# Real-Time Embedded Systems

## FreeRTOS and Thread Safe Programming

Alex Hoffman

Technical University of Munich
Chair of Real-Time Computer Systems (RCS)

August 13, 2020

Lehrstuhl für
Realzeit-Computersysteme

# What is an Operating System?

*An **operating system** is a software which manages hardware and software resources. It provides resource management functions to upper software layers.*

## General-purpose OSes

wikipedia.org

pngimg.com

logos.wikia.com


logos.wikia.com

## Real-time OSes


micrium.com


freertos.org


blogs.windriver.com


rdcs.eu

# What is a Real-Time OS?

*A **real-time operating system** is an operating system which provides predictable timing for the execution of time-critical tasks.*

## Microkernel


micrium.com


freertos.org


rdcs.eu

## Monolithic kernel

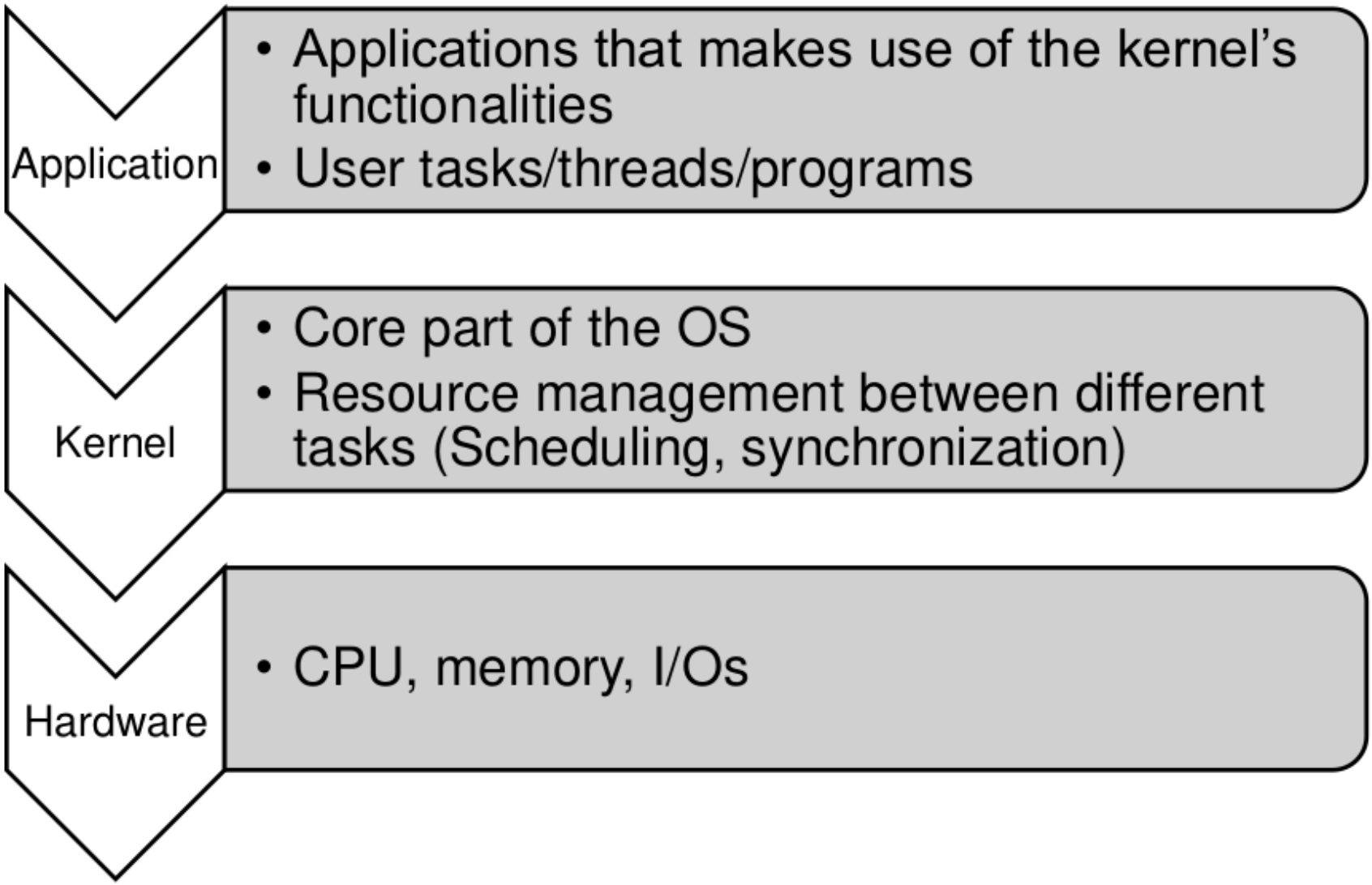

logos.wikia.com

# Backbone of all OSes - the kernel

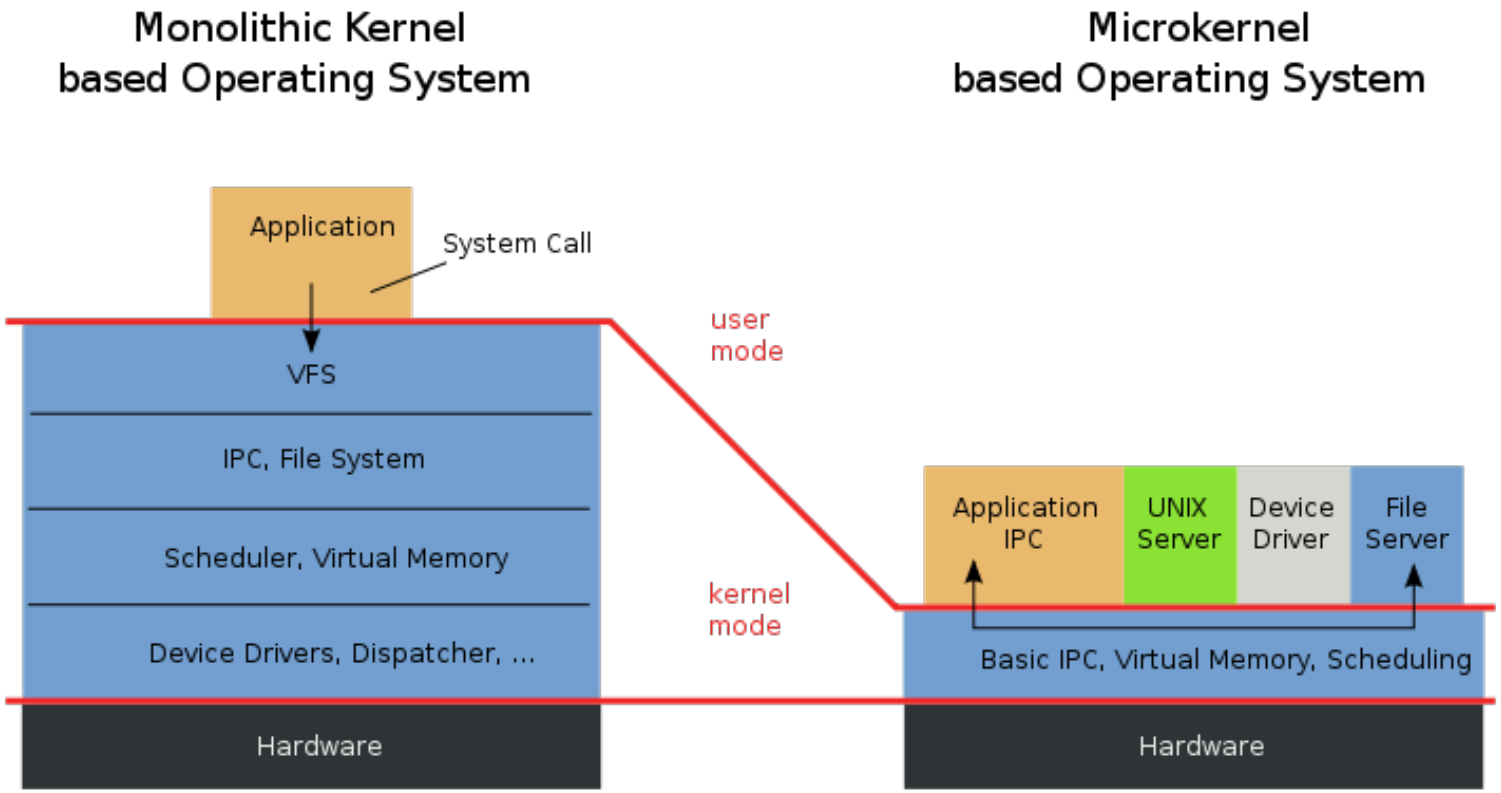
- Handles interfacing top level software with the system's hardware
- FreeRTOS is just a mini Real-Time kernel


madshrimps.be

# Where the Kernel Sits in a System



# Monolithic Kernels vs. Microkernels



# Before we get to FreeRTOS....

Operating systems need to be designed such that there is compatibility across systems.

How is this achieved?

**POSIX**

# What is POSIX?

The **Portable Operating System Interface (POSIX)** is a set of standards from the IEEE that define a number of important properties an OS should have.

Based upon UNIX OSs, POSIX defines: **Core Services**

- Processes: creation and control
- Inter-process communication (IPC): Passing of information or signals (eg. SIGKILL, SIGSEGV) between processes
- File system operations
- Pipeline (Pipe): a process communication pipeline for chaining processes using standard streams
- C library
- I/O Ports
- Process Triggers

# What is POSIX?

Based upon UNIX OSs, POSIX defines: **Real-Time and Thread Extensions**

- Scheduling
  - Priority scheduling
  - Thread scheduling
- Synchronisation
  - Real-time signals
  - Clocks and timers
  - Semaphores
  - Message passing - invoking behaviour in a process
  - Shared memory - IPC, memory locking etc
  - Asynchronous and synchronous I/O

# POSIX vs FreeRTOS

POSIX sets the standard for computer science. A pipe is a universal term in CS but a queue (not to be confused with a message queue) is more FreeRTOS specific.

All FreeRTOS functionality can be explained via POSIX functionality.

Core FreeRTOS API components:

- Tasks
- Task notifications
- Queues
- Messages
- Semaphores/mutexes

# FreeRTOS (1)

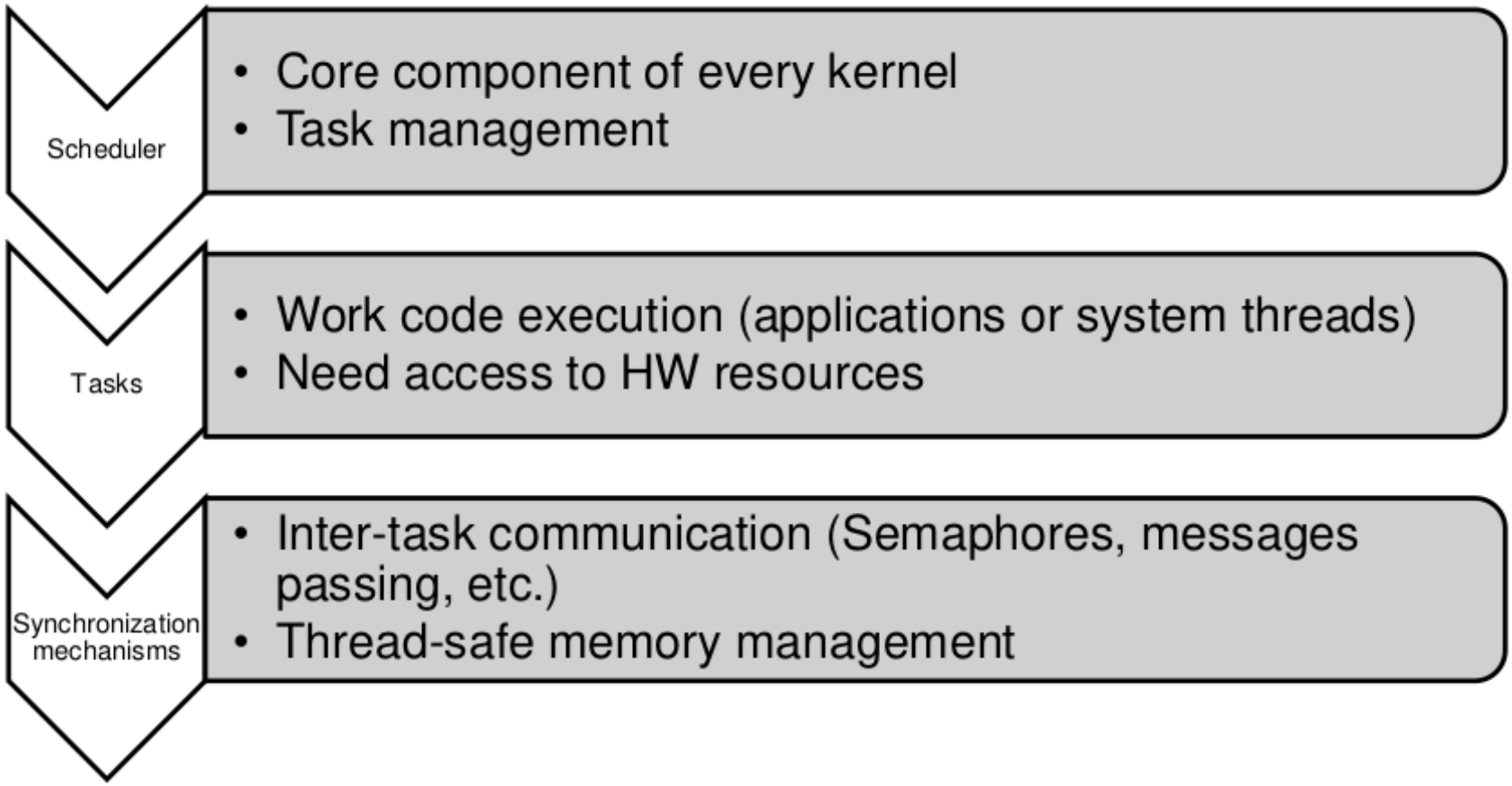
- Considered microkernel, but:
  - OS is compiled with the system
  - No possibility to load application dynamically
  - More of a library fashioned OS (API)
- Written in C, only hardware "ports" in assembly
  $\rightarrow$ can be used on many different architectures
- No device drivers or advanced memory management
- Is the example RTOS in the ESPLaboratory

