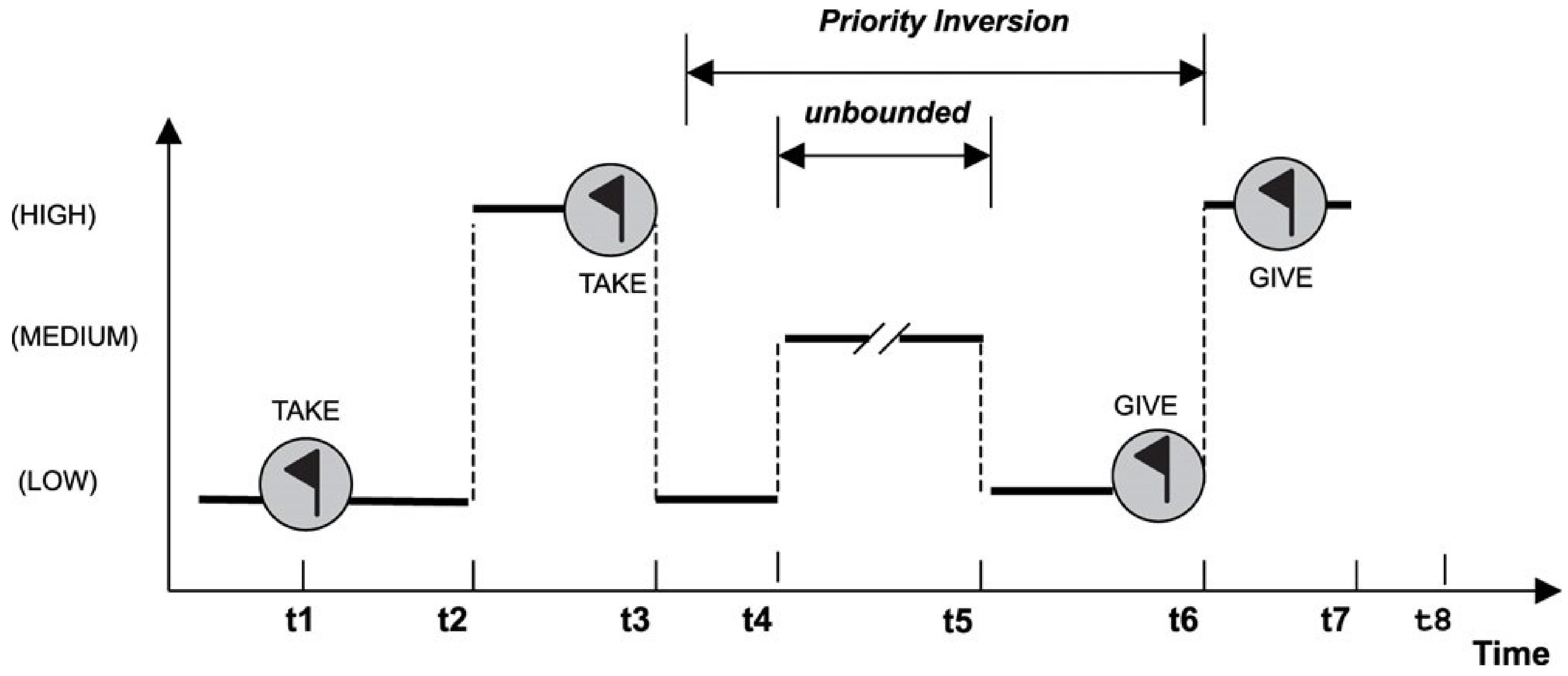
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



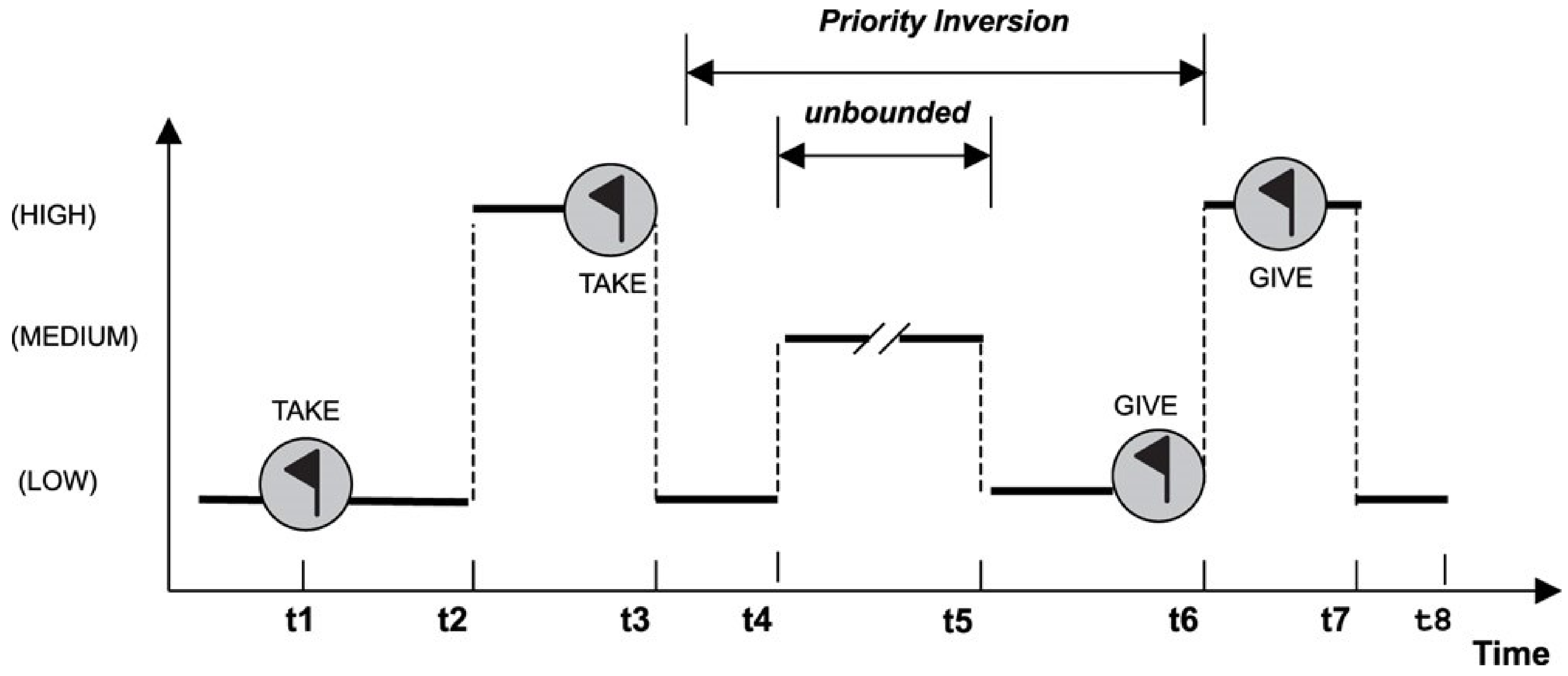
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



