SYSC 5701
Operating System Methods for Real-Time Applications

Memory Issues

Winter 2014

Memory Management is a Gnarly Issue

- Pearce suggests: offload the problem to the application … how do some “real” RTOS’s cope?
- µC/OS II: has (optional) API to manage partitions
  - Application supplies memory for partitions
  - Each partition contains blocks of same size
  - Partitions are managed by kernel (safe)
- FreeRTOS: has required API
  - add-on above kernel
  - various implementations

µC/OS II

- Partition – a contiguous memory region
  - Kernel cuts up into blocks of equal size
  - Application can Get/Put blocks
- Can have multiple partitions of different sized blocks
- Configuration (OS_CFG.H)
  - Enable management services: OS_MEM_EN = 1
  - Max. number of partitions:
    OS_MAX_MEM_PART = MaxNumberOfPartitions
    - Statically allocated array of memory control blocks (not partitions!)

Application specifies these at config! (compile time)

µC/OS II

- Create a partition at runtime:

Application supplies these!

```
OS_MEM *OSMemCreate(
    void *addr,        // start address of memory block
    INT32U nblks,      // number of blocks to create
    INT32U blksize,    // size of each block
    INT8U *perr        // return code (for create status)
);
```

- Returns: OS_MEM* = ptr to memory control block

µC/OS II

- Get a block from a partition at runtime:

```
void *OSMemGet(
    OS_MEM *pmem,      // ptr to memory control block
    INT8U *perr        // return code (success?)
        // size of block is implicit to partition!!
);
```

- Returns: void* = pointer to block from specified partition

µC/OS II

- Put (return) a block to a partition:

```
INT8U OSMemPut(
    OS_MEM *pmem,      // ptr to memory control block
    void *pblk         // ptr to block to return
);
```

- Returns: INT8U = return code

FreeRTOS

- Memory allocation API is in the portable layer
- Portable layer: outside of source files that implement the core RTOS functionality
- Allows an application-specific implementation appropriate to real time system being developed
- Provides some implementations … but application can supply its own implementation

FreeRTOS

- FreeRTOSConfig.h customizes the kernel to the application being built
- Every FreeRTOS application must have a FreeRTOSConfig.h header file in the application directory (not RTOS directory!)
- **configTOTAL_HEAP_SIZE = xxxx**
  - Total amount of RAM available to RTOS kernel – only used if application uses particular provided sample memory allocation schemes

Application specifies this at config! (compile time)

FreeRTOS

- When kernel requires RAM, calls:

  void *pvPortMalloc(size_t xWantedSize)
  - Returns ptr to allocated block of requested size
  - Returns NULL if no memory allocated
- When kernel releases RAM, calls:

  void pvPortFree(void *pv)
- Uses whatever implementation has been linked to the kernel code

FreeRTOS

- Provided implementation: Heap_1.c
- Does not permit memory to be freed once it has been allocated (i.e. no pvPortFree calls)
  - Deterministic
  - OK when all kernel-managed objects are created initially at startup and exist for entire running of application
  - Pros: Simple, no runtime overhead after startup
  - Cons: no dynamic create/delete

FreeRTOS

- Provided implementation: Heap_2.c
- Best fit algorithm, allows blocks to be freed, does not coalesce adjacent free blocks to create larger blocks
  - NOT Deterministic
  - OK when kernel-managed objects are created (deleted) dynamically, but only a small set of sizes of blocks involved – e.g. fixed sized control blocks & messages
  - Pros: dynamic create/delete
  - Cons: runtime overhead, random-sized blocks will likely increase fragmentation

FreeRTOS

- Provided implementation: Heap_3.c
- Simple thread-safe wrapper on C's malloc( ) & free( )
  - NOT Deterministic
  - OK when kernel-managed objects are created (deleted) dynamically, and random-sized blocks
  - Must now include C library for implementation of malloc() and free() [code size & efficiency?!]
  - configTOTAL_HEAP_SIZE not used
  - Pros: dynamic create/delete
  - Cons: runtime overhead

FreeRTOS

- Provided implementation: Heap_4.c
- First fit algorithm, coalescence algorithm
  - NOT Deterministic
  - OK when kernel-managed objects are created (deleted) dynamically, and random-sized blocks
  - Probably more efficient than C library (smaller code?)
  - Pros: dynamic create/delete
  - Cons: runtime overhead

FreeRTOS

- Application provided implementation: Heap_x.c
- Implemented however the application might like to manage memory … must implement:

  void *pvPortMalloc(size_t xWantedSize)

  void pvPortFree(void *pv)
  - Used by kernel when needed
  - Pros: use system-specific memory regions in customized ways (?)
  - Cons: more code/details for application programmer