Memory Management in FreeRTOS

FreeRTOS needs to assign memory for each created task, queue, or semaphore. Dynamic data structures may also be created within application functions.

C functions `malloc()` and `free()` can be used to manipulate the heap, but they typically have slow unpredictable time, and are not safe to use with preemptions.

Memory management functions are part of the port-dependent FreeRTOS code.

Memory Management in FreeRTOS

FreeRTOS provides five sample memory allocation impleme-ntations. Designer may select one of these according to the application requirements, or else write his own memory management code.

Prototype of functions used in all versions are `void *pvPortMalloc(size)`, which allocates memory of specified size and returns a pointer to the allocated memory, and `void vProtFree(void *pv)` which frees a previously allocated memory area.

The versions are provided in the files heap_1.c, heap_2.c, heap_3.c, heap_4.c, and heap_5.c.

Memory Management in FreeRTOS

- heap_1.c

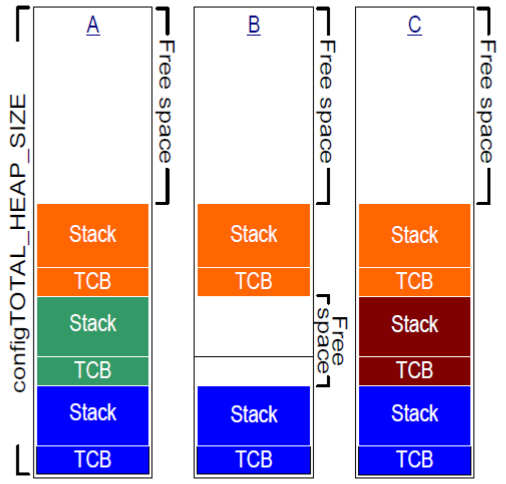
This version does not allow freeing memory, and thus assumes that all memory is allocated statically before starting the scheduler, and tasks, queues and semaphores will not be deleted.

This provides a simple guarantee of deterministic (predictable) operation, however may not be suitable for systems with limited memory where dynamic allocation is needed.

- heap_2.c

This version allows freeing of allocated blocks and uses best-fit assignment. However, it does not merge free blocks together.

Memory Management in FreeRTOS



Memory Management in FreeRTOS

The Task Control Block (TCB) stores information associated with a task. Some TCB data fields are essential and others may be omitted if not needed (to save memory).

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	*

Memory Management in FreeRTOS

This version is suitable if new objects (task, queue, ..,etc.) have exactly the same size as the deleted ones. It is not deterministic but more efficient than standard `malloc()` and `free()` .

- heap_3.c

This version uses the standard `malloc()` and `free()` functions, but suspends scheduler during their execution (using `vTaskSuspendAll()` and `xTaskResumeAll()`) to make them thread-safe.

- heap_4.c

This version uses first-fit and merges adjacent free blocks. It provides less fragmentation and better use of available memory.

Memory Management in FreeRTOS

- heap_5.c

This version is similar to heap_4.c but allows the heap to span multiple non adjacent (non-contiguous) memory regions. It is the version used in the Windows simulator.

The function `xPortGetFreeHeapSize()` can be used to get the amount of unused area in heap, e.g. to optimize the choice of the heap size .

Memory Management in FreeRTOS

FreeRTOS (versions ≥ 9) allows static allocation of memory by the application writer instead of automatic allocation by the system.

If configSUPPORT_STATIC_ALLOCATION is set to 1, the API functions [object]CreateStatic() can be used instead of [object]Create() functions.

These function has additional arguments to specify the location assigned to the object in memory.

If configSUPPORT_DYNAMIC_ALLOCATION is set to 0 then RTOS objects can only be created using RAM provided by the application writer.

File System Management

OS manages the data stored on secondary storage devices, which store large amounts of data in a nonvolatile manner.

OS simplifies access to data on these devices by providing the abstract view of files, directories or folders, paths, …etc.

The overall system performance is affected by:

- ❑ Allocation of storage space to files.
- ❑ Organization of data on storage space.
- ❑ How the OS keeps track of storage space usage.

File System Management

Currently, most embedded operating systems provide file management functions. Differences from general purpose systems result from:

- o Code size and power consumption limitations.
- o Solid-state storage devices are typically used

Flash memory based devices provide relatively fast and inexpensive storage, but have special requirements, e.g. for block erasures.

In addition, erase/write cycles results in quicker wear compared to other storage devices. Thus, wear levelling techniques which distribute access over the storage space and avoid using the same blocks excessively are needed.

Allocation of Storage Space to Files

OS assigns each file an integer number of *assignment units* (may be called blocks, clusters, …). The assignment unit is one or more contiguous sectors (smallest storage units).

When selecting the unit size we try to:

○ reduce lost disk space as a result of partially used units.

○ reduce the size of the data structure needed to keep track of disk space usage.

i.e. a compromise must be made.

Also, as file size increases, it may be fragmented, i.e. assigned non-contiguous units, which slows down the file access.