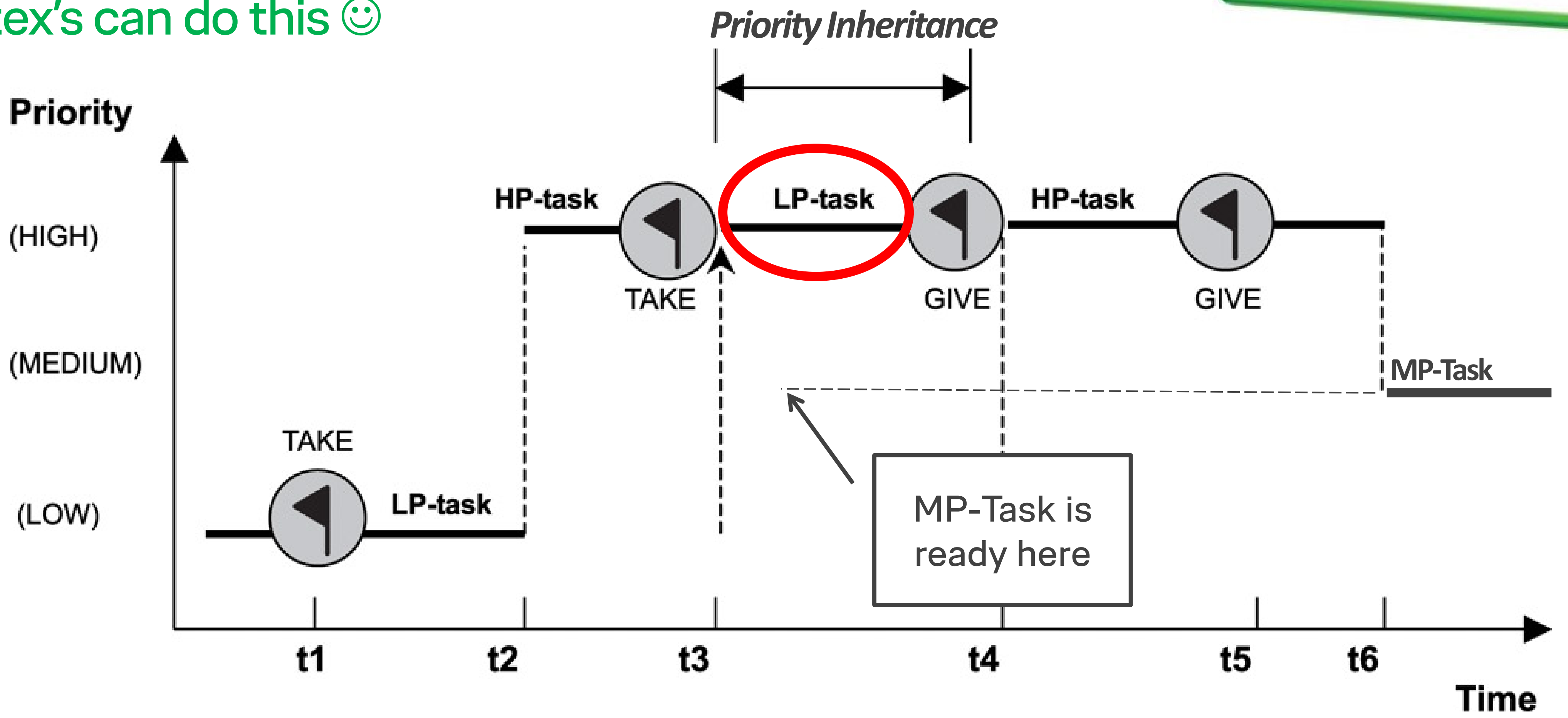
Priority Inversion

To understand *Priority Inheritance* we must understand *Priority Inversion*



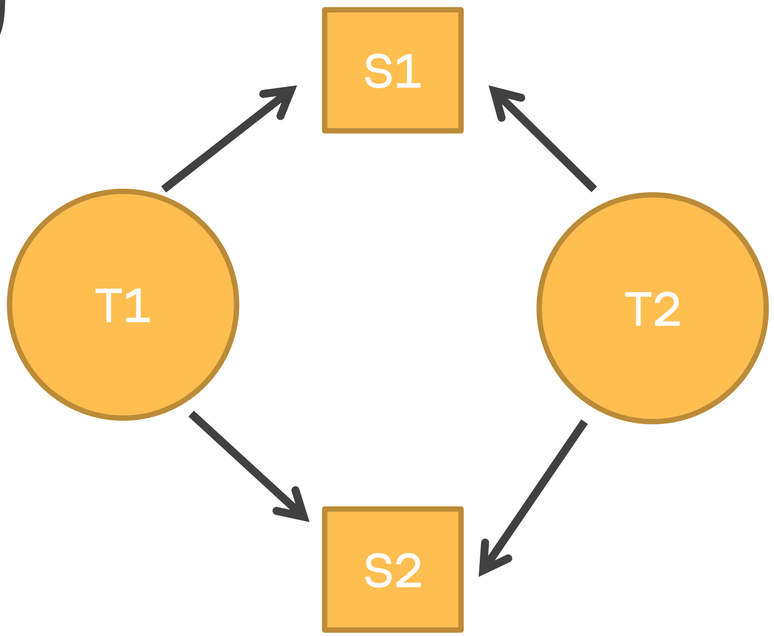
Priority Inheritance

Mutex's can do this ☺



The low priority task is temporary given the priority of the high priority task to let it execute until it gives the semaphore back!

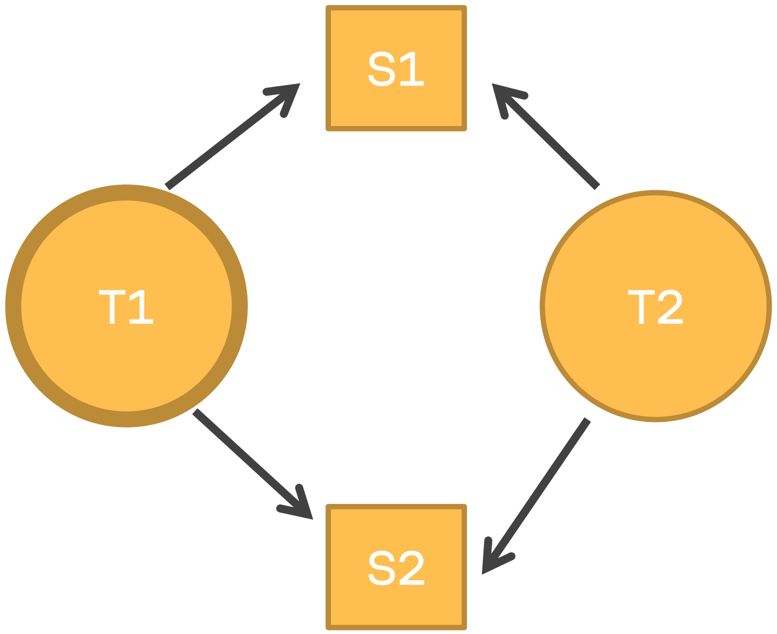
Deadlock (in the simplest form)



Be warned: semaphores can lead to deadlocks or priority inversion!

