

---

CSCE 4114  
Queue Management

David Andrews

[dandrews@uark.edu](mailto:dandrews@uark.edu)



# Agenda

---

- How to Create a Queue
- How a Queue Manages Data it Contains
- How to Send/Receive Data on Queue
- Blocking on a Queue
- Task Priorities When Writing/Reading
- Blocking on Multiple Queues
- Overwriting Data in a Queue



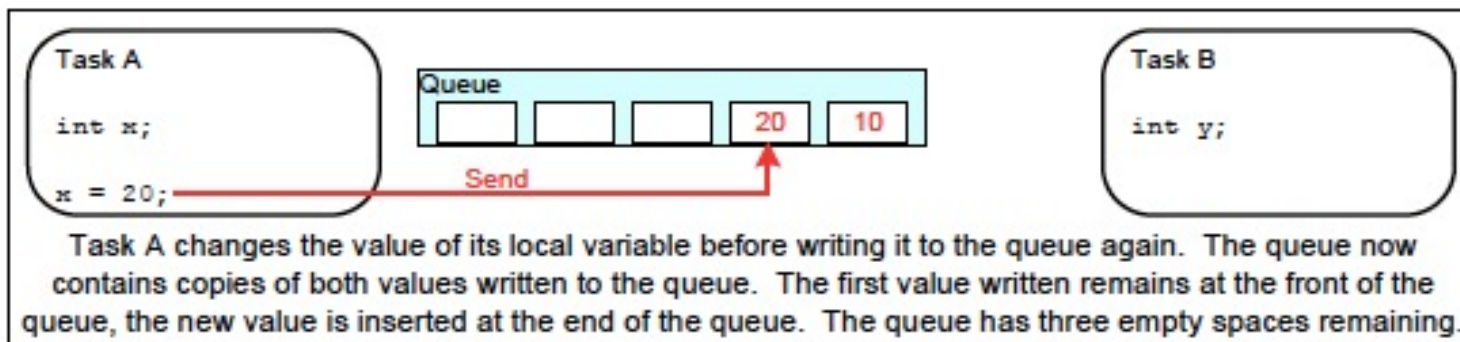
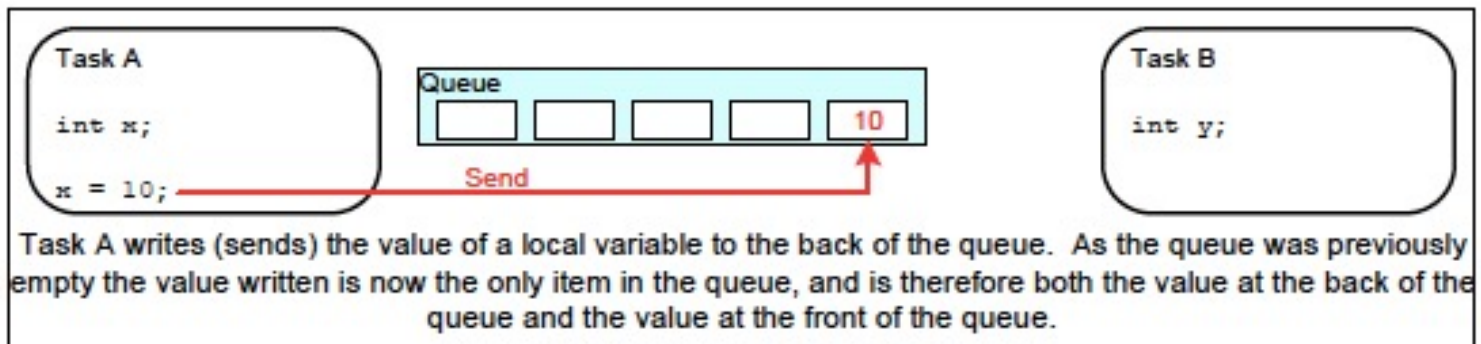
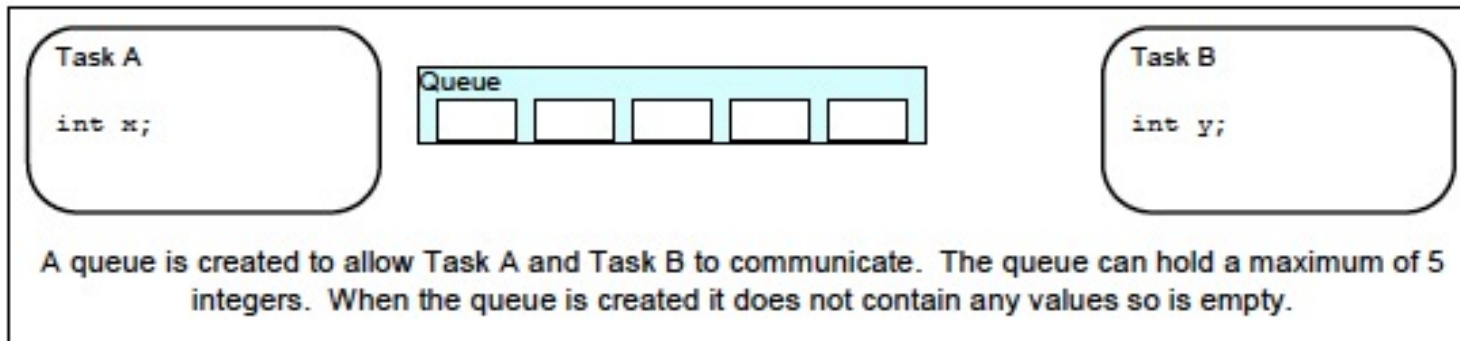
# Queue Basics

---

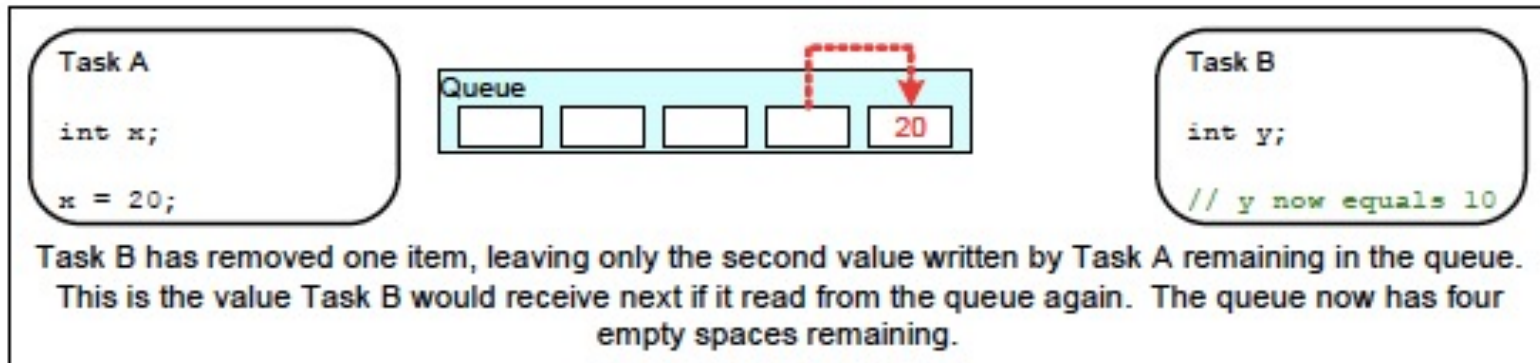
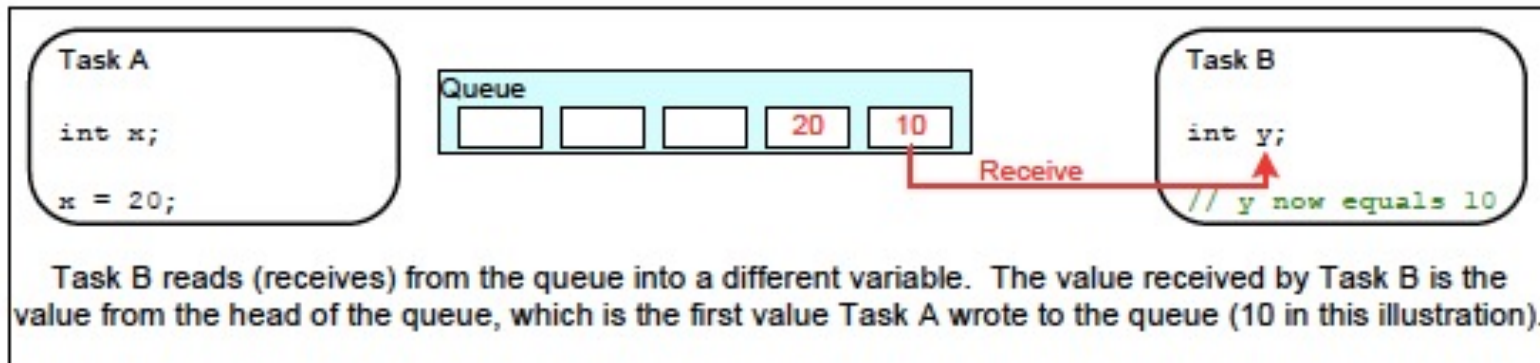
- Queues are Primatively:
- Message channels with storage
  - User Defined Depths and Data Types
- Synchronization Mechanisms
  - Can Block and Synch like Semaphores
- Queue's are Default FIFO Structures....  
but operations provided for Stack Type  
operations.....



