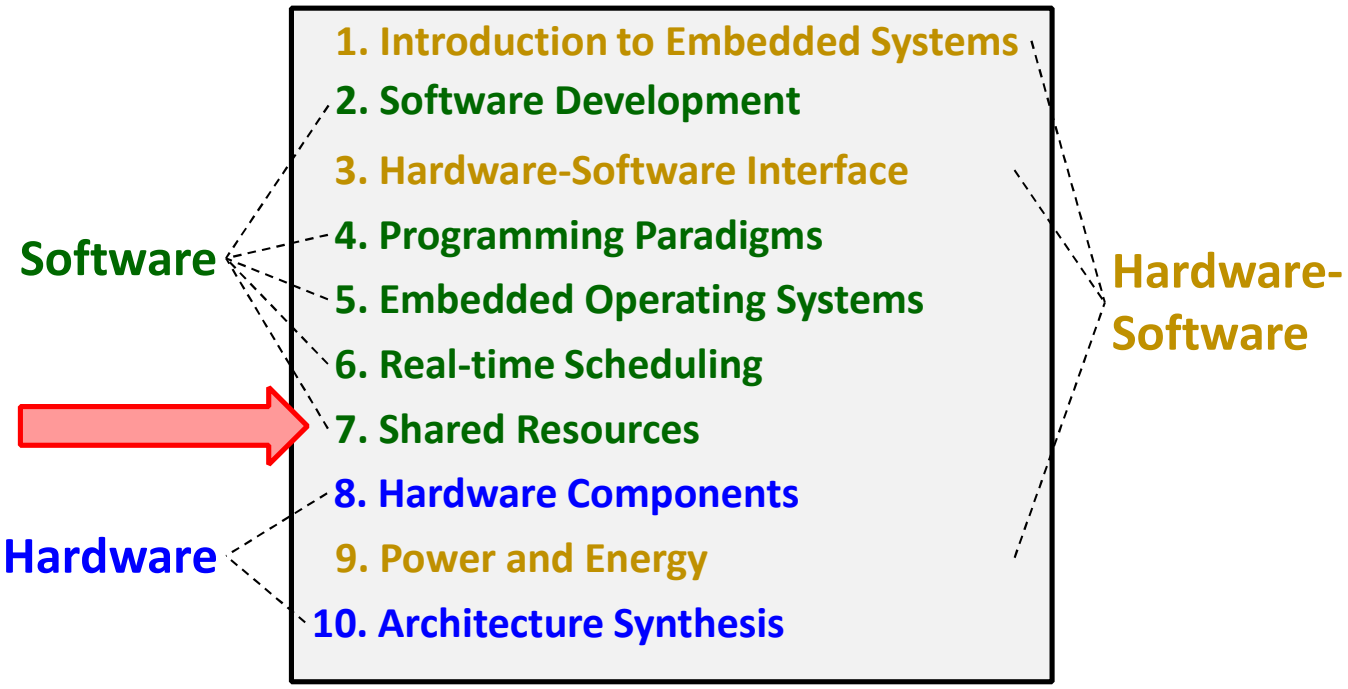
Embedded Systems

7. Shared Resources

Where we are …

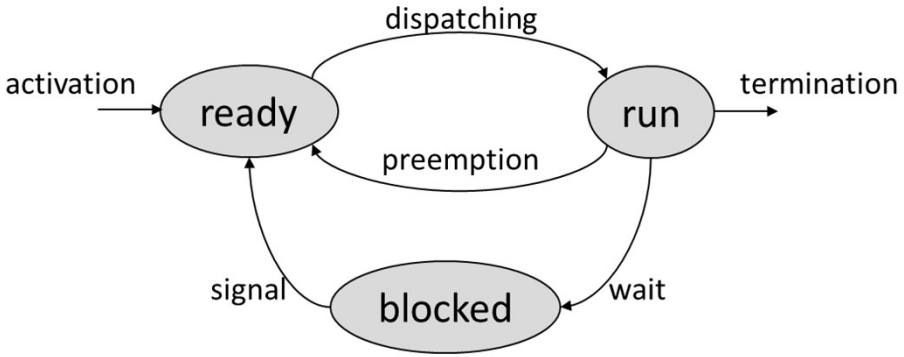

1. Introduction to Embedded Systems
2. Software Development
3. Hardware-Software Interface
4. Programming Paradigms
5. Embedded Operating Systems
6. Real-time Scheduling
7. Shared Resources
8. Hardware Components
9. Power and Energy
10. Architecture Synthesis
Software
Hardware
Hardware-Software

Ressource Sharing

Resource Sharing

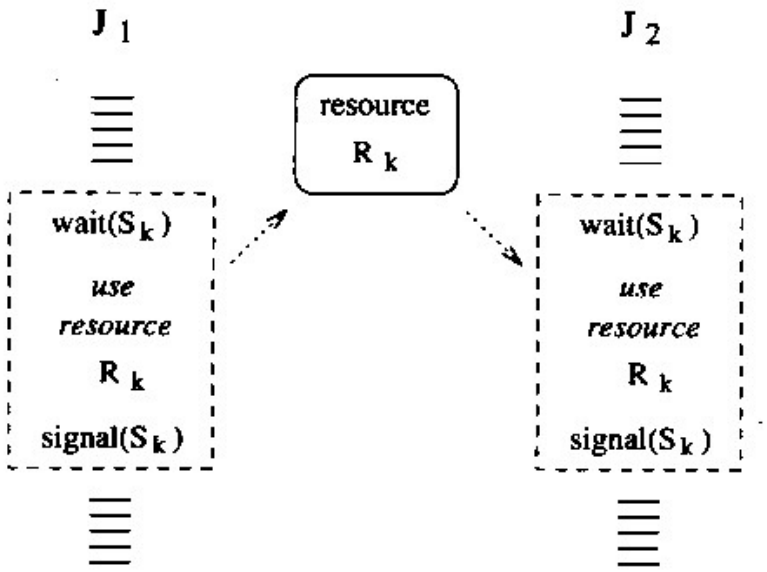
- Examples of *shared resources*: data structures, variables, main memory area, file, set of registers, I/O unit, … .
- Many shared resources do not allow simultaneous accesses but require *mutual exclusion*. These resources are called *exclusive resources*. In this case, no two threads are allowed to operate on the resource at the same time.

- There are several methods available to *protect exclusive resources*, for example
  - *disabling interrupts* and preemption or
  - using concepts like *semaphores and mutex* that put threads into the blocked state if necessary.



Protecting Exclusive Resources using Semaphores

- Each *exclusive resource* R_i must be protected by a different *semaphore* S_i. Each critical section operating on a resource must begin with a `wait(`S_i`)` primitive and end with a `signal(`S_i`)` primitive.



- All tasks blocked on the same resource are kept in a queue associated with the semaphore. When a running task executes a *wait* on a *locked semaphore*, it enters a *blocked state*, until another tasks executes a *signal* primitive that *unlocks the semaphore*.