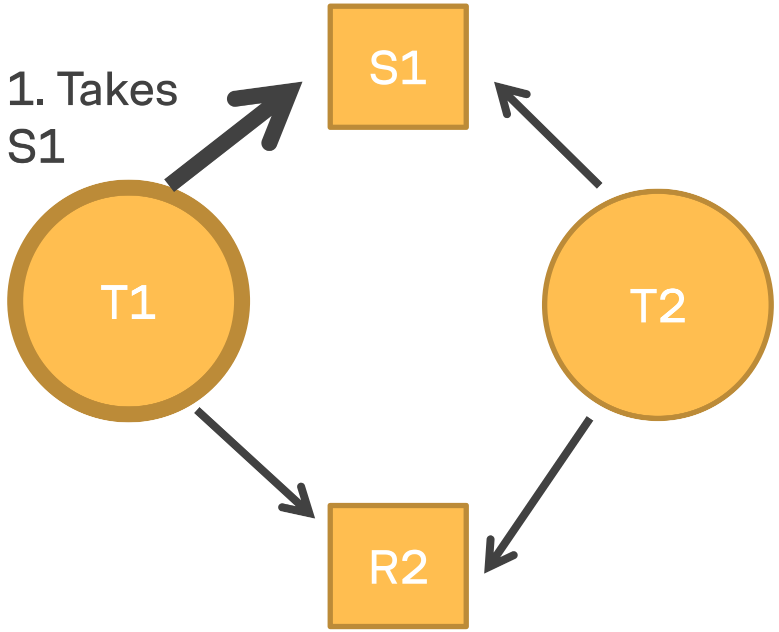
Deadlock

T1 runs first.



Deadlock

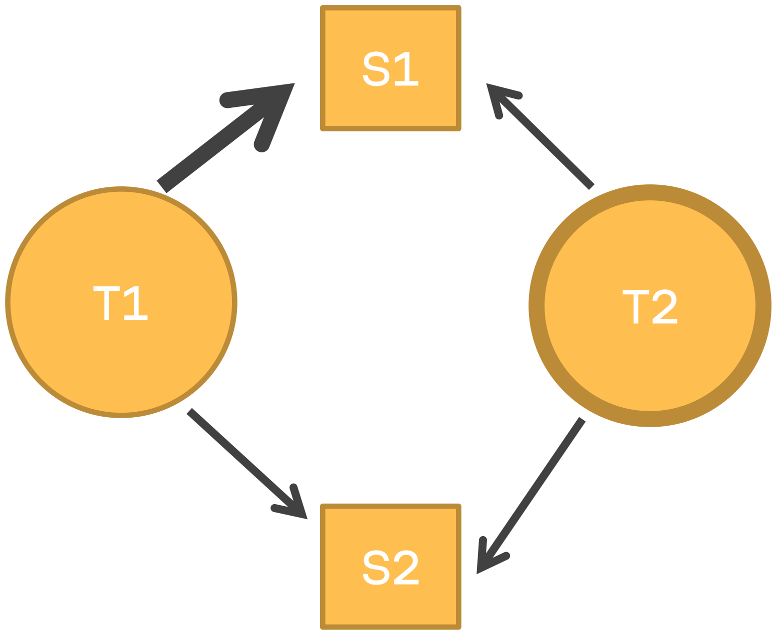
T1 takes S1



Deadlock

T1 is preempted and T2 runs.