# FreeRTOS (2)

- Provides time and **multitasking** support
- Gives good possibility to structure your code
  $\rightarrow$ Reduces complexity of user-written code
- Open-source but also commercial versions available
- Well documented and easy to use API
- Large community ($\rightarrow$ lots of online support)
- Available for many hardware platforms (generic)

- Large memory footprint compared to other microkernel RTOSes
- Comparably slow due to the generic API ($\rightarrow$ overhead)

# Components of an (RT)OS Kernel



# Tasks

POSIX defines processes and threads. The implementation depends on the OS

**Process:**

- Instance of a computer program

```
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
alxhoff   20205  3.8  0.8 2972404 134432 ?      Sl   09:58   2:23 texmaker
```

- Contains the program's code
- Can be composed of multiple threads of execution which execute concurrently

```
  PID  SPID TTY          TIME CMD
20205 20205 ?        00:02:23 texmaker
20205 20207 ?        00:00:04 QXcbEventQueue
20205 20208 ?        00:00:00 QDBusConnection
20205 20209 ?        00:00:00 texmake:disk$0
20205 20210 ?        00:00:00 texmaker
```

# Threads

Smallest sequence of program code that can be handled by the scheduler. Allows for a process to be broken up into smaller packets of work.

- Executed concurrently *A concurrent system is one where a thread execution is not dependent on the completion of all other thread executions*
- Share resources such as memory
- Implementation is OS dependant
  - Linux does not differentiate between processes and threads
  - Threads in Linux are simply processes that share an other resources with their parent processes
  - Created using the `pthread_create` function on Linux, as part of the POSIX thread (pthread) library

# Advantages of threads

The abstraction of a program into threads has the following advantages:

- Modularization of code
  - Responsiveness can be improved be segmenting, for example, input and output.
    *A single threaded program that is blocked waiting for I/O would cause the program to "freeze". If I/O and program output are in different threads this problem is negated.*
- Faster execution using multiple hardware resources such as CPU cores or CPUs.
- Lower resource usage as resources can be shared, for example a web server handling multiple client requests.

# Advantages of threads - cont.

- Better system utilization by using hardware components concurrently. For example, reading data from disk and from the network concurrently.
- Communication, threads can boycott the need for IPCs using other methods such as data, code and files.
- Parallelization, allows work to be split into parallel subtasks. For example, image processing.

# Multitasking

**Multitasking vs Multithreading:**

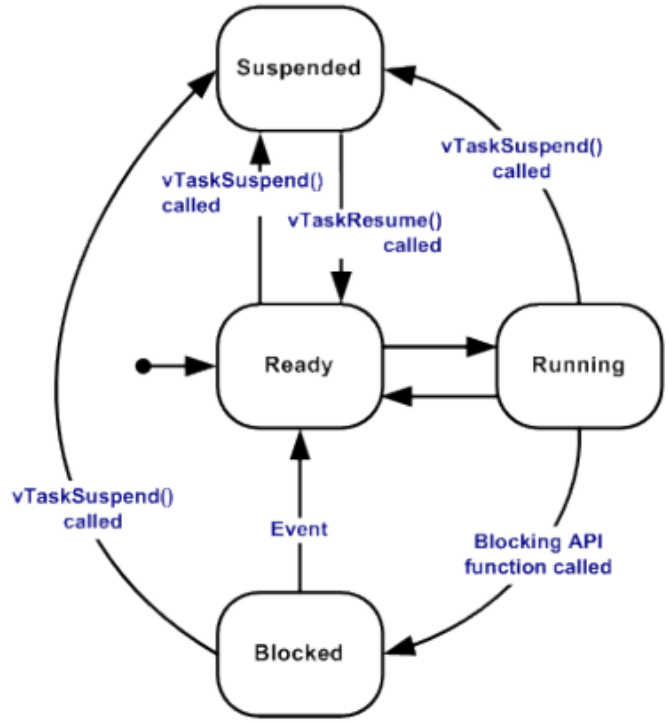
- Multithreading is the aforementioned division of a program into smaller "chunks" of work
- Multiprocessing is a hardware property of a computer system
  - The use of two or more CPU cores to allocate the execution of tasks between CPUs

But what if your system is a single core system? How do I still benefit from multithreading?
**Context switching**

# Task states

- Running – the process currently doing work
- Ready – is available to run as soon as the OS allows it
- Blocked – waiting for an event to allow it to run
- Suspended – not checked by the OS, task is shut off

FreeRTOS task states:

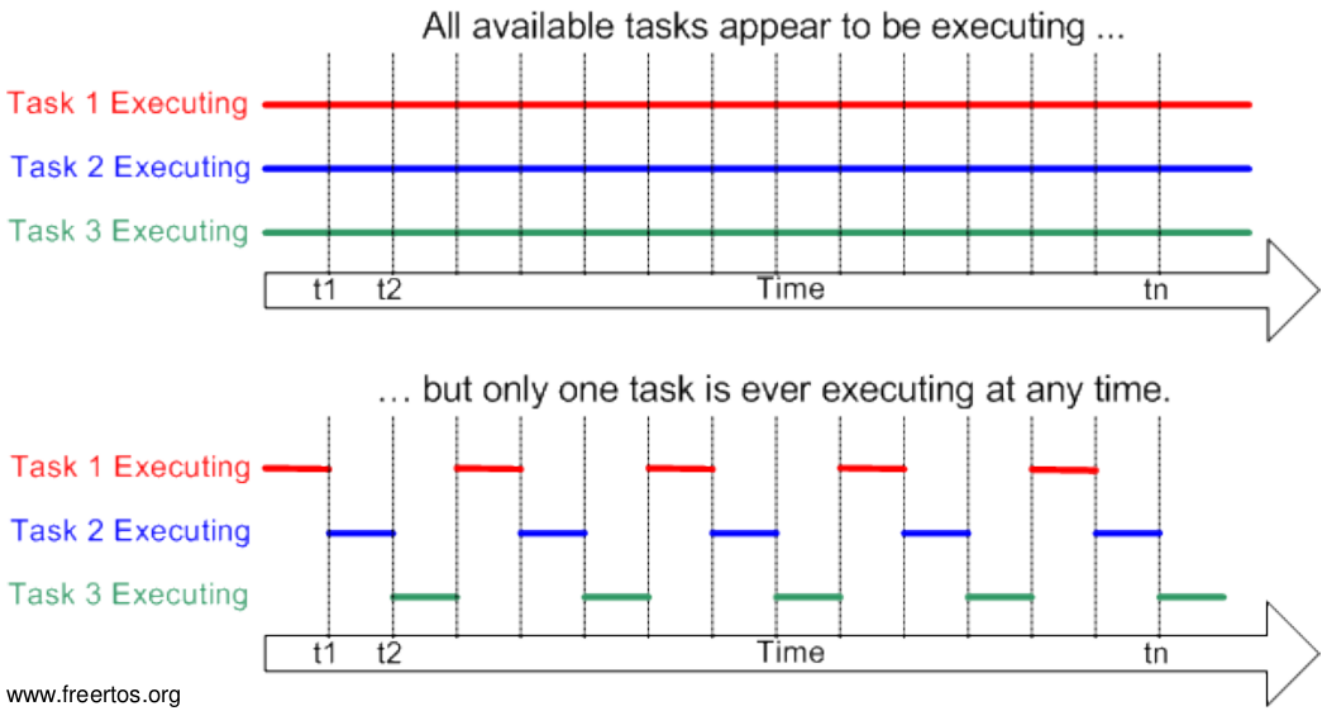



www.freertos.org

# Single core multithreading

Using context switching pseudo-parallel and concurrent execution of tasks is possible.

Swapping tasks on and off of the CPU is handled by the scheduler, arguably the core component of any OS (kernel).



# Contex switching

The ***context*** of a system is the contents of the CPU's registers and program counter at any point in time.

A ***context switch*** happens when one process is to take control of the CPU resource, "switching" out the current process.

This is done by saving the context and kernel-mode stack of the currently executing process in memory, such that it can be resumed later. The next scheduled process's context and stack are loaded from memory into the CPU, such that execution can be started or resumed from the location indicated by the program counter.

# Context switch 1/7

**Scenario:**

- Task A is executing
- Timer interrupt arrives at processor
- Scheduler is invoked and decides that task B has to run next
- Task B is executing