# Queue Basics



# Queue Basics



# xQueueCreate() API

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Listing 40. The xQueueCreate() API function prototype

Table 18. xQueueCreate() parameters and return value

Parameter Name	Description
uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size in bytes of each data item that can be stored in the queue.
Return Value	If NULL is returned, then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.

A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.





# xQueueCreate() API

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );
```

Listing 40. The xQueueCreate() API function prototype

Table 18. xQueueCreate() parameters and return value

Parameter Name	Description
uxQueueLength	The maximum number of items that the queue being created can hold at any one time.
uxItemSize	The size in bytes of each data item that can be stored in the queue.
Return Value	If NULL is returned, then the queue cannot be created because there is insufficient heap memory available for FreeRTOS to allocate the queue data structures and storage area.

A non-NULL value being returned indicates that the queue has been created successfully. The returned value should be stored as the handle to the created queue.



# Writing into the Queue

---

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,  
                             const void * pvItemToQueue,  
                             TickType_t xTicksToWait );
```

---

Listing 41. The xQueueSendToFront() API function prototype

---

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,  
                             const void * pvItemToQueue,  
                             TickType_t xTicksToWait );
```

---

Listing 42. The xQueueSendToBack() API function prototype

---

xQueueSend()      Identical behaviors





# Writing into the Queue

---

```
BaseType_t xQueueSend      (QueueHandle_t Xqueue,  
                             const void * pvItemToQueue,  
                             TickType_t xTicksToWait);
```



# Writing into the Queue

---

BaseType\_t xQueueSend (QueueHandle\_t Xqueue,  
const void \* pvItemToQueue,  
TickType\_t xTicksToWait);

xQueue

The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.



# Writing into the Queue

---

BaseType\_t xQueueSend (QueueHandle\_t Xqueue,  
const void \* pvItemToQueue,  
TickType\_t xTicksToWait);

**xQueue** The handle of the queue to which the data is being sent (written). The queue handle will have been returned from the call to xQueueCreate() used to create the queue.

**pvItemToQueue** A pointer to the data to be copied into the queue.  
  
The size of each item that the queue can hold is set when the queue is created, so this many bytes will be copied from pvItemToQueue into the queue storage area.



# Writing into the Queue

---

BaseType\_t xQueueSend (QueueHandle\_t Xqueue,  
const void \* pvItemToQueue,  
TickType\_t xTicksToWait);

xTicksToWait      The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.



# Writing into the Queue

---

BaseType\_t xQueueSend (QueueHandle\_t Xqueue,  
const void \* pvItemToQueue,  
TickType\_t xTicksToWait);

xTicksToWait      The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.

Both xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is zero and the queue is already full.



# Writing into the Queue

---

BaseType\_t xQueueSend (QueueHandle\_t Xqueue,  
const void \* pvItemToQueue,  
TickType\_t xTicksToWait);

xTicksToWait      The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, should the queue already be full.

Both xQueueSendToFront() and xQueueSendToBack() will return immediately if xTicksToWait is zero and the queue is already full.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out), provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.





# Writing into the Queue

---

```
BaseType_t xQueueSend      (QueueHandle_t Xqueue,  
                             const void * pvItemToQueue,  
                             TickType_t xTicksToWait);
```

Key Idea\* Task will be suspended if trying to place new data into a full queue.....

Can use this to synchronize: how ?



# Reading (Receiving) from the Queue

---

```
BaseType_t xQueueReceive (QueueHandle_t Xqueue,  
                          const void * pvBuffer,  
                          TickType_t xTicksToWait);
```

xQueue

The handle of the queue from which the data is being received (read).  
The queue handle will have been returned from the call to  
xQueueCreate() used to create the queue.



# Reading (Receiving) from the Queue

---

BaseType\_t xQueueReceive (QueueHandle\_t Xqueue,  
const void \* pvBuffer,  
TickType\_t xTicksToWait);

**xQueue**            The handle of the queue from which the data is being received (read).  
The queue handle will have been returned from the call to  
xQueueCreate() used to create the queue.

**pvBuffer**            A pointer to the memory into which the received data will be copied.

The size of each data item that the queue holds is set when the queue  
is created. The memory pointed to by pvBuffer must be at least large  
enough to hold that many bytes.



# Reading (Receiving) from the Queue

---

```
BaseType_t xQueueReceive (QueueHandle_t Xqueue,  
const void * pvBuffer,  
TickType_t xTicksToWait);
```

**xTicksToWait**      The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.

If **xTicksToWait** is zero, then **xQueueReceive()** will return immediately if the queue is already empty.

Setting **xTicksToWait** to **portMAX\_DELAY** will cause the task to wait indefinitely (without timing out) provided **INCLUDE\_vTaskSuspend** is set to 1 in **FreeRTOSConfig.h**.



# Reading (Receiving) from the Queue

---

```
BaseType_t xQueueReceive (QueueHandle_t Xqueue,  
                          const void * pvBuffer,  
                          TickType_t xTicksToWait);
```



# Reading (Receiving) from the Queue

---

```
BaseType_t xQueueReceive (QueueHandle_t Xqueue,  
                          const void * pvBuffer,  
                          TickType_t xTicksToWait);
```

**xTicksToWait**      The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.





# Reading (Receiving) from the Queue

---

```
BaseType_t xQueueReceive (QueueHandle_t Xqueue,  
                          const void * pvBuffer,  
                          TickType_t xTicksToWait);
```

**xTicksToWait**      The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.

If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty.



# Reading (Receiving) from the Queue

---