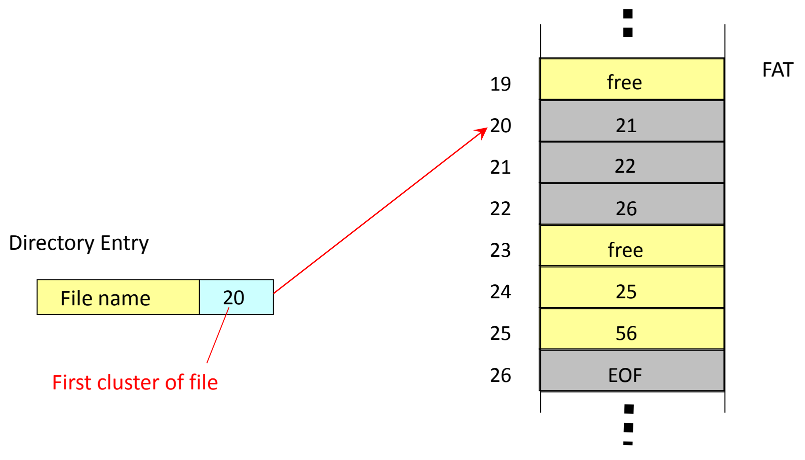
File Allocation Table (FAT)

FAT system was originally developed for small magnetic disks. It is not suitable for currently available large-capacity hard disks. However, it is widely used for solid-state storage devices in embedded applications.

System keeps track of space allocated to files using a table with an entry for each cluster on device. Size of this entry is 8, 12, 16, or 32 bits according to version. Entry contains one of the following:

- Code for free cluster.
- Number of next cluster in file for which this cluster is assigned.
- End of File code.
- Code for bad cluster.

File Allocation Table (FAT)



Each file has a chain of clusters in FAT ending by an EOF.

File Allocation Table (FAT)

Example

A disk using FAT system has a cluster size of 2K bytes. Directory of this disk indicates that the number of the first cluster in the file FILE1 is 10 and the first cluster in file FILE2 is 14. A part of the FAT of this disk is shown below.

cluster no	…. 10	11	12	13	14	15	16	17	18	19	20
FAT entry	…. 11	12	16	Free	15	19	17	18	EOF	17	EOF…

a) How much disk space is assigned to file FILE1? How much of this space is wasted if this file contains 11000 bytes of data?

b) What is the error in the above FAT? What is its possible cause?

a) From the table, FILE1 is assigned clusters 10, 11, 12, 16, 17, 18. Thus, it is assigned 6 clusters, or 12K on disk. This occurred since five clusters will not be enough for the 11000 bytes of the file.

The wasted disk space (i.e. assigned to FILE1 but not holding data, and cannot be used by any other file is

Wasted space $= 12 \times 1024 - 11000 = 1288\ bytes$

b) Both the entries of cluster 16 and 19 point to 17. Thus table indicates that cluster 17 is part of both FILE1 and FILE2 (this serious error is called cross-linked files).

Such errors occur when system make changes in FAT while in memory but fails to update FAT on disk, e.g. after a power failure or disk not "safely" removed.

File Allocation in UNIX

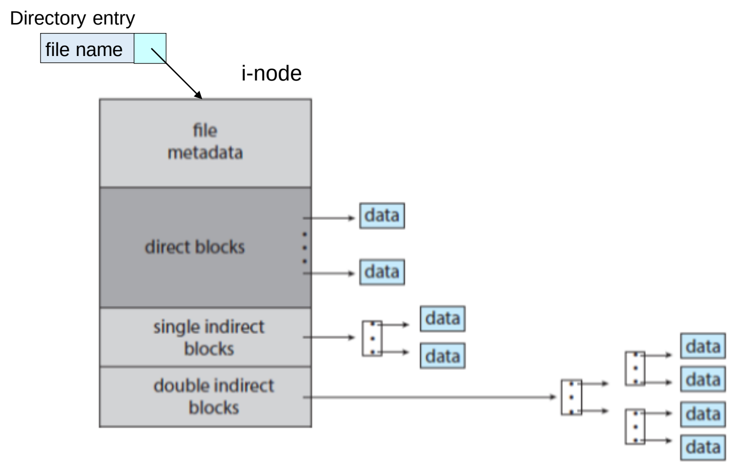
Information about each file in the system is stored in a structure called an *i-node*.

I-node of a file stores its attributes and control information, as well as addresses of the first blocks assigned to the file.

Directory entry of a file points to the place of its i-node on the disk. This is brought to memory when the file is opened.

All i-nodes have a fixed size. To account for larger files, indirect addressing (single, double, …) is used.

File Allocation in UNIX



The maximum file size is determined by the above arrangement.

File Allocation in UNIX

In addition, system maintains a *list of free blocks* on the disk. The size of this list shrinks as more data is stored on disk.

Errors may result in inconsistencies in the i-nodes and free list information.

Compared to FAT, the method used by UNIX is best suited to large disks with small files (Why?)

Directories and subdirectories in UNIX (and most of the other systems) are treated as special files.

File Allocation in UNIX

Example

A UNIX system has a hard disk with a block size of 2 Kbytes. This system keeps track of disk space assignment using i-nodes containing 10 direct pointers as well as pointers for single and double indirect addressing. Assume that block addresses are 4-bytes long.

a) Find the maximum file size in the above disk.
b) If a file of 8221 Kbytes is stored in the above disk, find the corresponding wasted disk space.
c) What are the number of disk accesses required to open this file? Make any necessary assumptions.