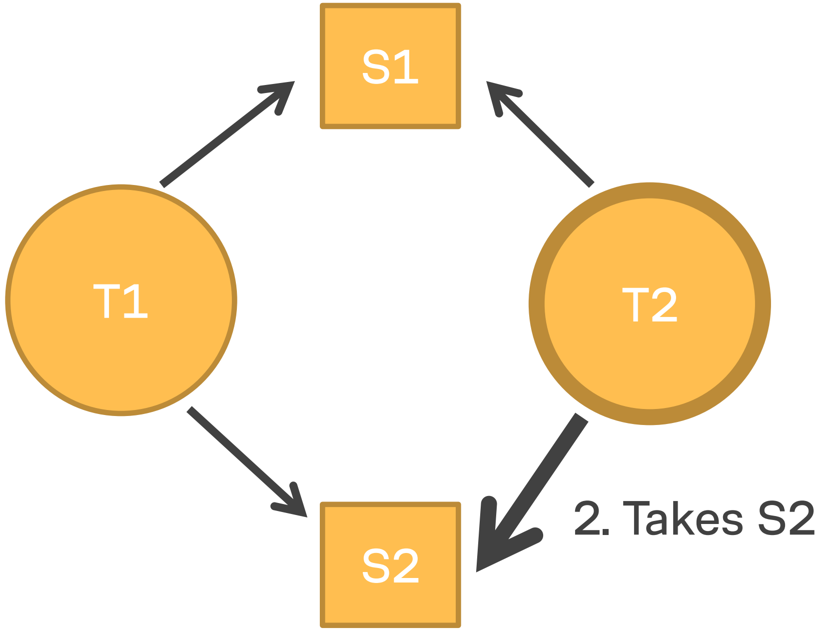
Has taken S1



Deadlock

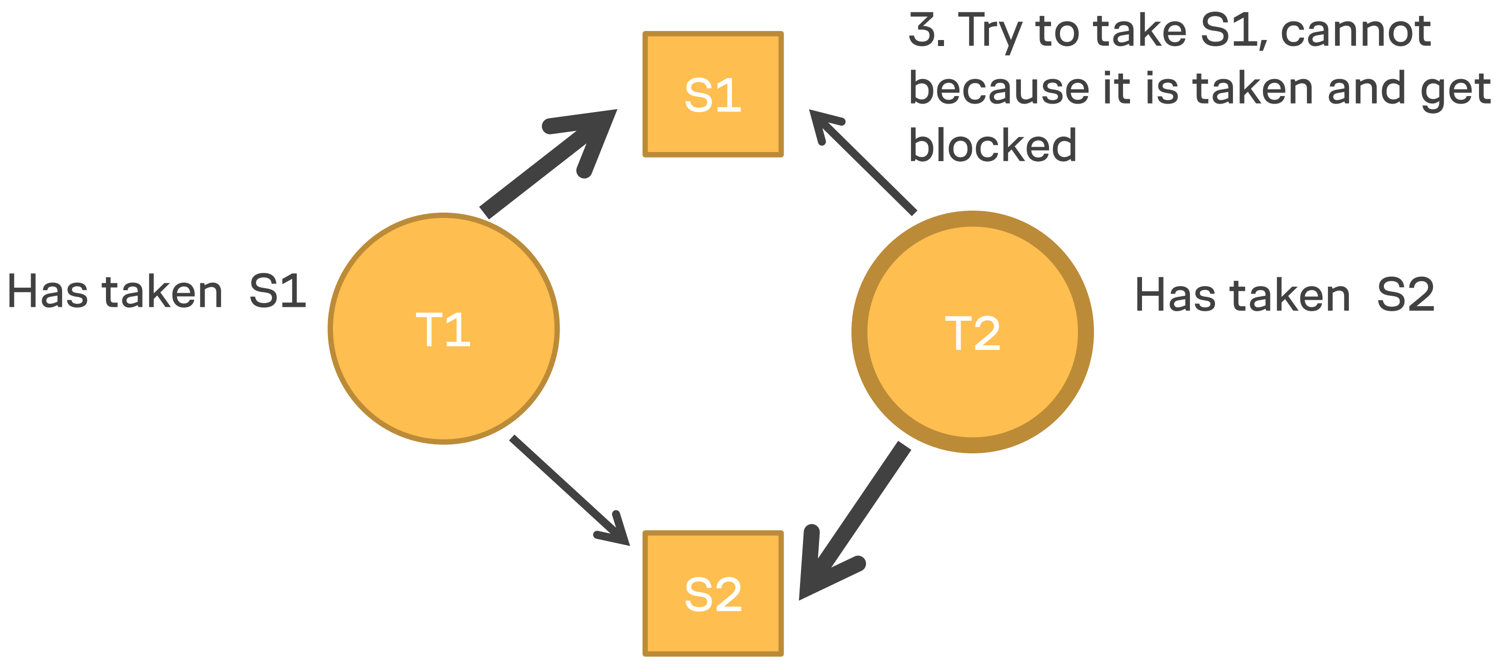
T2 takes S2

Has taken S1



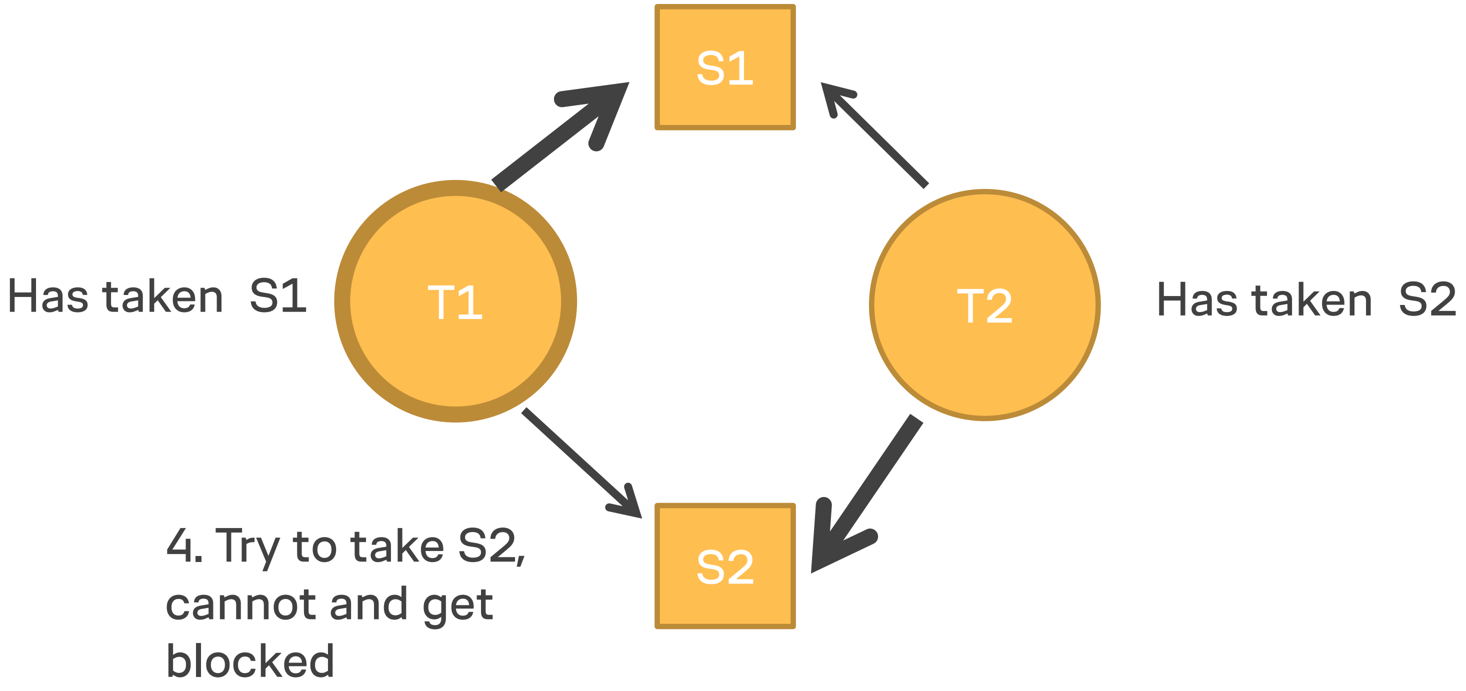
Deadlock

T2 tries to take S1.