```
BaseType_t xQueueReceive (QueueHandle_t Xqueue,  
const void * pvBuffer,  
TickType_t xTicksToWait);
```

**xTicksToWait**      The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.

If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.



# Reading (Receiving) from the Queue

---

BaseType\_t xQueueReceive (QueueHandle\_t Xqueue,  
const void \* pvBuffer,  
TickType\_t xTicksToWait);

**xTicksToWait**      The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, should the queue already be empty.

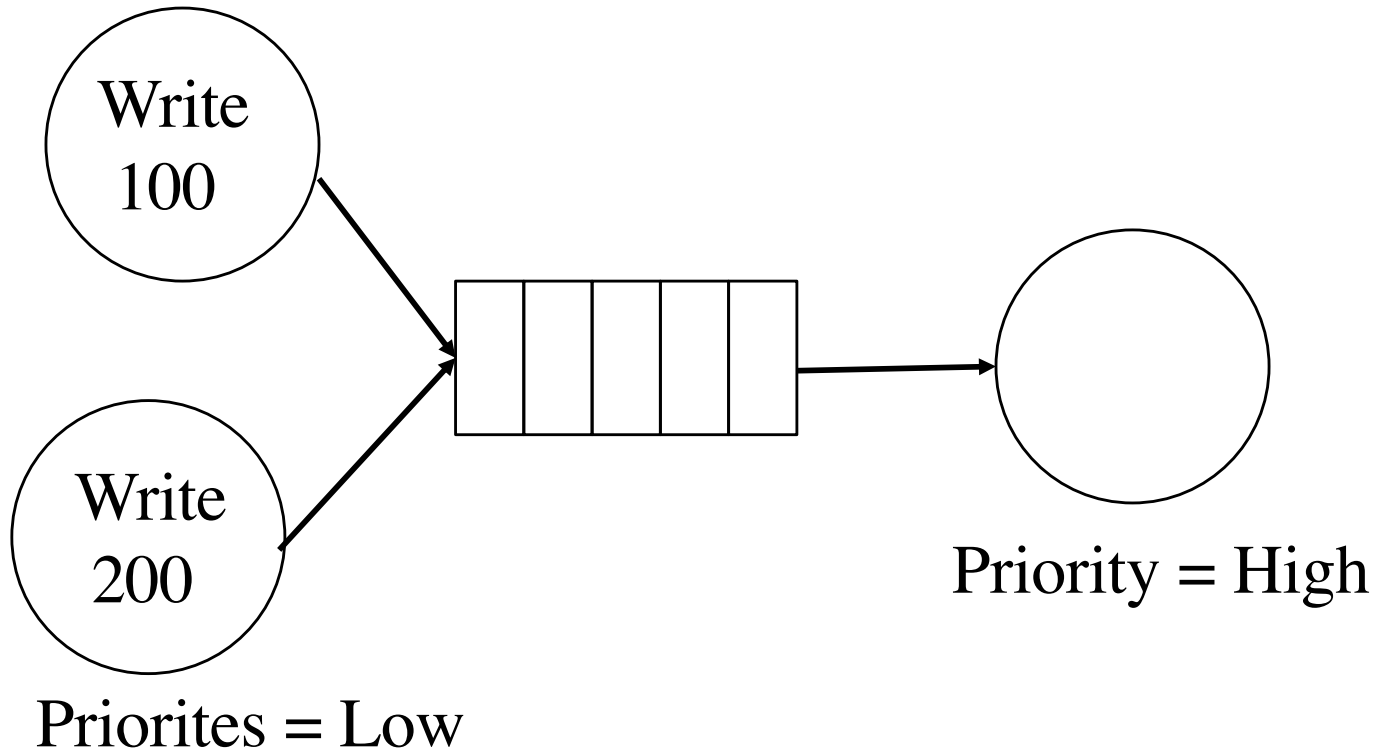
If xTicksToWait is zero, then xQueueReceive() will return immediately if the queue is already empty.

Setting xTicksToWait to portMAX\_DELAY will cause the task to wait indefinitely (without timing out) provided INCLUDE\_vTaskSuspend is set to 1 in FreeRTOSConfig.h.



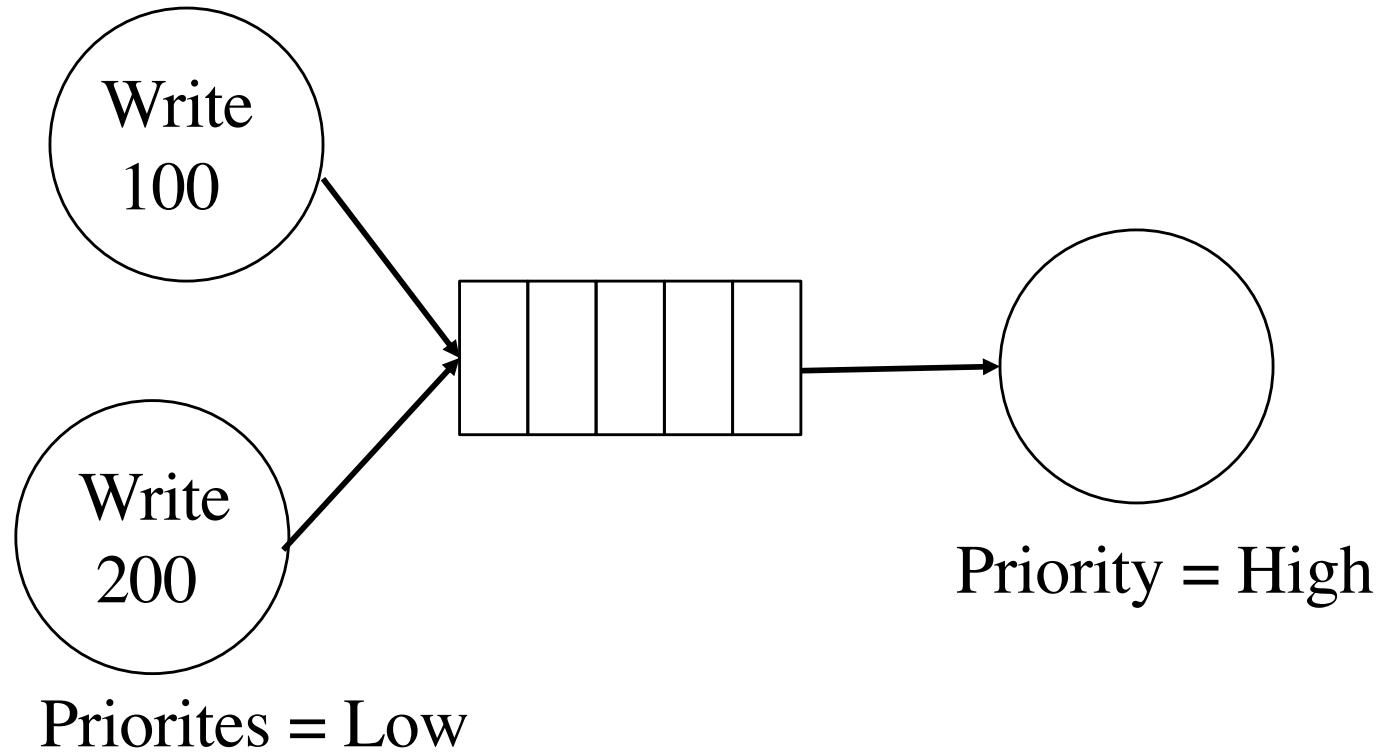
