Technika cyfrowa i mikroprocesorowa

Systemy czasu rzeczywistego, podstawowe zagadnienia. Omówienie problemów pracy wielowątkowej w systemach embedded.

Wojciech Tarnawski
www.w-tarnawski.pl
wojciech.tarnawski@pwr.edu.pl

Systemy czasu rzeczywistego

- Systemy czasu rzeczywistego
- FreeRTOS
- SafeRTOS
- Wątki, scheduler
- Priorytety
- Problem synchronizacji
- Semafory, mutex
- Kolejki FIFO/LIFO
- Kolejki jednoelementowe
- Podsumowanie

Systemy czasu rzeczywistego

System czasu rzeczywistego (ang. real-time system, real-time computing, RTC) – urządzenie techniczne, którego wynik i efekt działania jest zależny od chwili wypracowania tego wyniku. Istnieje wiele różnych definicji naukowych takiego systemu. Ich wspólną cechą jest zwrócenie uwagi na równoległość w czasie zmian w środowisku oraz obliczeń realizowanych na podstawie stanu środowiska. Z tego wyścigu dwóch stanów: zewnętrznego i wewnętrznego, wynikają kryteria ograniczające czas wypracowywania wyniku.

źródło: https://pl.wikipedia.org/wiki

Systemy czasu rzeczywistego

Często pod pojęciem „system czasu rzeczywistego” rozumie się systemy zbudowane z wykorzystaniem komputera, pracującego pod kontrolą systemu operacyjnego czasu rzeczywistego. W skład takiego systemu włącza się także jego niezbędne otoczenie, takie jak deterministyczne sieci transmisyjne (np. Modbus, CAN itd.),układy wejściowe i wyjściowe oraz urządzenia kontrolowane przez komputer (np. napędy, przekaźniki, itd.).

Aby system składający się z komponentów był systemem czasu rzeczywistego, konieczne jest spełnianie wymogów systemu czasu rzeczywistego przez każdy z komponentów. W przypadku systemów informatycznych oznacza to, że zarówno sprzęt, system operacyjny, jak i oprogramowanie aplikacyjne muszą gwarantować dotrzymanie zdefiniowanych ograniczeń czasowych.

W realizacji oprogramowania działającego w czasie rzeczywistym niezbędna jest analiza wydajności działania aplikacji.

źródło: https://pl.wikipedia.org/wiki

Systemy czasu rzeczywistego

Podział systemów czasu rzeczywistego:

1. systemy o ostrych/rygorystycznych ograniczeniach czasowych (ang. hard real-time)
2. systemy o mocnych ograniczeniach czasowych (ang. firm real-time) - gdy fakt przekroczenia terminu powoduje całkowitą nieprzydatność wypracowanego przez system wyniku, jednakże nie oznacza to zagrożenia dla ludzi lub sprzętu; pojęcie to stosowane jest głównie w opisie teoretycznym baz danych czasu rzeczywistego,
3. systemy o miękkich lub łagodnych ograniczeniach czasowych (ang. soft real-time)

Systemy czasu rzeczywistego

Rygorystyczne ograniczenie czasowe (ang. Hard Deadline)

to takie ograniczenie które zawsze pozostaje spełnione. Jeśli choć raz zostało przekroczone uważa się że nie zostało spełnione. Wymaga się aby istniała procedura walidacyjna pozwalająca na sprawdzenie czy warunek ten został spełniony.

Rygorystyczny system czasu rzeczywistego (ang. Hard Real TimeSystem)

to system w którym wymaga się spełnienia rygorystycznych ograniczeń czasowych.

Przykłady rygorystycznych systemów czasu rzeczywistego:

- System sterowania elektrownią atomową
- System sterowania samolotem
- System sterowania zapłonem samochodowym

Systemy czasu rzeczywistego

Łagodne ograniczenie czasowe (ang. Soft Deadline)

to takie ograniczenie czasowe które czasami może być przekroczone i przekroczenie pewnego czasu powoduje negatywne skutki tym poważniejsze, im bardziej ten czas został przekroczony.

Jak zdefiniować pojęcie czasami:

1. Kategoria prawdopodobieństwa – np. ograniczenie spełnione jest

w 99% przypadków.

2. Funkcja użyteczności – podaje ocenę korzyści w zależności od

czasu uzyskania odpowiedzi.

Łagodny system czasu rzeczywistego (ang. Soft Real Time System)

To system w którym wymaga się spełnienia łagodnych ograniczeń czasowych.

Przykłady łagodnych systemów czasu rzeczywistego:

- Multimedia
- Sterowanie telefonem komórkowym
- Centrala telefoniczna

Systemy czasu rzeczywistego

Systemy czasu rzeczywistego znajdują zastosowanie:

- w przemyśle do nadzorowania procesów technologicznych,
- systemy rozproszone- składające się z wielu elementów,
- do nadzorowania eksperymentów naukowych,
- w urządzeniach powszechnego użytku, jak sterowniki układów ABS i ESP czy wtrysku paliwa do silników samochodowych, bądź też urządzenia gospodarstwa domowego,
- w medycynie,
- w lotnictwie, zastosowaniach wojskowych i kosmicznych,
- w autimatycznych kasach biletowych, oprogramowaniu bibliotek i podobnych usługach realizowanych w wielodostępnych systemach rozproszonych,
- skomplikowane układy sterowania, które realizują kilka różnych funkcji.

źródło: https://pl.wikipedia.org/wiki

FreeRTOS

FreeRTOS – system operacyjny czasu rzeczywistego dla urządzeń wbudowanych.

- FreeRTOS został zaprojektowany, pod kątem najkrótszego i najprostszego kodu źródłowego. Jądro składa się tylko z trzech plików kodu. Aby kod był czytelny, łatwy do portowania i konserwacji jest napisany głównie w języku C. Zastosowano również wstawki assemblerowe.
- FreeRTOS zapewnia metody do tworzenia wielu wątków bądź zadań, mutexów, semaforów i timerów. Posiada ponadto specjalistyczne funkcje dla aplikacji o niskim poborze prądu. Obsługiwane są priorytety wyjątków. Aplikacje mogą być całkowicie przydzielane statycznie.
- Nie ma bardziej zaawansowanych funkcji, które zwykle można znaleźć w systemach operacyjnych, takich jak Linux lub Microsoft Windows, takich jak sterowniki urządzeń, zaawansowane zarządzanie pamięcią, konta użytkowników i sieć. Nacisk kładziony jest na zwartość i szybkość wykonania. FreeRTOS można traktować raczej jako „bibliotekę wątków” niż „system operacyjny”,

źródło: https://pl.wikipedia.org/wiki

SafeRTOS

SAFERTOS® is a pre-certified safety Real Time Operating System (RTOS) for embedded processors. It delivers superior performance and pre-certified dependability, whilst utilizing minimal resources.

- Developed by WHIS, a safety systems company
- Supports a wide range of international development standards
- Based on the FreeRTOS functional model, with simple migration
- Available pre-certified to IEC 61508-3 SIL 3 by TÜV SÜD - Bezpieczeństwo funkcjonalne elektrycznych/elektronicznych/programowalnych elektronicznych systemów związanych z bezpieczeństwem
- Available pre-certified to ISO 26262 ASIL D by TÜV SÜD - (Automotive Safety Integrity Level) - awaria któregokolwiek z komponentów nie powodowała zagrożenia dla osób znajdujących się wewnątrz lub na zewnątrz pojazdu.
- Part of the WITTENSTEIN group, established in 1948

https://www.highintegritysystems.com/safertos/

Wątki - zadania

- program dzielimy na mniejsze zadania/wątki
- każdy wątek wykonuje założone operacje zgodnie z algorytmem
- wątki mogą mieć priorytety
- dostęp do wspólnych zasobów sprzętowych wymaga synchronizacji i zabezpieczenia
- wymiana informacji pomiędzy wątkami wymaga specjalnych struktur i funkcji
- obsługa przerwań sprzętowych i przesyłanie danych wymaga zastosowania specjalnych funkcji

Wątki - zadania

Stan zadania

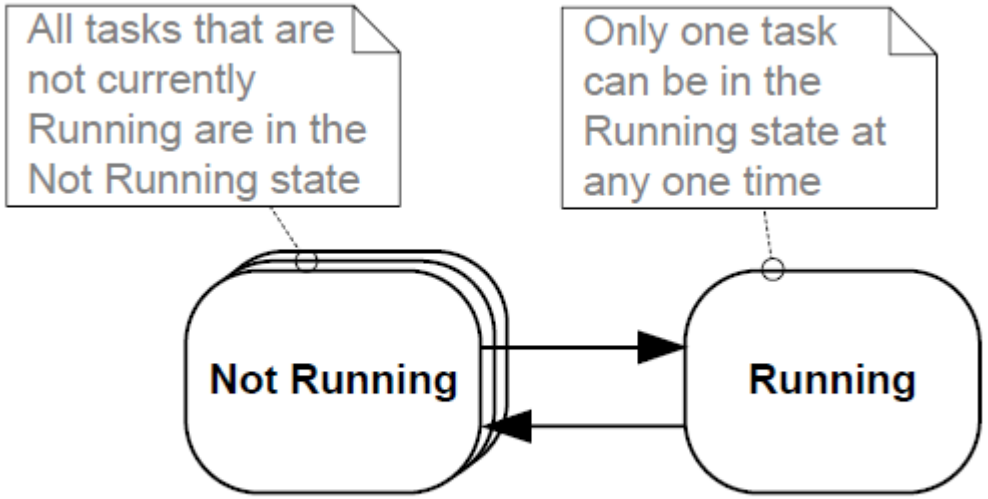


Figure 9. Top level task states and transitions

źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance of a task
created using this example function will have its own copy of the lVariableExample
variable.  This would not be true if the variable was declared static – in which case
only one copy of the variable would exist, and this copy would be shared by each
created instance of the task. (The prefixes added to variable names are described in
section 1.5, Data Types and Coding Style Guide.) */
int32_t lVariableExample = 0;

