EE458 - Embedded Systems
Lecture 12 – Semaphores

• Outline
  – Introduction to Semaphores
  – Binary and Counting Semaphores
  – Mutexes
  – Typical Applications
  – RTEMS Semaphores

• References
  – RTC: Chapter 6
  – CUG: Chapter 9
Semaphores
Introduction

- A *semaphore* is a kernel object that one or more tasks can acquire or release for the purpose of synchronization or mutual exclusion.

- *Mutual exclusion* is a provision by which only one task at a time can access a shared resource (port, memory block, etc.)
Semaphores
Introduction

- Think of a semaphore as a key. Your task can make a request for the key. If it is available, your task can check out the key and proceed. If it is not available (another task has it checked out) your task will block until the key becomes available (the other task checks it back in).
Semaphores
Introduction

• *Acquiring the semaphore* is analogous to checking out the key. *Releasing the semaphore* is analogous to checking the key back in. Multiple semaphores can be used if desired. (Keys to different doors.)

• There may be multiple tasks that are waiting (blocked) on a semaphore. When released, either the oldest task on the queue or the highest priority task is given the semaphore. (RTEMS supports both methods.)
Semaphores

Binary Semaphores

• An integer variable is used to implement a semaphore. We have been discussing binary semaphores in which a value of 0 means that the semaphore is unavailable. A value 1 means the semaphore is available.

• Semaphores are global resources. Any task can release the semaphore even if it was acquired by another task. This is useful for task synchronization. (This is analogous to a friend returning the key for you.)
Semaphores

Counting Semaphores

- A *counting semaphore* uses a count that allows it to be released multiple times (up to some maximum value). (This is analogous to having multiple copies of a key to a single door available for check out.)

- When it is acquired the count value is decremented. When the count reaches 0 any task trying to acquire the semaphore will block. As the semaphore is released the count is incremented.
Semaphores

Additional Semaphore Info

- Semaphores can be created in the available (value > 0) or unavailable (value = 0) states. Proper initialization is very important!!!
- You must ensure that every semaphore acquisition is paired with a release.
- Most OSes optionally allow a call to acquire a semaphore to time-out after a number of ticks if the semaphore is not available. (RTEMS does.)
Semaphores
Priority Inversion

- *Priority inversion* can occur when HI and LO priority tasks share a semaphore.

- Assume HI is blocked and LO acquires the semaphore. HI unblocks and tries to acquire the semaphore. Since LO has it, HI will block allowing LO to run. (This is all OK so far.) But now MED unblocks and preempts LO. MED is not using the semaphore but HI is waiting for MED to complete, so that LO can run and release the semaphore.
Semaphores
Mutexes

- Mutual exclusion semaphores (mutexes) are similar to binary semaphores except they provide ownership and priority inversion avoidance. (The terms lock and unlock are often used with mutexes instead of acquire and release.)

- When a task locks a mutex only that task can release it. This is ownership. (In contrast, a semaphore can be released by any task.) RTEMS does not support ownership.
Semaphores
Mutexes

• Two common methods are used for avoiding priority inversion with mutexes:
  – The *priority inheritance protocol* raises the priority of the lower priority task to that of the higher priority task when the high priority task requests the mutex. The priority drops back when the mutex is released.
  – With the *ceiling priority protocol*, tasks acquiring a mutex have their priority raised to a max level.

• RTEMS supports both methods.
Semaphores
Typical Directives

- **Create, delete**: The same calls may be used to create both binary and counting semaphores. Separate routines are usually provided for mutexes. (RTEMS uses the same calls for semaphores and mutexes.)

- **Acquire, release**: Routines may be named *take* and *give*, *pend* and *post*, or *p* and *v*. (*lock* and *unlock* are common for mutexes) The acquire call may optionally take a timeout argument.
Semaphores
Typical Applications

• Here are some of the applications for which semaphores are used:
  – wait and signal synchronization
  – credit tracking synchronization
  – single shared resource access synchronization
  – multiple shared resource access synchronization
Semaphores
Wait and Signal Synch

• A binary semaphore is created with an initial value of 0 (unavailable). Task HI runs and blocks on acquire. This allows LO to run. LO completes its task and releases the semaphore. HI immediately preempts LO and continues running until it tries to acquire the semaphore again. HI blocks (HI already has the semaphore), allowing LO to run until LO releases the semaphore again.
Semaphores
Wait and Signal Synch

• This is known as a *unilateral rendezvous*. A *bilateral rendezvous* can be accomplished using two semaphores.