# Lets Play.....

---



# Lets Play.....

---

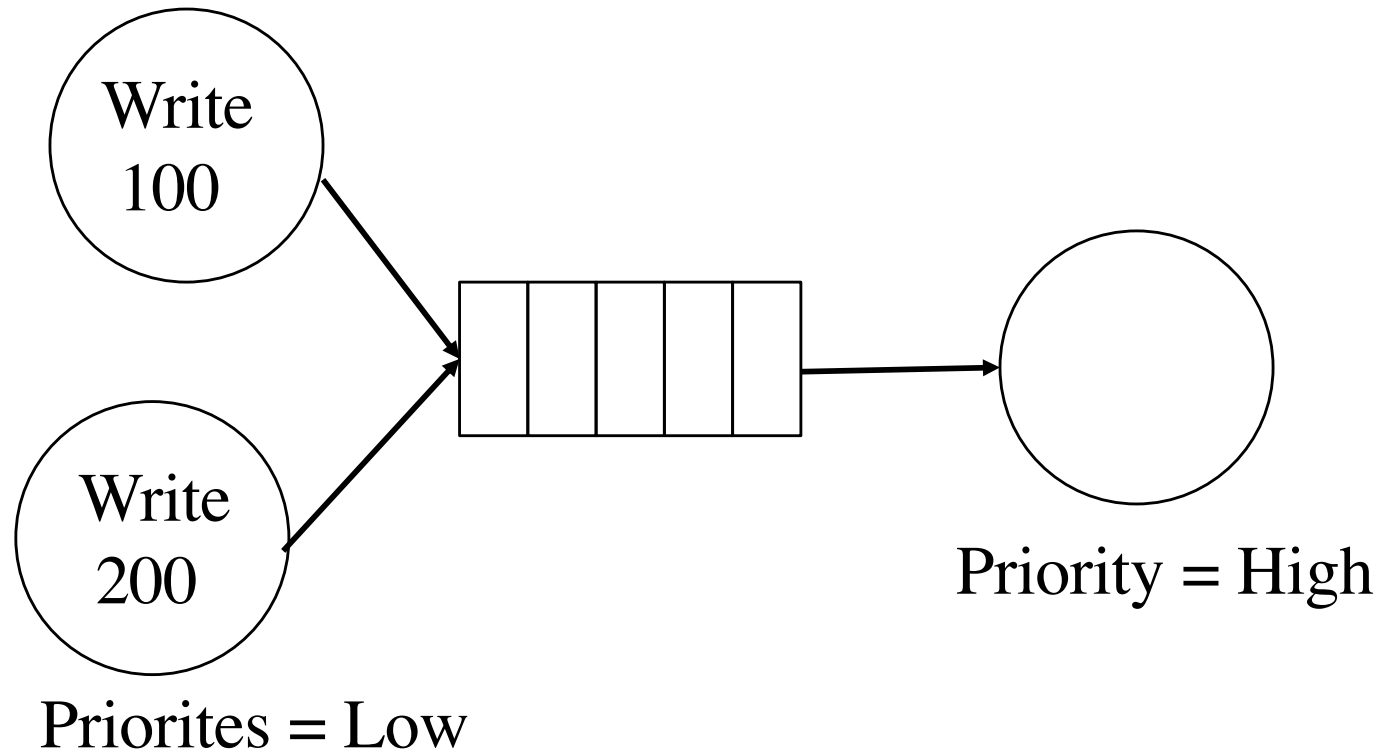


1. Receiver starts by reading and is blocked



# Lets Play.....

---



1. Receiver starts by reading and is blocked
2. Task 1 writes into Queue.....What happens ?





# Lets Play.....

---

Receiver

Sender2

Sender1

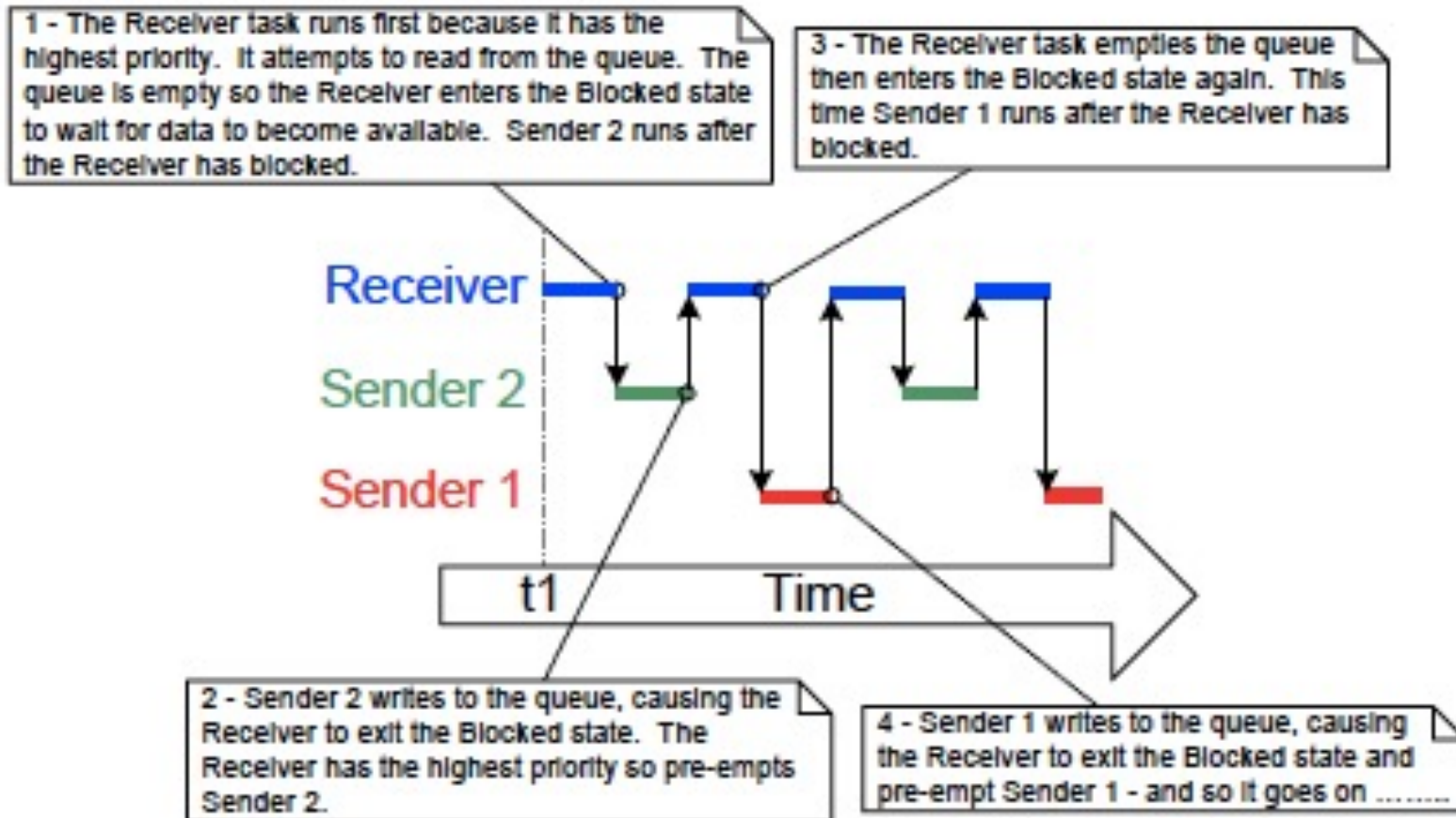


t1

Time



# Lets Play.....



# Expanding a Little....

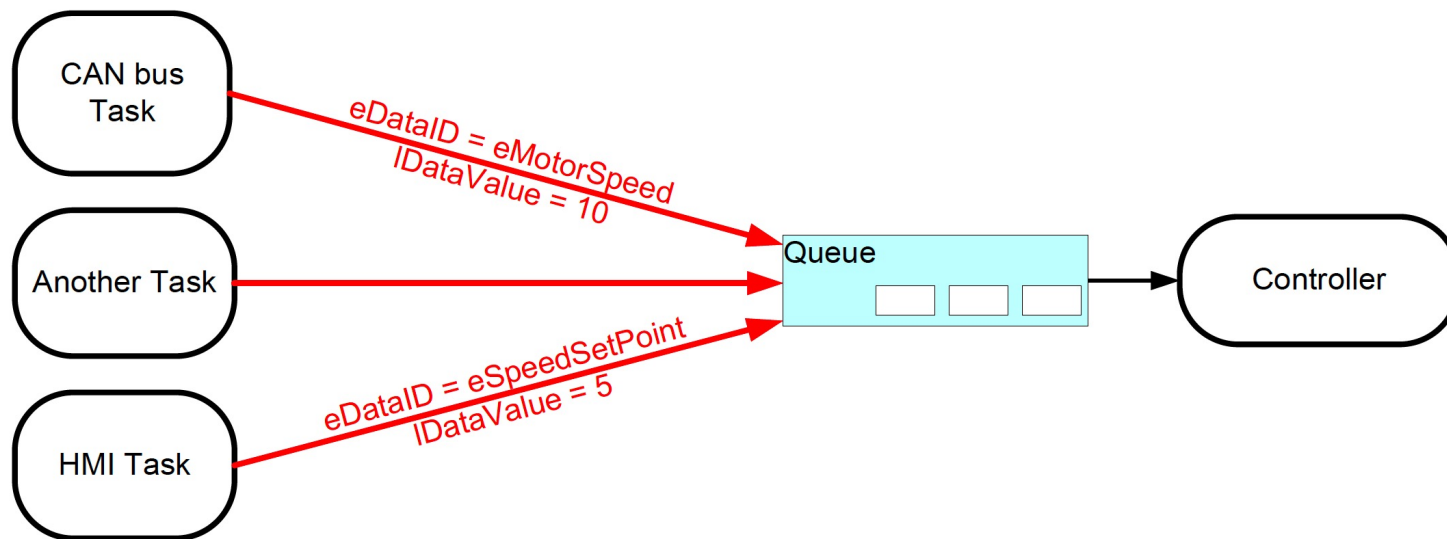


Figure 34. An example scenario where structures are sent on a queue

Controller processing multiple input streams

Q: How to tell which entry is from what source ?



# Expanding a Little....

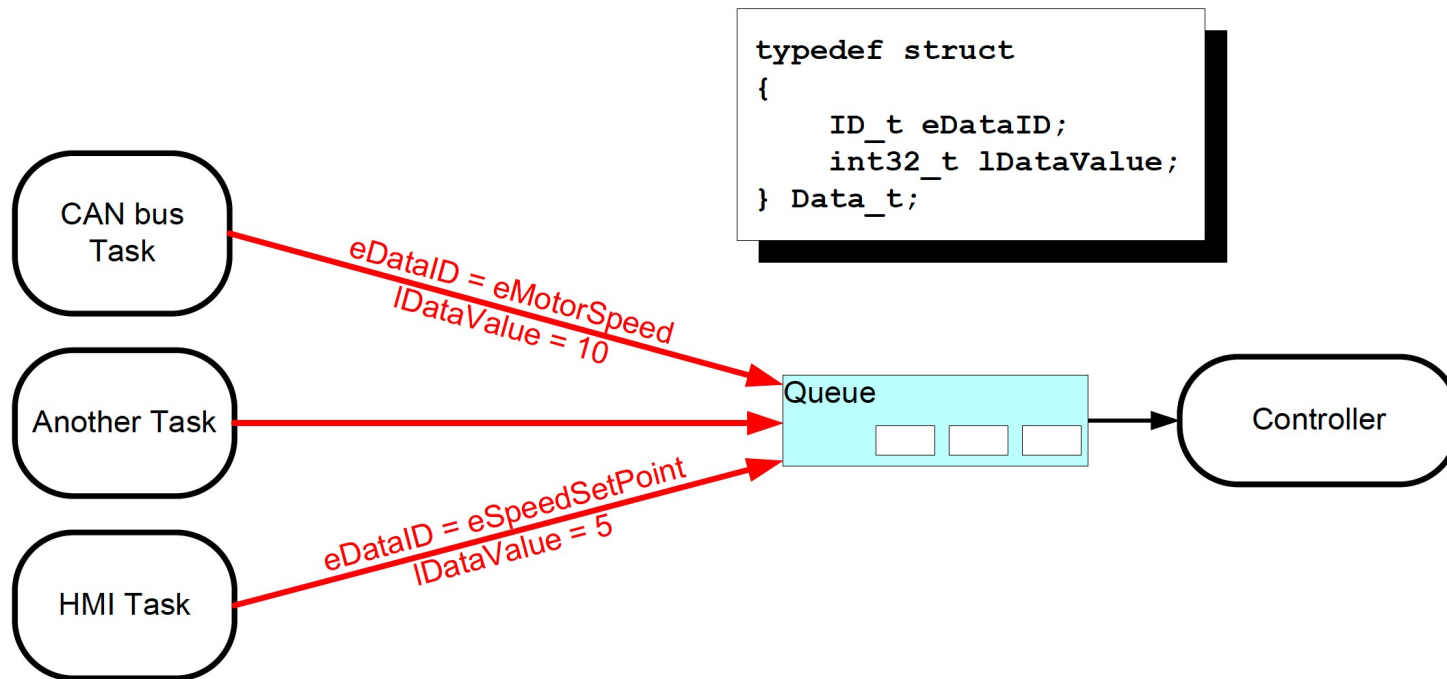


Figure 34. An example scenario where structures are sent on a queue

A: Use a struct with a key (identifier) with data



# Expanding a Little....

---