Example FreeRTOS

To ensure data consistency is maintained at all times access to a resource that is shared between tasks, or between tasks and interrupts, must be managed using a ‘mutual exclusion’ technique.

One possibility is to disable all interrupts:

```
...
taskENTER_CRITICAL();
   ... /* access to some exclusive resource */
taskEXIT_CRITICAL();
...
```

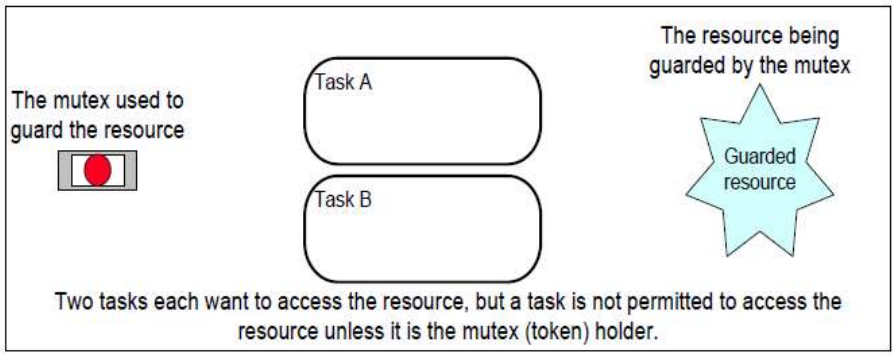
This kind of critical sections must be kept very short, otherwise they will adversely affect interrupt response times.

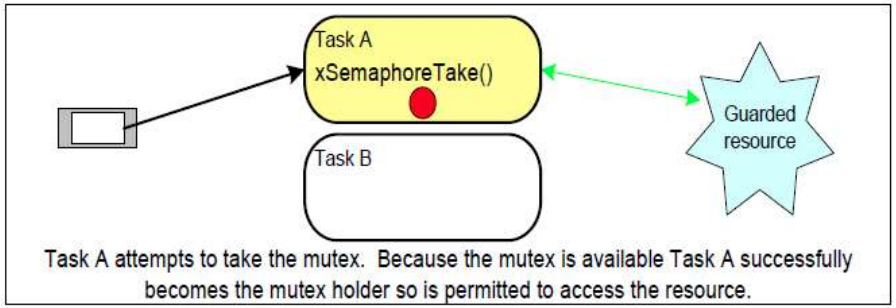
Example FreeRTOS

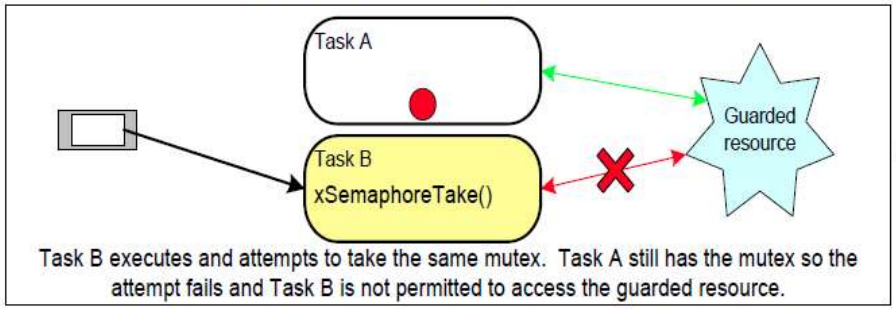
Another possibility is to use mutual exclusion: In FreeRTOS, a *mutex* is a special type of *semaphore* that is used to *control access* to a resource that is shared between two or more tasks. *A semaphore that is used for mutual exclusion must always be returned:*

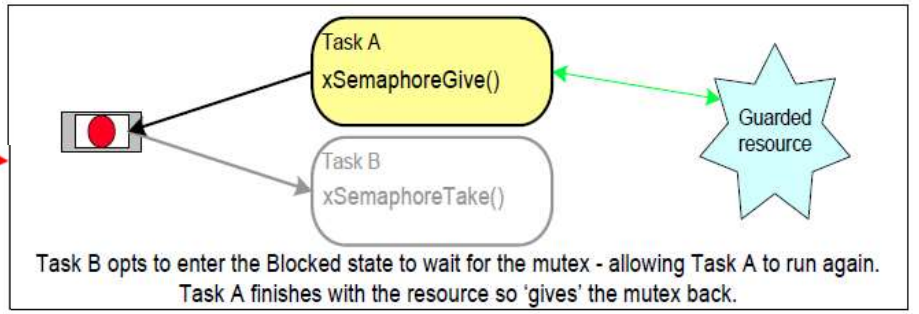
- When used in a mutual exclusion scenario, the mutex can be thought of as a token that is associated with the resource being shared.
- For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back.
- Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource.

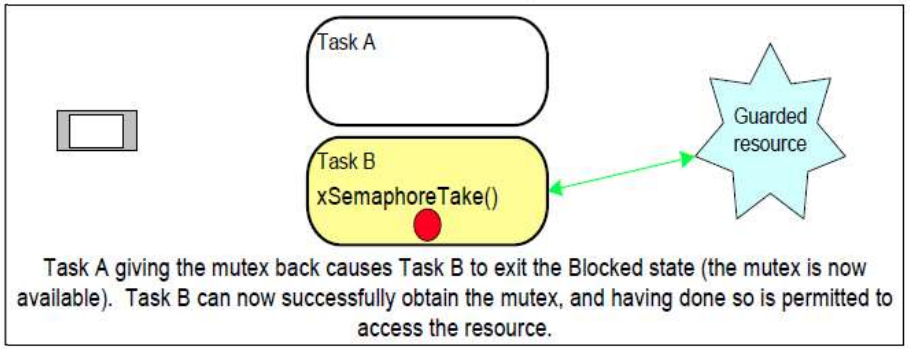
Example FreeRTOS

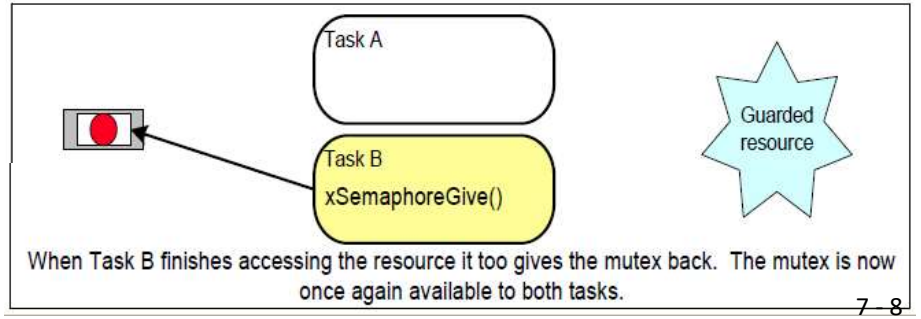












Example FreeRTOS

Example:

create mutex semaphore

```
SemaphoreHandle_t xMutex;

int main( void ) {
   xMutex = xSemaphoreCreateMutex();
   if( xMutex != NULL ) {
      xTaskCreate(vTask1,"Task1",1000,NULL,1,NULL);
      xTaskCreate(vTask2,"Task2",1000,NULL,2,NULL);
      vTaskStartScheduler();
   }
   for( ;; );
}
```

some defined constant for infinite timeout; otherwise, the function would return if the mutex was not available for the specified time

```
void vTask1( void *pvParameters ) {
   for( ;; ) {
      ...
      xSemaphoreTake(xMutex,portMAX_DELAY);
         ... /* access to exclusive resource */
      xSemaphoreGive(xMutex);
      ... }
}
```