    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop, then the task
    must be deleted before reaching the end of its implementing function.  The NULL
    parameter passed to the vTaskDelete() API function indicates that the task to be
    deleted is the calling (this) task.  The convention used to name API functions is
    described in section 0, Projects that use a FreeRTOS version older than V9.0.0
must build one of the heap_n.c files.  From FreeRTOS V9.0.0 a heap_n.c file is only
required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if
configSUPPORT_DYNAMIC_ALLOCATION is left undefined.  Refer to Chapter 2, Heap Memory
Management, for more information.
Data Types and Coding Style Guide. */
    vTaskDelete( NULL );
}
```


źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop.
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  The
            nothing to do in here.  Later examples will replace this cru
            loop with a proper delay/sleep function. */
        }
    }
}
```


źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* This example does not use the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```


źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

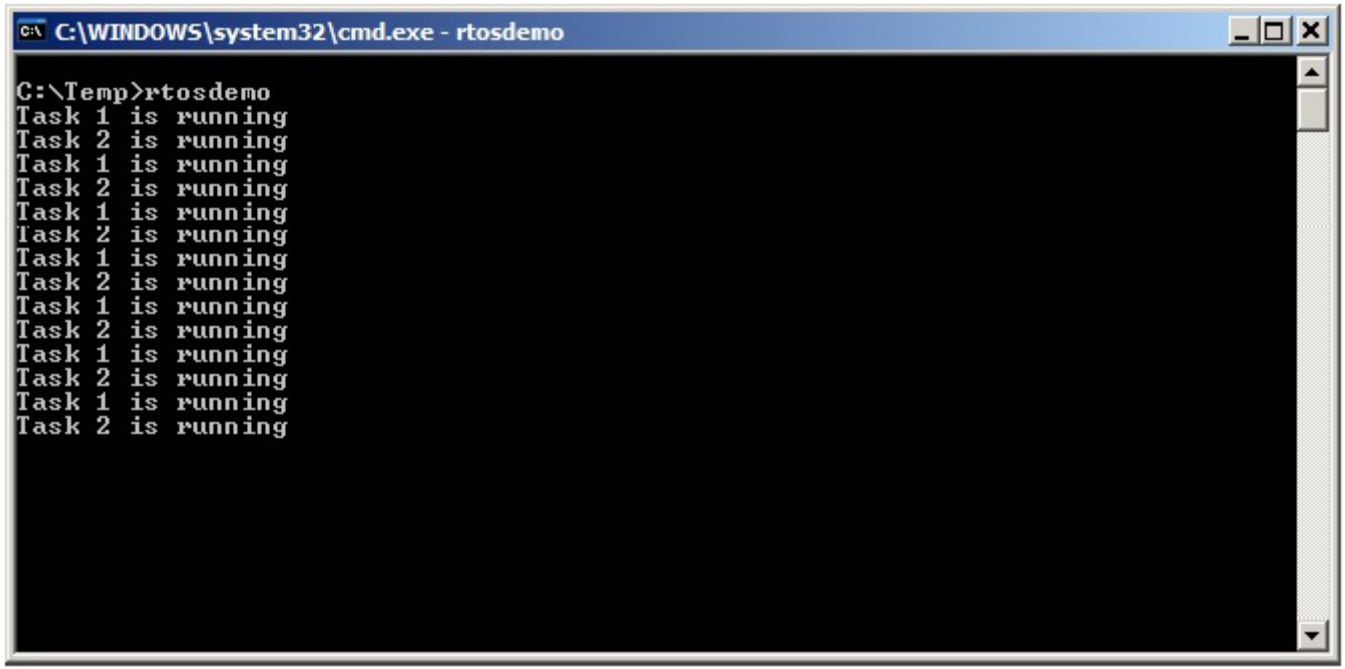


Figure 10. The output produced when Example 1 is executed[1]

źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

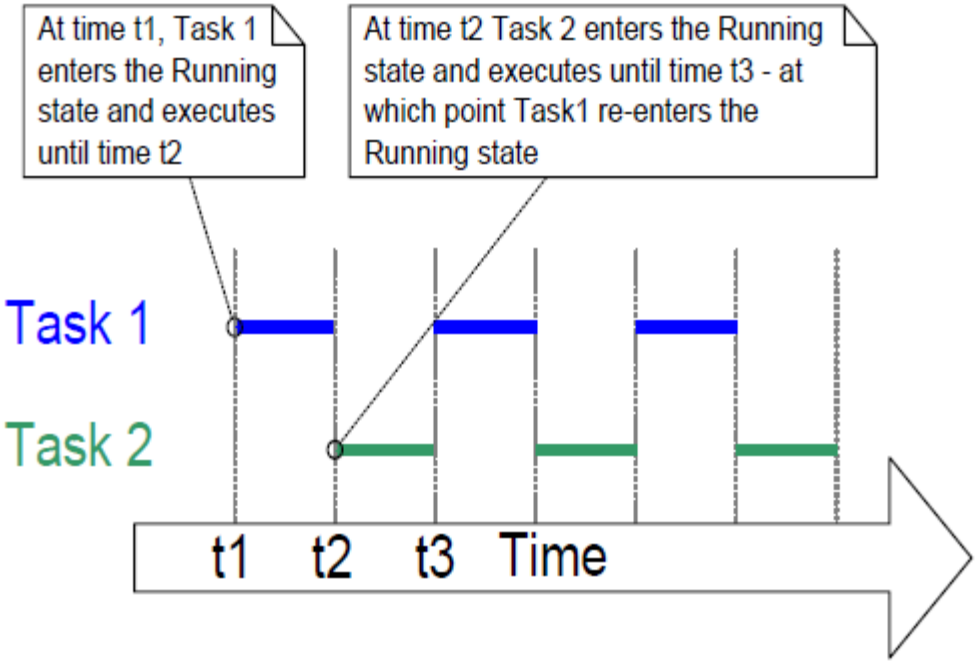
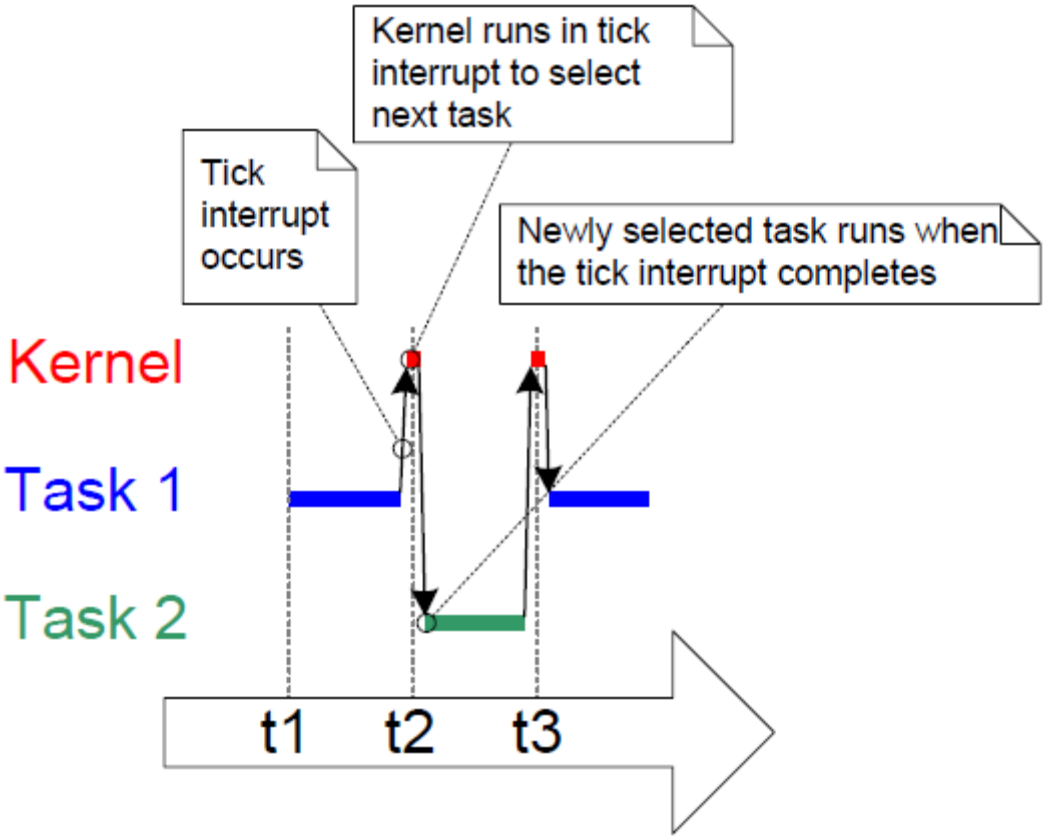


