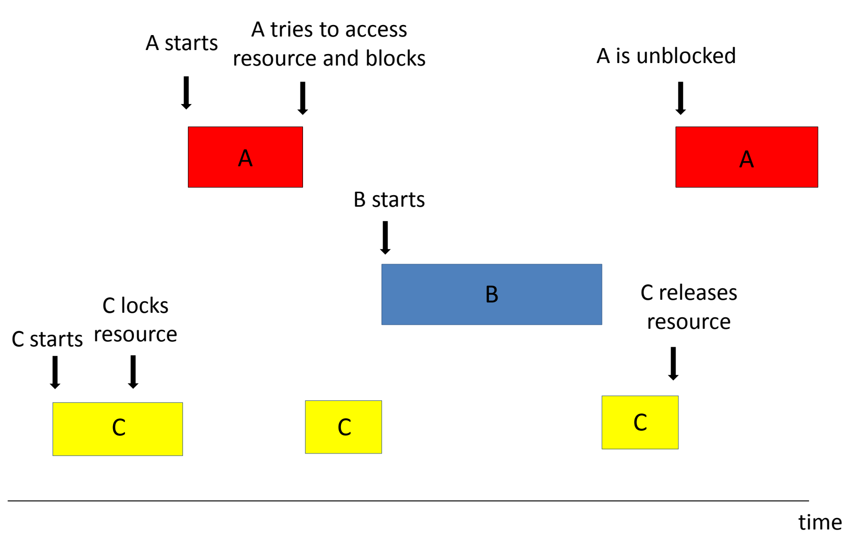
Response Time Analysis with Task Blocking

We can use Worst-Case Response Time Analysis to study real-time schedulability when tasks are not independent, but interacting with each other.

Tasks may be forced to be blocked waiting for other tasks, thus increasing their worst-case response time. In fact, task may be blocked waiting for tasks of lower priority, which is known as the "priority inversion" problem.

Consider three tasks A,B, and C. Assume A has the highest priority and C has the lowest priority. A and C access a common resource.

Priority Inversion



Priority Inversion

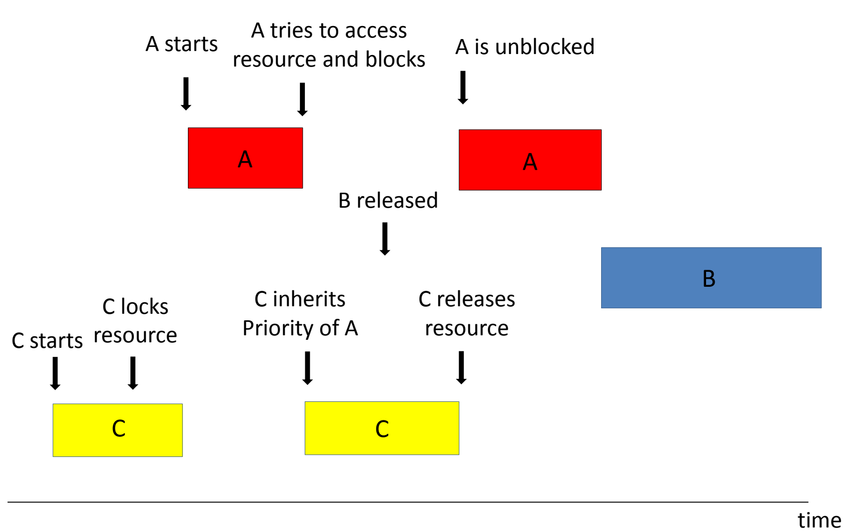
Note that A was forced to wait for C (necessary for consistency) and for B as well (which is not really necessary).

Effect of priority inversion on high-priority tasks can be alleviated using the "Priority Inheritance" protocol.

If a high-priority task τ_H is blocked waiting for a resource held by a low-priority task τ_L, task τ_L operates at the priority level of τ_H until it releases the resource (i.e. end of "critical section"). Any task τ_M with intermediate priority cannot thus preempt τ_L.

Note that τ_M may nevertheless be preempted by τ_L, which temporarily has higher priority (push-through blocking).

Priority Inheritance



Priority Inheritance

With priority inheritance, worst-case response time can be calculated as:

$$R_i = p_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil p_j$$

where B_i is a blocking factor, accounting for interference by lower priority tasks.

B_i will include the WCET of a resource's low-priority critical section if this resource is accessed by a task with priority less than i, AND task with priority equal to or higher than i.

Note that this covers both direct and push-through blocking delays, and that the low priority task can interfere with the higher priority task only once.

Priority Inheritance

<u>Example (1):</u>

Task i	p_i	T_i	d_i	
A	9	75	75	Uses k with time= 5
B	20	35	35	
C	5	20	20	Uses resource k

If fixed priorities are selected according to RM.