```
typedef struct
{
    ID_t eDataID;
    int32_t lDataValue;
} Data_t;
```



# Expanding a Little....

---

```
typedef struct
{
    ID_t eDataID;
    int32_t lDataValue;
} Data_t;

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



# Expanding a Little....

---

```
typedef struct
{
    ID_t eDataID;
    int32_t lDataValue;
} Data_t;

/* 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;
```



# Expanding a Little....

---

```
typedef struct
{
    ID_t eDataID;
    int32_t lDataValue;
} Data_t;
```

```
/* 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. */
};
```





# Expanding a Little....sender....

---

```
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( ;; )
    {

        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" );
        }
    }
}
```



# Expanding a Little....receiver....

```
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( ;; )
    {
        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }

        /* Receive from the queue.
        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" );
        }
    }
}
```



# Expanding a Little....main....

---

```
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 )
    {
        xTaskCreate( vSenderTask, "Sender1", 1000, &(amp;xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &(amp;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( ;; );
}
}
```



# Execution in words....

---

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.



# Execution in words....

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.



# Execution in words....

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.
t3	Task Sender 2 finds the queue is already full, so enters the Blocked state to wait for its first send to complete. Task Receiver is now the highest priority task that is able to run, so enters the Running state.



# Execution in words....

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.
t3	Task Sender 2 finds the queue is already full, so enters the Blocked state to wait for its first send to complete. Task Receiver is now the highest priority task that is able to run, so enters the Running state.
t4	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, resulting in task Receiver being pre-empted as soon as it has removed one item from the queue. Tasks Sender 1 and Sender 2 have the same priority, so the scheduler selects the task that has been waiting the longest as the task that will enter the Running state—in this case that is task Sender 1.





# Execution in words....

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.
t3	Task Sender 2 finds the queue is already full, so enters the Blocked state to wait for its first send to complete. Task Receiver is now the highest priority task that is able to run, so enters the Running state.
t4	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, resulting in task Receiver being pre-empted as soon as it has removed one item from the queue. Tasks Sender 1 and Sender 2 have the same priority, so the scheduler selects the task that has been waiting the longest as the task that will enter the Running state—in this case that is task Sender 1.
t5	Task Sender 1 sends another data item to the queue. There was only one space in the queue, so task Sender 1 enters the Blocked state to wait for its next send to complete. Task Receiver is again the highest priority task that is able to run so enters the Running state.
	Task Sender 1 has now sent four items to the queue, and task Sender 2 is still waiting to send its first item to the queue.





# Execution in words....

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.
t3	Task Sender 2 finds the queue is already full, so enters the Blocked state to wait for its first send to complete. Task Receiver is now the highest priority task that is able to run, so enters the Running state.
t4	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, resulting in task Receiver being pre-empted as soon as it has removed one item from the queue. Tasks Sender 1 and Sender 2 have the same priority, so the scheduler selects the task that has been waiting the longest as the task that will enter the Running state—in this case that is task Sender 1.
t5	Task Sender 1 sends another data item to the queue. There was only one space in the queue, so task Sender 1 enters the Blocked state to wait for its next send to complete. Task Receiver is again the highest priority task that is able to run so enters the Running state.  Task Sender 1 has now sent four items to the queue, and task Sender 2 is still waiting to send its first item to the queue.
t6	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, so task Receiver is pre-empted as soon as it has removed one item from the queue. This time Sender 2 has been waiting longer than Sender 1, so Sender 2 enters the Running state.