Figure 11. The actual execution pattern of the two Example 1 tasks

źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

Scheduler- planista



źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Wątki - zadania

Możliwe stany zadania

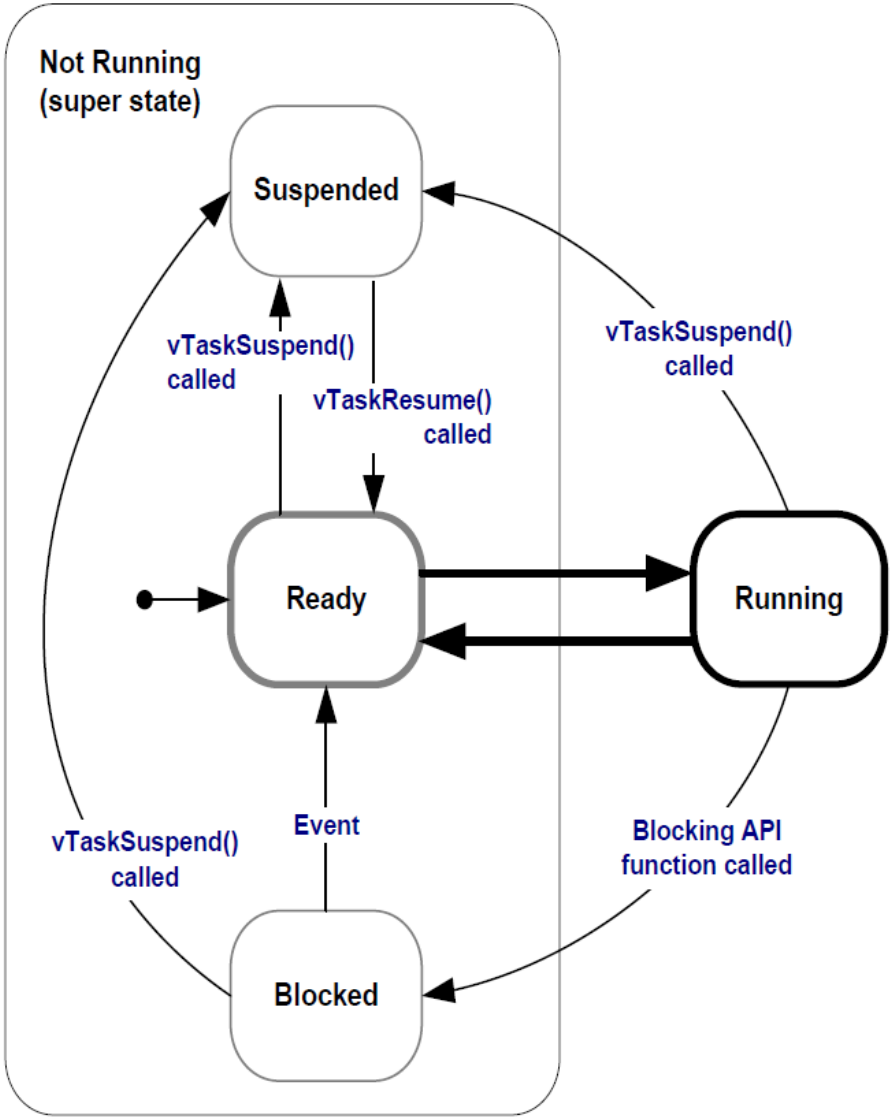


Figure 15. Full task state machine

źródło: Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide

Priorytety

- mechanizm umożliwiający przyznawanie więcej czasu procesora dla wybranego wątku,
- możliwość wywłaszczenia, przerwania zadania przez inne zadanie, które posiada większy priorytet

- problem: „zagłodzenia”

Priorytety

Task1 priorytet: 1, Task2 priorytet: 2

```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

int main( void )
{
    /* Create the first task at priority 1.  The priority is the second to last
    parameter. */
    xTaskCreate( vTaskFunction, "Task 1", 1000, (void*)pcTextForTask1, 1, NULL );

    /* Create the second task at priority 2, which is higher than a priority of 1.
    The priority is the second to last parameter. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* Will not reach here. */
    return 0;
}
```

Priorytety

Task1 priorytet: 1, Task2 priorytet: 2

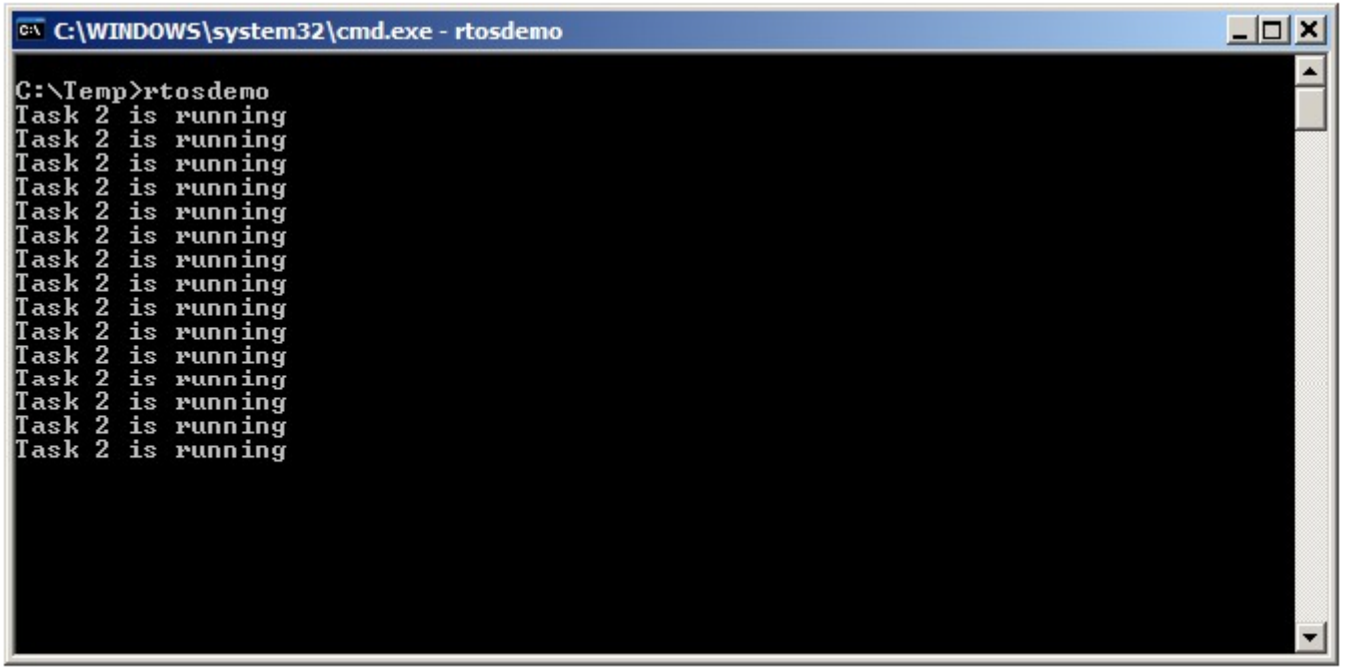
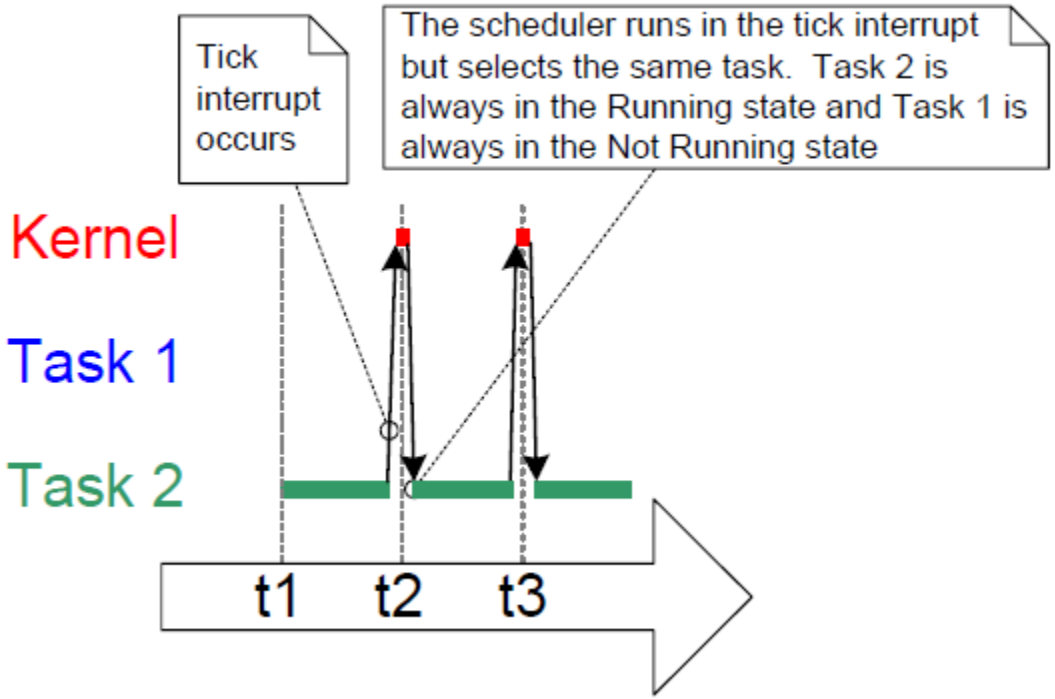


Figure 13. Running both tasks at different priorities

Priorytety

Task1 priorytet: 1, Task2 priorytet: 2



Priorytety

Funkcja vTaskDelay()

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* The string to print out is passed in via the parameter.  Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period.  This time a call to vTaskDelay() is used which places
        the task into the Blocked state until the delay period has expired.  The
        parameter takes a time specified in 'ticks', and the pdMS_TO_TICKS() macro
        is used (where the xDelay250ms constant is declared) to convert 250
        milliseconds into an equivalent time in ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

Priorytety

Funkcja vTaskDelay()