$R_C = 5 + 5 = 10 < d_C$

$R_B = 20 + 5 + \left\lceil \frac{R_B}{20} \right\rceil \times 5$ which converges to 35= d_B.

$R_A = 9 + \left\lceil \frac{R_A}{20} \right\rceil \times 5 + \left\lceil \frac{R_A}{35} \right\rceil \times 20$ which converges to 69< d_A.

Message queues and mailboxes

Some communication mechanisms, such as semaphores, require a shared memory among tasks. The method of *message passing* is more general.

A minimum set of system calls that handle message passing is the following:

send (*destination*, &*message*)

receive (*source*, &*message*)

Many design options exist for the message format, addressing methods, synchronization modes, and queuing discipline. All these can affect program timing.

Message queues and mailboxes

Message Format

Typically, message is a sequence of bytes with fixed or variable length. Correct interpretation of message content is the responsibility of the communicating tasks, not the operating system.

Addressing Method

- *Direct addressing*: using task id.

  Allows only one-to-one communication.

- *Indirect addressing* : using *mailboxes*

  Allows many-to-one, one-to-many or many-to-many modes.

Queuing and Synchronization

A message sent but not yet received is queued by the system. Queue will have a pre-specified maximum capacity.

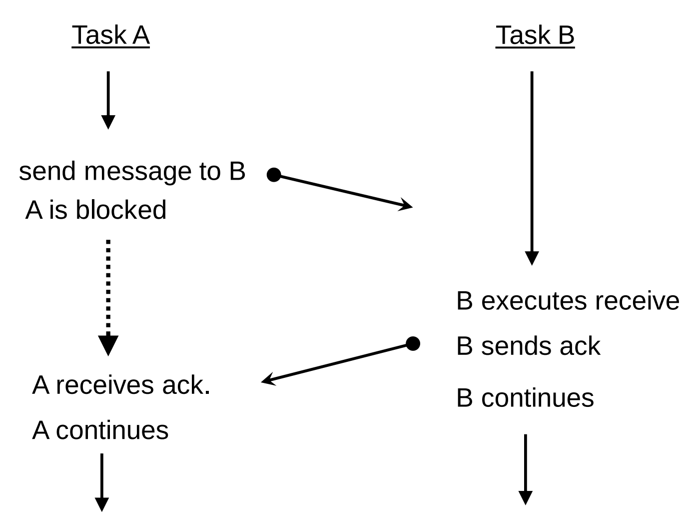
If no message is available, the receiver will typically be blocked until one arrives. Alternatively, it can return immediately with an error code.

Sender can operate in one of two modes:

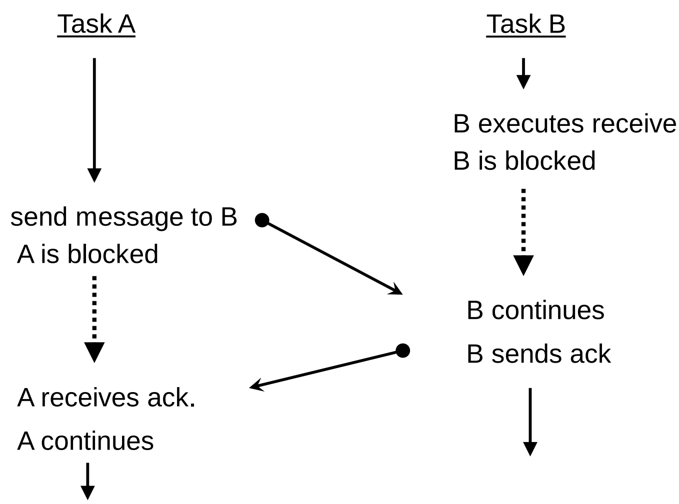
- in asynchronous send, sender will continue operation regardless of whether the message was received or not.

- In synchronous send, the sender will be blocked until it receives an acknowledgment from the receiver.

Message Synchronization Modes



Message Synchronization Modes



Message queues and mailboxes

Example (2): Mutual exclusion using messages

A controlling task (call it SYNCH) will be used to control the access to the shared resource. The mailbox Synch_mail will also be created.

To access resource, any task will use the following sequence:

```
send(Synch_mail,’request’);
receive(Synch,’permit’);
Access_resource;
send(Synch_mail,’clear’);
```

Message queues and mailboxes

<u>Example (3):</u> The Reader/Writer Problem

A buffer of size n with reader and writer tasks running at different speeds.

