Memory Management

(slides are based on Prof. Dr. Jian-Jia Chen and http://www.freertos.org)

Anas Toma

LS 12, TU Dortmund

January 24, 2019

Memory Management

- Subdividing memory to accommodate multiple tasks
- Memory needs to be allocated to ensure a reasonable supply of ready tasks to consume available processor time
- Memory allocation:
  - Programmers do not know where the program will be placed in memory when it is executed
  - While the program is executing, it may be swapped to external memory (disks etc.) and returned to main memory at a different location (for general OS)
  - Memory references must be translated in the code to actual physical memory address
  - This is implemented by calling malloc() in Linux and Unix
- Memory deallocation:
  - The unused memory space is given back to OS so that other tasks can use this part of memory again
  - This is implemented by calling free() in Linux and Unix

Memory Management in Embedded Systems

- The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. The malloc() and free() functions can sometimes be used for this purpose, but ...
  1. they are not always available on embedded systems,
  2. take up valuable code space,
  3. are not thread safe, and
  4. are not deterministic (the amount of time taken to execute the function will differ from call to call)
- An alternative scheme is required. One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.
  - The memory allocation API is included in the RTOS portable layer
  - When the real time kernel in FreeRTOS requires RAM, instead of calling malloc() it makes a call to pvPortMalloc(). When RAM is being freed, instead of calling free() the real time kernel makes a call to vPortFree().

Default Schemes in FreeRTOS

- heap_1.c: This is the simplest scheme of all. It does not permit memory to be freed once it has been allocated, but despite this is suitable for a large number of applications.
  - Can be used if the application never deletes a task or queue (no calls to vTaskDelete() or vQueueDelete() are ever made).
  - Is always deterministic (always takes the same amount of time to return a block).
- heap_2.c: This scheme uses a best fit algorithm and, unlike scheme 1, allows previously allocated blocks to be freed. It does not combine adjacent free blocks into a single large block.
  - It can be used even when the application repeatedly calls vTaskCreate()/vTaskDelete() or vQueueCreate()/vQueueDelete()
  - It could possible result in memory fragmentation problems should your application create blocks of queues and tasks in an unpredictable order.
  - It is not deterministic - but is also not particularly inefficient.
- heap_3.c: This is just a wrapper for the standard malloc() and free() functions. It makes them thread safe.

Case 1: heap_1.c

```c
void *pvPortMalloc( size_t xWantedSize ){
void *pvReturn = NULL;
    /* Ensure that blocks are always aligned to the required number of bytes. */
    #if portBYTE_ALIGNMENT != 1
        if( xWantedSize & portBYTE_ALIGNMENT_MASK )
        {
            /* Byte alignment required. */
            xWantedSize += ( portBYTE_ALIGNMENT - ( xWantedSize & portBYTE_ALIGNMENT_MASK ) );
        }
    #endif
    vTaskSuspendAll();
    {
        /* Check there is enough room left for the allocation. */
        if( ( ( xNextFreeByte + xWantedSize ) < configTOTAL_HEAP_SIZE ) &&
            ( ( xNextFreeByte + xWantedSize ) > xNextFreeByte )  )/* Check for overflow. */
        {
            /* Return the next free byte then increment the index past this
            block. */
            pvReturn = &( xHeap.ucHeap[ xNextFreeByte ] );
            xNextFreeByte += xWantedSize;
        }
    }
    xTaskResumeAll();
    #if( configUSE_MALLOC_FAILED_HOOK == 1 )
    {
        if( pvReturn == NULL )
        {
            extern void vApplicationMallocFailedHook( void );
            vApplicationMallocFailedHook();
        }
    }
    #endif
    return pvReturn;
}
```

Placement Algorithms

Different strategies may be taken as to how space is allocated to processes: (reference "Operating System Concepts" by Abraham Silberschatz, Peter B. Galvin (Author), and Greg Gagne.)

- First Fit: Allocate the first hole that is big enough. Searching may start either at the beginning of the set of holes or where the previous first-fit search ended.
- Best Fit: Allocate the smallest hole that is big enough. The entire list of holes must be searched unless it is sorted by size. This strategy produces the smallest leftover hole.
- Worst Fit: Allocate the largest hole. In contrast, this strategy aims to produce the largest leftover hole, which may be big enough to hold another process.
- Experiments have shown that both first fit and best fit are better than worst fit in terms of decreasing time and storage utilization. First fit is generally faster.

Case 2: heap_2.c

- It is more involving by using a best fit algorithm.
- We will look into the source code directly.

Memory Protection

- Using a Memory Protection Unit (MPU) can protect applications from a number of potential errors, ranging from undetected programming errors to errors introduced by system or hardware failures.
- In FreeRTOS: FreeRTOS-MPU
  - It can be used to protect the kernel itself from invalid execution by tasks and protect data from corruption.
  - It can also protect system peripherals from unintended modification by tasks and guarantee the detection of task stack overflows.

Creating Restricted Tasks in FreeRTOS

- The created task can run in either Privileged or User modes.
    - When Privileged mode it used the task will have access to the entire memory map
    - When User mode is used the task will have access to only its stack.
    - In both cases the MPU will not automatically catch stack overflows.
- If a task wants to use the MPU then the following additional information has to be provided:
    - The address of the task stack.
    - The start, size and access parameters for up to three user definable memory regions.
    - The memory regions allocated to a task can be changed using vTaskAllocateMPURegions().
- It is implemented in xTaskCreateRestricted() in task.h
- A Privileged mode task can call portSWITCH_TO_USER_MODE() to set itself into User mode. A task that is running in User mode cannot set itself into Privileged mode.