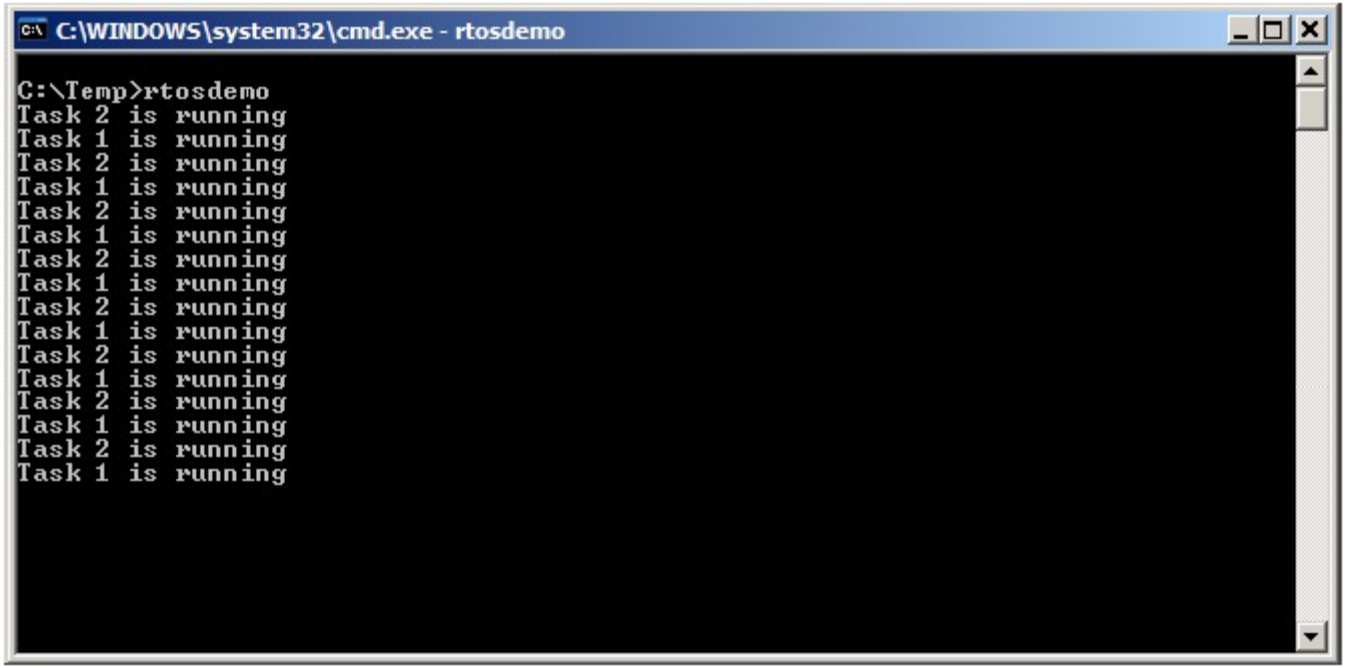


Figure 16. The output produced when Example 4 is executed

Priorytety

Funkcja vTaskDelay()

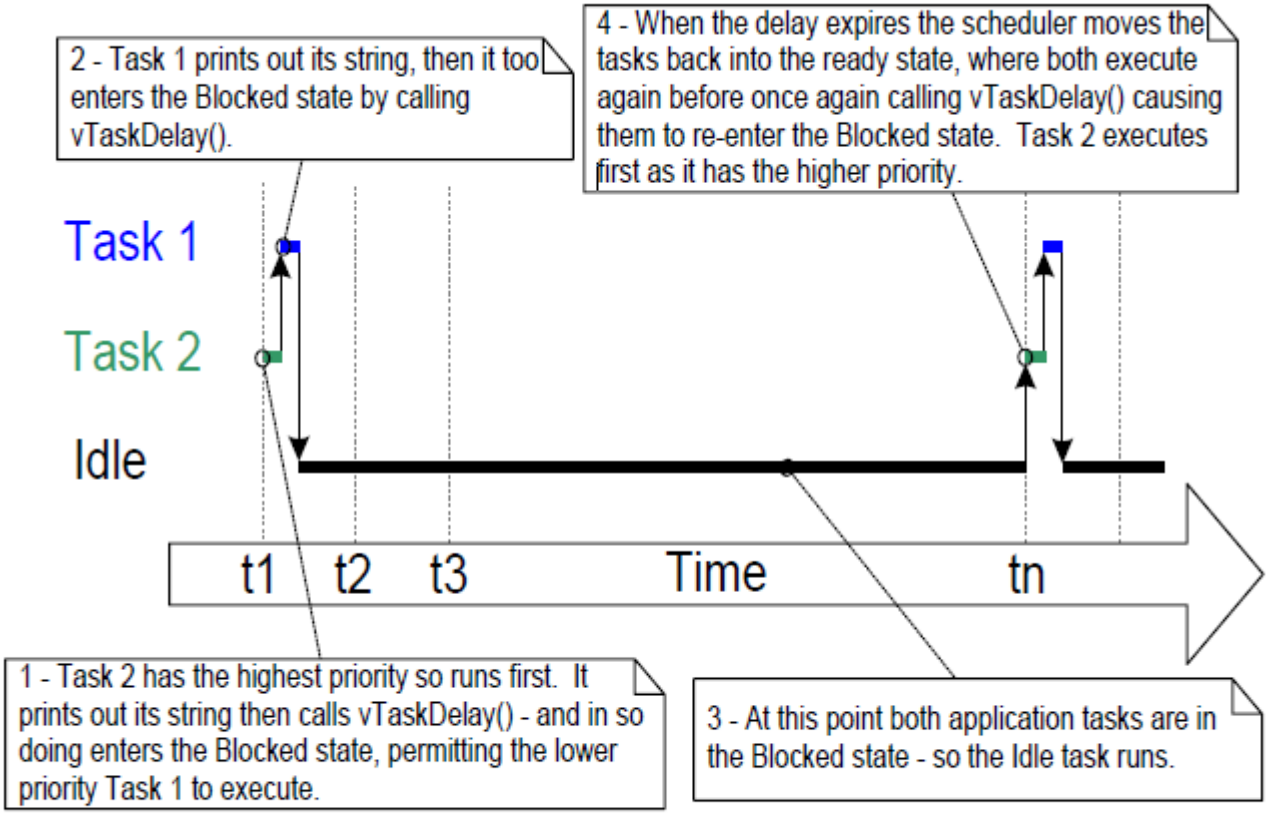
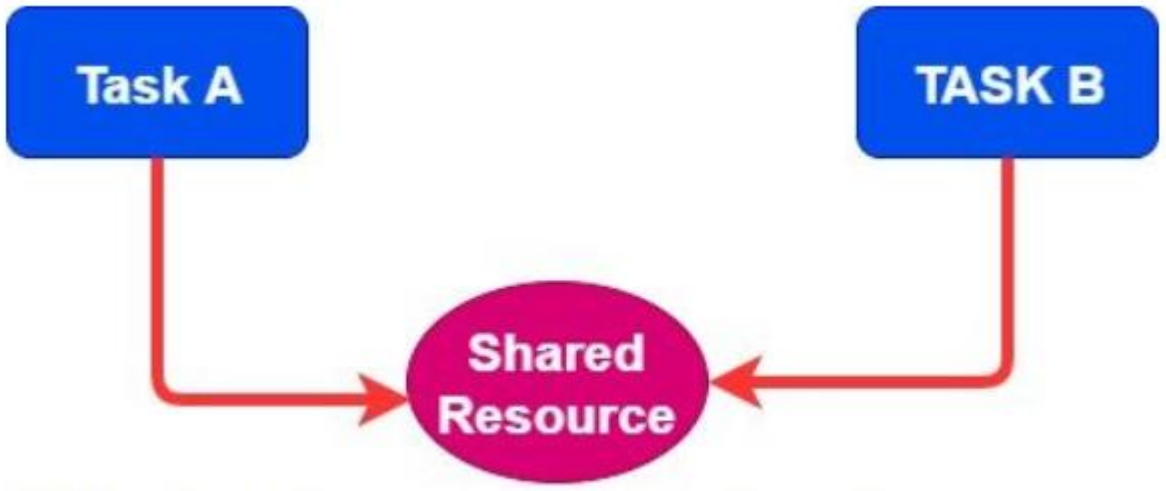


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Problem synchronizacji



Problem synchronizacji

Sekcja krytyczna – fragment kodu programu, w którym korzysta się z zasobu dzielonego, a co za tym idzie, w danej chwili może być wykorzystywany przez co najwyżej jeden wątek. Systemy czasu rzeczywistego posiadają mechanizmy, które dbają o synchronizację. Jeśli więcej wątków żąda wykonania kodu sekcji krytycznej, dopuszczany jest tylko jeden wątek, pozostałe są wstrzymywane. Dąży się do tego, aby kod sekcji krytycznej był krótki – wykonywał się szybko.

Przykłady sekcji krytycznej:

- zabezpieczenie dostępu do zasobów sprzętowych
- zabezpieczenie dostępu do zmiennych

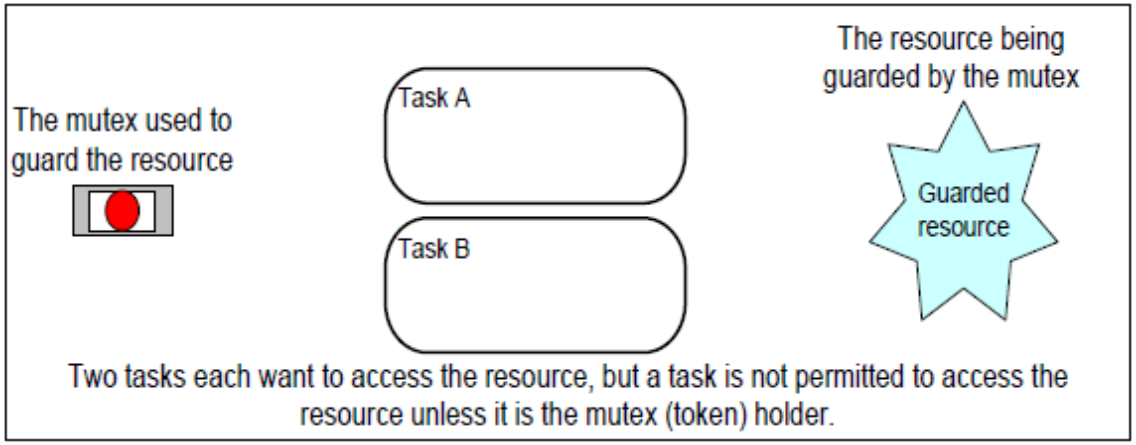
Do zabezpieczania dostępu, synchronizacji wykorzystuje się:

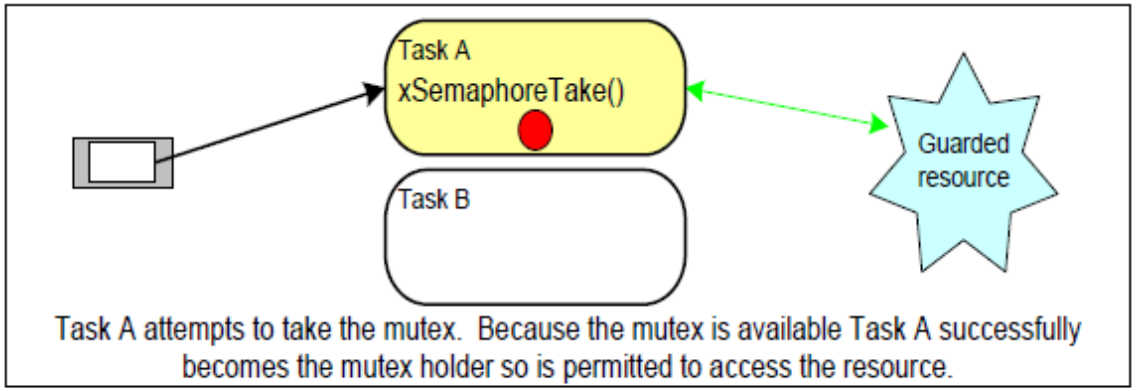
- mutexy, semafory
- kolejki danych
- dedykowane rozwiązania systemu czasu rzeczywistego

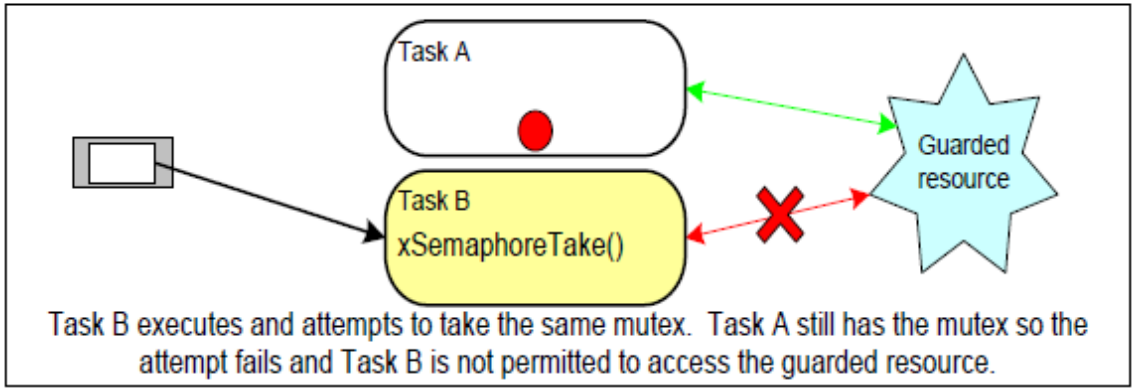
Semafory

mutexy

- mutex
- semafor binarny



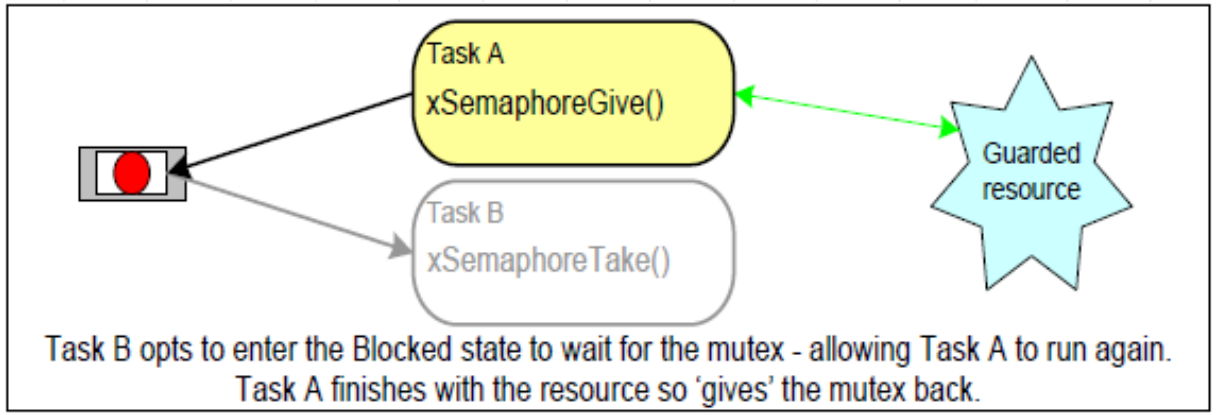




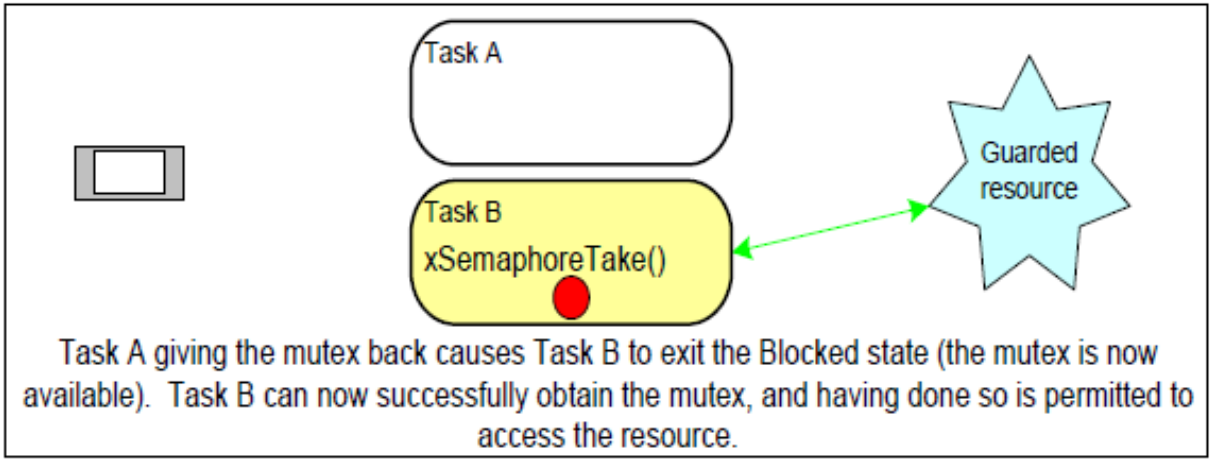
Semafory

mutexy

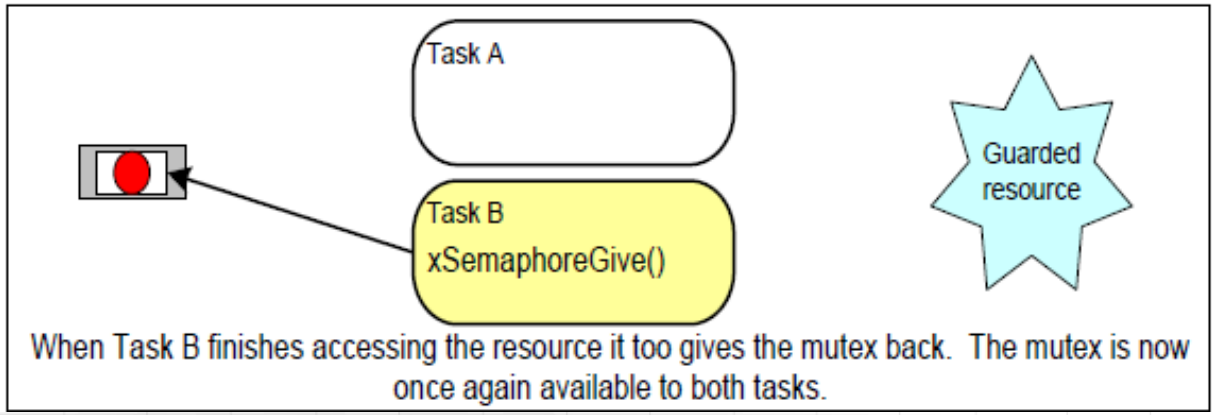
- mutex
- semafor binarny



Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.



Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.