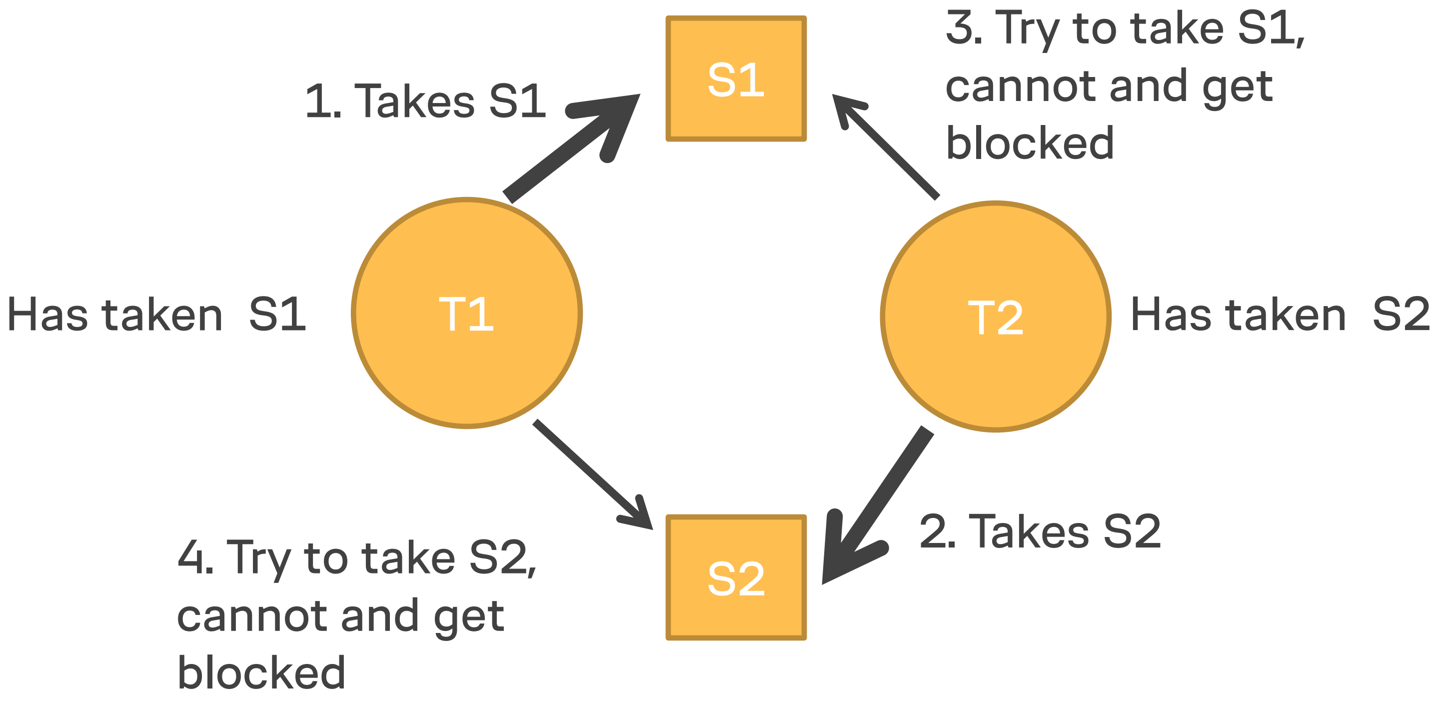
Deadlock

T1 resumes and tries to take S2



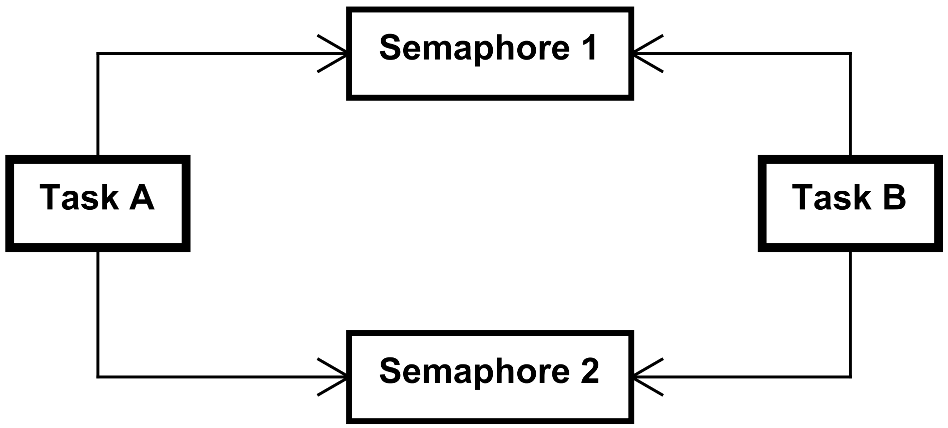
Deadlock

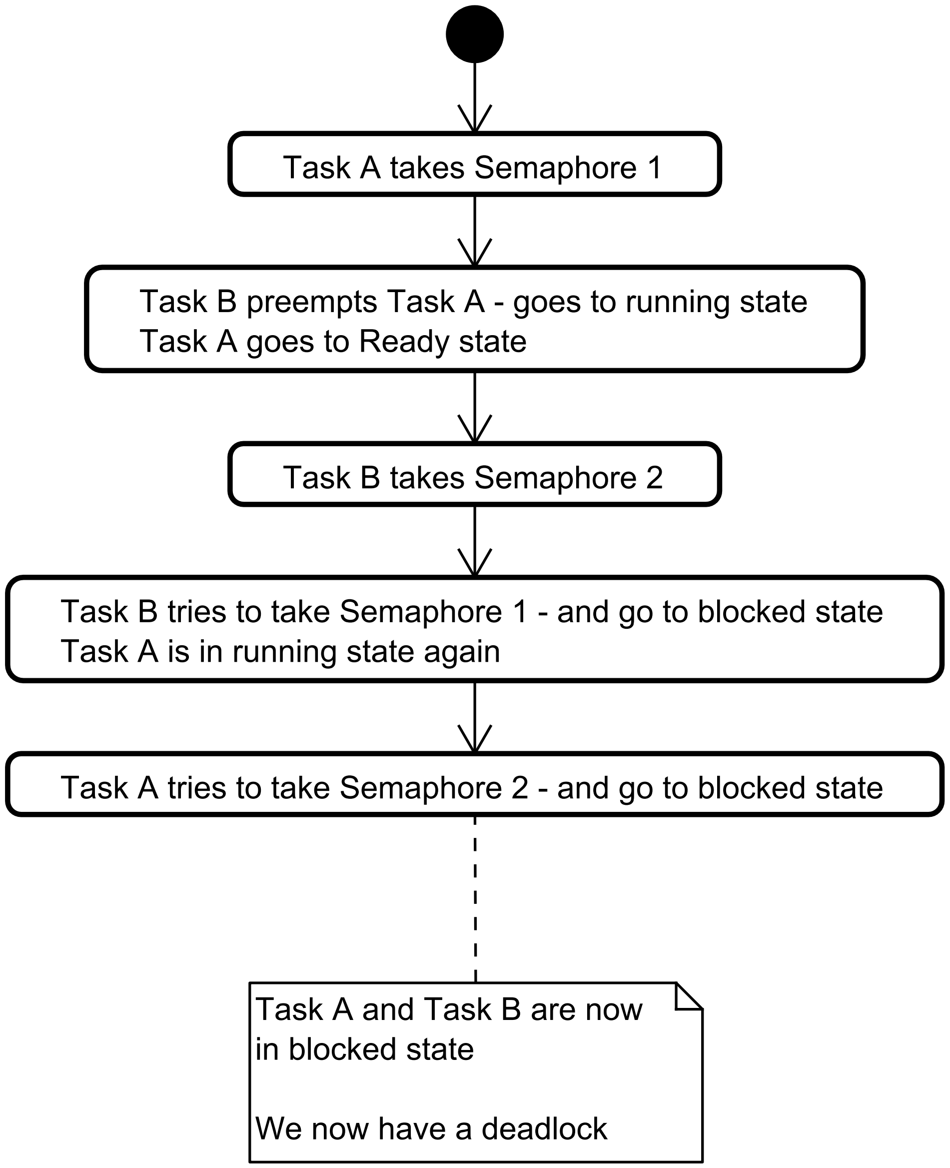
T1 blocked, T2 blocked - both tasks **stays blocked forever.** This is a Deadlock!!