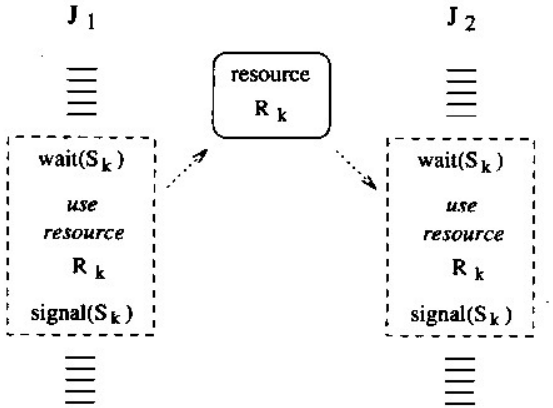
```
void vTask2( void *pvParameters ) {
   for( ;; ) {
      ...
      xSemaphoreTake(xMutex,portMAX_DELAY);
         ... /* access to exclusive resource */
      xSemaphoreGive(xMutex);
      ... }
}
```

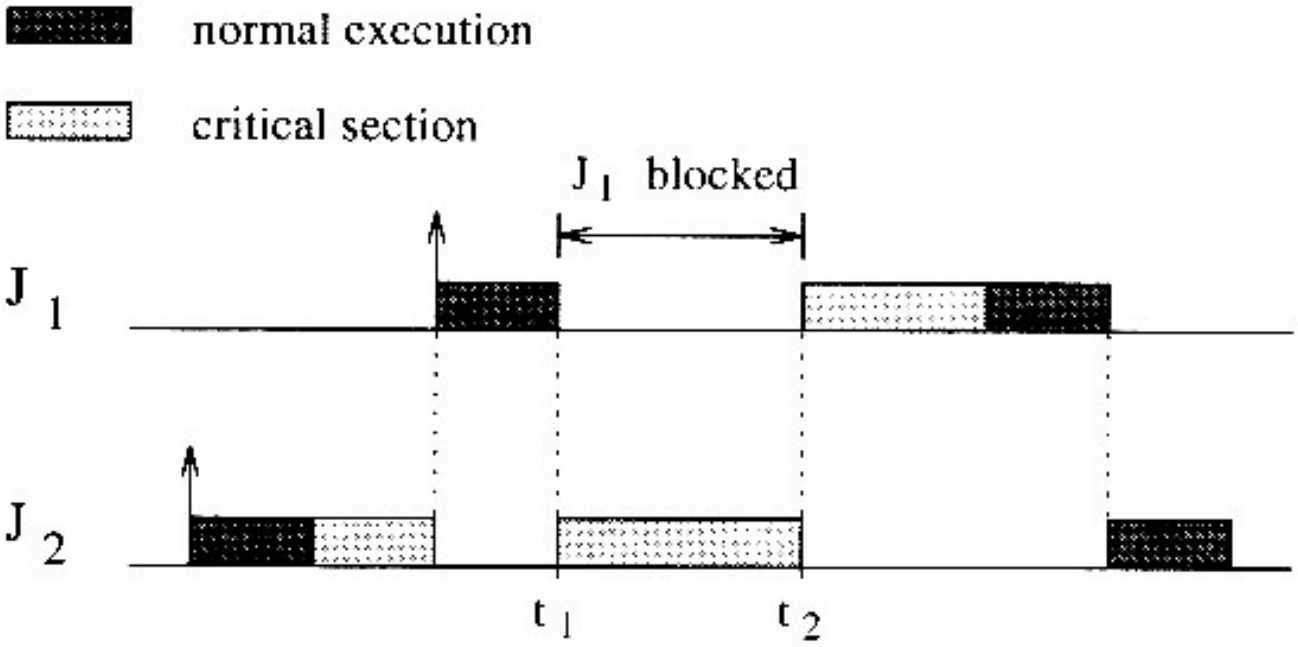
Ressource Sharing

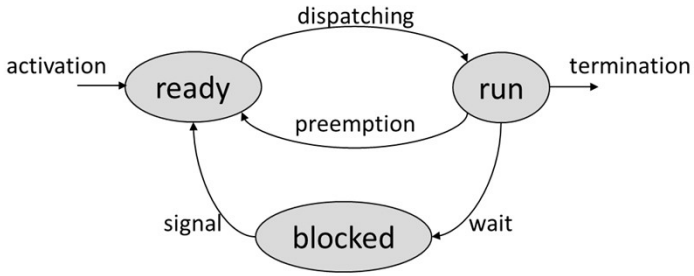
Priority Inversion

Priority Inversion (1)

Unavoidable blocking:

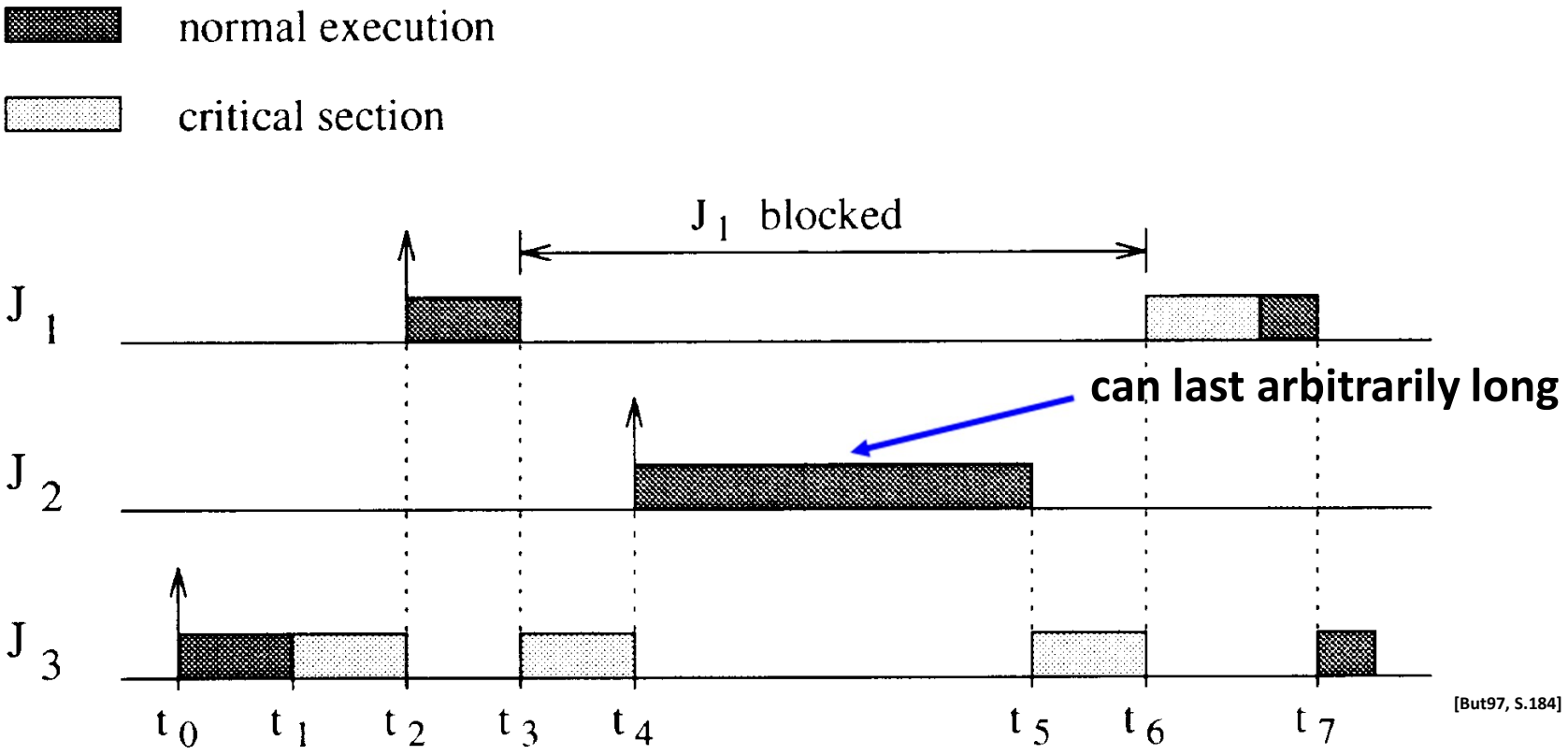






Priority Inversion (2)

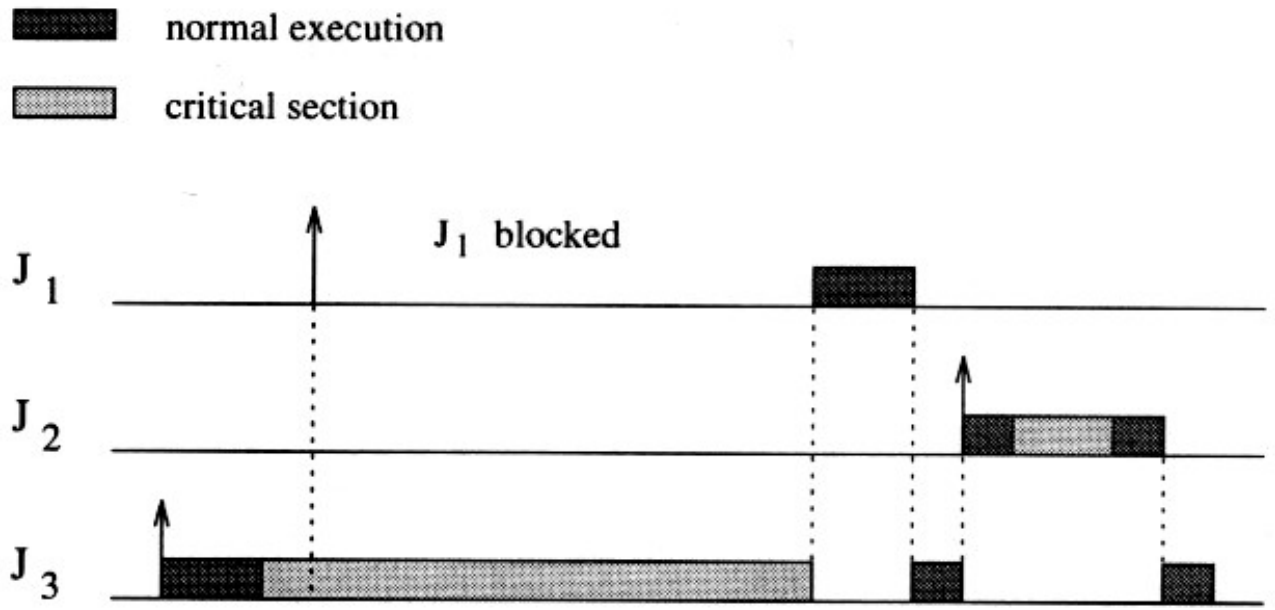
Priority Inversion:



[But97, S.184]

Solutions to Priority Inversion

Disallow preemption during the execution of all critical sections. Simple approach, but it creates unnecessary blocking as unrelated tasks may be blocked.



Resource Access Protocols

Basic idea: Modify the priority of those tasks that cause blocking. When a task J_i blocks one or more higher priority tasks, it temporarily assumes a higher priority.

Specific Methods:

- Priority Inheritance Protocol (PIP), for static priorities
- Priority Ceiling Protocol (PCP), for static priorities
- Stack Resource Policy (SRP),

  for static and dynamic priorities
- others …

Priority Inheritance Protocol (PIP)

Assumptions:

n tasks which cooperate through *m* shared resources; fixed priorities, all critical sections on a resource begin with a `wait(`S_i`)` and end with a `signal(`S_i`)` operation.

Basic idea:

When a task J_i blocks one or more higher priority tasks, it temporarily assumes (inherits) the highest priority of the blocked tasks.

Terms:

We distinguish a fixed *nominal priority* P_i and an *active priority* p_i larger or equal to P_i. Jobs J_1, …J_n are ordered with respect to nominal priority where J_1 has *highest priority*. Jobs do not suspend themselves.

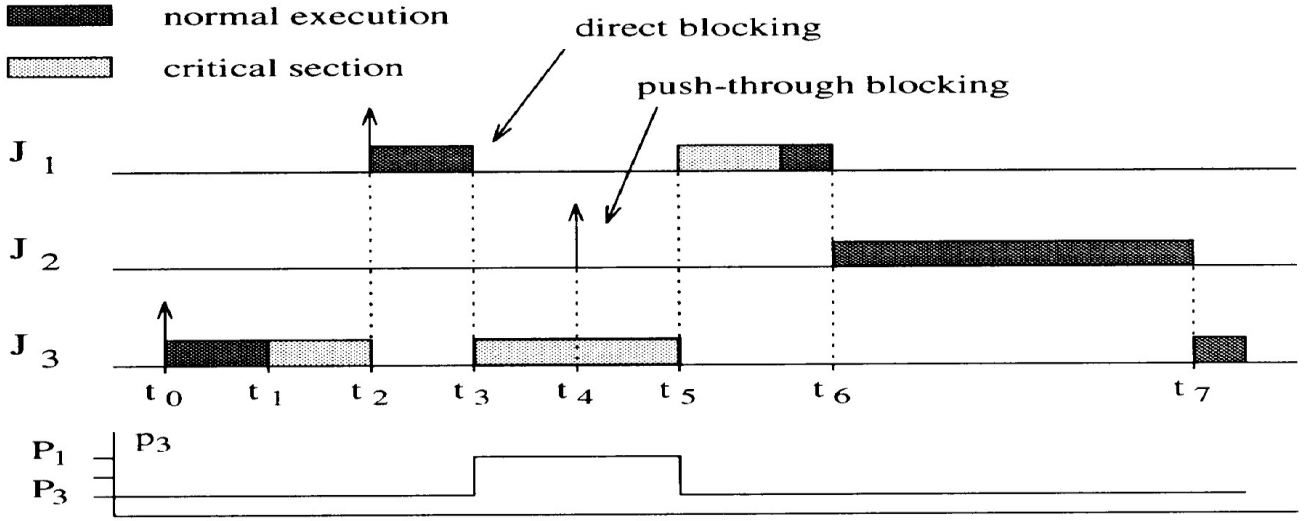
Priority Inheritance Protocol (PIP)

