Internal Structure of FreeRTOS

The Task Control Block (TCB) is the main data structure used by FreeRTOS to represent a task and to store any information associated with it. Some TCB data fields are essential and others may be omitted if not needed (to save memory).

The scheduler uses a Ready Task List for each priority level, pointing to the TCBs of ready tasks in this level. These lists are used to decide which task to run in each clock tick.

In addition, there are lists for suspended tasks, delayed tasks, pending-ready tasks, and waiting-for-termination tasks.

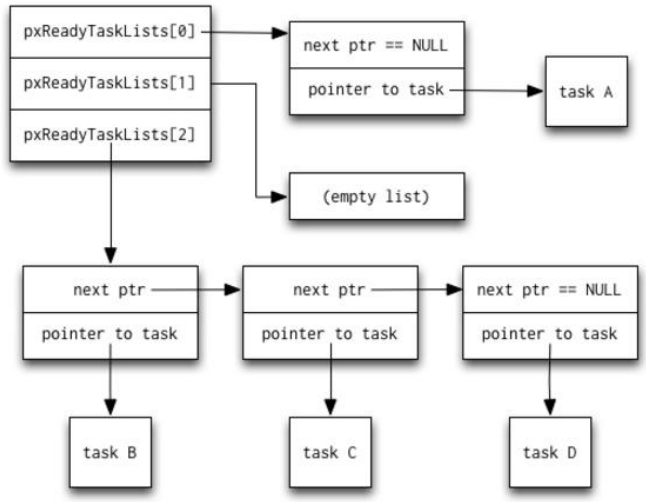
The stack of task holds its context when a context switching occurs.

FreeRTOS Task Control Block (TCB)

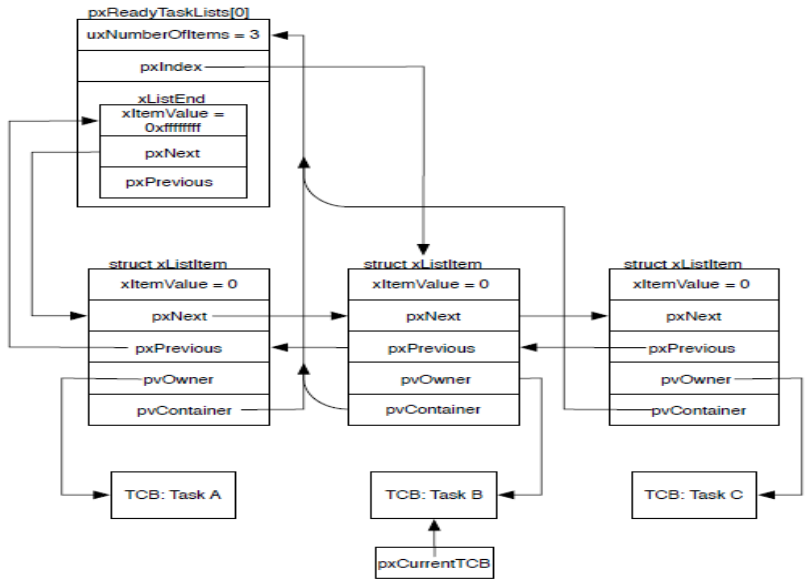
Contents of a FreeRTOS Task Control Block (TCB)

Field	Purpose	Optional
`pcTaskName[]`	Human-readable task name	-
`uxTCBNumber`	Task identification number	*
`uxPriority`	Active priority	-
`uxBasePriority`	Baseline priority	*
`pxStack`	Lowest task stack address	-
`pxEndOfStack`	Highest task stack address	*
`pxTopOfStack`	Current top of the task stack	-
`xGenericListItem`	Link to the scheduler's lists	-
`xEventListItem`	Link to an event wait list	-
`uxCriticalNesting`	Interrupt disable nesting level	*
`ulRunTimeCounter`	CPU time consumed	*
`pxTaskTag`	User-defined, per-task pointer	*
`xMPUSettings`	Memory protection information	*

FreeRTOS Ready Task Lists



FreeRTOS Ready Task List (More detailed)



Optional Components of FreeRTOS Application

The following functions used in the FreeRTOS demo are optional and can be omitted or modified:

vApplicationIdleHook

This hook (or call-back) function is executed in the idle task if `configUSE_IDLE_HOOK` is set to 1.

vApplicationTickHook

This function is executed each clock tick if `configUSE_TICK_HOOK` is set to 1. It should use `fromISR` API functions.

Optional Components of FreeRTOS Application

vApplicationStackOverflowHook

Executed when a stack overflow occurs if a nonzero value is given to `configCHECK_FOR_STACK_OVERFLOW`.

vApplicationMallocFailedHook

Executed in case of error in memory allocation (i.e. `pvPortMalloc()` returns `null`) if `configUSE_MALLOC_FAILED_HOOK` is set to 1.

vAssertCalled

This function is used if the macro `configASSERT(`*`expression`*`)` is defined. This macro is used to abort program and display an error message if *`expression`* is false. (This increases code size and execution time and is thus used only during debugging.)

Optional Components of FreeRTOS Application

prvInitialiseHeap

Needed only if `heap_5.c` is used for memory management.

prvSaveTraceFile

Used to generate debugging trace files if `configUSE_TRACE_FACILITY` and `configGENERATE_RUN_TIME_STATS` are set to 1.

