SYS/BIOS On-Line Training

Semaphores
Semaphores

- Semaphores are the basic blocking primitive
  - Semaphores store a count $\geq 0$
  - Posting a semaphore increments count
  - Pending on a semaphore decrements count

Semaphore_post(mySem);

Semaphore_pend(mySem);

Semaphore_post(mySem);
(Counting Semaphore)

Semaphore_post(mySem);
(Binary Semaphore)
Task Blocks if Pend on Sem With Count = 0

Semaphore_pend(mySem);
Task Continues if Pend on Sem With Count > 0

Semaphore_pend(mySem);

myTask (RUNNING)

mySem

2

mySem

1
Task Unblocks if Post on Count=0 Semaphore

Semaphore_post(mySem);

Semaphore_pend(mySem);

*
Multiple Tasks Can Block on Same Semaphore

Semaphore_post(mySem);  
lowPriTask (BLOCKED)

Semaphore_pend(mySem); Semaphore_pend(mySem);

hiPriTask (BLOCKED)  
medPriTask (BLOCKED)

Semaphore_pend(mySem);

mySem 0  
lowPriTask (READY)  
hiPriTask (RUNNING)  
medPriTask (BLOCKED)  
mySem 0
Binary Semaphore

Used to Protect a critical section or resource from multiple tasks

Initial count of binary semaphore mySem=1

Void lowPriTask(UArg arg0, UArg arg1)
{
    for (;;) {
        Semaphore_pend(mySem, BIOS_WAIT_FOREVER);

        resource += 1; /* do work on locked resource */

        Semaphore_post(mySem);
    }
}

Void hiPriTask(UArg arg0, UArg arg1)
{
    for (;;) {
        Semaphore_pend(mySem, BIOS_WAIT_FOREVER);

        resource += 1; /* do work on locked resource */

        Semaphore_post(mySem);
        Task_sleep(5); /* allow low pri task to work */
    }
}
### Counting Semaphores

- A counting semaphore can control access to multiple resources.
- The restaurant analogy:
  - Tasks are customers, Tables are resources, the Maitre d’ is the semaphore.
  - Initial semaphore count = number of tables.
  - Seating a table is done with a Semaphore_pend() – if no tables available (count = 0), the semaphore blocks (nobody can be seated).
  - Customers leaving a table done with a Semaphore_post().
  - Key points: a single semaphore can do it all, atomicity is important.
## Task Run-Time API

```c
Task_Params taskParams;
Task_Handle myTask;
Error_block eb;

Task_Params_init(&taskParams);
taskParams.stackSize = 512;
taskParams.priority = 15;
Error_init(&eb);
myTask = Task_create((Task_FuncPtr)hiPriTask, &taskParams, &eb);
```

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task_create</td>
<td>Allocate and initialize a new instance object and return its handle</td>
</tr>
<tr>
<td>Task_delete</td>
<td>Finalize and free this previously allocated instance object, setting the referenced</td>
</tr>
<tr>
<td>Task_disable</td>
<td>Disable the task scheduler</td>
</tr>
<tr>
<td>Task_exit</td>
<td>Terminate execution of the current task</td>
</tr>
<tr>
<td>Task_getEnv</td>
<td>Get task environment pointer</td>
</tr>
<tr>
<td>Task_getIdleTask</td>
<td>Returns a handle to idle task object</td>
</tr>
<tr>
<td>Task_getMode</td>
<td>Retrieve the Mode of a task</td>
</tr>
<tr>
<td>Task_getPri</td>
<td>Get task priority</td>
</tr>
<tr>
<td>Task_Params_init</td>
<td>Initialize this config-params structure with supplier-specified defaults before</td>
</tr>
<tr>
<td>Task_restore</td>
<td>Restore Task scheduling state</td>
</tr>
<tr>
<td>Task_self</td>
<td>Returns a handle to the currently executing Task object</td>
</tr>
<tr>
<td>Task_setEnv</td>
<td>Set task environment</td>
</tr>
<tr>
<td>Task_setPri</td>
<td>Set a task's priority</td>
</tr>
<tr>
<td>Task_sleep</td>
<td>Delay execution of the current task</td>
</tr>
<tr>
<td>Task_stat</td>
<td>Retrieve the status of a task</td>
</tr>
<tr>
<td>Task_yield</td>
<td>Yield processor to equal priority task</td>
</tr>
</tbody>
</table>
Creating a Semaphore With XGCONF

Name: string passed to global.h for runtime access of object

count: initial count of sem when created

mode: selects binary or counting semaphore type