Algorithm:

- Jobs are scheduled based on their *active priorities*. Jobs with the same priority are executed in a FCFS discipline.
- When a job J_i tries to *enter a critical section* and the resource is blocked by a lower priority job, the job J_i is blocked. Otherwise it enters the critical section.
- When a job J_i is *blocked*, it transmits its active priority to the job J_k that holds the semaphore. J_k resumes and executes the rest of its critical section with a priority $p_k=p_i$ (it *inherits* the priority of the highest priority of the jobs blocked by it).
- When J_k exits a critical section, it *unlocks* the semaphore and the highest priority job blocked on that semaphore is awakened. If no other jobs are blocked by J_k, then p_k is set to P_k, otherwise it is set to the highest priority of the jobs blocked by J_k.
- Priority inheritance is *transitive*, i.e. if 1 is blocked by 2 and 2 is blocked by 3, then 3 inherits the priority of 1 via 2.

Priority Inheritance Protocol (PIP)

Example:

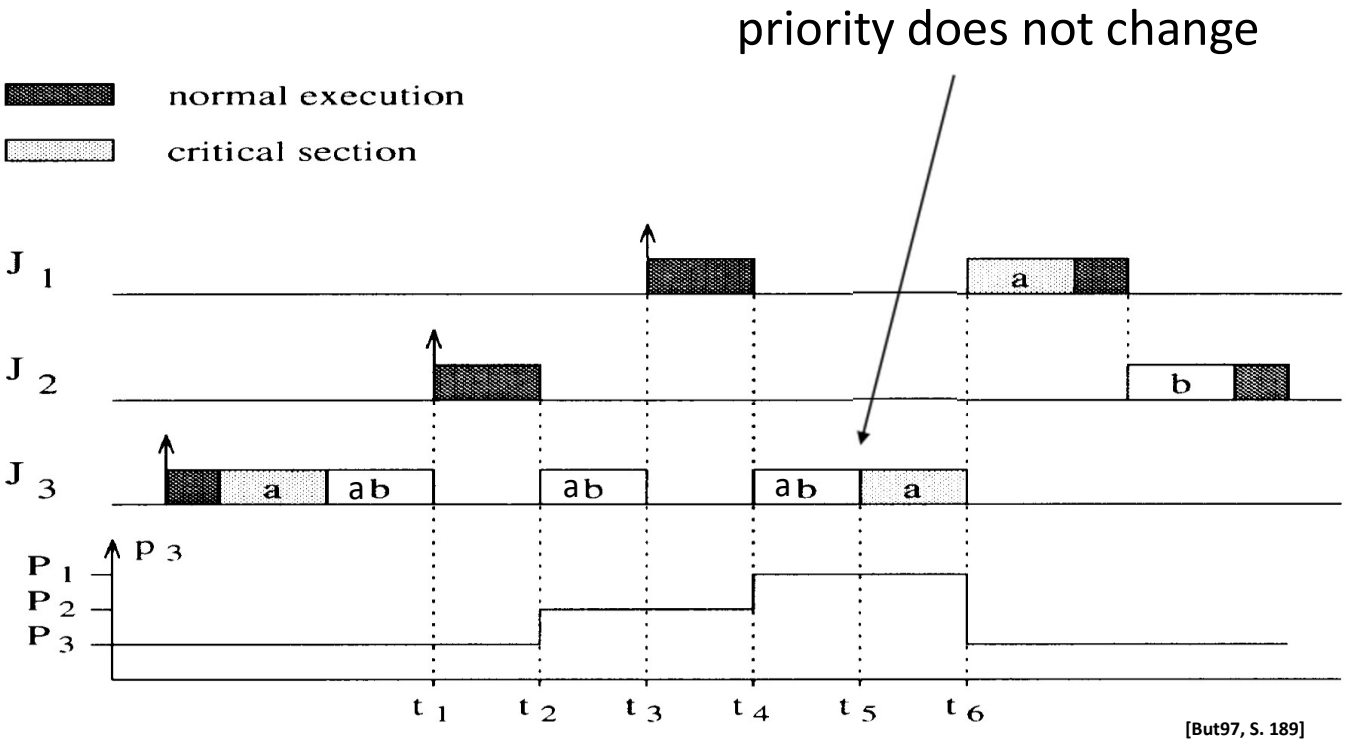


Direct Blocking: higher-priority job tries to acquire a resource held by a lower-priority job

Push-through Blocking: medium-priority job is blocked by a lower-priority job that has inherited a higher priority from a job it directly blocks

Priority Inheritance Protocol (PIP)