When running FreeRTOS on microcontrollers, a `prvSetupHardware()` function will always be needed to configure the hardware.

File Management in FreeRTOS

Third party FAT system is available for FreeRTOS. This is an open source, small footprint, thread aware DOS/Windows compatible file system that supports FAT12, FAT16, and FAT32 systems.

API functions are provided for:

- Disk management: partition, format, mount.
- File operations: open, close, rename, delete, read, write, …etc.
- Directory/folder functions: make directory, change directory, remove directory, ….etc.

A RAM-disk demo is provided for the simulator (FreeRTOS-Plus for version 9 and in FreeRTOS Labs for version 10).

File Management in FreeRTOS

Directory/Folder Functions

- ff_mkdir()
- ff_chdir()
- ff_rmdir()
- ff_getcwd()

File Utility Functions

- ff_errno()
- ff_feof()
- ff_rename()
- ff_remove()
- ff_stat()
- ff_filelength()
- ff_findfirst()
- ff_findnext()

File Read and Write Functions

- ff_fopen()
- ff_fclose()
- ff_fwrite()
- ff_fread()
- ff_fputc()
- ff_fgetc()
- ff_fgets()
- ff_fprintf()
- ff_fseek()
- ff_ftell()
- ff_seteof()
- ff_rewind()
- ff_truncate()

Disk Management Functions

- FF_Partition()
- FF_Format()
- FF_Mount()

Embedded Linux

Linux started in the 1990s as a free, open source version of UNIX for Intel x86 PCs.

It was successfully ported to a wide range of 32-bit and 64-bit architectures.

Linux is highly modular and configurable, which makes it possible to remove all unnecessary functionality to allow Linux to run from minimum hardware.

Currently, a large portion of embedded market is based on Linux. Android, which is based on the Linux kernel, can be considered a form of embedded Linux adapted to touch-based mobile devices.

Embedded Linux

The complete operating system can be obtained in the form of a Linux *distribution*. A distribution contains all the components needed in a Linux installation. These include the Linux kernel together with (usually open source) utility programs, graphical environments, development tools, etc.

Typically, distributions make use of a package manager that simplifies the initial installation and subsequent upgrading of system and to manage installation and removal of other packages on the system.

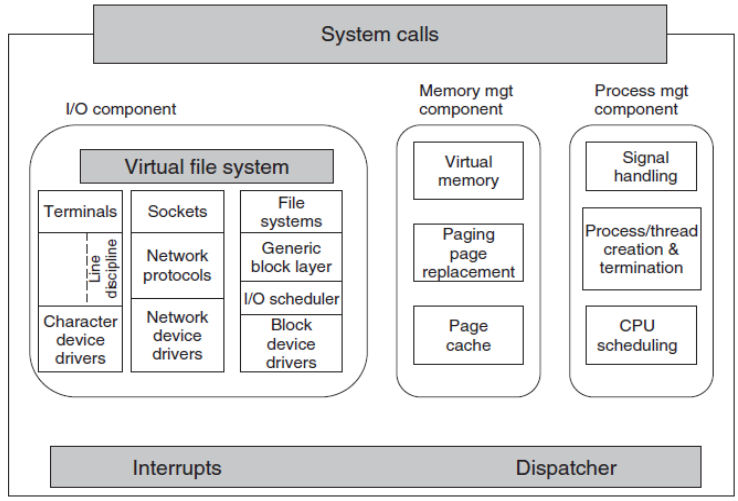
Embedded Linux

Embedded devices typically require support for a specific set of devices, peripherals, and protocols. An embedded Linux kernel will be far smaller than an ordinary Linux kernel.

Distribution can be customized to specific hardware, applications, and required interface.

Embedded Linux distributions are usually compiled on one platform but are intended to be executed on another.

Linux Kernel



Source: [Tanenbaum 15]

Linux Processes

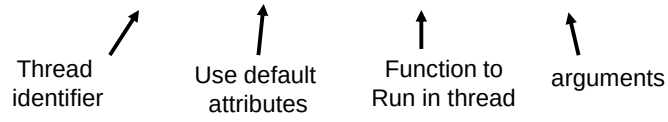
Linux processes can be created using the fork()system call of traditional UNIX. It generates an exact duplicate of the calling process. It then returns 0 to the child process, and the child's process id to the parent process. If some error occurs, fork returns -1.

Typically, the child runs different executable file by using the exec() function which replaces the process image by a new image.

Linux POSIX Threads

Process can have multiple threads using the POSIX compliant pthread library. It provides a set of functions for thread management.

```
#include <pthread.h>
…
 Pthread_t tid;
…
 Pthread_create (&tid, NULL, function, NULL);
```



Linux POSIX Threads

A thread can terminate for different causes:

- when it executes the last instruction of the associated function (normal termination).
- when it calls the pthread_exit() function.
- when another thread executes pthread_cancel()on it.
- when its parent process terminates with a normal termination or a call to the exit() function.

It is possible to wait for the termination of a thread through the function pthread_join().

Linux `clone()` function

The system call clone() can also be used to create threads. It is passed a set of flags that determine how much sharing is to take place between the parent and child tasks.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

Source: [Tanenbaum 15]

The clone() function is not portable to other systems. However, fork() and pthread_create() are actually implemented in Linux using clone().