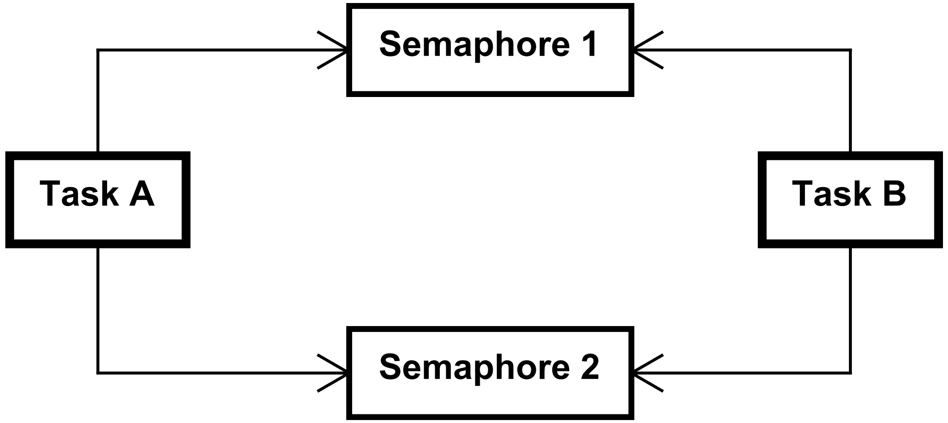
Deadlock Explained (in the simplest form)

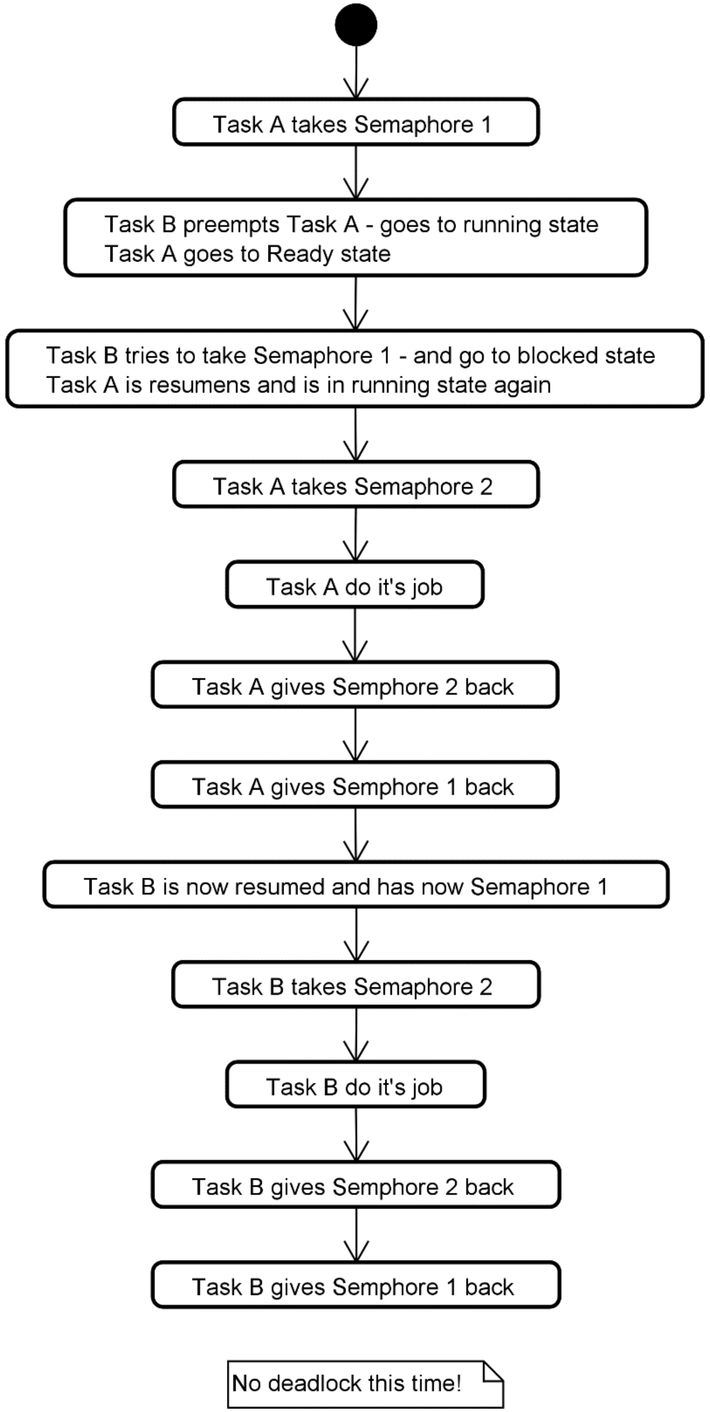
Be warned: semaphores can lead to deadlocks and/or priority inversion!





Prevent Deadlock





Prevent deadlocks

- Acquire all necessary semaphores before proceeding
- Acquire all semaphores in the same order and release them in the opposite order
- Time out if semaphore does not become available

Demands for Safety Critical Systems

Liveness

- No deadlocks
- No live-locks
- No starvation
- No priority inversion

And meeting all hard deadlines (a lot more about this later)

Delays - vTaskDelay

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

Used to

- Delay a task for a given number of ticks
- The actual time that the task remains blocked depends on the tick rate
  - Setup in *FreeRTOSConfig.h* or in the port
- The constant `portTICK_PERIOD_MS` can be used to calculate real time from the tick rate – with the resolution of one tick period
  - Macro `pdMS_TO_TICKS(msToDelay)` can also be used
  - `portMAX_DELAY` can be used to wait forever

Read more here: https://www.freertos.org/a00127.html

Delays - vTaskDelay

Can you see any problems in using vTaskDelay() ?

```
for (;;)
{
   for (int i = 0; i < 5; i++)
   {
      puts("Send");
      xQueueSend(intQueue, &counter,
      portMAX_DELAY);
      counter++;
   }
   vTaskDelay(pdMS_TO_TICKS(200));
}
```

How long time are there between the task executes?