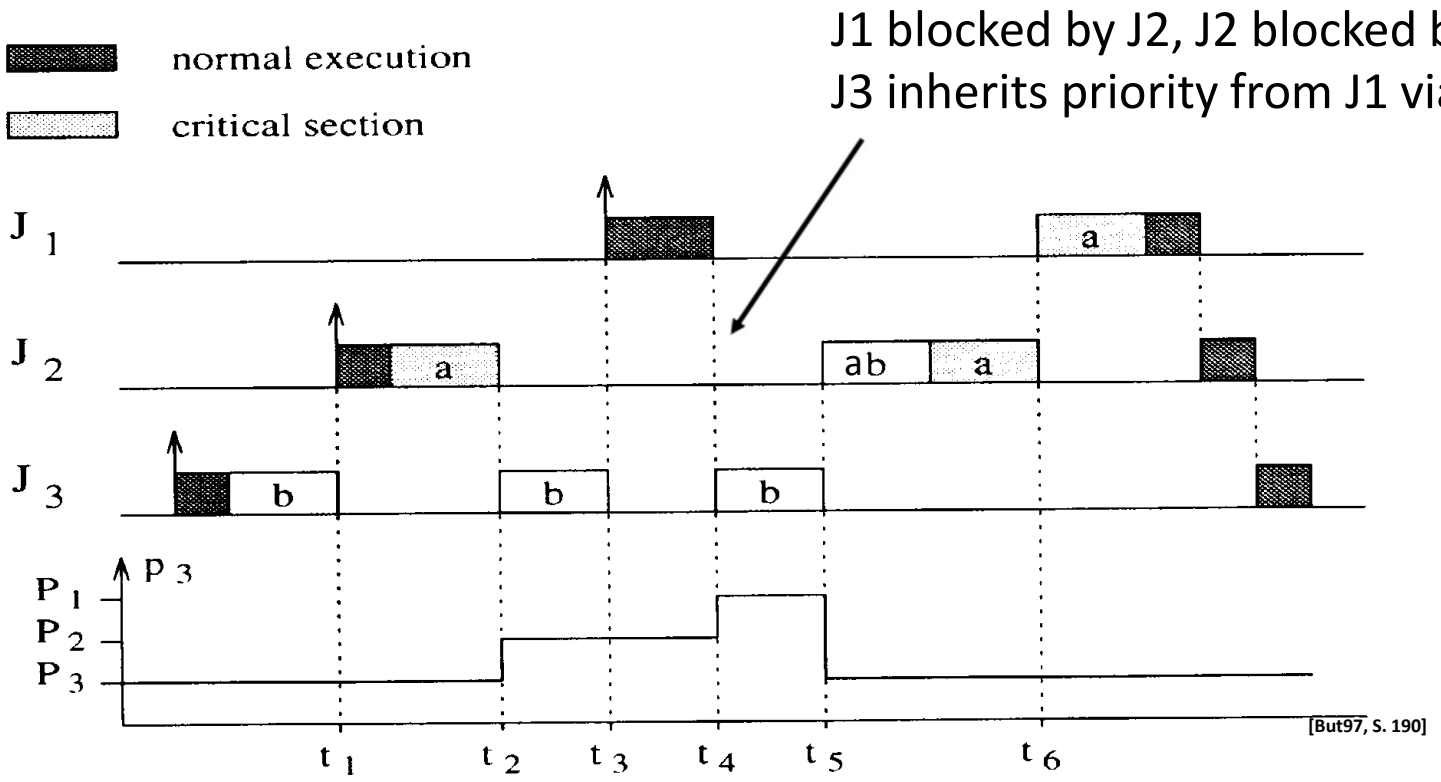
Example with nested critical sections:



[But97, S. 189]

Priority Inheritance Protocol (PIP)

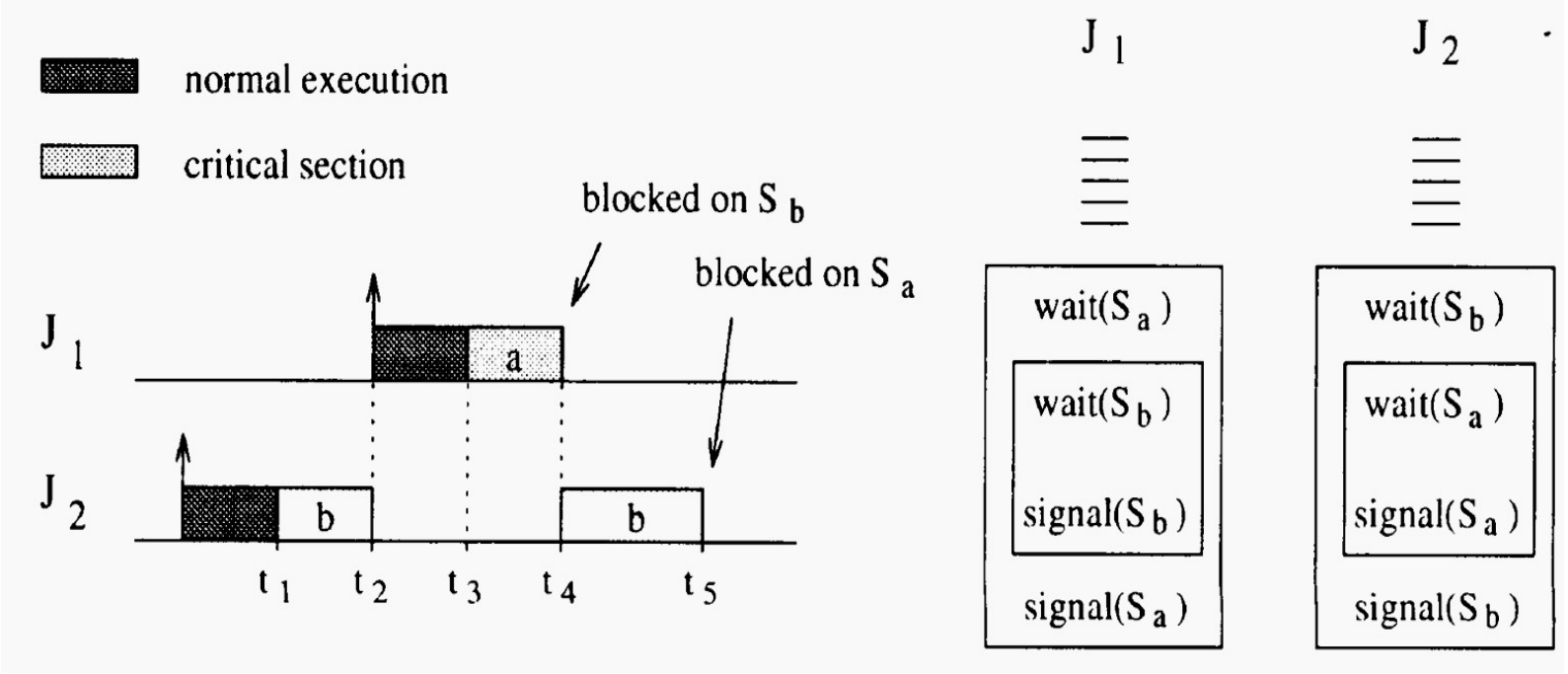
Example of transitive priority inheritance:



Priority Inheritance Protocol (PIP)

Still a Problem: Deadlock

…. but there are other protocols like the Priority Ceiling Protocol …



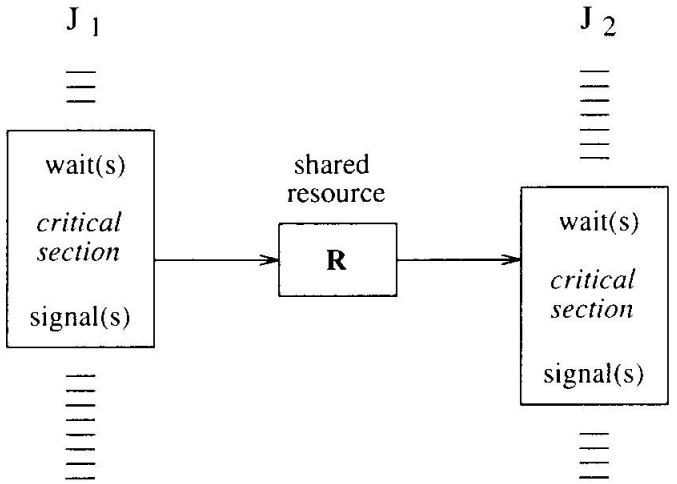
[But97, S. 200]

Communication and Synchronization

Communication Between Tasks

Problem: the use of shared memory for implementing communication between tasks may cause priority inversion and blocking.