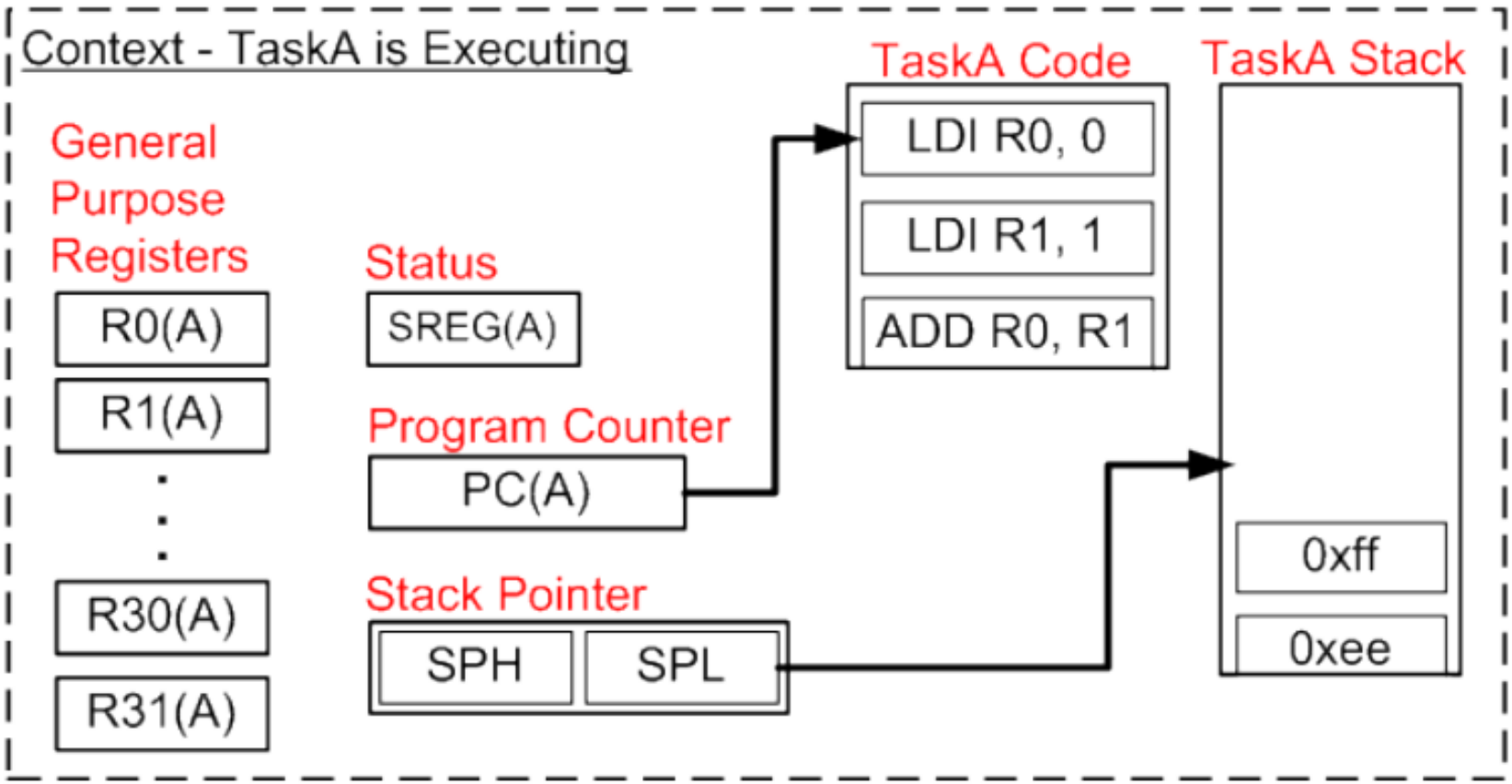
**Problem:**

- Registers contain the context of task A
- Cannot just simply start task B

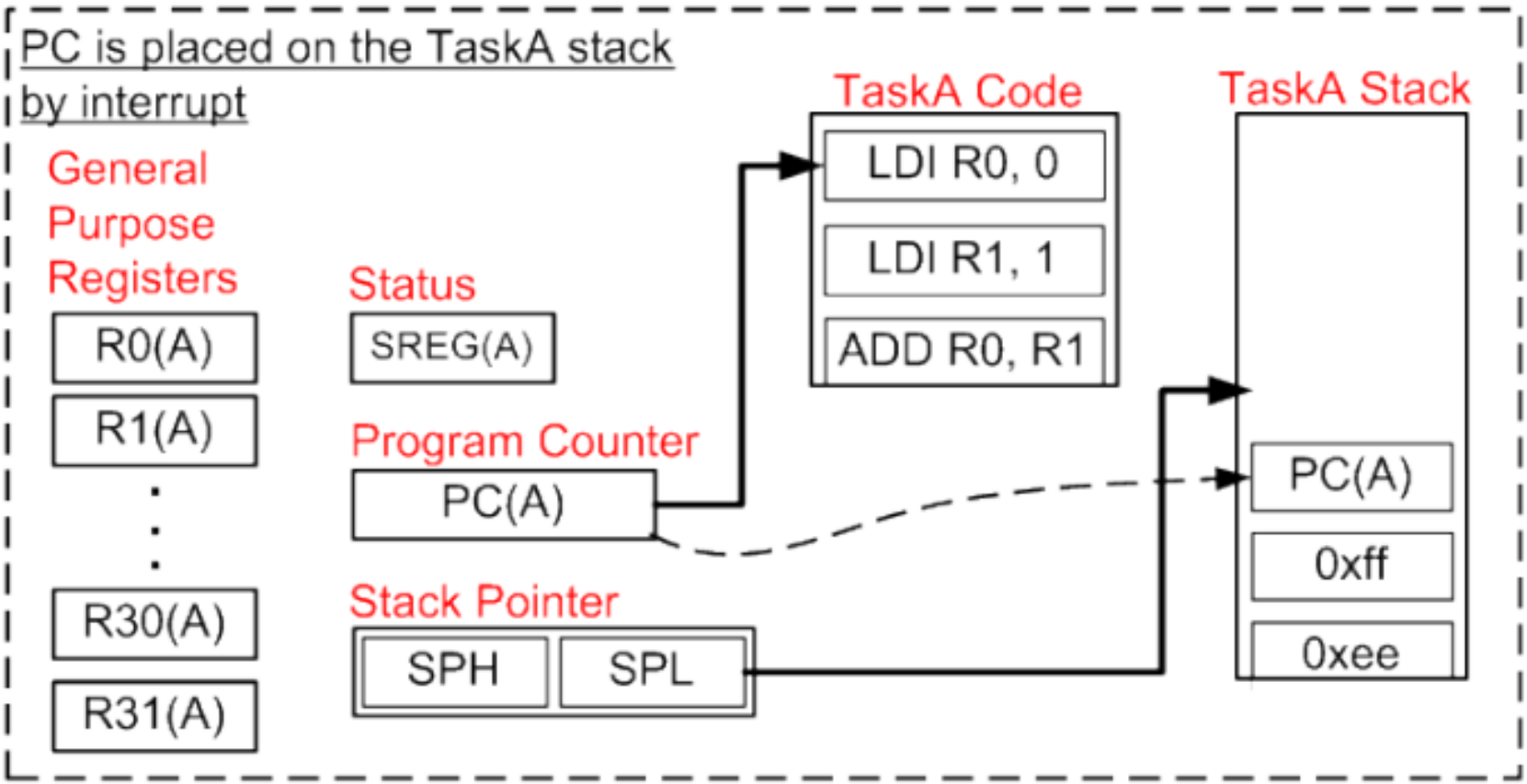
**Solution:**

- Store the context of A and restore the context of B

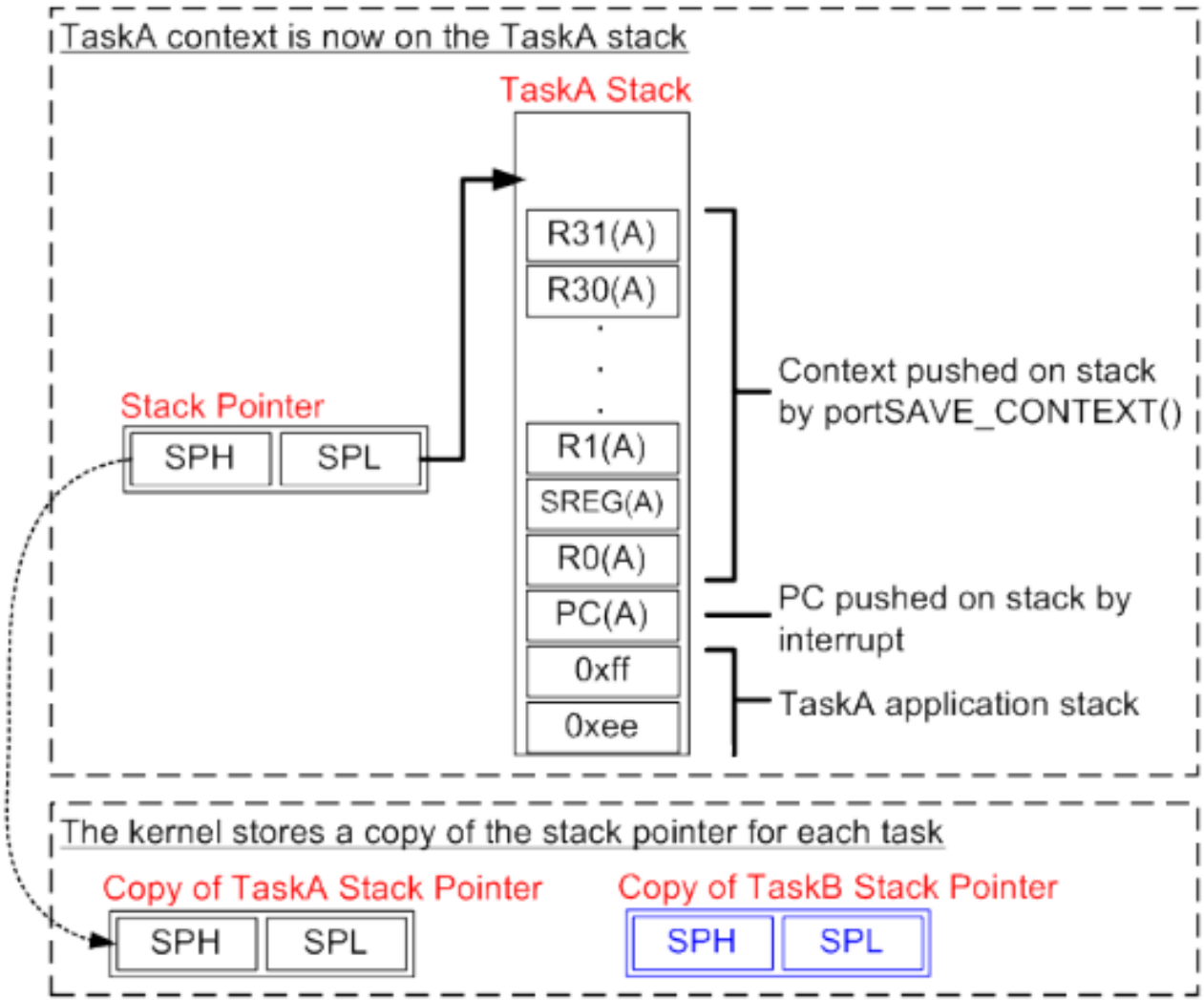
# Context switch 2/7



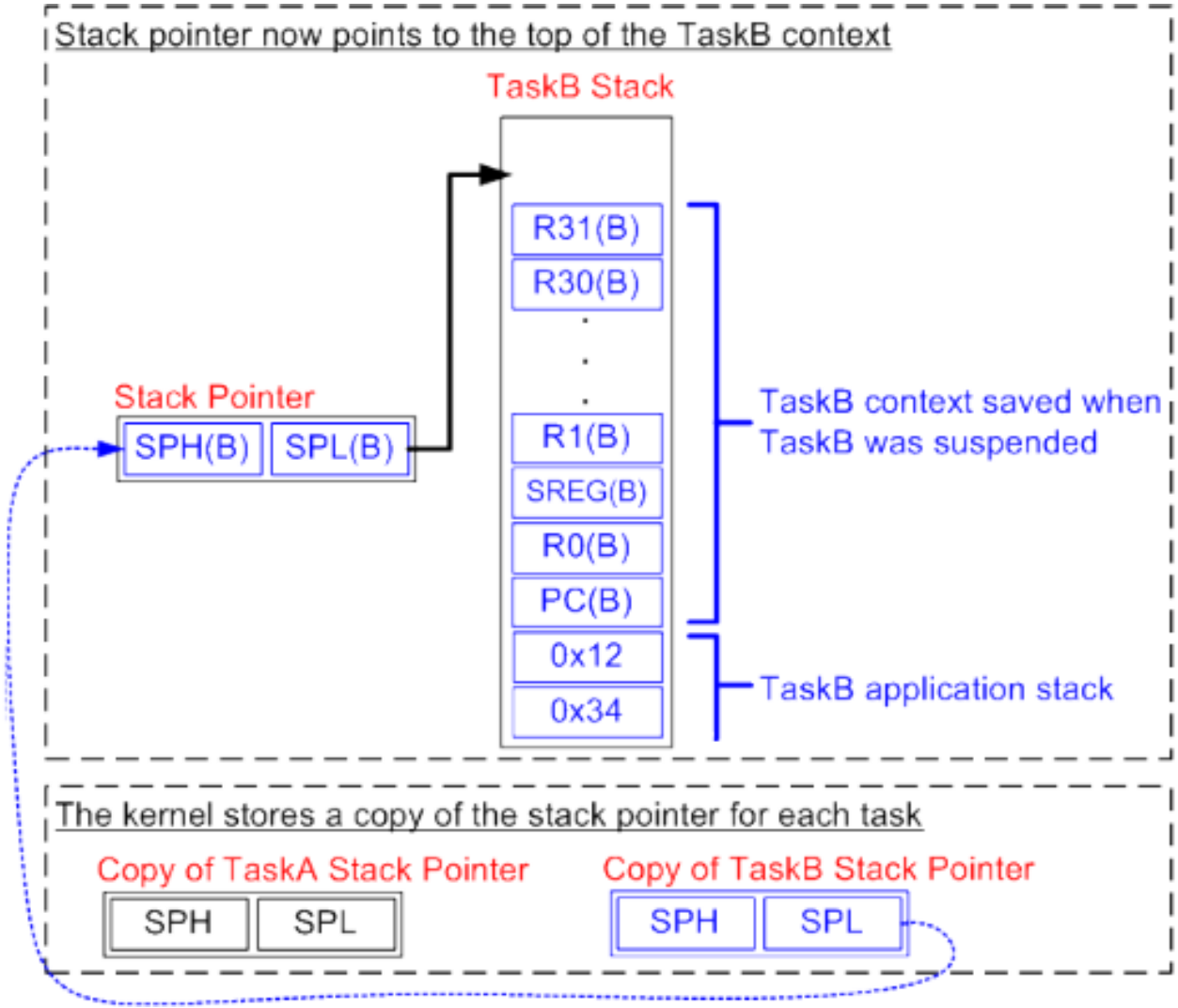
# Context switch 3/7



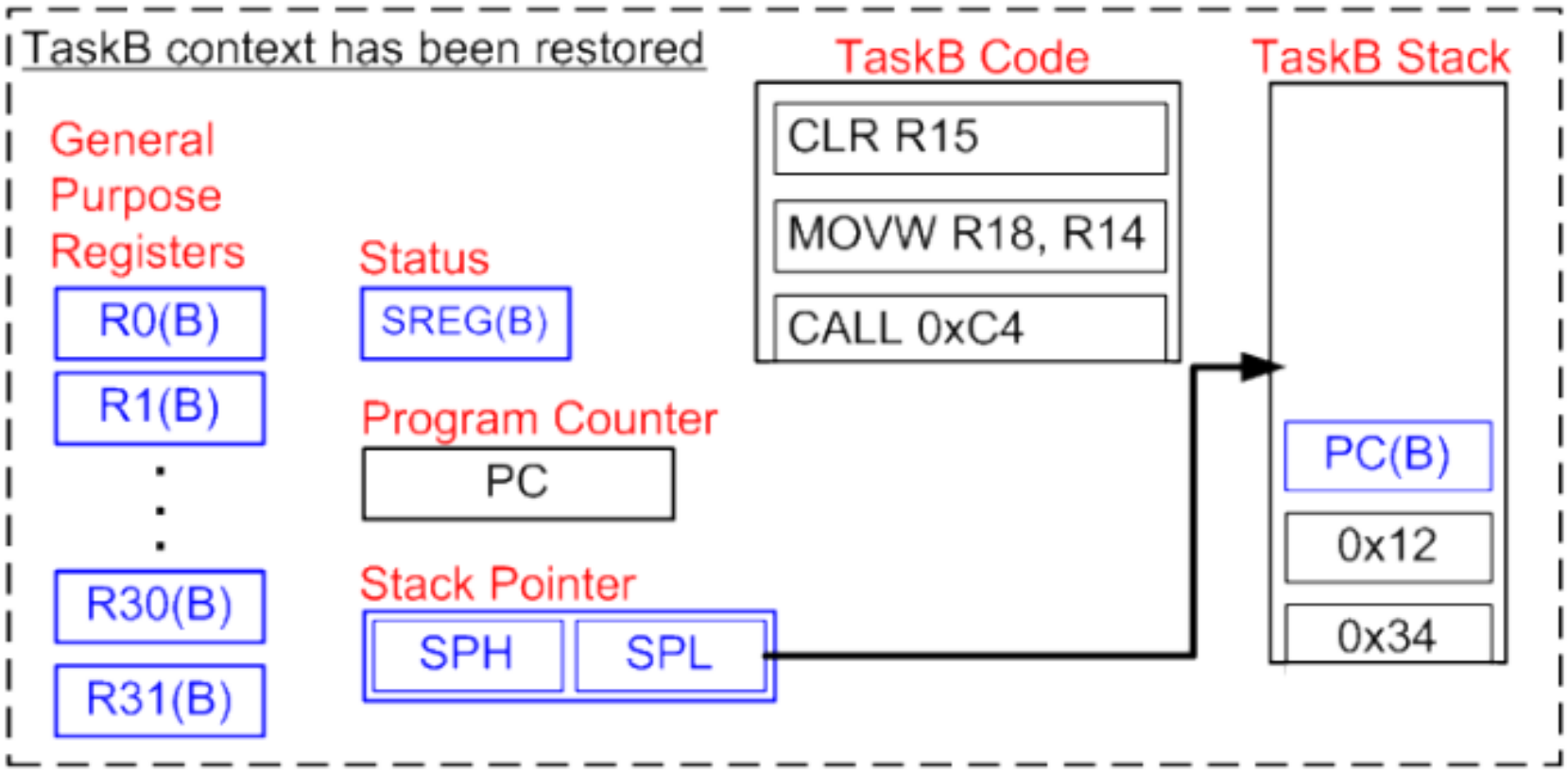
# Context switch 4/7



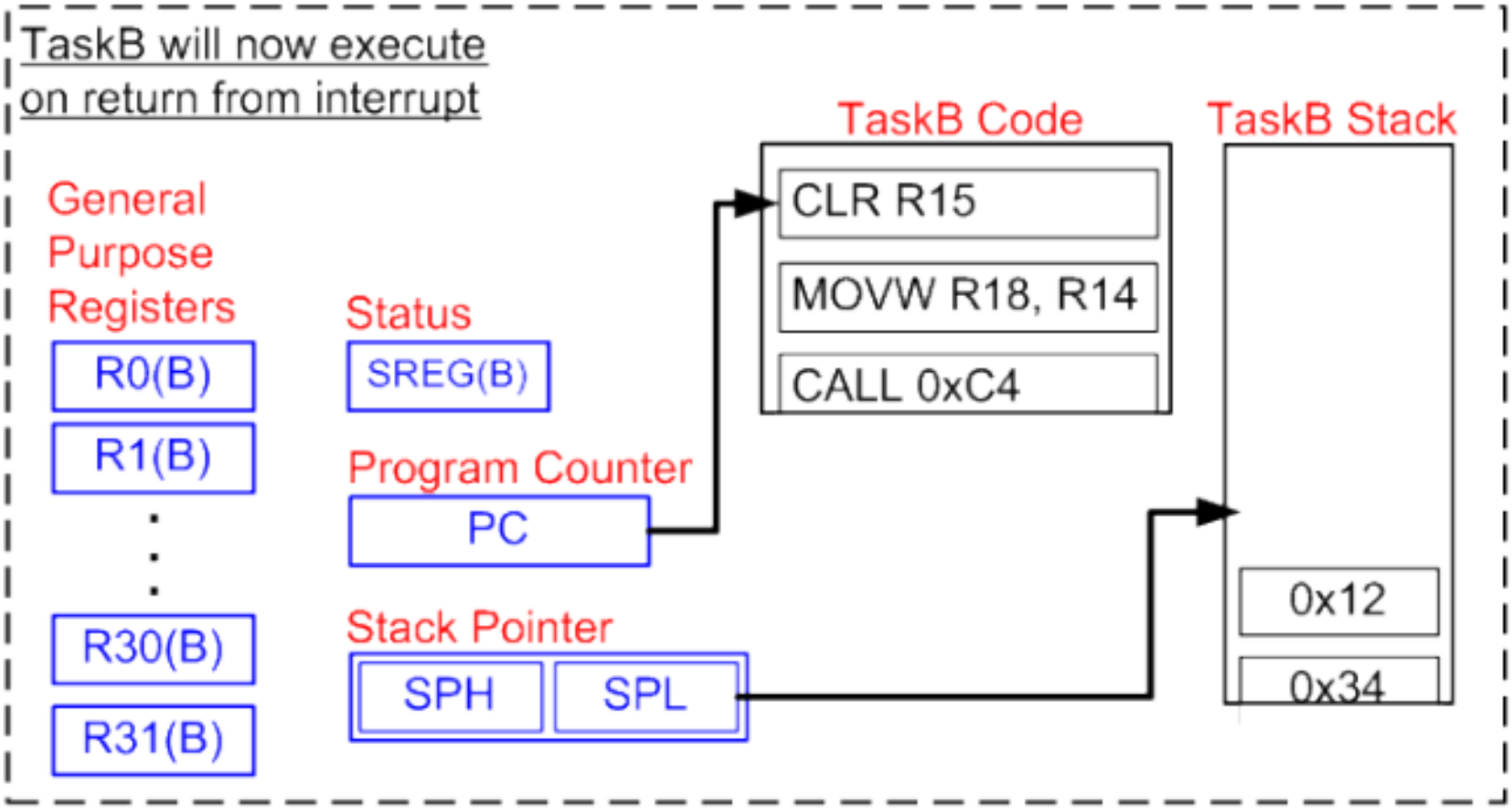
# Context switch 5/7



# Context switch 6/7



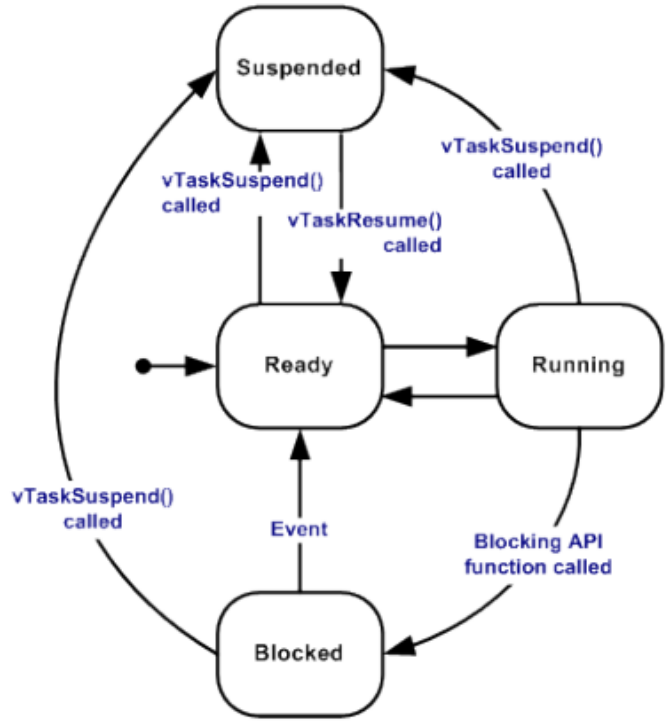
# Context switch 7/7



# Tasks (1)

- Execution of application work spread over multiple tasks
- Every task has a priority, low priority == low number
- Different task states →
- Each task has its own context (stack etc.)
  → allows for context switching between tasks
- No dependency on other tasks or scheduler
- Periodic or event triggered task switching
- Co-routines also available but rarely used

FreeRTOS task states:




www.freertos.org

# Tasks (2)

- Usually implemented as infinite loop
- The loop needs to be paused at some point in time (endless execution results in an error)
- Tasks in FreeRTOS are not allowed to return, but should be deleted if not needed

# FreeRTOS Task API 1

- xTaskCreate( task code, name, stack size, parameters, priority, task handle)
- vTaskDelay(delay duration)
- vTaskDelayUntil(previous wake time, time increment)
- vTaskSuspend(task handle)
- vTaskResume(task handle)

# FreeRTOS Task API 2

Example task code:

```
/* Task to be created. */
void vTaskCode( void * pvParameters ){
  for( ;; ){
    /* Task code goes here */
  }
}
```

```
TaskHandle_t xHandle = NULL;
/* Function that creates a task. */
void vOtherFunction( void ){
BaseType_t xReturned;
    /* Create the task, storing the handle. */
    xReturned = xTaskCreate( vTaskCode, "NAME",
    STACK_SIZE, ( void * ) 1, tskIDLE_PRIORITY, &
    xHandle );
    if( xReturned == pdPASS ){  /* The task was created
    .  Use the task's handle to delete the task. */
        vTaskDelete( xHandle );
    }
}
```

# Scheduling

The scheduler is responsible for allocating the CPU's resources (time executing on CPU) between running tasks on the system.

The scheduler utilizes scheduling policies (algorithms) to determine which task should run.