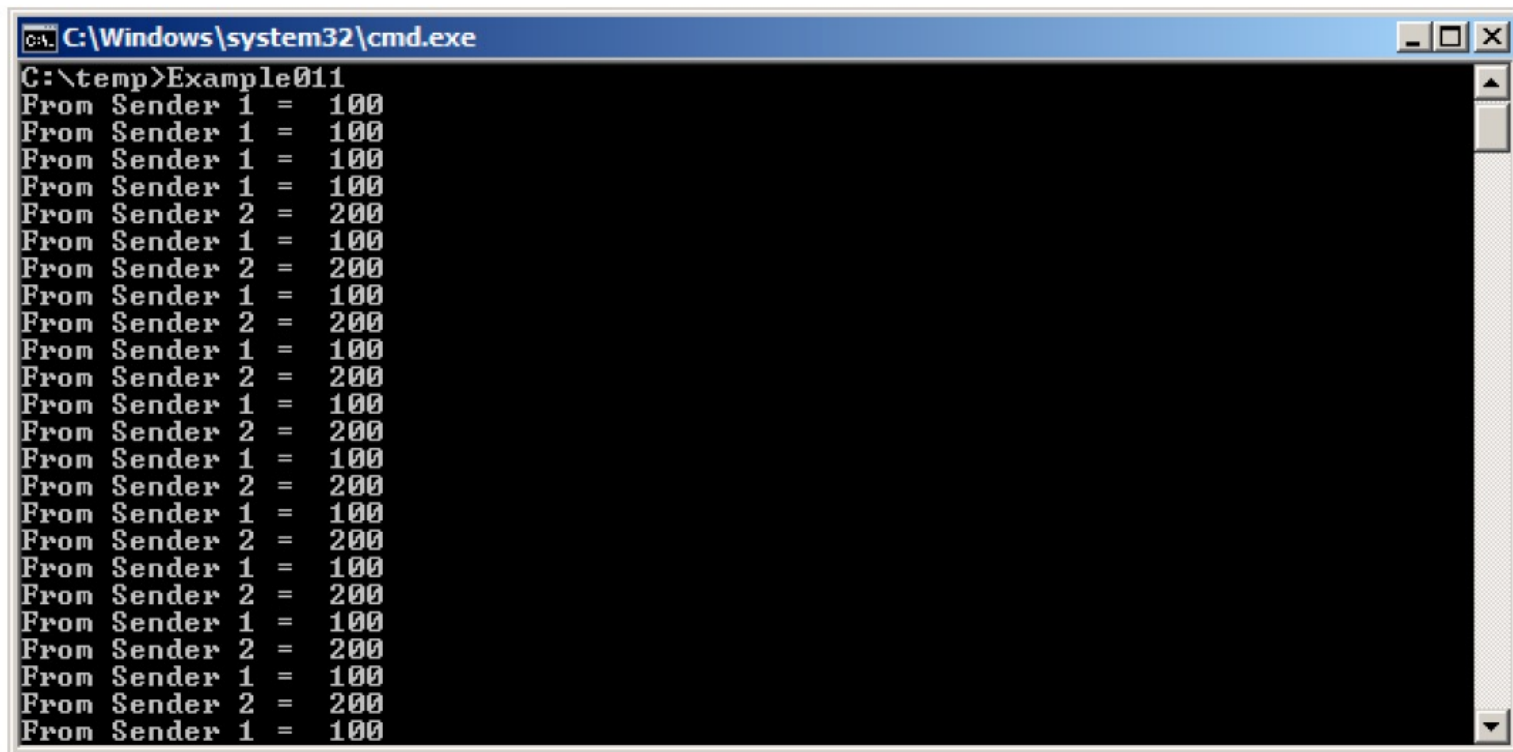


# Execution in words....

Time	Description
t1	Task Sender 1 executes and sends 3 data items to the queue.
t2	The queue is full so Sender 1 enters the Blocked state to wait for its next send to complete. Task Sender 2 is now the highest priority task that is able to run, so enters the Running state.
t3	Task Sender 2 finds the queue is already full, so enters the Blocked state to wait for its first send to complete. Task Receiver is now the highest priority task that is able to run, so enters the Running state.
t4	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, resulting in task Receiver being pre-empted as soon as it has removed one item from the queue. Tasks Sender 1 and Sender 2 have the same priority, so the scheduler selects the task that has been waiting the longest as the task that will enter the Running state—in this case that is task Sender 1.
t5	Task Sender 1 sends another data item to the queue. There was only one space in the queue, so task Sender 1 enters the Blocked state to wait for its next send to complete. Task Receiver is again the highest priority task that is able to run so enters the Running state.  Task Sender 1 has now sent four items to the queue, and task Sender 2 is still waiting to send its first item to the queue.
t6	Two tasks that have a priority higher than the receiving task's priority are waiting for space to become available on the queue, so task Receiver is pre-empted as soon as it has removed one item from the queue. This time Sender 2 has been waiting longer than Sender 1, so Sender 2 enters the Running state.
t7	Task Sender 2 sends a data item to the queue. There was only one space in the queue so Sender 2 enters the Blocked state to wait for its next send to complete. Both tasks Sender 1 and Sender 2 are waiting for space to become available on the queue, so task Receiver is the only task that can enter the Running state.



# Expanding a Little...Output...



```
C:\Windows\system32\cmd.exe
C:\temp>Example011
From Sender 1 = 100
From Sender 1 = 100
From Sender 1 = 100
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
From Sender 2 = 200
From Sender 1 = 100
```

Figure 35 The output produced by Example 11



# Expanding a Little...sched sequence....

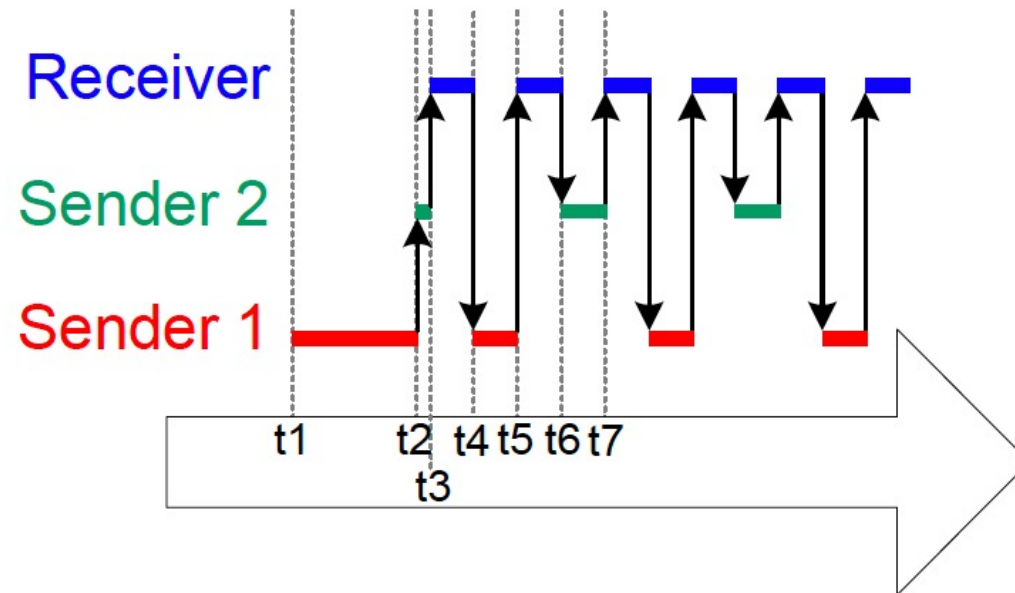
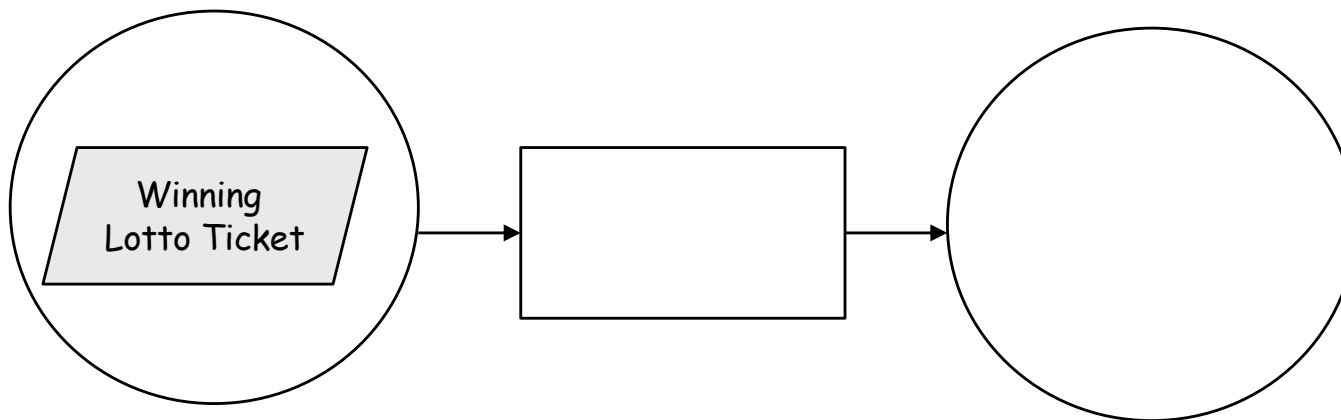