Therefore, either the implementation of the shared medium is "thread safe" or the data exchange must be *protected by critical sections*.



Communication Mechanisms

Synchronous communication:

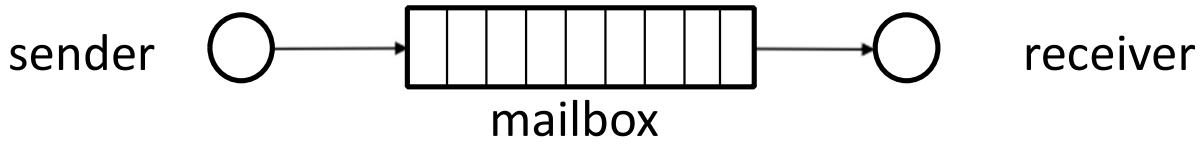
- Whenever two tasks want to communicate they must be synchronized for a message transfer to take place (rendez-vous).
- They have to wait for each other, i.e. both must be at the same time ready to do the data exchange.

- *Problem:*
  - In case of dynamic real-time systems, estimating the maximum blocking time for a process rendez-vous is difficult.
  - Communication always needs synchronization. Therefore, the timing of the communication partners is closely linked.

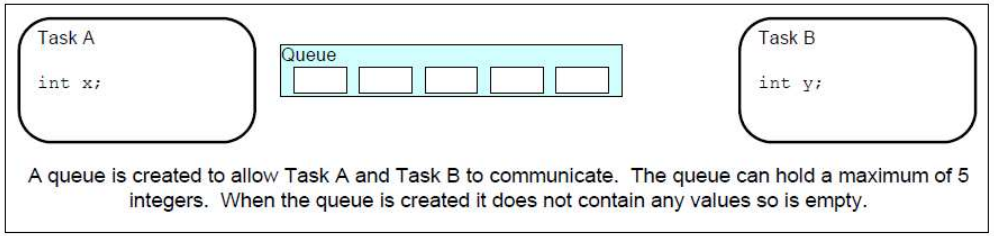
Communication Mechanisms

Asynchronous communication:

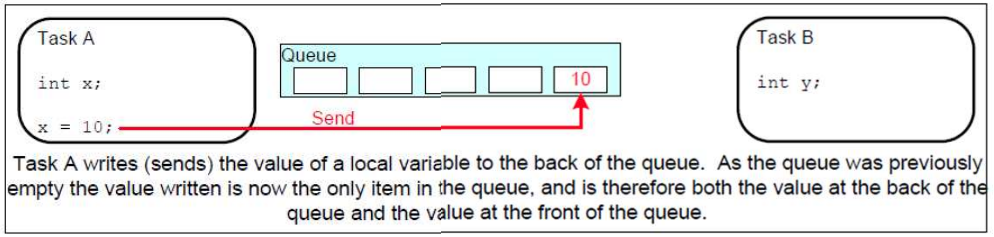
- Tasks do not necessarily have to wait for each other.
- The sender just deposits its message into a channel and continues its execution; similarly the receiver can directly access the message if at least a message has been deposited into the channel.
- More suited for real-time systems than synchronous communication.
- *Mailbox:* Shared memory buffer, FIFO-queue, basic operations are send and receive, usually has a fixed capacity.
- *Problem:* Blocking behavior if the channel is full or empty; alternative approach is provided by cyclical asynchronous buffers or double buffering.



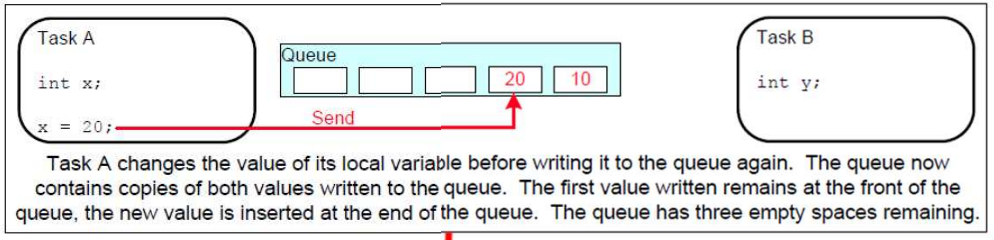
Example: FreeRTOS



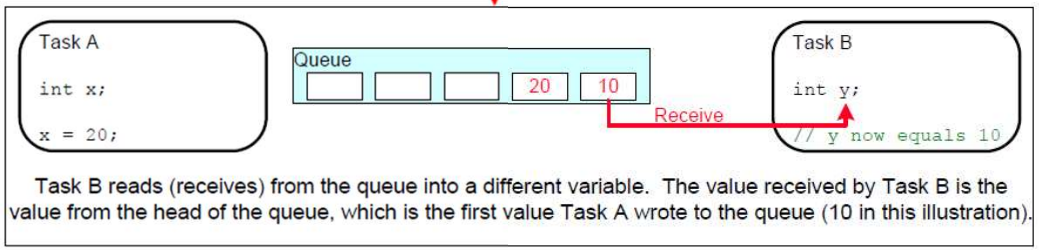
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



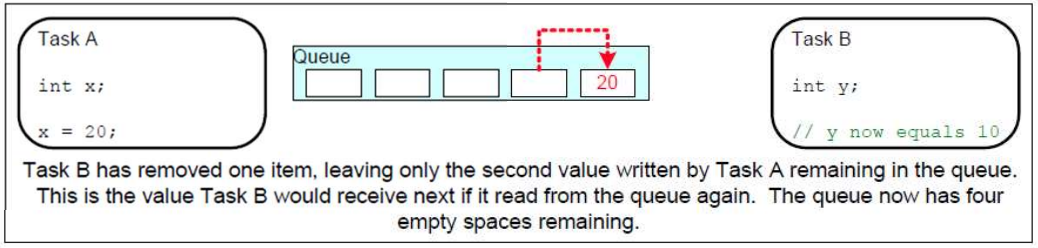
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



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).



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.