When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

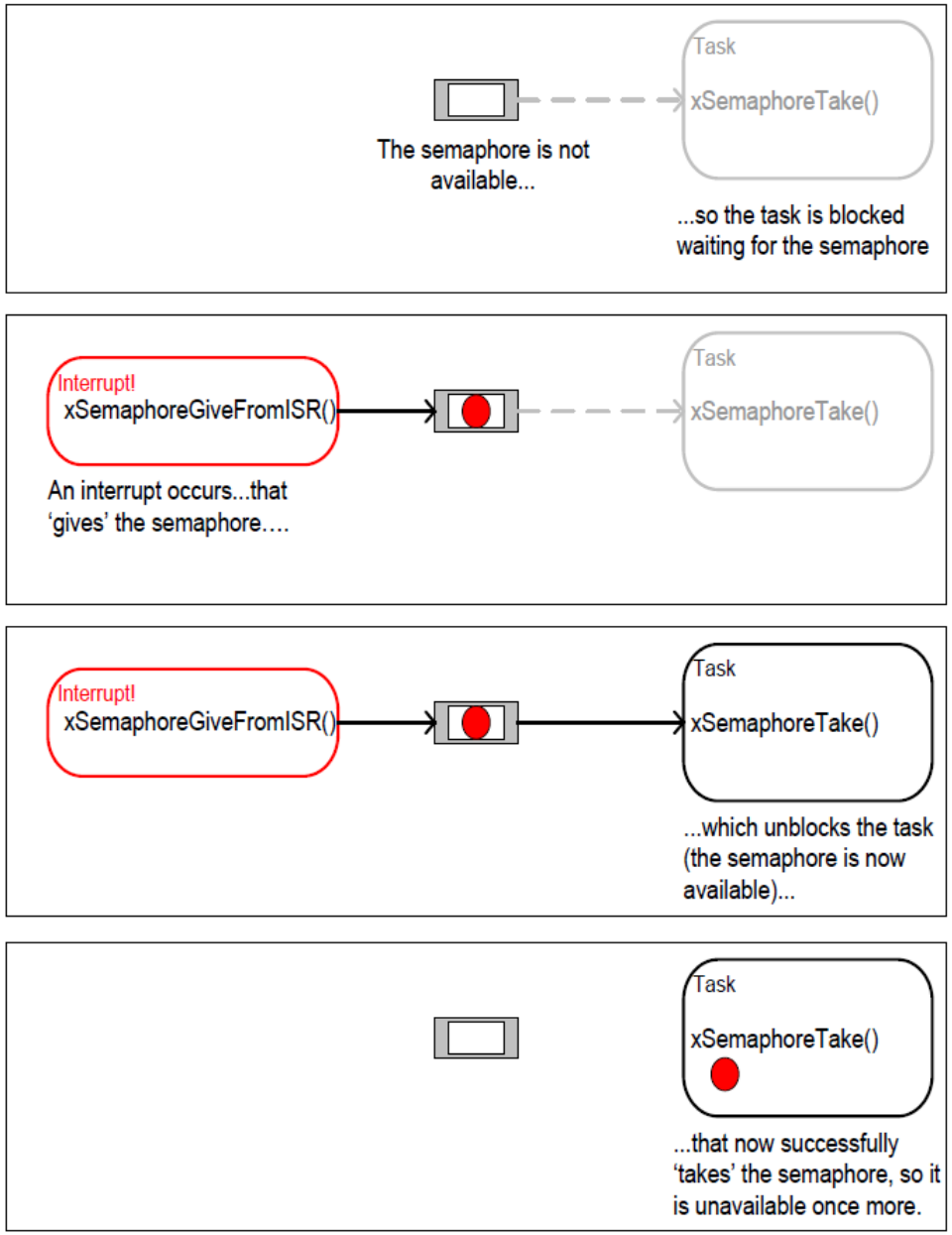
Semafory, mutexy

Jakie jest zagrożenie przy stosowaniu mutexów, semaforów binarnych?

Semafory, mutexy

- semafor binarny
- przerwania

Przykład:
- zdarzenie otwarcia drzwi



Semafory, mutexy

- semafor binarny
- przerwania

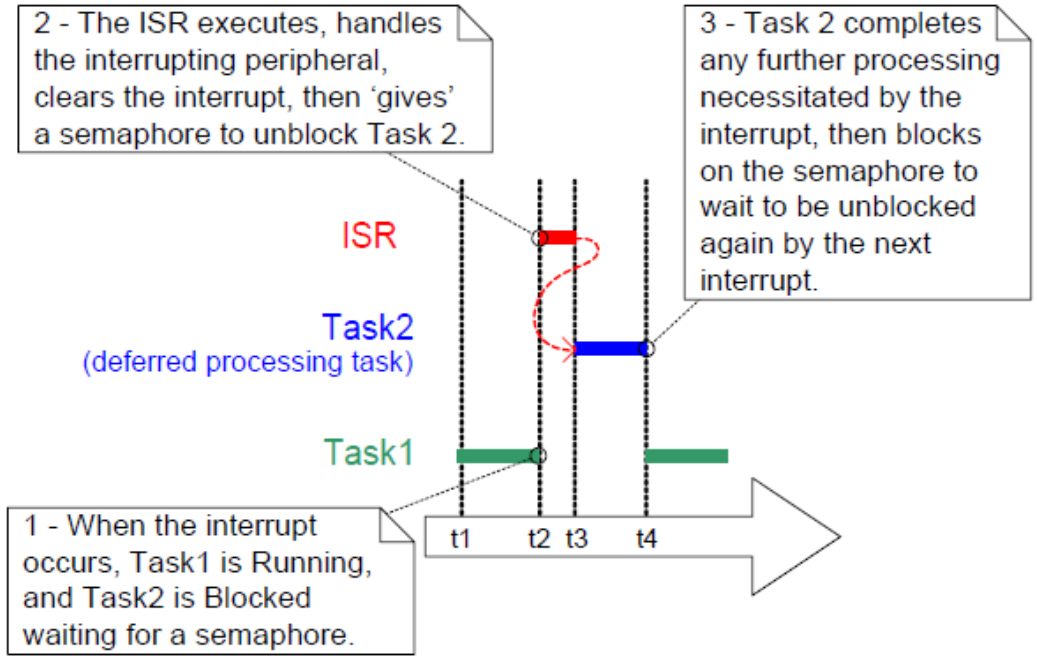


Figure 49. Using a binary semaphore to implement deferred interrupt processing

Semafory, mutexy

- semafor liczący
- przerwania

Przykład:

- baloniki

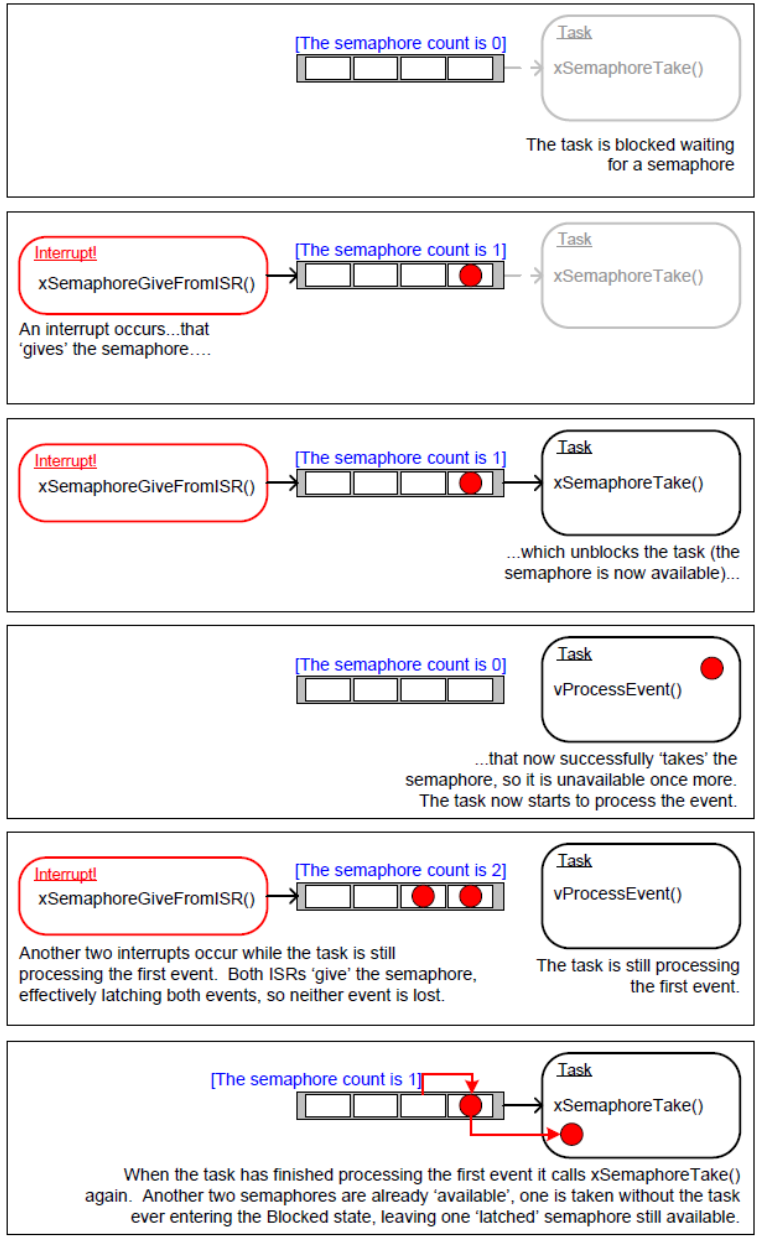


Figure 55. Using a counting semaphore to 'count' events

Kolejki

A co zrobić jak chcemy dzielić dane pomiędzy wątkami?

Klasyczny problem producenta-konsumenta

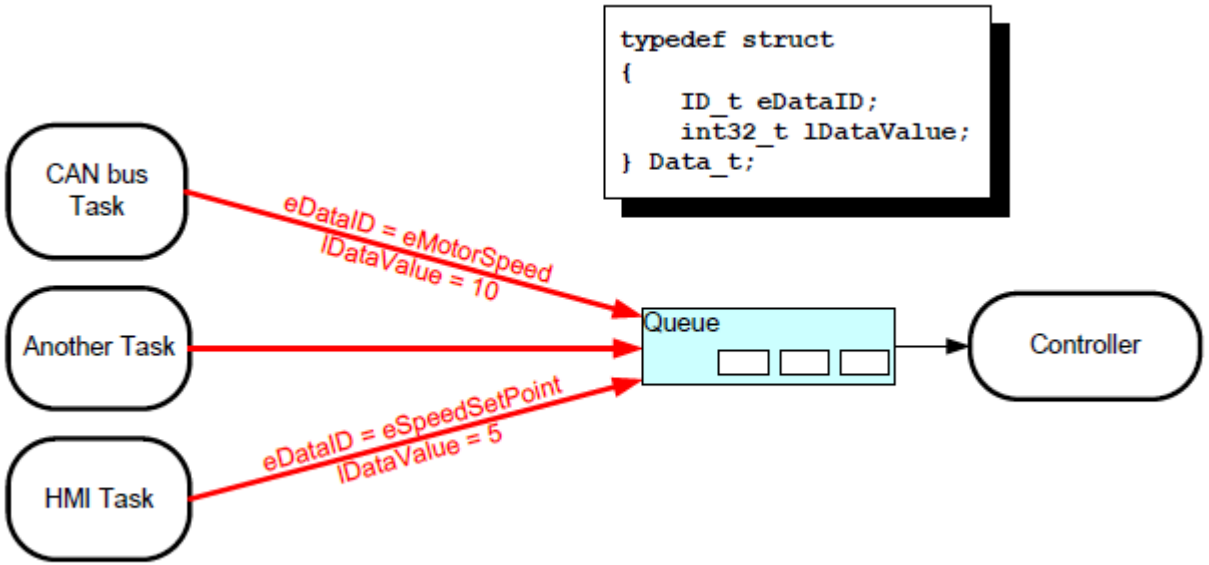
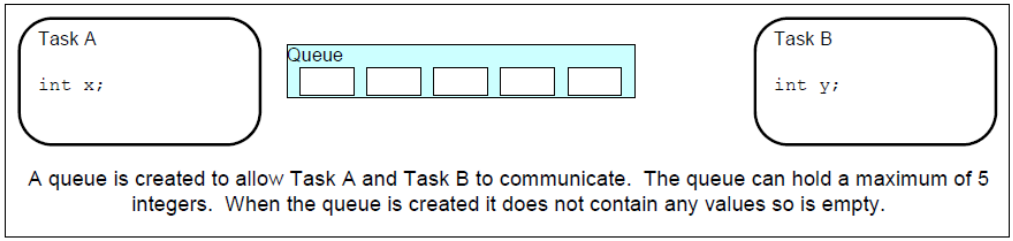