Figure 36. The sequence of execution produced by Example 11



# Mailboxes (Ch 4.7)

---

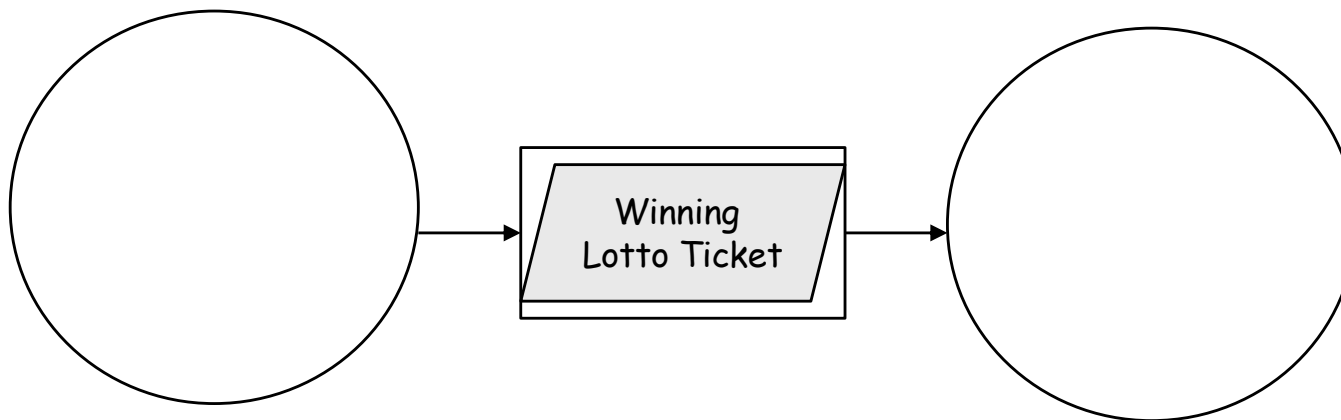


Information is put in and removed from a Queue



# Mailboxes (Ch 4.7)

---

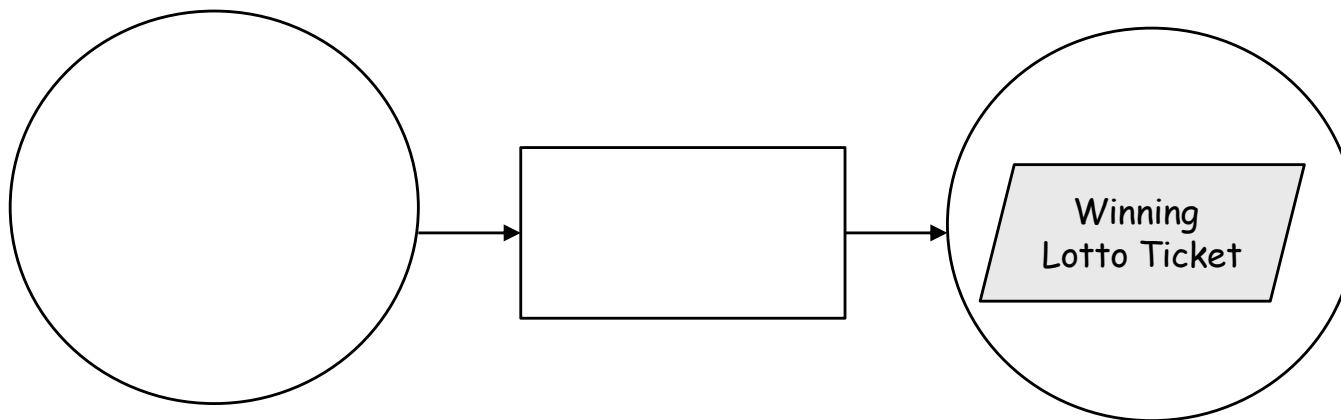


Information is put in and removed from a Queue



# Mailboxes (Ch 4.7)

---



Information is put in and removed from a Queue

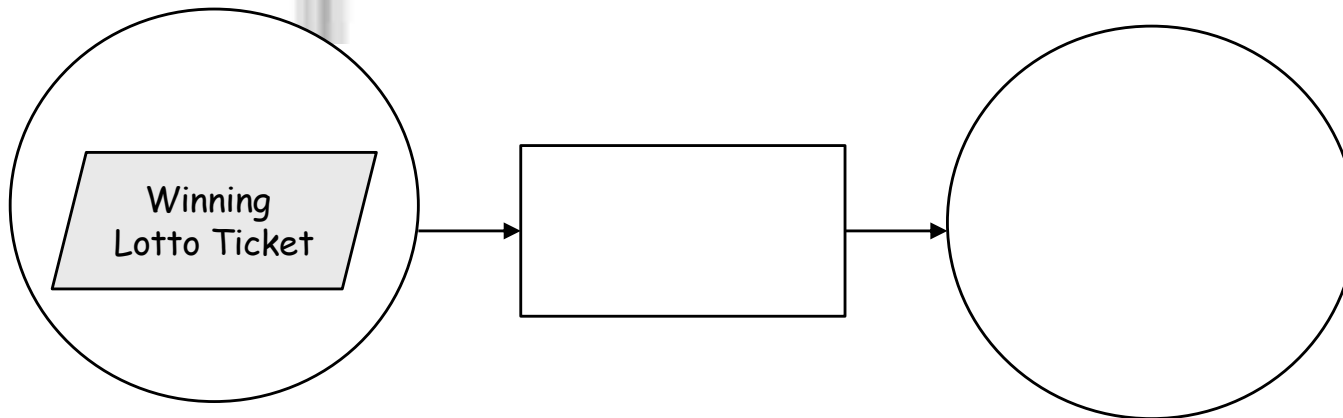


# Mailboxes (Ch 4.7)

---



Mailbox = Queue of length 1



Information is put in Queue



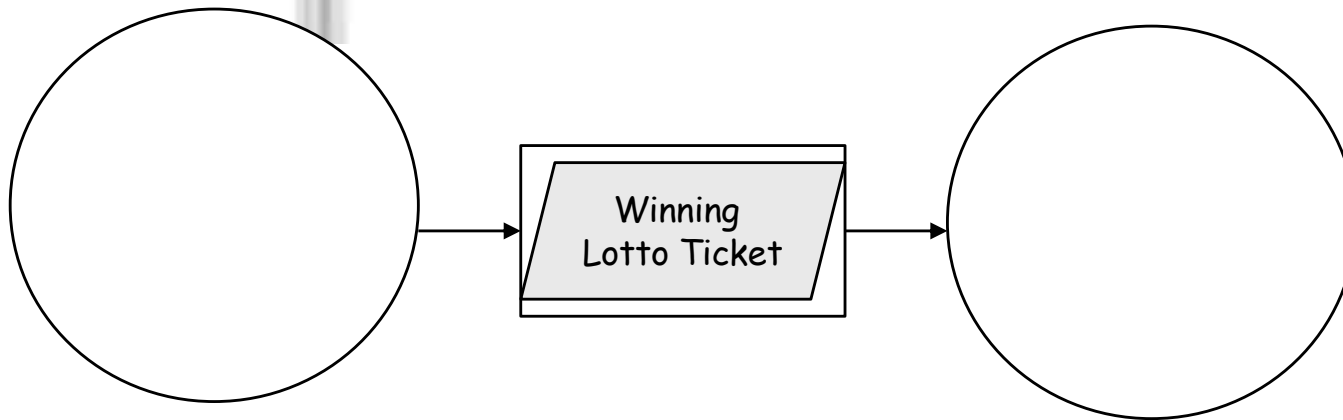


# Mailboxes (Ch 4.7)

---



Mailbox = Queue of length 1



Information is put in Queue

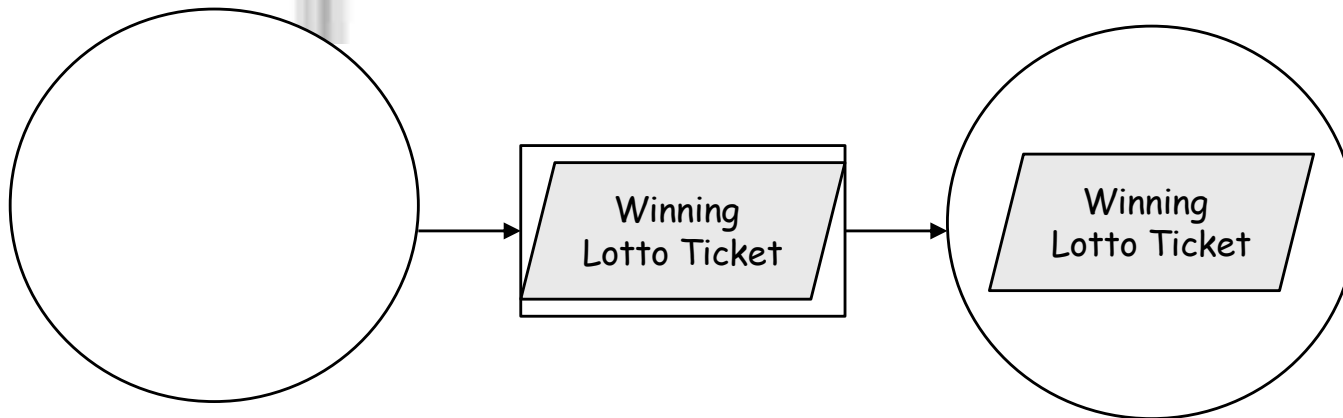


# Mailboxes (Ch 4.7)

---



Mailbox = Queue of length 1



Information is put in Queue: Copied Out by a Reader but remains In Queue (Mailbox):

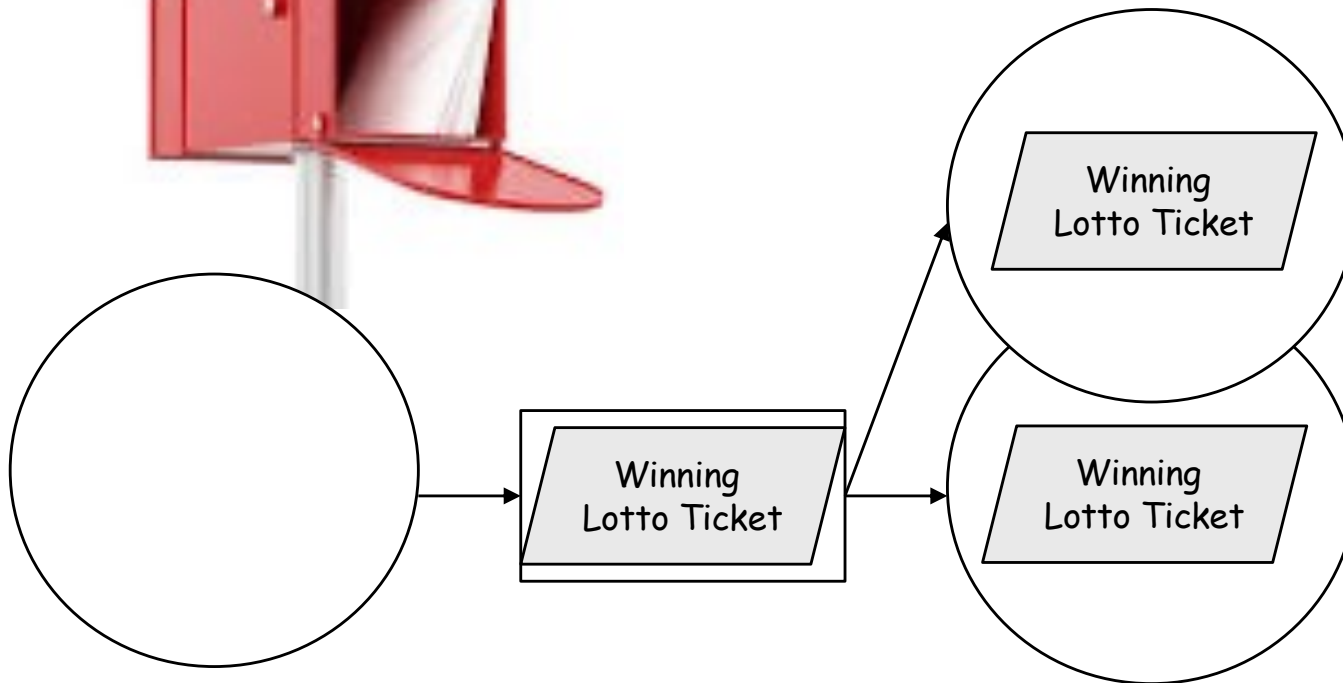


# Mailboxes (Ch 4.7)

---



Mailbox = Queue of length 1



Information is put in Queue: Copied Out by a Reader but remains In Queue (Mailbox): and Read by Multiple Readers !