• This method can also be used to synchronize task activity with an interrupt service routine (ISR). The ISR should only release the semaphore, never acquire it. An ISR should never block!
Semaphores
Credit Tracking Synch

• An ISR (or HI task) can release a counting semaphore to indicate the number of occurrences of an event. A task (or another LO task) can acquire the semaphore and process the event.

• This is useful if the ISR (or HI task) releases semaphores in bursts. There must be sufficient catch-up time between bursts for the other task to process the events.
Semaphores
Single Shared Resource Access

• Single shared resource access is one of the more common uses of semaphores. We have a resource (printer, serial port, area of memory) that should only be accessed by a single task at a time.

• Use of the resource is bracketed by calls to acquire and release a semaphore (usually either a binary semaphore or a mutex).
Semaphores
Single Shared Resource Access

• You should take care to protect ALL global resources that are shared by tasks:
  
  ```
  static int counter;
  void foo() { counter++; }
  ```

• Is there a problem with the above code if only one task calls `foo()`? What if two separate tasks call `foo()`? Would the problem exist if `counter` could be incremented in a single machine language instruction?
Areas of code which manipulate shared data are known as **critical sections** and they must be protected. In the example on the previous slide the code might run properly 99.999% of the time without being protected. Bugs like this can be nearly impossible to find.

Although semaphores are often used to protect critical sections other methods may be available: disabling the scheduler, disabling interrupts.
Semaphores
Single Shared Resource Access

- If two or more resources are being protected by semaphores, you must take care to prevent deadlock (aka deadly embrace). Task 1 has exclusive access to resource R1, while Task 2 has exclusive access to resource R2. If T1 needs access to R2 and T2 needs access to R1, deadlock occurs.

- To prevent: (1) acquire all resources first, (2) acquire all resources in the same order, (3) release resources in reverse order.
Semaphores
Mult. Shared Resource Access

- A counting semaphore can be used to protect multiple equivalent shared resources. For example, a memory manager might have 10 blocks of memory available. A counting semaphore would be created and initialized to 10. Up to 10 tasks could simultaneously acquire the semaphore and use one of the memory blocks. An 11th task would block. Tasks release the semaphore to indicate that they are done with the memory block.
Semaphores

RTEMS Semaphores

- The directive `rtems_semaphore_create()` is used to create binary and counting semaphores. Attribute sets are passed as an argument to create different types.

- RTEMS defines both binary and simple binary semaphores. A simple binary semaphore does not allow nested access and can be deleted when locked. Simple binary semaphores must be used for task synchronization!
Semaphores
RTEMS Semaphores

• The full set of semaphore attributes includes:

  RTEMS_FIFO - tasks wait in FIFO order (default)
  RTEMS_PRIORITY - tasks wait in priority order
  RTEMS_BINARY_SEMAPHORE - only 0 and 1
  RTEMS_COUNTING_SEMAPHORE – any val. (def.)
  RTEMS_SIMPLE_BINARY_SEMAPHORE
  RTEMS_NO_INHERIT_PRIORITY - (default)
  RTEMS_INHERIT_PRIORITY - use prior. inheritance
  RTEMS_PRIORITY_CEILING - use priority ceiling
  RTEMS_NO_PRIORITY_CEILING - (default)
  RTEMS_LOCAL - local task (default)
  RTEMS_GLOBAL - global task
Semaphores
RTEMS Semaphores

• Here is the prototype for the create routine:

```c
rtems_status_code rtems_semaphore_create(
    rtems_name           name,
    rtems_unsigned32     count,
    rtems_attribute      attribute_set,
    rtems_task_priority  priority_ceiling,
    rtems_id             *id
);
```

• count is the initial value. priority_ceiling is only used when the priority ceiling attribute is used. The id is a return value and is used in other directives.
Semaphores
RTEMS Semaphores

- Attribute values are ORed to obtained a desired attribute set. For all default values use RTEMS_DEFAULT_ATTRIBUTES.
- The priority inheritance and priority ceiling attributes are only supported when the priority queue (not the default FIFO queue) attribute (RTEMS_PRIORITY) is also specified.