Example: FreeRTOS

Creating a queue:

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

returns handle to created queue

the maximum number of items that the queue being created can hold at any one time

the size in bytes of each data item

Sending item to a queue:

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

returns pdPASS if item was successfully added to queue

a pointer to the data to be copied into the queue

the maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue

Example: FreeRTOS

Receiving item from a queue:

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

returns pdPASS if data was successfully read from the queue

the maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue

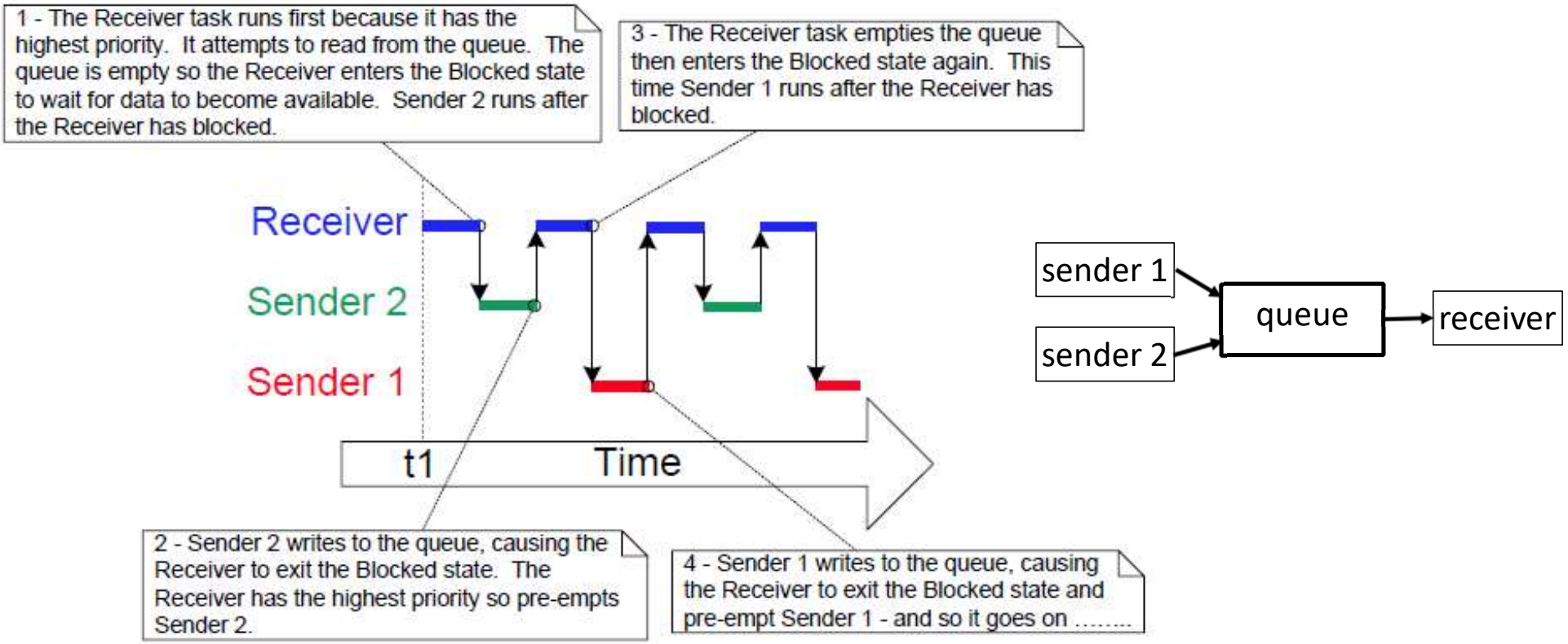
a pointer to the memory into which the received data will be copied

Example:

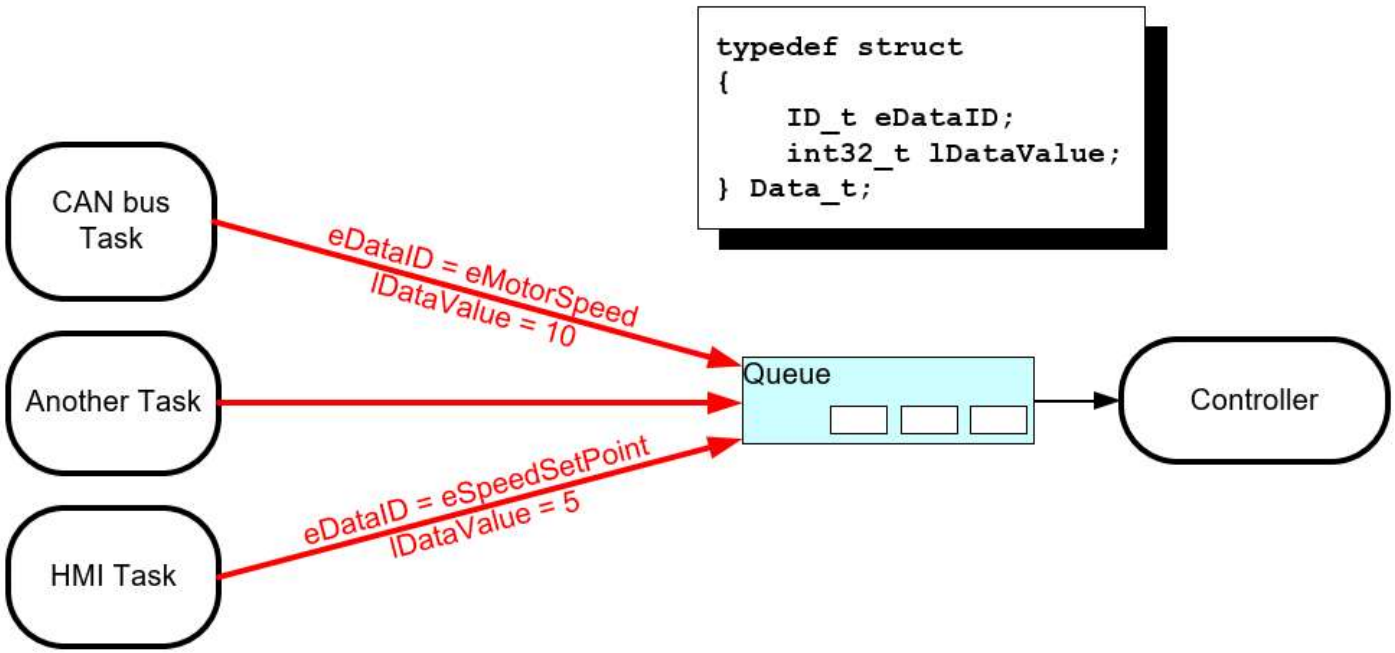
- Two sending tasks with equal priority 1 and one receiving task with priority 2.
- FreeRTOS schedules tasks with equal priority in a round-robin manner: A blocked or preempted task is put to the end of the ready queue for its priority. The same holds for the currently running task at the expiration of the time slice.

Example: FreeRTOS

Example cont.:



Example: FreeRTOS



Communication Mechanisms

Cyclical Asynchronous Buffers (CAB):

- *Non-blocking communication between tasks*.
- A reader gets the most recent message put into the CAB. A message is not consumed (that is, extracted) by a receiving process but is maintained until overwritten by a new message.
- As a consequence, once the first message has been put in a CAB, a task can never be blocked during a receive operation. Similarly, since a new message overwrites the old one, a sender can never be blocked.
- Several readers can simultaneously read a single message from the CAB.

writing

```
buf_pointer = reserve(cab_id);
<copy message in *buf_pointer>
putmes(buf_pointer, cab_id);
```

reading

```
mes_pointer = getmes(cab_id);
<use message>
unget(mes_pointer, cab_id);
```