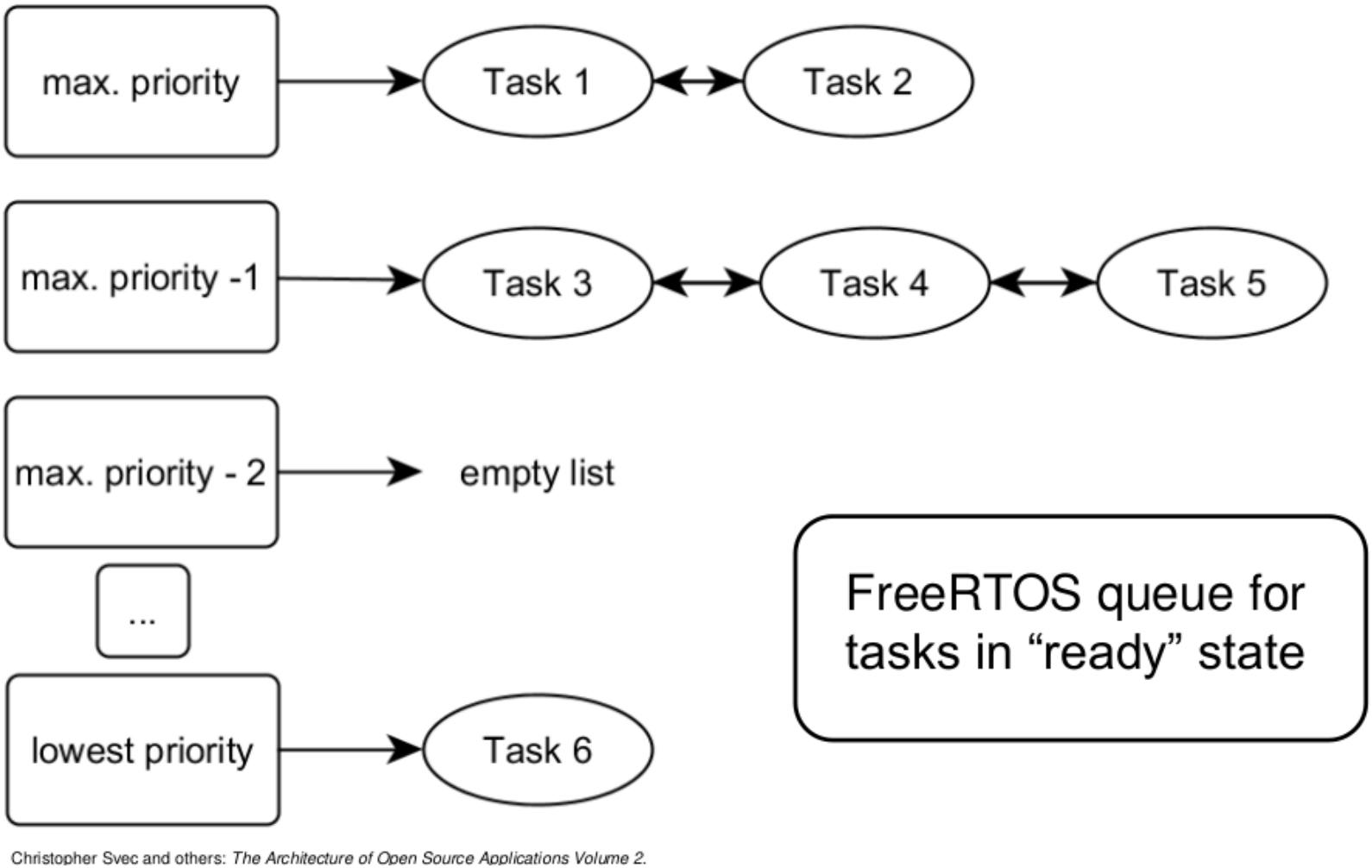
Linux uses the *Completely Fair Scheduling* algorithm that tries to insure that within a fixed time interval each thread on the system runs at least once, done by using runtimes and weights to calculate priorities.

FreeRTOS uses user assigned task priorities to perform its scheduling.

# Scheduler (1)

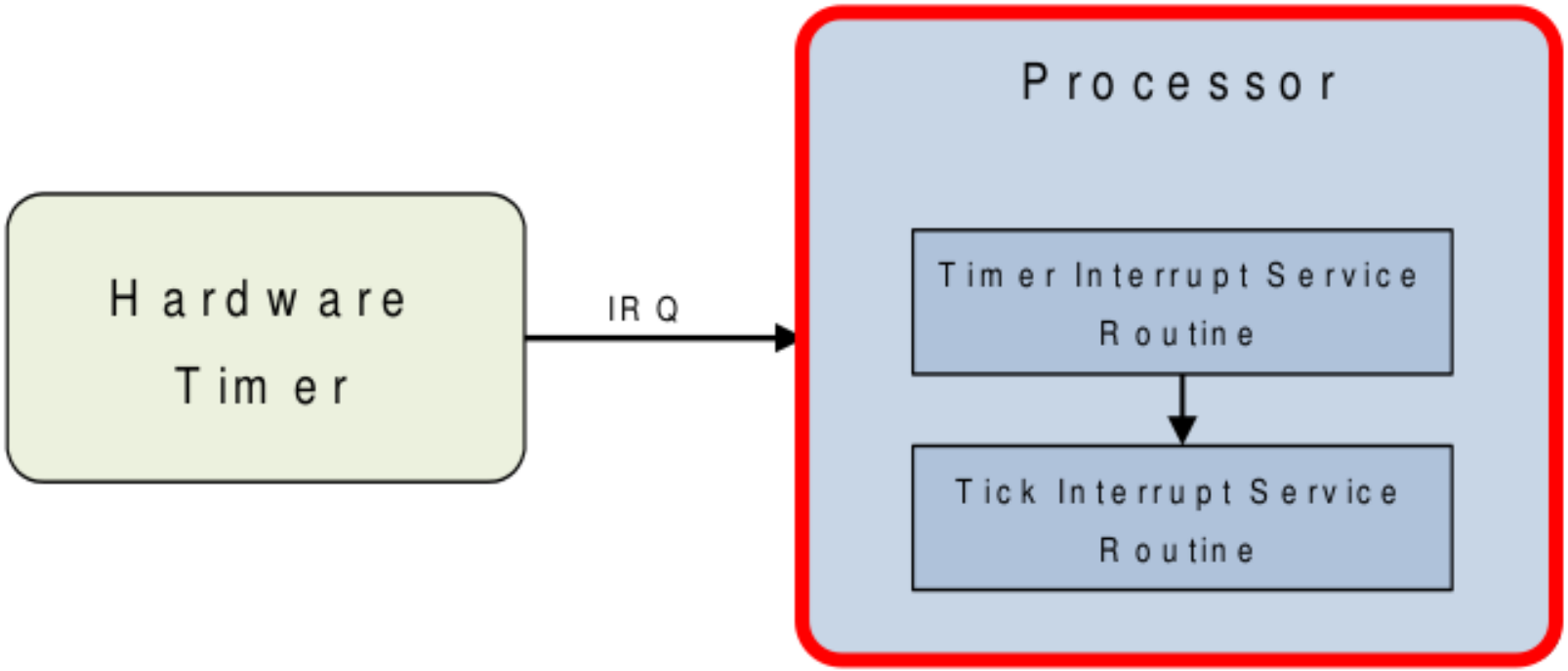
- The scheduler decides which task is allowed to run and when
- Scheduling policy: algorithm which is used by the scheduler to make its decisions
  - Preemptive: tasks with higher priority can interrupt
  - Cooperative: next task is allowed to run when the processor is free by the current task
  - Hybrid
- Priority and round-robin based scheduling policy
- Scheduler has to be started from the "`main()`" function of the program by calling "`vTaskStartScheduler()`"

# Scheduler (2)




Christopher Svec and others: *The Architecture of Open Source Applications Volume 2.*

# How is the scheduler called?




A hardware timer (SysTick on ARM) is configured to provide the kernel with a periodic interval that drives the scheduler and in turn context switches. Also provides the FreeRTOS kernel with its tick.

# Can multiple tasks execute within a tick?

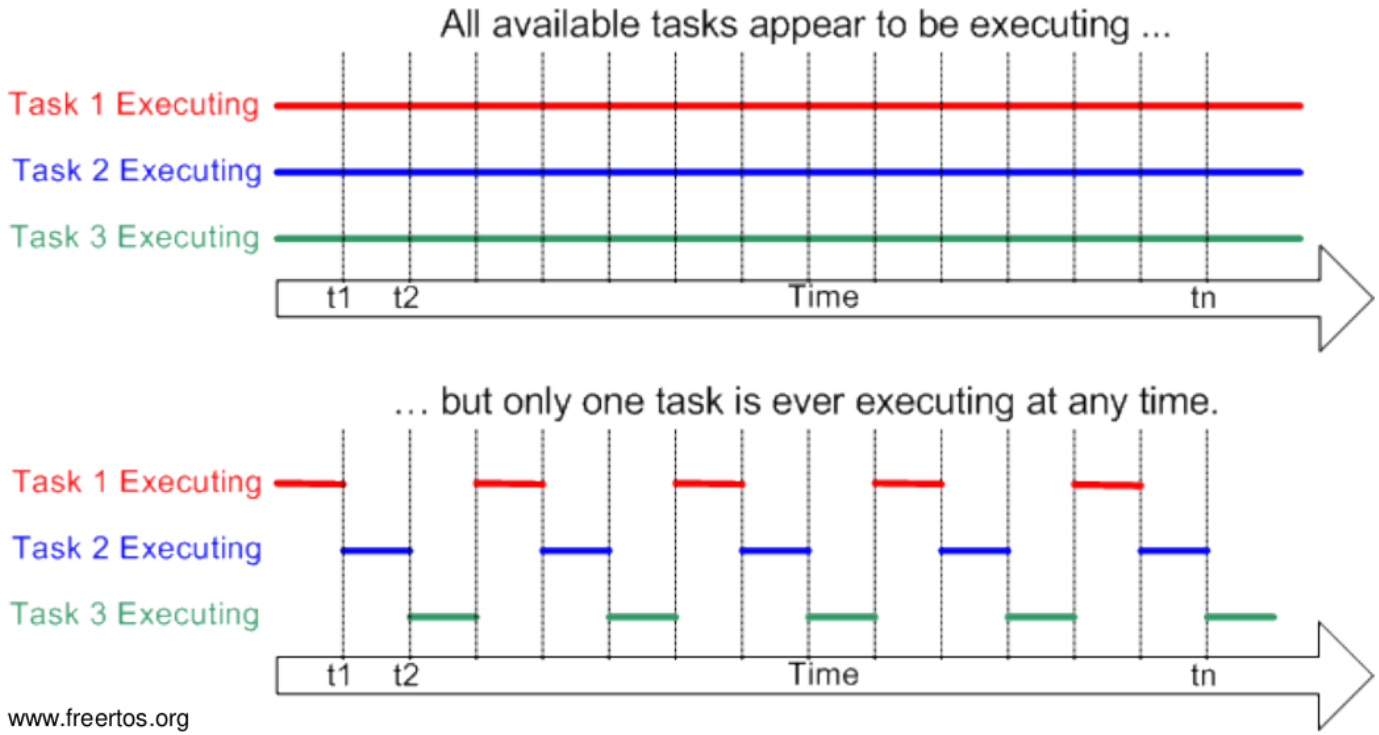
- Scheduling determines the execution order within a tick.
- FreeRTOS or other OSs (eg. Linux) also force manual context switching/rescheduling after certain events (eg. current task becoming blocked)

```
677     void vTaskDelay( portTickType xTicksToDelay )
678     {
679     portTickType xTimeToWake;
680     signed portBASE_TYPE xAlreadyYielded = pdFALSE;
681
682         /* A delay time of zero just forces a reschedule. */
683         if( xTicksToDelay > ( portTickType ) 0 )
684         {
685             vTaskSuspendAll();
686 +---- 36 lines: {-----------------------------------------------------------
722             xAlreadyYielded = xTaskResumeAll();
723         }
724
725         /* Force a reschedule if xTaskResumeAll has not already done so, we may
726         have put ourselves to sleep. */
727         if( !xAlreadyYielded )
728         {
729             portYIELD_WITHIN_API();
730         }
731     }
```

*taskYEILD/portYEILD is a request to perform a context switch to the next scheduled task. Called here in the vTaskDelay FreeRTOS function.*

# Task/Context Switch (1)

- Call to scheduler periodically (every "Tick") or event based
- Tick length can be adjusted in source code
- Pseudo-parallel execution of tasks
- Scheduler invokes context switch
- Context switch is also triggered when a task is blocked/suspended