<u>Reader task</u>

```
receive(writer,'full');
Read_data_item;
send(writer,'empty');
```

<u>Writer task</u>

```
receive(reader,'empty');
Write_data_item;
send(reader,'full');
```


Message Synchronization Modes

Even with asynchronous send, the sender may be blocked if the receiving queue or mailbox is full.

In a remote invocation mode, sender waits for a reply from the receiver and not just an acknowledgment of receipt.

The queue of messages can be managed using FCFS, or *priorities* may be given to the messages.

Introduction to FreeRTOS

FreeRTOS is an open source real-time kernel suitable for small to medium sized microcontrollers (32K to 512K of flash memory and 16K to 256K of RAM).

It is written mostly in C with few extra assembly routines. This results in high degree of portability.

Kernel is a library of modules which are linked to application code to build the application executable image. Several add-on libraries are available. Kernel itself has a small footprint (as low as 9KB).

FreeRTOS Tasks

Applications are arranged as a collection of independent threads of execution, called "tasks". In most implementations there is no memory protection, so all tasks share the same memory space.

Tasks can be created by main program and also a task can create other tasks.

In each task, a specific C function is executed. It is possible to run the same function in any number of separate independent tasks.

Tasks are created using the API function `xTaskCreate( )`.

FreeRTOS Tasks

```
xTaskCreate(  ATaskFunction,
              /* Name of executed function.*/
               "LED",
               /* Name of Task used in debugging.*/
               1024,
              /*Size of task stack in words.*/
               NULL,
              /*Parameters passed to the task.*/
               2 ,
              /*The priority of the task.*/
              NULL );
               /* A handle to the task if needed.*/
```


FreeRTOS Tasks

```
*/Task function should take a void pointer parameter
 and return void.*/
 void ATaskFunction (void *pvParameters)
 {
    /* variable declarations goes here. */
   for ( ; ; )
   {
/*Task functionality always within an infinite loop*/
    }

   /* Task functions never return to caller.*/
 }
```

FreeRTOS Tasks

To use API functions within program we need to include files `FreeRTOS.h` and `task.h`.

Task has an initial priority represented by an unsigned word. 0 corresponds to lowest priority, and highest priority is determined by the constant `configMAX_PRIORITIES` defined in file `FreeRTOSConfig.h`.

Task scheduler starts running by calling the API function:

```
vTaskStartScheduler ( );
```

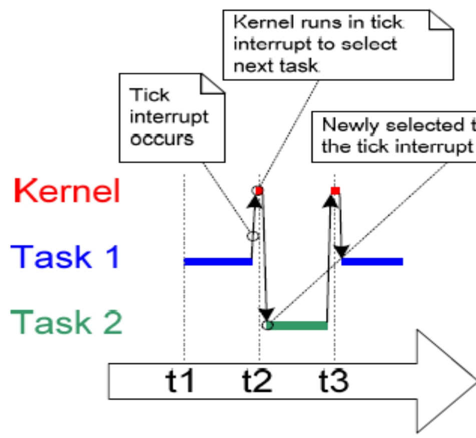

FreeRTOS Task Scheduling

Scheduler runs higher priority tasks first. These tasks preempt lower priority tasks.

Equal priority tasks run in round-robin with time slice specified by `configTICK_RATE_HZ`.

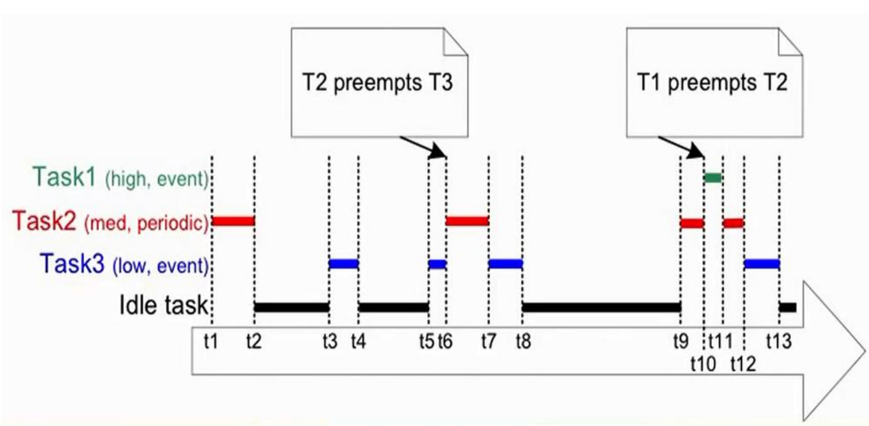
System time is represented by ticks of the periodic interrupt. For example, the API function `vTaskDelay (100)` puts the calling task in blocked state for 100 ticks.

FreeRTOS Task Scheduling



Source: [Barry 10]

FreeRTOS Task Scheduling



Source: [Barry 10]