COMP595EA

Embedded Applications
Operating System Services

Inter-Task Communication

- Tasks often need to communicate
  - Could be done through global memory
  - Difficulties arise due to:
    - handshaking
    - communication of streams
- Similar to communication tasks encountered in programming processes or threads.

Communication Types

- Three types of communications often offered
  - Message Queues
  - Mailboxes
  - Pipes
- RTOS guarantees that the functions provided for using these mechanisms are reentrant.

Queues

- Support two operations
  - Enqueue: used by a task to send data to another task
  - Dequeue: Used by the other task to receive data.
- If the queue is empty the dequeuing task blocks until a message is ready. (typically)
- Usually messages are of some fixed length.
  - More complicated operating systems may allow arbitrary length messages.

Queue Details

- Queue details:
  - Most RTOS require that the queue be initialized before use.
  - It may be your responsibility to preallocate the memory that the queue will use.
  - Identification of which queue you want to use varies between RTOSes
  - Writing to a full queue may cause the task to block or it may lose data or it may crash.

More Queue Details

- More details
  - Many RTOS provide a function that will not block a read from an empty queue but will return an error instead.
  - Queue size may be inflexible. Many RTOS only allow you to write exactly the number of bytes taken up by a void pointer.
    - To provide arbitrary size messages a pointer to the data can be enqueued
    - small amounts can be sent by casting to void pointer type.

Mailboxes

- Mailboxes are similar to queues.
  - Capacity of mailboxes is usually fixed after initialization
  - Some RTOS allow only a single message in the mailbox. (mailbox is either full or empty)
  - Some RTOS allow prioritization of messages

Pipes

- Pipes provide another communication mechanism like queues.
  - Typically byte-oriented (streams)
  - Accepts arbitrary length messages.
    - (but read() has no knowledge of where a write() started or left off)

Pitfalls

- RTOS do not restrict which task can read or write to a particular queue, mailbox or pipe.
  - Coordination has to be designed in by the programmer.
- RTOS do not guarantee that data is interpreted correctly. (interpreting a pointer as a float may cause a problem if it was the result of an int cast.)
- Running out of space is usually a disaster in RTOS.

Pointer Passing

- Another common pitfall is created when passing pointers between tasks.
  - One method for avoiding the problem is to treat pointers as object that can exist in one task at a time.
  - writing the pointer to a queue essentially gives up the object and the writing task should not use the object anymore.

Timers

- RTOS and embedded systems provide timing mechanisms since many things in the real world are temporally dependent.
- One simple service is a timer which delays execution.
  - task blocks until period of time expires.
  - example: dial tones.

Timer Questions

- How do I know the unit of time that the timer functions in?
  - Ans: You don't. typically the timers operating on the system clock. One clock cycle produces one tick of time.
- How accurate are delays?
  - Typically the delay is 0-2 ticks longer than the period requested.

More Questions

- How does the RTOS know how to setup the hardware time on my platform?
  - Built in by the developers of the RTOS. If the RTOS doesn't support your platform you're pretty much out of luck.
- What is the “normal” length of a system tick?
  - There really isn't one. most microprocessors will operate from 0 cycles per second (DC) to whatever the maximum frequency rating is.

Last Question

- What if I need really accurate timing?
  - Make the system tick as short as possible.
  - Use a separate hardware timer chip.

More Timers

- RTOS will provide a variety of timers:
  - limit how long a task will block waiting to read from a queue.
  - limit how long a task will block waiting for a semaphore
  - schedule a function to execute after some period of time.

Events

- Many RTOS provide services for managing "events"
- different than X-window events
- events are typically implemented as a boolean flag of sorts
  - An event is cleared by resetting the flag.
  - tasks can choose to block and wait on the flag, until
  - another task can signal the event.

Event Details

- A major difference from traditional operating system events is
  - All tasks currently blocked on the event are unblocked when the event occurs (as opposed to the event being "delivered" to a particular handler.
  - More than one task can block waiting for the event
  - tasks can wait on groups of events. any event in the group will cause the task to be unblocked.
  - method of clearing the event varies with RTOSes

Comparison

- Semaphores are usually fastest and simple
- Events are a little more complicated and take up more time.
- Queues allow large quantities of information to be communicated but take up more time and resources.

Memory Management

- Most RTOS provide some memory management techniques.
  - Even if it is just malloc() and free()
- malloc() and free() are typically slow and expensive.
- allocation of fixed buffers is usually done in RTOS to save computational cycles.

MultiTask! example

- memory is set in “pools”
- each pool consists of a number of memory buffers identical in size.
- different pools can consist of different sized buffers
- allocate is done through getbuf() or reqbuf()
  - getbuf() blocks when no buffers are available
- each method accept a parameter controlling which pool to allocate from.

Interrupt Routine Rules

- Interrupt routines in RTOS must follow two rules that do not apply to task code:
  - An interrupt routine must not call any RTOS functions that might block.
    - could block the highest priority task
    - might not reset the hardware or allow further interrupts
  - An interrupt routine must not call any RTOS function that might cause the RTOS to switch tasks
    - causing a higher priority task to run may cause the interrupt routine to take a very long time to complete.

Rule 2 solutions

- RTOS may intercept calls to interrupt routines and will not switch tasks until the interrupt completes. (So a write to a queue during an interrupt routine won't cause a task blocked on reading to execute until interrupt routine finishes.)
  - Requires programmer intervention to inform RTOS where the interrupt routines are and which hardware interrupt corresponds with which routine.

More Solutions

- RTOS may provide an instruction that disables the scheduler during an interrupt.
  - requires the programmer to make a call to this RTOS function in any interrupt routines that are about to call an RTOS function that may unblock a higher priority task.
- Some RTOS provide alternate functions for routines that would unblock a task and a function that interrupt routine calls prior to exit to cause the schedule to reschedule tasks.

Nested Interrupts

- If a higher priority interrupt can interrupt a lower priority interrupt routine then a mechanism must exist to tell the RTOS not to reschedule after the higher priority interrupt finishes
  - Basically: the RTOS cannot be allow to resechedule until all interrupts complete.