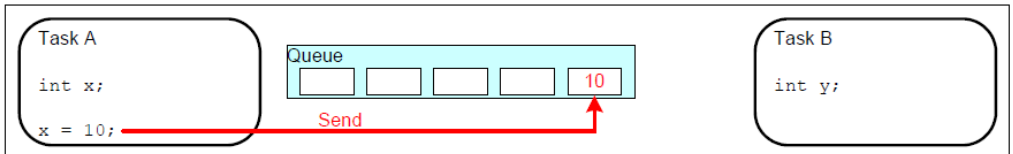
Figure 34. An example scenario where structures are sent on a queue

Kolejki

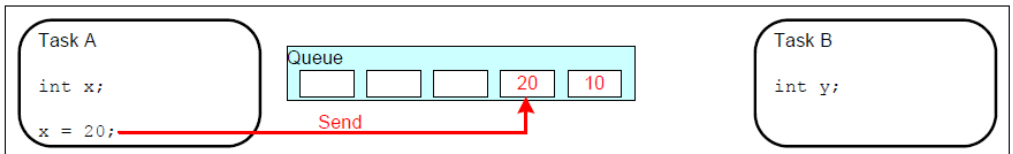
Kolejka FIFO



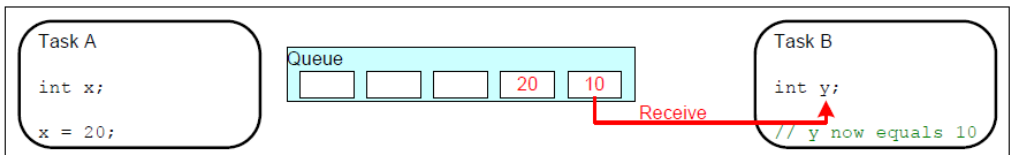
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

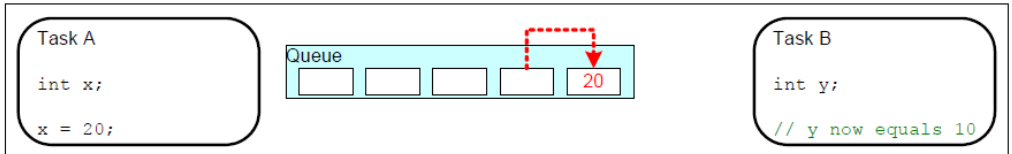


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

Kolejka LIFO

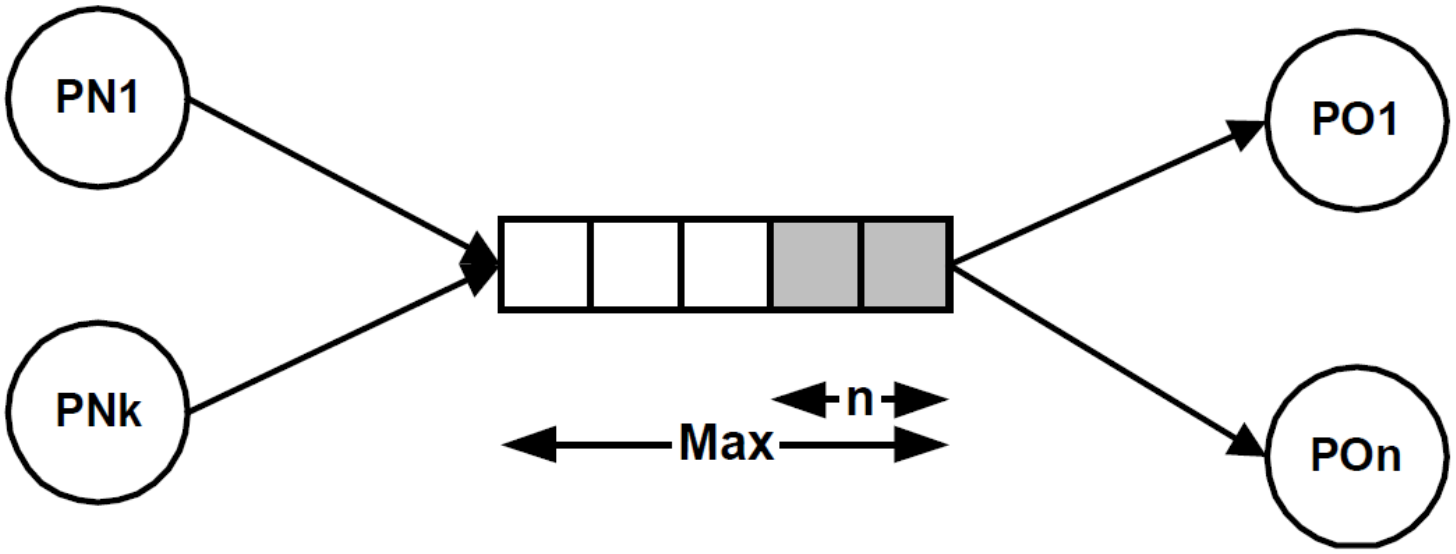


Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Kolejki

2 producentów

2 konsumentów



Procesy nadające

Kolejka Q

Procesy odbierajace

Kolejki

```
/* Define an enumerated type used to identify the source of the data. */
typedef enum
{
    eSender1,
    eSender2
} DataSource_t;

/* Define the structure type that will be passed on the queue. */
typedef struct
{
    uint8_t ucValue;
    DataSource_t eDataSource;
} Data_t;

/* Declare two variables of type Data_t that will be passed on the queue.
static const Data_t xStructsToSend[ 2 ] =
{
    { 100, eSender1 }, /* Used by Sender1. */
    { 200, eSender2 }  /* Used by Sender2. */
};
```

```
static void vSenderTask( void *pvParameters )
{
BaseType_t xStatus;
const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Send to the queue.

        The second parameter is the address of the structure being sent.  The
        address is passed in as the task parameter so pvParameters is used
        directly.

        The third parameter is the Block time - the time the task should be kept
        in the Blocked state to wait for space to become available on the queue
        if the queue is already full.  A block time is specified because the
        sending tasks have a higher priority than the receiving task so the queue
        is expected to become full.  The receiving task will remove items from
        the queue when both sending tasks are in the Blocked state. */
        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete, even after waiting for 100ms.
            This must be an error as the receiving task should make space in the
            queue as soon as both sending tasks are in the Blocked state. */
            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

Kolejki

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the structure that will hold the values received from the queue. */
Data_t xReceivedStructure;
BaseType_t xStatus;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {

        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received
            value and the source of the value. */
            if( xReceivedStructure.eDataSource == eSender1 )
            {
                vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
                vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
        else
        {
            /* Nothing was received from the queue.  This must be an error as this
            task should only run when the queue is full. */
            vPrintString( "Could not receive from the queue.\r\n" );
        }
    }
}
```