Delays - vTaskDelayUntil

```
void vTaskDelayUntil( TickType_t *pxPreviousWakeTime,
                      const TickType_t xTimeIncrement )
```

Used to

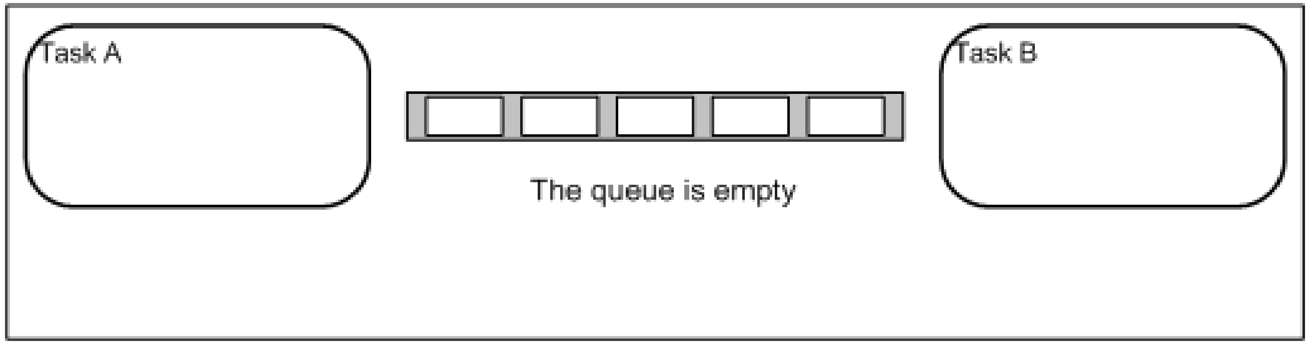
- Delay a task a **precise** specified time
- Ensures periodic tasks to have **constant** execution frequency/period

```
// Initialise the xLastWakeTime variable with the current time.
TickType_t xLastWakeTime = xTaskGetTickCount();
for (;;)
{
    vTaskDelayUntil(&xLastWakeTime, pdMS_TO_TICKS(200));
    puts("Hi from My Task");
}
```

Read more here: https://www.freertos.org/vtaskdelayuntil.html

Queues

Queues are the primary form of inter-task communications



Thread safe FIFO
(First In First Out) buffers

New data being sent to

- The back of the queue
- can also be sent to the front

Messages are sent through queues **by copy**

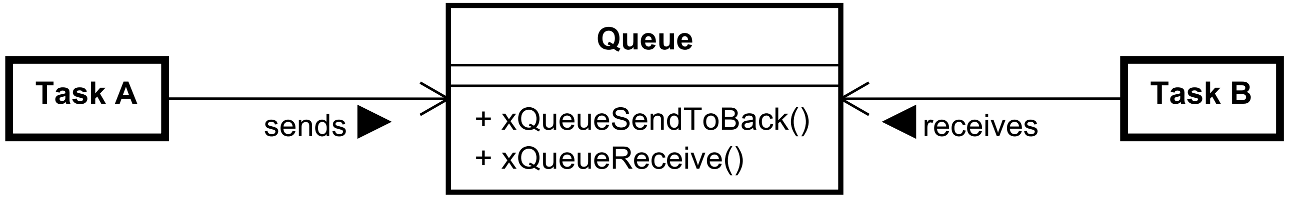
Normally mutually exclusive

Read more here: https://www.freertos.org/Embedded-RTOS-Queues.html

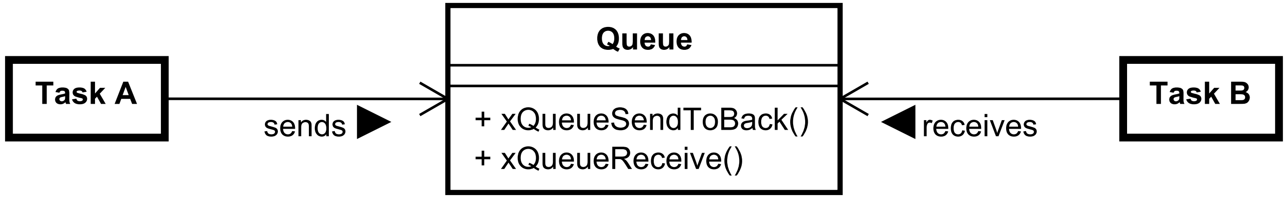
Queues

Most used Queue Functions

- xQueueCreate
- xQueueSendToBack
- xQueueSendToBackFromISR
- xQueueSendToFront
- xQueueSendToFrontFromISR
- xQueueReceive
- xQueueReceiveFromISR
- uxQueueMessagesWaiting
- uxQueueSpacesAvailable
- xQueueReset
- xQueuePeek



Queue Exercises



1. Create two tasks (A & B)
   a) Let A sent an **integer** into a queue each 100 ms
   b) Let B receive from the queue and print out what it gets

2. Create two tasks (A & B**) A with highest priority**
   a) Let A sent an **5 integers** into a queue each 100 ms
   b) Let B receive from the queue and print out what it gets

3. Change the priorities of the task in 2. to let **B have the highest priority**
   a) Do a) and b) again

4. Change the task and the queue to be able to sent **doubles** and do exercise 1) again

MessageBuffers

Message buffers allow **variable length discrete messages** to be passed from an interrupt service routine to a task, or from one task to another task.

- For example, messages of length 10, 20 and 123 bytes can all be written to, and read from, the same message buffer

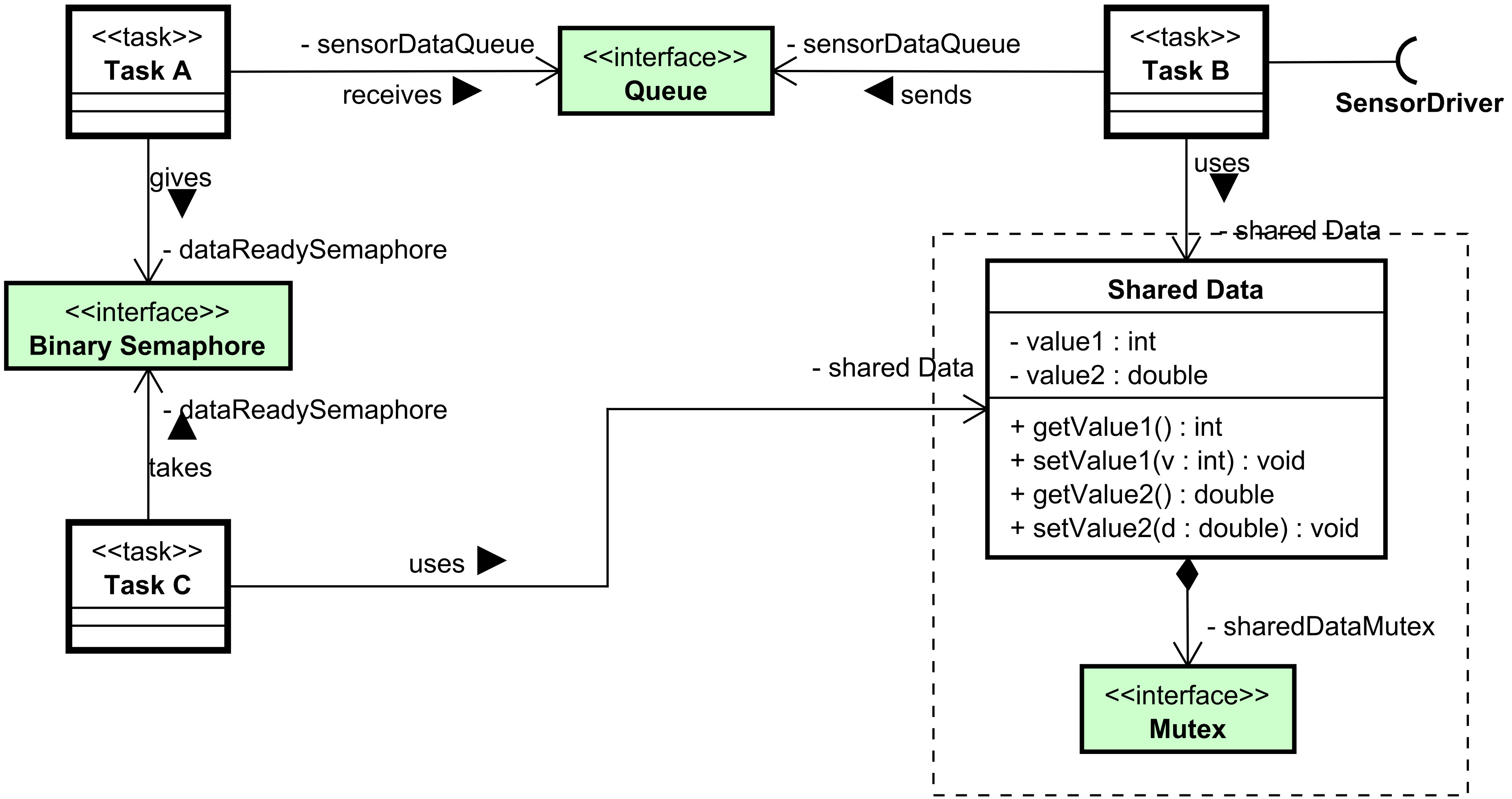
Message buffers are used in **One task to One task** communication