# Task/Context Switch (2)

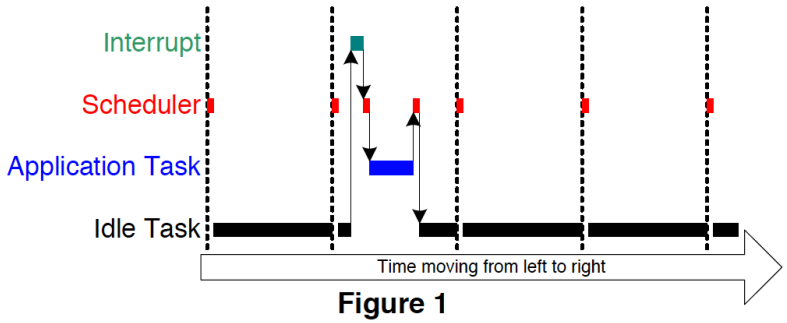


Figure 1

Kernel tick

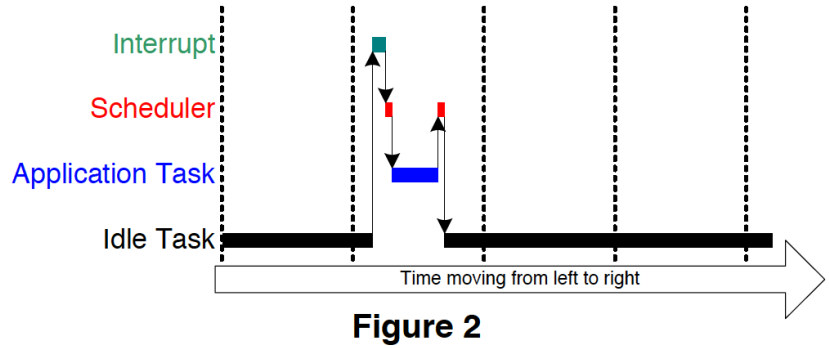


Figure 2

Tickless kernel

- Scheduler should handle delegation of work
- Performing large amounts of work in interrupt handlers can be dangerous to system performance
  $\rightarrow$ Can degrade system preemptiveness
- Tick frequency is a trade-off between system performance and overhead

# Scheduling in FreeRTOS

- Only one task can be running a given moment in time on a single core processor
- The scheduler chooses the highest priority task in the *Ready* state to enter the *Running* state
- Blocked tasks wait until an event and are moved back to the *Ready* state when the event occurs, ready for scheduling
  - Temporal events occur at particular points in time
  - Synchronization events occur when a task or interrupt sends information using a task notification, queue, event group or a type of semaphore

# Thread-safe programming

- Threads using shared resources (same address space) need to ensure that the resources are shared in a manner that respects the system's concurrency capabilities.
- Concurrent access and code safety is largely defined by software libraries.
- Concurrent reads might be thread-safe, but can present race condition problems.
- Concurrent reading and writing is never thread safe, leads to undefined behaviour.

# Example scenario

- Task A wants to increment a variable value
- A higher priority task B pre-empts task A and also wants to increment the same variable

**Read, modify, write operations**:

```
/* The C code being compiled. */
PORTA |= 0x01;

/* The assembly code produced when the C code is compiled. */
LOAD   R1,[#PORTA] ; Read a value from PORTA into R1
MOVE   R2,#0x01    ; Move the absolute constant 1 into R2
OR     R1,R2       ; Bitwise OR R1 (PORTA) with R2 (constant 1)
STORE  R1,[#PORTA] ; Store the new value back to PORTA
```

**Problem:** Modifying this variable is a non-atomic operation in that it takes more than one instruction to complete and can be interrupted.

# Resulting problem

- Task A loads the variable's value into a register
- Task A is pre-empted by Task B before the operation is completed. Task A's context (including registers) is saved
- Higher priority Task B reads, updates and writes the variable value, then entering a blocked state allowing task A to resume
- Task A's context is restored and it then increments it's copy of the variable and writes the variable back to memory
- The stored value was not only incremented by task A and not by task B, thus making the data effectively corrupted (out of date)

# Similar Problems

- Accessing I/O - Peripherals can't differentiate between input streams
- Non-atomic access to variables - large variables requiring multiple operation to read a stored value
- Non-reentrant functions - functions who access data that is non local to the calling task's stack (eg. a function employing static variables)

# Solution: Locking

- Locking prevents simultaneous access to shared resources
- Must obtain control of the lock before accessing resource
- Obtaining the lock made thread-safe by using atomic increment and decrement operations to insure they are not interrupted

*Analogy: when the class has an object (such as a fluffy ball) that must be passed around that allows a single individual to talk at any point in time*

# Semaphores

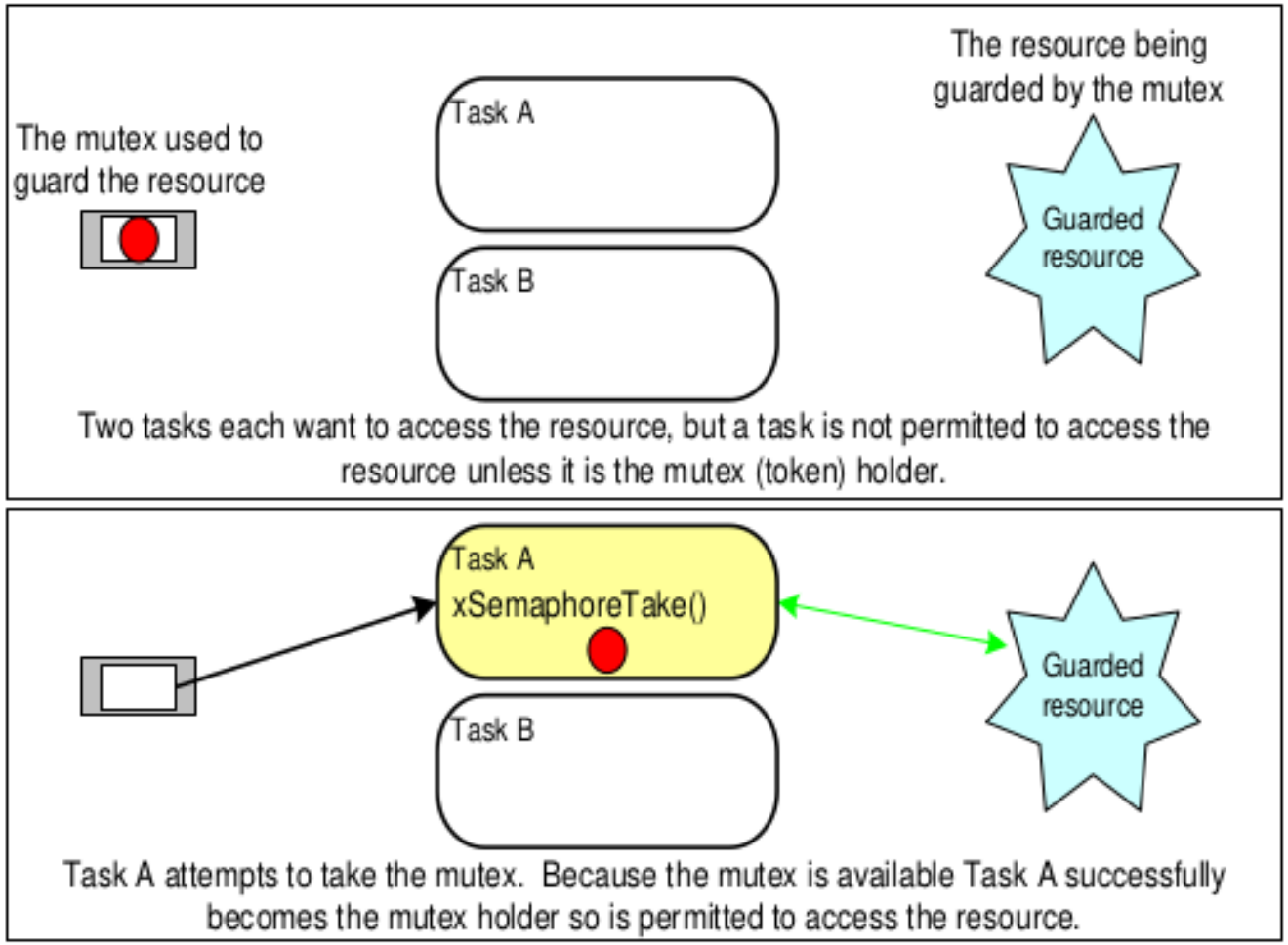
- An integer whos value cannot fall below zero that are used for both mutual exclusions and synchronization.
- Do not protect against priority inheritance (explained soon).
- Can either be incremented or decremented, value cannot drop below zero.
- If value is allowed to be incremented above 1 then the semaphore is a counting semaphore - useful for counting events and resource management where multiple copies of a resource are available.
- If semaphore is capped at 1 then it is a binary semaphore - used for synchronization and resource management where only one resource is available.

# FreeRTOS API

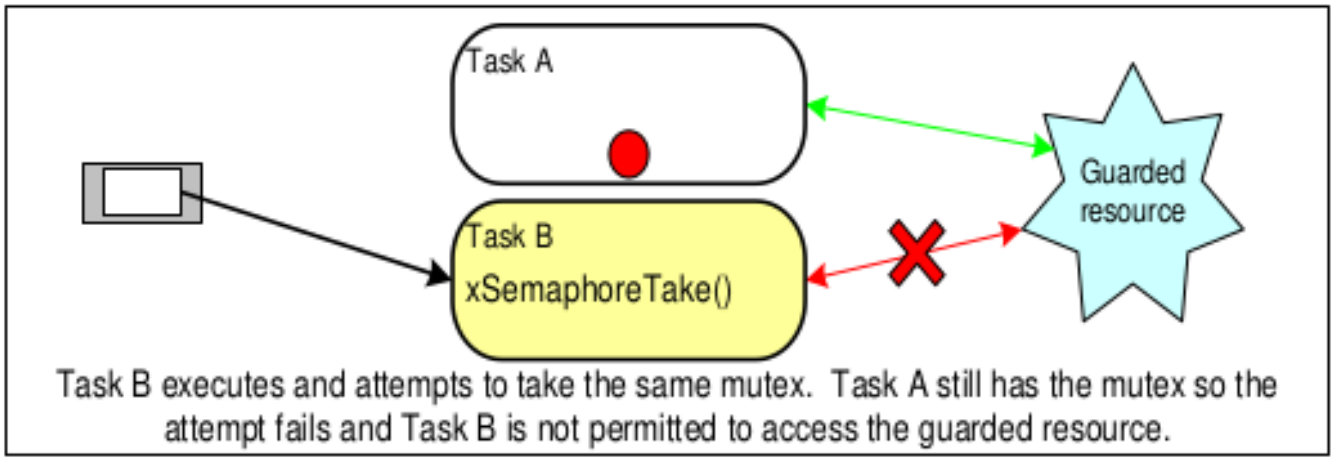
- xSemaphoreCreateBinary()
- xSemaphoreTake(semaphore, blocktime)
- xSemaphoreGive(semaphore)
- xSemaphoreCreateCounting(max count, initial count)
- Can be given by one task and taken by another

**Block time:** Time that the task trying to obtain the semaphore should go into a blocked state while waiting.

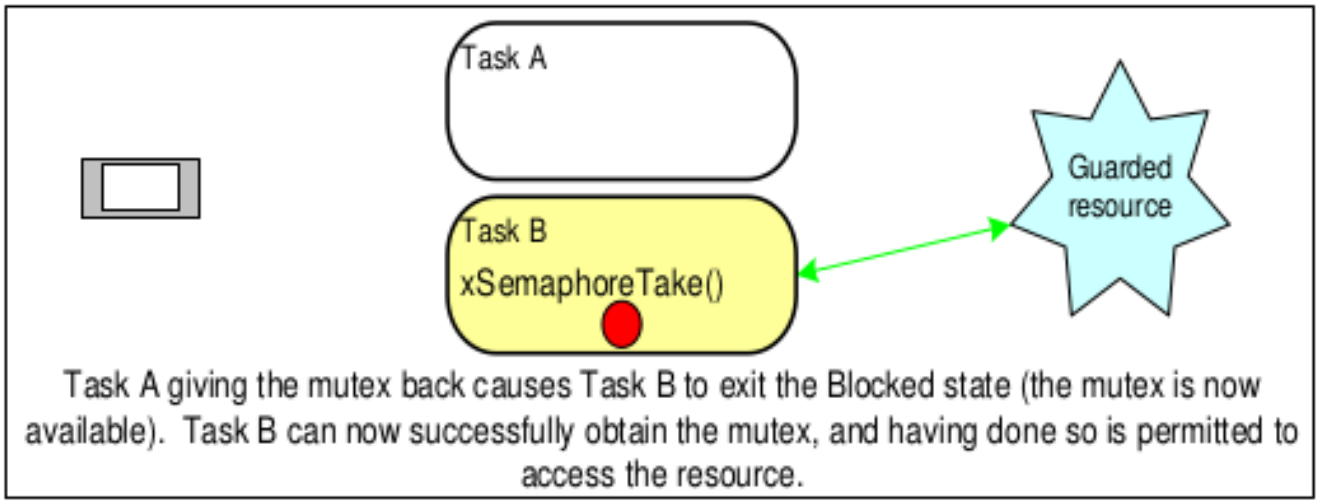
# FreeRTOS Semaphore Example 1



# FreeRTOS Semaphore Example 2



# FreeRTOS Semaphore Example 3



# Task Synchronization 1

Why is synchronization needed?