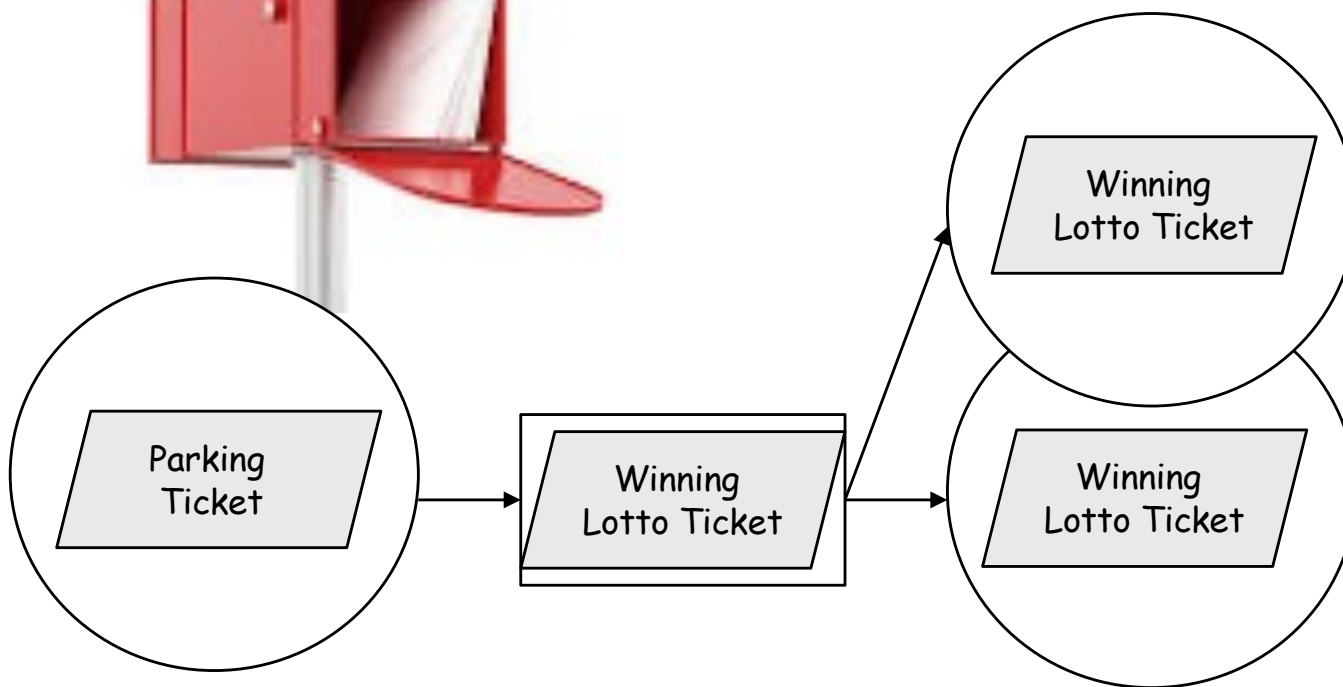


# Mailboxes (Ch 4.7)

---



Mailbox = Queue of length 1



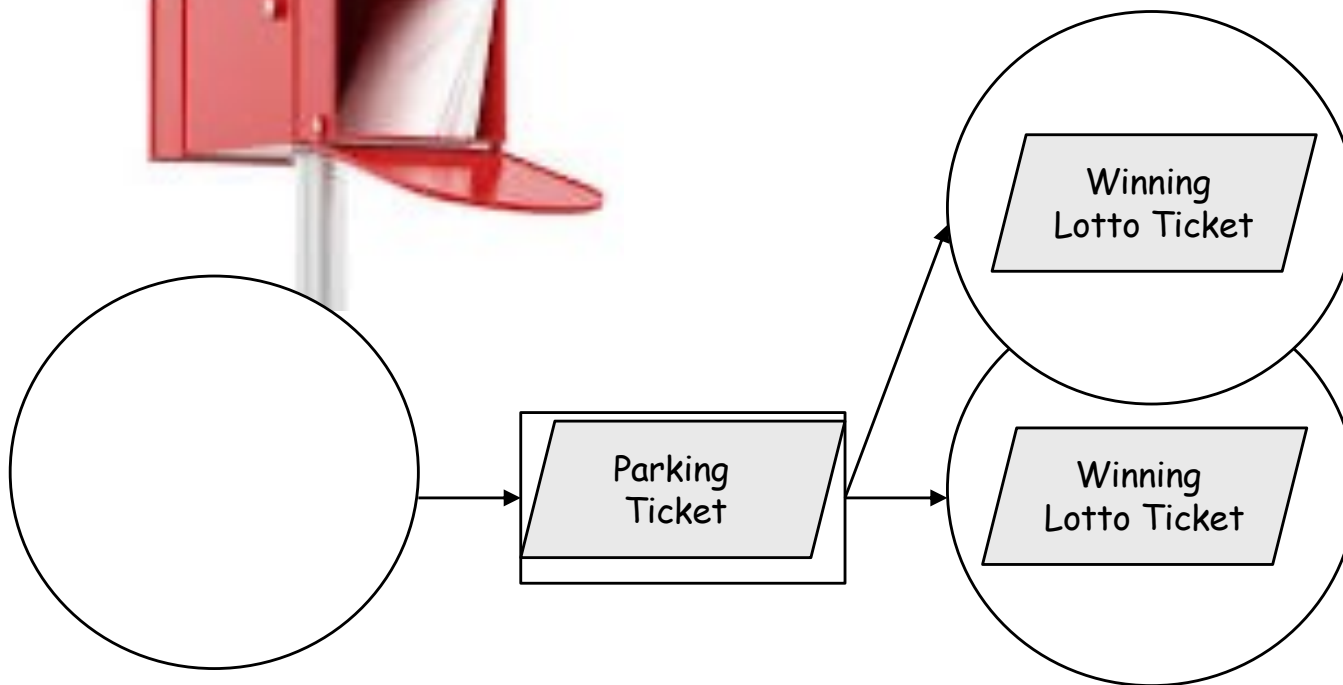
Information is put in Queue: Copied Out by a Reader but remains In Queue (Mailbox): and Read by Multiple Readers !



# Mailboxes (Ch 4.7)



Mailbox = Queue of length 1



Information is put in Queue: Copied Out by a Reader but remains In Queue (Mailbox): and Read by Multiple Readers ! : Until Overwritten



# Mailboxes (Ch 4.7)



```
/* 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 ) );
}
```



# Mailboxes (Ch 4.7)

---



```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue,  
                             const void * pvItemToQueue );
```

Unlike `xQueueSendToBack()`, if the queue is already full, then `xQueueOverwrite()` will overwrite data that is already in the queue.

`xQueueOverwrite()` should only be used with queues that have a length of one. That restriction avoids the need for the function's implementation to make an arbitrary decision as to which item in the queue to overwrite, if the queue is full.



# Mailboxes (Ch 4.7)



```
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 );
}
```

```
BaseType_t xQueuePeek( QueueHandle_t xQueue,
                      void * const pvBuffer,
                      TickType_t xTicksToWait );
```





# Mailboxes (Ch 4.7)

---



```
BaseType_t xQueuePeek( QueueHandle_t xQueue,  
                      void * const pvBuffer,  
                      TickType_t xTicksToWait );
```

xQueuePeek() is used to receive (read) an item from a queue *without* the item being removed from the queue.



# Mailboxes (Ch 4.7)



```
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;
}
```