IMPORTANT NOTE: Message buffer implementation assumes there is only **one task or interrupt** that will write to the buffer (the writer), and only **one task or interrupt** that will read from the buffer (the reader)

It is safe for the writer and reader to be different tasks or interrupts, but **it is not safe** to have multiple different writers or multiple different readers

Read more here: https://www.freertos.org/RTOS-message-buffer-example.html

RTOS and UML Diagrams



Message Buffer Exercises

1. Create two tasks (A & B)
   a) Let A sent an **different length** strings (zero terminated) into the message buffer
   b) Let B receive from the message buffer and print out what it gets

Event Groups - Event Bits (Event Flags)

Event bits are used to indicate if an event has occurred or not. Event bits are often referred to as event flags. For example, an application may

- Define a bit (or flag) that means “A message has been received and is ready for processing” when it is set to 1, and “there are no messages waiting to be processed” when it is set to 0
- Define a bit (or flag) that means “The application has queued a message that is ready to be sent to a network” when it is set to 1, and “there are no messages queued ready to be sent to the network” when it is set to 0
- Define a bit (or flag) that means “It is time to send a heartbeat message onto a network” when it is set to 1, and “it is not yet time to send another heartbeat message” when it is set to 0

Event Groups - Event Bits (Event Flags) How-to

```
#include "event_groups.h"
```

Declare global

```
/* Event Groups */
EventGroupHandle_t _myEventGroup = NULL;
#define BIT_TASK_A_READY (1 << 0)
#define BIT_TASK_B_READY (1 << 1)
```

Define the event group some where in *main* or in a initialation task

```
/* Create Event Groups */
_myEventGroup = xEventGroupCreate();
```

Event Groups - Event Bits (Event Flags) How-to

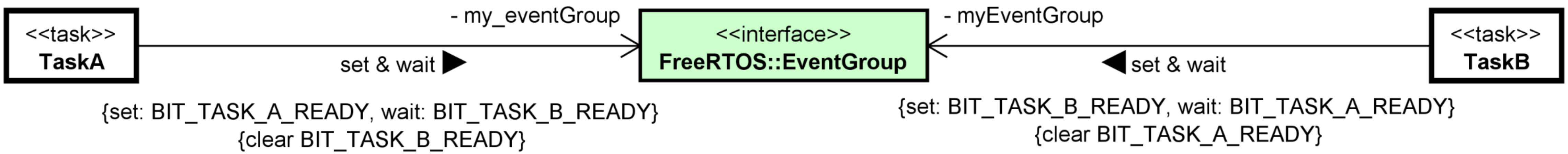
Set Event bits in group

```
xEventGroupSetBits(_myEventGroup, BIT_TASK_A_READY);
```

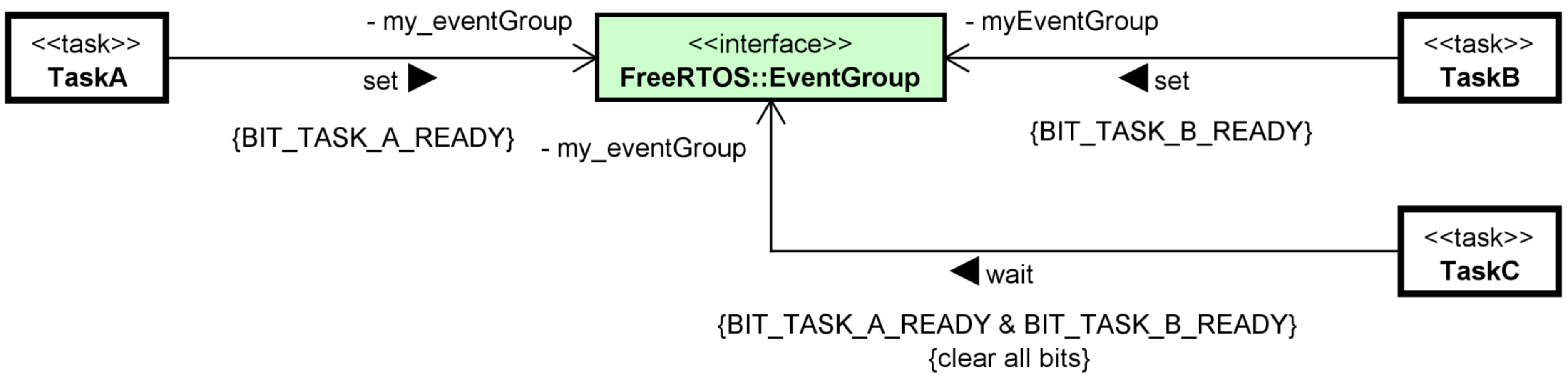
Wait for Event bits to be set in Group

```
xEventGroupWaitBits(
   _myEventGroup, /* The event group being tested. */
   BIT_TASK_B_READY, /* The bits to wait for. */
   pdTRUE, /* Bits will be cleared before return*/
   pdTRUE, /* Wait for bits to be set */
   portMAX_DELAY); /* Maximum time to wait*/
```

Event Groups – and UML How-to



Exercise Event Groups



Make a system where two tasks each set an individual bit in an event group and have a third task wait for these two bits to be set