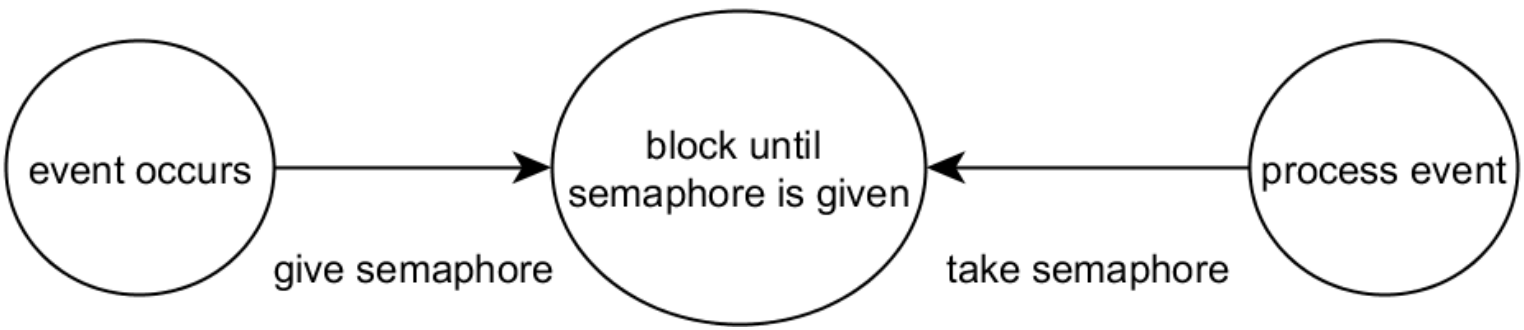
- Pass data from one task to the other
- Manage concurrent access to resources
- Signal another task that a specific even has happened
- Unblock tasks which are blocked
- Timers
- Not all FreeRTOS functionality is covered in the lecture, so have a look on the website

# Task Synchronization 2

Example mechanism in FreeRTOS.

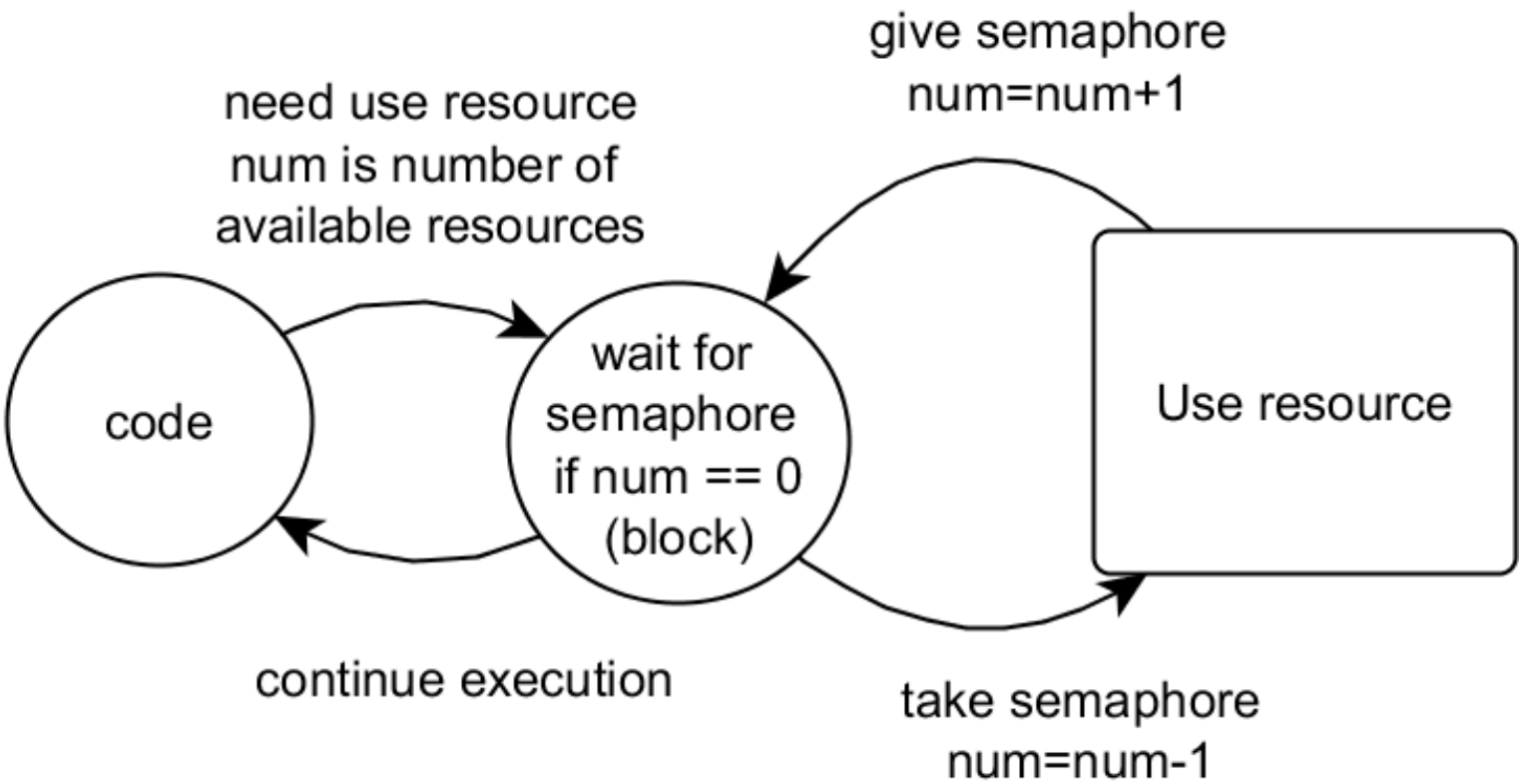
FreeRTOS semaphores can be given by one thread and taken by another.

Synchronization signal:

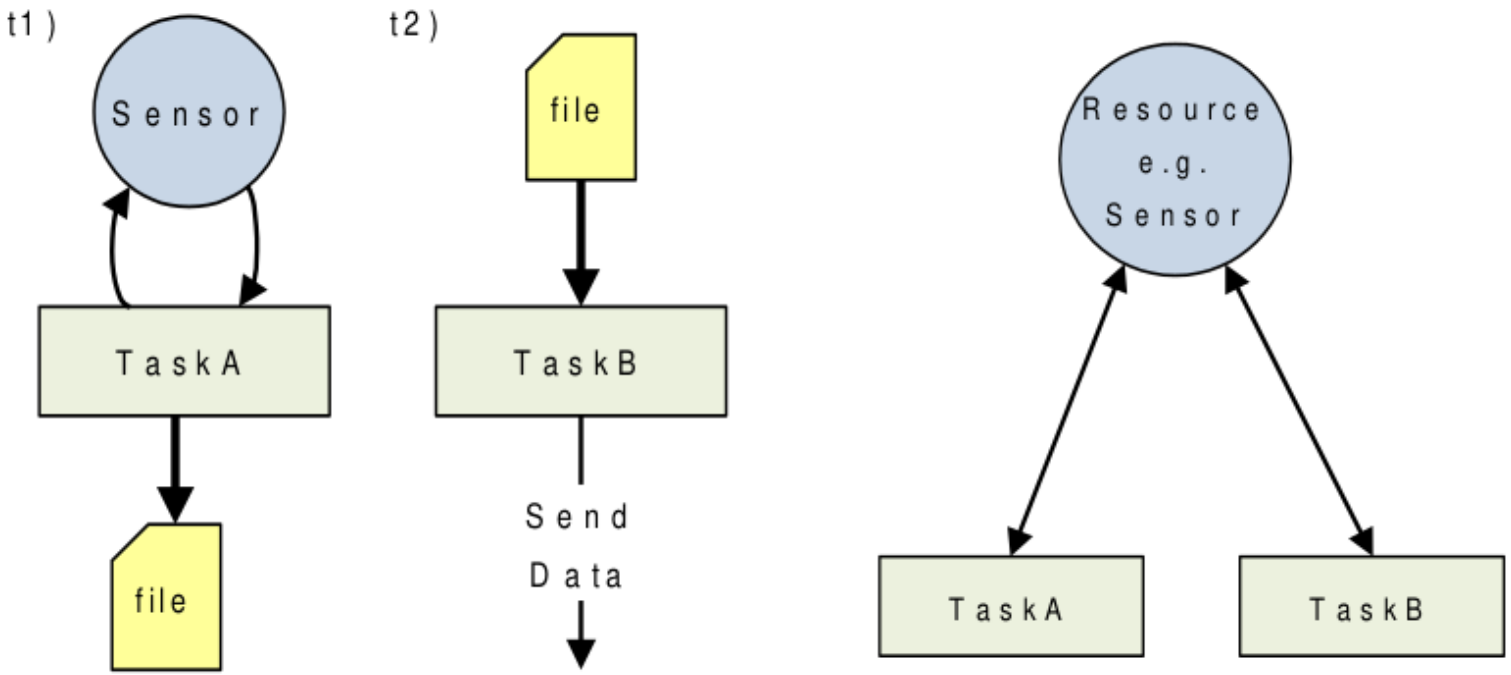


# Event Processing with Counting Semaphore

Eg. Synchronizing number of events that have happened/need to be processed



# Task Synchronization 3

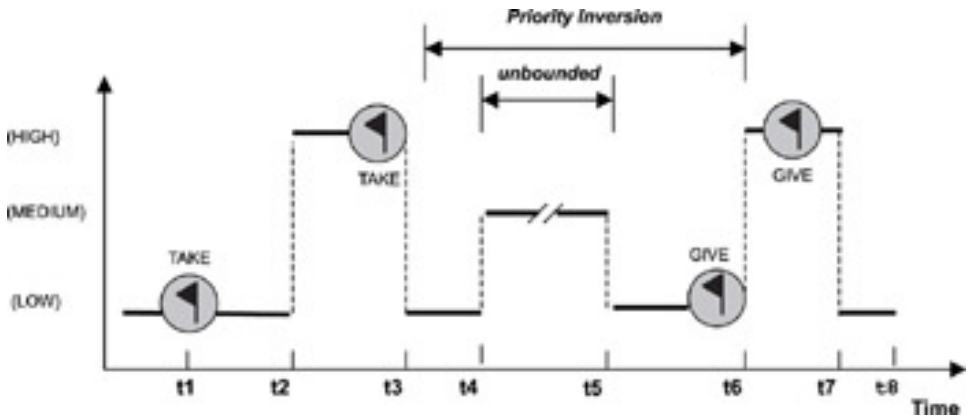


# Priority Inversion

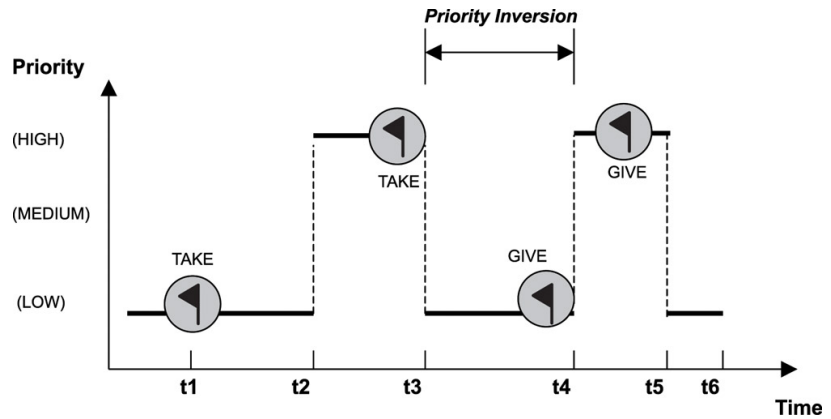
*A problem caused by binary semaphores when used for resource locking*

Task with lower priority runs before a higher priority task as it is stuck in a blocked state waiting on a lower priority task.