Kolejki

```
int main( void )
{
    /* The queue is created to hold a maximum of 3 structures of type Data_t. */
    xQueue = xQueueCreate( 3, sizeof( Data_t ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will write to the queue.  The
        parameter is used to pass the structure that the task will write to the
        queue, so one task will continuously send xStructsToSend[ 0 ] to the queue
        while the other task will continuously send xStructsToSend[ 1 ].  Both
        tasks are created at priority 2, which is above the priority of the receiver. */
        xTaskCreate( vSenderTask, "Sender1", 1000, &( xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &( xStructsToSend[ 1 ] ), 2, NULL );

        /* Create the task that will read from the queue.  The task is created with
        priority 1, so below the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```

Kolejki

The output produced by Example 11 is shown in Figure 35.

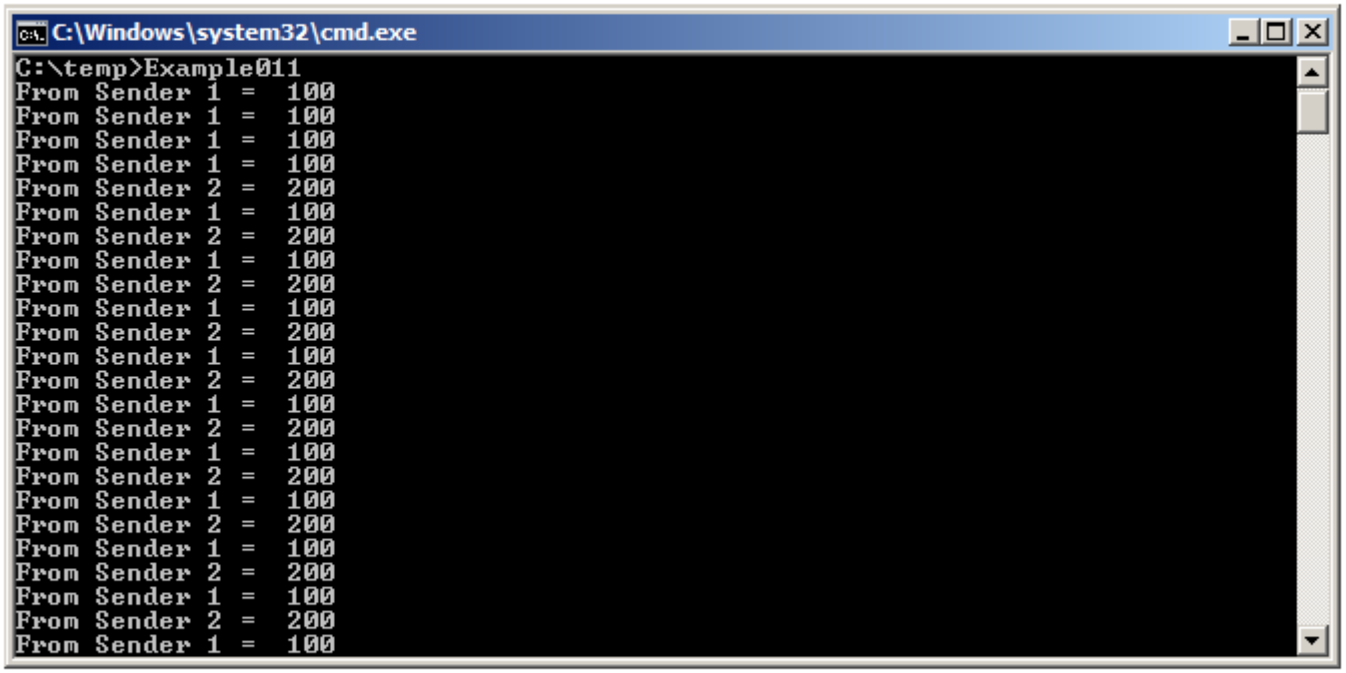


Figure 35 The output produced by Example 11

Kolejki

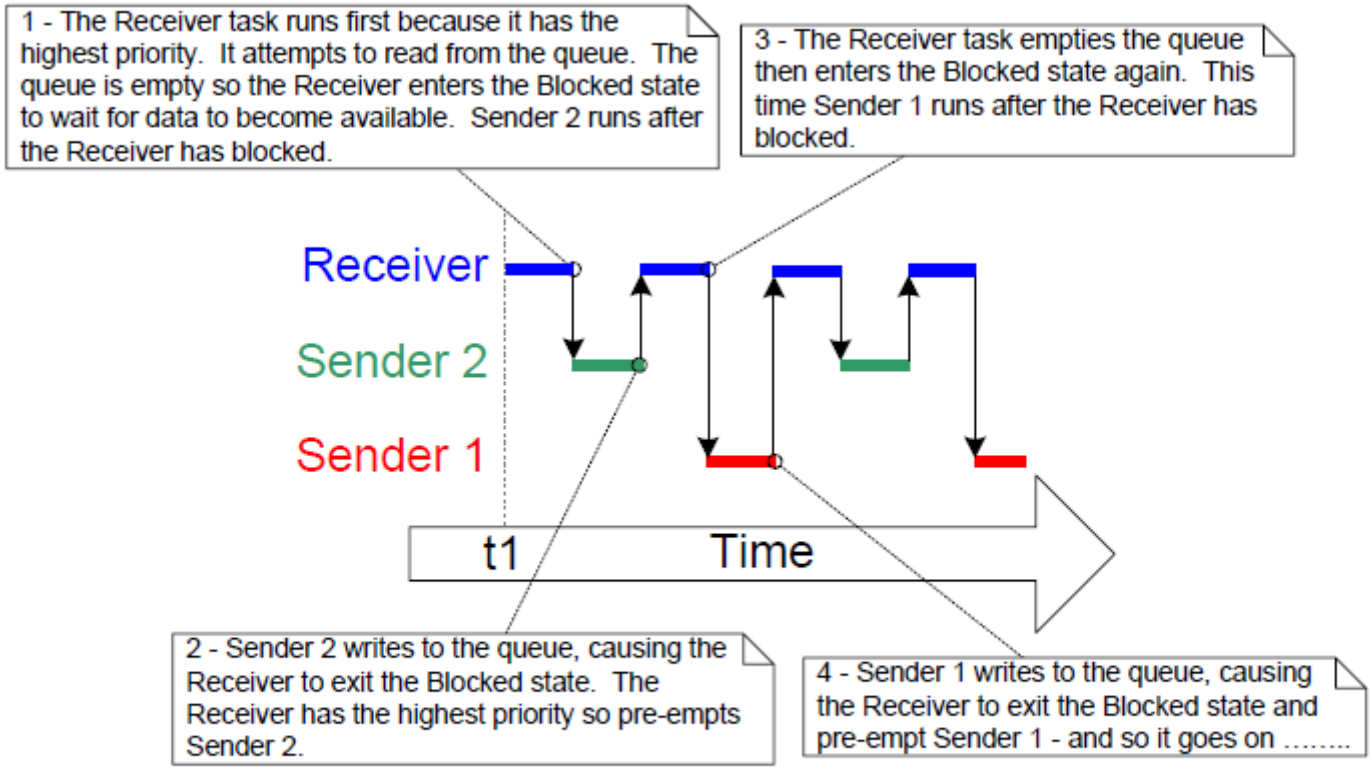


Figure 33. The sequence of execution produced by Example 10

Jaki wybrać mechanizm jak chcemy dzielić pomiędzy wątkami tylko aktualną daną?

Scenariusz:

1. wątek pomiarowy – mierzy napięcie na przetworniku ADC
2. wątek sterujący – na podstawie zmierzonego napięcia ustawia wyjście sterująca w odpowiednim stanie
3. wątek logujący – co 1 sekundę odczytuje stan napięcia i zapisuje informacje o jego wartości na karcie SD
4. wątek komunikacyjny – wysyła aktualne napięcie po UART

Kolejka jednoelementowa – „mailbox”

```
/* A mailbox can hold a fixed size data item.  The size of the data item is set
when the mailbox (queue) is created.  In this example the mailbox is created to
hold an Example_t structure.  Example_t includes a time stamp to allow the data held
in the mailbox to note the time at which the mailbox was last updated.  The time
stamp used in this example is for demonstration purposes only - a mailbox can hold
any data the application writer wants, and the data does not need to include a time
stamp. */
typedef struct xExampleStructure
{
    TickType_t xTimeStamp;
    uint32_t ulValue;
} Example_t;


/* A mailbox is a queue, so its handle is stored in a variable of type
QueueHandle_t. */
QueueHandle_t xMailbox;

void vAFunction( void )
{
    /* Create the queue that is going to be used as a mailbox.  The queue has a
    length of 1 to allow it to be used with the xQueueOverwrite() API function, which
    is described below. */
    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );

}
```

Kolejka jednoelementowa – „mailbox”

```
void vUpdateMailbox( uint32_t ulNewValue )
{
/* Example_t was defined in Listing 67. */
Example_t xData;

    /* Write the new data into the Example_t structure.*/
    xData.ulValue = ulNewValue;

    /* Use the RTOS tick count as the time stamp stored in the Example_t structure. */
    xData.xTimeStamp = xTaskGetTickCount();

    /* Send the structure to the mailbox - overwriting any data that is already in the
    mailbox. */
    xQueueOverwrite( xMailbox, &xData );
}
```

stosujemy xQueueOverwrite() a nie xQueueSendToBack()

Kolejka jednoelementowa – „mailbox”

```
BaseType_t vReadMailbox( Example_t *pxData )
{
TickType_t xPreviousTimeStamp;
BaseType_t xDataUpdated;

    /* This function updates an Example_t structure with the latest value received
    from the mailbox.  Record the time stamp already contained in *pxData before it
    gets overwritten by the new data. */
    xPreviousTimeStamp = pxData->xTimeStamp;

    /* Update the Example_t structure pointed to by pxData with the data contained in
    the mailbox.  If xQueueReceive() was used here then the mailbox would be left
    empty, and the data could not then be read by any other tasks.  Using
    xQueuePeek() instead of xQueueReceive() ensures the data remains in the mailbox.
    A block time is specified, so the calling task will be placed in the Blocked
    state to wait for the mailbox to contain data should the mailbox be empty.  An
    infinite block time is used, so it is not necessary to check the value returned
    from xQueuePeek(), as xQueuePeek() will only return when data is available. */
    xQueuePeek( xMailbox, pxData, portMAX_DELAY );

    /* Return pdTRUE if the value read from the mailbox has been updated since this
    function was last called.  Otherwise return pdFALSE. */
    if( pxData->xTimeStamp > xPreviousTimeStamp )
    {
        xDataUpdated = pdTRUE;
    }
    else
    {
        xDataUpdated = pdFALSE;
    }

    return xDataUpdated;
}
```

stosujemy xQueuePeek() a nie xQueueReceive()

Podsumowanie

Zalety

- Stosowanie mechanizmów systemów czasu rzeczywistego ułatwia pisanie większych programów.
- Umożliwia łatwe wydzieleni części programu i przekazanie go różnym programistą do implementacji, łatwiejsza praca zespołowa.
- Przez podział na moduły program łatwiej rozwijać i testować.

Wady:

- Wymaga większych zasobów sprzętowych.
- Wymaga poznania architektury wykorzystywanego systemu i jego funkcji.
- Komplikuje proces wymiany danych pomiędzy zadaniami przez konieczność użycia odpowiednich struktur i funkcji
- Dzielenie zasobów sprzętowych należy realizować z wykorzystaniem sekcji krytycznych – konieczność stosowanie dodatkowych mechanizmów.