Problem (priority inversion)




Solution (priority inheritance)




http://www.embeddedlinux.org.cn

# MUTual EXclusion (MUTEX)

- A mutex is a token that is associated with a shared resource
- Must be taken and given by same task
- Not used for synchronization

**FreeRTOS API:**

- xSemaphoreCreateMutex()
- xSemaphoreTake(mutex, blocktime)
- xSemaphoreGive(mutex)

# Mutex Example

```c
/* TUM_Event.h */
// Functions exposed to retrieve locally stored mouse
    value from other tasks
signed short xGetMouseX(void);
signed short xGetMouseY(void);
```

```c
/* TUM_Event.c */
// Variables local to the TUM_Event.c file, only
    retrievable through xGetMouse functions
typedef struct mouse {
    xSemaphoreHandle lock;
    signed short x;
    signed short y;
} mouse_t;

mouse_t mouse; // Shared resource local to TUM_Event.c
```

# Mutex Example

```
1 /* TUM_Event.c */
  void initMouse(void) {
      mouse.lock = xSemaphoreCreateMutex(); // Init mutex
  }

6 void vInitEvents(void) {
      initMouse();  // Init mouse mutex
      ...
  }
```

# Mutex Example

```
1  /* TUM_Event.c */
   void vEventsTask(void *pvParameters) {
     ...
          else if (event.type == SDL_MOUSEMOTION) {
              xSemaphoreTake(mouse.lock, portMAX_DELAY); //
         Lock taken
6             mouse.x = event.motion.x; //Shared resource
        updated
              mouse.y = event.motion.y;
              xSemaphoreGive(mouse.lock); // Lock given
        back
           }
           ...
11 }
```

# Mutex Example

Variables only accessible through TUM_Event.h API, guaranteeing that locking is used when values are retrieved.

```
/* TUM_Event.c */
signed short xGetMouseY(void) {

    signed short ret; // Copy of data to be returned

    xSemaphoreTake(mouse.lock, portMAX_DELAY); // Take
     lock
    ret = mouse.y; // Copy value of interest
    xSemaphoreGive(mouse.lock); // Return lock

    return ret; // Return copy of value stored in
     shared resource
}
```

# Inter-Process Communication (IPC)

POSIX defines:

- Pipes - a one way (half-duplex) flow of data between processes, a bridging of an output stream to an input stream
  Using the command `ls | more` on Linux, the | operator creates a pipe between the standard output of `ls` is redirected to the pipe, `more` then reads its input from the pipe.
- Message Queues - Based on named pipes (persistent) and sends messages that are comprised of a fixed-size header and a variable-length text (data)
- Semaphores (previously mentioned)
- Shared Memory - Allows two or more processes to access common data structures by using a *shared memory segment*

# FreeRTOS IPC

FreeRTOS defines:

- Stream Buffers (POSIX Pipes) - Allows a stream of bytes to be passed from one task to another. A single writer and a single reader
- Queues (POSIX Message Queues) - Used to send messages between tasks as well as interrupts and tasks. Usually FIFO (First In First Out)
- Semaphores (previously mentioned)
- Shared Memory - In most FreeRTOS projects this will be use of **LOCKED** global variables

# Streams

**Message** based communication applies processing to a singular message. The message is a distinct chunk of information, well defined transmissions size.

**Stream** based communication applies processing to one or more records (ie. messages). Streams do not have boundaries to the transmissions sizes as that is inherently not possible with streams.

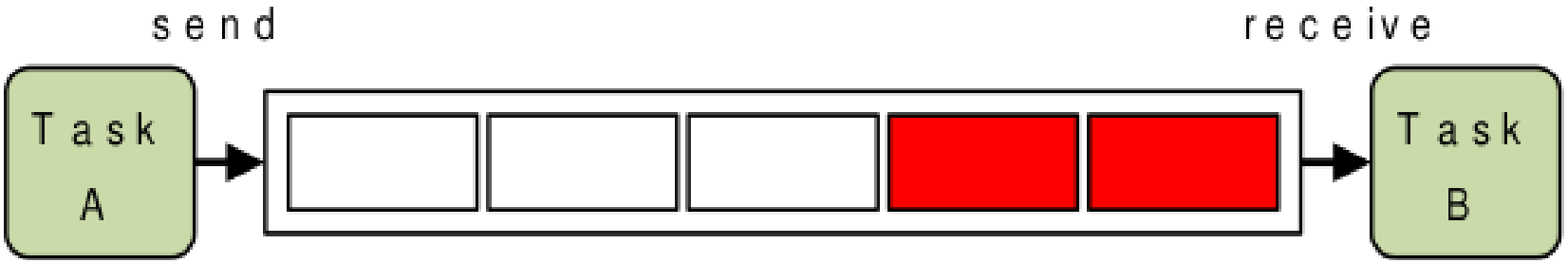
# FreeRTOS Stream Buffer API

- xStreamBufferCreate(buffer size, trigger level)
- xStreamBufferSend(stream handle, data, data length, block time)
- xStreamBufferReceive(stream handle, receive buffer, receive data length, block time)

Block time: how long the task should wait while waiting for the steam buffer to become available or for the trigger level to be reached.

Trigger level: the number of bytes that must be in the stream buffer before a task wishing to read from the buffer is unblocked and allowed to read the buffer.

# Queues



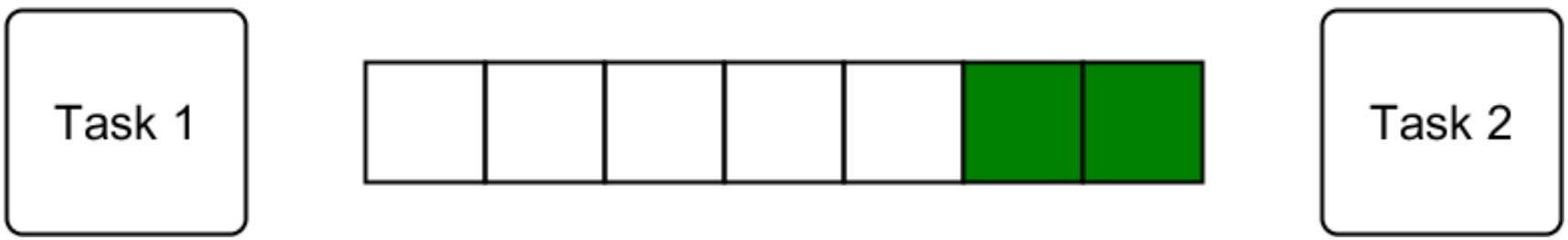

Queues send messages by copy, data is copied into the queue instead of just a reference to the data. While references (pointers) can be sent as the data.

Advantages:

- No need for buffers external to the queue
- Variables sent in the queue can be immediately reused
- Memory allocation is handled by the kernel
- Variable data sizes
- Crosses protected memory boundaries (between tasks)
- Easy to use API

# Task Sync: Queues

- Buffer used to send data from one task to other tasks
- Length and data size of one element can be customized
- Tasks can block on a queue until an element is received
- Data is copied into the queue and from the queue (not the reference - *different address space*)
- Example: Queue with length seven whose two elements are occupied and the rest is empty



# FreeRTOS Queue API

- xQueueCreate(queue length, size of message)
- vQueueDelete(queue handle)
- xQueueSend(queue handle, reference to item to send, block time)
- xQueueSendToBack(queue handle, reference to item to send, block time)
- xQueueSendToFront(queue handle, reference to item to send, block time)
- xQueueReceive(queue handle, buffer to store data, block time)
- xQueuePeek(queue handle, buffer to store data, block time)

Block time: how long the task should wait while trying to send the message if the queue is in use by another task.

# Example Usage from FreeRTOS Simulator 1

Queue allows for a pseudo global variable to be safely used between tasks

```
/* TUM_Event.h */
extern QueueHandle_t inputQueue;  // TUM_Event.h
    exposes a global queue handle

typedef struct buttons { // Data structure sent in the
    queue provided in header file's API
    unsigned char a;
    unsigned char b;
    unsigned char c;
    unsigned char d;
    unsigned char e;
    unsigned char f;
} buttons_t;
```

# Example Usage from FreeRTOS Simulator 2

```c
/* TUM_Event.c */
#include "TUM_Event.h"
void vInitEvents(void) {
  inputQueue = xQueueCreate(1, sizeof(buttons_t)); // Queue is initialised to have a length of 1
  xTaskCreate(vEventsTask, "EventsTask", 100, NULL, tskIDLE_PRIORITY, &eventTask); // Background task to handle button events from graphics library
}

void vEventsTask(void *pvParameters) {
  while (SDL_PollEvent(&event)) {
  ...  // Read and process button events into a message for queue
  }
  xQueueOverwrite(inputQueue, &buttons); // Send buton event message
}

```

# Example Usage from FreeRTOS Simulator 3

```
/* Main.c */
void xGetButtonInput(buttons_t *but) {
 xQueuePeek(inputQueue, but, 0); // Item from queue is
     read in but not removed so that the button status
     is never NULL
}
```

The task does not block when attempting to get the data. Replacing `0` with `portMAX_DELAY` would cause task to block indefinitely waiting for the queue to have a valid value.

# FreeRTOS Task Notifications

- FreeRTOS specific feature that acts as a combination of semaphore and message queue.
- Each task has a 32-bit *notification value*
- From another task the notification value can be:
  - Set without being overwritten (logical OR)
  - Overwritten (logic AND)
  - Set bit wise (logical masked OR)
  - Incremented
- Tasks can block waiting on a notification (synchronization)
- Given task notifications can transmit small amounts of data while unblocking target task via notification value
- 45% faster than binary semaphores

# FreeRTOS Task Notification API

- xTaskNotify(task to notify, notification value, action)
- xTaskNotifyGive(task to notify)
- ulTaskNotifyTake(clear notification value, block time)

Action:

- eNoAction
- eSetBits
- eIncrement
- eSetValueWithOverwrite
- eSetValueWithoutOverwrite

# Task Sync: Summary

| Mechanism | Description | Usage |
|---|---|---|
| Task notification | Notify **one particular** task about an event | Light-weight notification |
| Binary sem. | Counting sem. that can be taken only once | Task synchronization |
| Counting sem. | Can be taken multiple times | Resource management, event counting |
| Queue | Buffer | Data transfer |
| Mutex | Mutual exclusion, priority inheritance | Resource access control |


*Complexity in terms of performance overhead

# Software Timers

- Simple timer functionality
- Run a particular piece of code at a defined point in time (e.g. Watchdog timers)
- Timer can trigger once or periodically
- Callback function to be called when timer expired must be specified
- Callback functions must be non-blocking
- Granularity depends on the kernel tick - Software timers, not hardware timers

# Memory Management

- Implementation of "`malloc()`", "`free()`", etc. are not necessarily safe:
  - Not thread-safe
  - Not always available on embedded systems
  - Deterministic execution time of function is not guaranteed
  - FreeRTOS offers implementations (e.g. `pvPortMalloc()`
- FreeRTOS offers five different heap implementation:
  - heap_1: not possible to free the allocated memory
  - heap_2: frees memory but does no defragmentation
  - heap_3: adds thread safety to standard “malloc” and “free”
  - heap_4: add **defragmentation** for freed memory
  - heap_5: like heap_4, but it is possible to split heap over non-adjacent memory regions

# FreeRTOS Configuration

Customization of OS functionality done in the FreeRTOSConfig.h file

```
#define configUSE_PREEMPTION   1
#define configTICK_RATE_HZ        1000
#define configUSE_16_BIT_TICKS   1
#define configMAX_PRIORITIES     4
...
#define INCLUDE_vTaskDelete    0
```

Corresponding preprocessor code inclusion

```
#if ( INCLUDE_vTaskDelete == 1 )
void vTaskDelete( xTaskHandle pxTaskToDelete )
{...}
#endif
```

Allowing code base scalability

# FreeRTOS General Comments

- Kernel customization can be done in the **FreeRTOSConfig.h** file (e.g. tick rate, features)
- Every item (e.g. semaphore, task, etc.) has to be created before it can be used
- Many functions have a "FromISR" equivalent which should be used in interrupt service routines
- There exists more FreeRTOS features than presented here
- use www.freertos.org for API reference
- Further reading:
  - FreeRTOS reference manual
  - Mastering the FreeRTOS Real-Time Kernel: A hands-on Tutorial Guide

# Sources

www.freertos.org

http://www.freertos.org/Documentation/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

http://www.freertos.org/Documentation/FreeRTOS_Reference_Manual_V9.0.0.pdf